What is Computer Science?
An Information Security Perspective

D. Page and N.P. Smart
<{page,nigel}@cs.bris.ac.uk>
# CONTENTS

## 1 Introduction
- 1.1 Intended audience ................................................. 14
- 1.2 Overview of content ............................................. 14
  - 1.2.1 Core material .................................................. 14
  - 1.2.2 Supplementary material ..................................... 16
  - 1.2.3 Embedded tasks ............................................... 16
  - 1.2.4 Notation ........................................................ 16
- 1.3 Frequently Asked Questions (FAQs) .............................. 18
- 1.4 Acknowledgements ................................................. 19

## I Foundations

### 2 Compressing and correcting digital media
- 2.1 A compact disk = a sequence of numbers .......................... 25
  - 2.1.1 Decimal and binary representation .......................... 25
  - 2.1.2 Decimal and binary notation ................................ 29
  - 2.1.3 Grouping bits into bytes .................................... 29
- 2.2 Data compression .................................................. 30
  - 2.2.1 A run-length based approach ................................. 31
  - 2.2.2 A dictionary-based approach ................................. 32
- 2.3 Error correction ................................................... 34
  - 2.3.1 An error detection approach ................................ 35
  - 2.3.2 An error correction approach ................................ 36
- 2.4 Recasting error correction as matrix arithmetic ................. 38
  - 2.4.1 An overview of vectors ...................................... 39
  - 2.4.2 An overview of matrices .................................... 40
  - 2.4.3 Addition and multiplication modulo 2 ..................... 41
  - 2.4.4 Using matrices for error correction ......................... 41
  - 2.4.5 Generalising the matrix-based (7,4)-code .................. 43

### 3 Writing and comparing algorithms
- 3.1 Algorithms .......................................................... 47
  - 3.1.1 What is an algorithm? ........................................ 48
  - 3.1.2 What is not an algorithm? .................................... 49
- 3.2 Algorithms for multiplication ...................................... 51
- 3.3 Algorithms for exponentiation ..................................... 54
- 3.4 Computational complexity .......................................... 56
  - 3.4.1 Step counting and dominant steps .......................... 56

## II Algorithms in practice
http://www.cs.bris.ac.uk

<table>
<thead>
<tr>
<th>Section</th>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2</td>
<td>Problem size and step counting functions</td>
<td>57</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Ignoring small problems and minor terms</td>
<td>58</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Studying the growth of functions</td>
<td>60</td>
</tr>
<tr>
<td>4</td>
<td>Playing hide-and-seek with virus scanners</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>Computers and programs</td>
<td>66</td>
</tr>
<tr>
<td>4.1.1</td>
<td>A theoretical computer</td>
<td>67</td>
</tr>
<tr>
<td>4.1.2</td>
<td>A real, Harvard-style computer</td>
<td>68</td>
</tr>
<tr>
<td>4.1.3</td>
<td>A real, von Neumann-style computer</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>Harvard versus von Neumann computers</td>
<td>76</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Beyond straight-line programs</td>
<td>76</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Toward self-modifying programs</td>
<td>77</td>
</tr>
<tr>
<td>4.3</td>
<td>A self-modifying virus</td>
<td>79</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Using XOR to mask numbers</td>
<td>79</td>
</tr>
<tr>
<td>4.3.2</td>
<td>A virus that masks the payload</td>
<td>80</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Preventing the virus without a virus scanner</td>
<td>81</td>
</tr>
<tr>
<td>5</td>
<td>How long is a piece of string?</td>
<td>85</td>
</tr>
<tr>
<td>5.1</td>
<td>String data structures</td>
<td>85</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Problem #1: representing characters</td>
<td>86</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Problem #2: representing strings</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>String algorithms</td>
<td>90</td>
</tr>
<tr>
<td>5.2.1</td>
<td>strlen: finding the length of a string</td>
<td>90</td>
</tr>
<tr>
<td>5.2.2</td>
<td>toupper: converting a string to upper-case</td>
<td>92</td>
</tr>
<tr>
<td>5.2.3</td>
<td>strcmp: testing if one string is the same as another</td>
<td>94</td>
</tr>
<tr>
<td>5.2.4</td>
<td>strcat: concatenating two strings together</td>
<td>95</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Problem #3: repeated concatenation</td>
<td>98</td>
</tr>
<tr>
<td>6</td>
<td>Demystifying web-search: the Mathematics of PageRank</td>
<td>103</td>
</tr>
<tr>
<td>6.1</td>
<td>PageRank: the essence of Google web-search</td>
<td>103</td>
</tr>
<tr>
<td>6.1.1</td>
<td>What actually is web-search?</td>
<td>103</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Web-search before Google</td>
<td>104</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Web-search after Google</td>
<td>105</td>
</tr>
<tr>
<td>6.2</td>
<td>Using graph theory to model and explore the web</td>
<td>106</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Graph traversal</td>
<td>110</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Graph exploration</td>
<td>113</td>
</tr>
<tr>
<td>6.3</td>
<td>Using probability theory to model web-browsing</td>
<td>113</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Sanitising the web-graph to avoid a subtle problems</td>
<td>114</td>
</tr>
<tr>
<td>6.3.2</td>
<td>A Mathematical approach to computing PageRank</td>
<td>115</td>
</tr>
<tr>
<td>6.4</td>
<td>Putting it all together: using PageRank to produce web-search results</td>
<td>120</td>
</tr>
<tr>
<td>II</td>
<td>Cryptography</td>
<td>125</td>
</tr>
<tr>
<td>7</td>
<td>Using short programs to make and break historical ciphers</td>
<td>127</td>
</tr>
<tr>
<td>7.1</td>
<td>Shift ciphers</td>
<td>129</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Encryption and decryption</td>
<td>129</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Cryptanalysis</td>
<td>133</td>
</tr>
<tr>
<td>7.2</td>
<td>Substitution ciphers</td>
<td>135</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Encryption and decryption</td>
<td>135</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Cryptanalysis</td>
<td>137</td>
</tr>
<tr>
<td>8</td>
<td>The Enigma of group theory</td>
<td>145</td>
</tr>
<tr>
<td>8.1</td>
<td>Permutations</td>
<td>146</td>
</tr>
<tr>
<td>8.2</td>
<td>An equation for the Enigma</td>
<td>149</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Choice of rotors</td>
<td>149</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Reflector</td>
<td>150</td>
</tr>
<tr>
<td>8.2.3</td>
<td>An Enigma key</td>
<td>150</td>
</tr>
<tr>
<td>8.2.4</td>
<td>The encryption operation</td>
<td>150</td>
</tr>
<tr>
<td>8.3</td>
<td>The Polish work on Enigma</td>
<td>151</td>
</tr>
</tbody>
</table>
13.3.2 A normal, valid request ................................................................. 253
13.3.3 A not-so-normal, attack request ................................................... 254
13.3.4 The end result .............................................................................. 255

14 Is the key to your cash being leaked by your cache? 259
14.1 Cache memories ................................................................................ 259
14.1.1 The design of a cache mechanism .................................................. 261
14.1.2 Reasoning about cache effectiveness .............................................. 266
14.2 On straight-line execution ................................................................. 266
14.2.1 Applying specialisation and loop unrolling .................................... 266
14.2.2 Estimating execution time ............................................................. 268
14.3 An example block cipher .................................................................. 269
14.3.1 Applying the substitution step using an S-box ............................... 269
14.3.2 Implementing a simpler, straight-line version of the cipher .......... 276
14.4 A side-channel attack based on cache behaviour ............................. 277
14.4.1 A brute-force attack .................................................................... 277
14.4.2 A side-channel attack .................................................................. 278

V Appendices ......................................................................................... 285

A A short BASH tutorial ......................................................................... 287
A.1 Some basic, background concepts and terminology .......................... 288
A.1.1 Issuing commands to execute processes ......................................... 290
A.1.2 Controlling execution using options and arguments ...................... 291
A.1.3 Controlling input and output streams .......................................... 291
A.2 A limited BASH command reference ............................................... 294
A.2.1 alias ............................................................................................. 294
A.2.2 cat ............................................................................................... 295
A.2.3 umask and chmod ........................................................................ 296
A.2.4 cut ............................................................................................... 298
A.2.5 diff ............................................................................................... 299
A.2.6 du ................................................................................................. 300
A.2.7 echo ............................................................................................. 301
A.2.8 fold ............................................................................................. 301
A.2.9 grep ............................................................................................. 302
A.2.10 bzip2, gzip and zip .................................................................... 303
A.2.11 head and tail ............................................................................... 305
A.2.12 ls ............................................................................................... 306
A.2.13 od ............................................................................................... 308
A.2.14 paste .......................................................................................... 310
A.2.15 sort ............................................................................................. 310
A.2.16 tr ................................................................................................. 312
A.2.17 uniq ........................................................................................... 313
A.2.18 wc ............................................................................................... 314
A.2.19 wget .......................................................................................... 315

B A tour of computer networking and cryptography-based network security 319
B.1 Some basic, background concepts and terminology ......................... 320
B.1.1 Exploring IP and DNS information for a host .............................. 321
B.1.2 Checking a host is active using ping ............................................ 323
B.1.3 Exploring the path between hosts using traceroute ...................... 324
B.1.4 Inspecting network configuration using netstat ............................ 324
B.2 Networked inter-process communication using netcat ..................... 328
B.2.1 A simple messaging system .......................................................... 328
B.2.2 Some simple HTTP-based applications ....................................... 330
B.3 Using OpenSSL-based cryptographic primitives ............................ 333
B.3.1 Symmetric encryption and decryption operations ......................... 334
B.3.2 Cryptographic hash and MAC operations .................................... 337
B.3.3 Asymmetric encryption and decryption operations ....................... 339
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Two adverts published in <em>Byte</em> magazine issues 11/88 and 2/89, from the late 1980s.</td>
<td>24</td>
</tr>
<tr>
<td>2.2</td>
<td>An illustration of pits and lands on a CD surface.</td>
<td>26</td>
</tr>
<tr>
<td>2.3</td>
<td>A Venn diagram describing the computation of error correction bits in a (7,4)-code.</td>
<td>37</td>
</tr>
<tr>
<td>2.4</td>
<td>A table detailing error correction for the (7,4)-code.</td>
<td>38</td>
</tr>
<tr>
<td>2.5</td>
<td>A flow chart description of FERMAT-TEST.</td>
<td>50</td>
</tr>
<tr>
<td>2.6</td>
<td>A table showing values of ( y ) and the number of bits in their binary representation.</td>
<td>57</td>
</tr>
<tr>
<td>2.7</td>
<td>A table showing the number of additions performed by MULTIPLE-REPEAT and MULTIPLE-HORNER, and the values of the associated step counting functions.</td>
<td>57</td>
</tr>
<tr>
<td>2.8</td>
<td>A graph illustrating the growth of several functions as the problem size is increased.</td>
<td>59</td>
</tr>
<tr>
<td>3.1</td>
<td>A flow chart description of FERMAT-TEST.</td>
<td>70</td>
</tr>
<tr>
<td>3.2</td>
<td>Computing the sum 10 + 20, as executed on a Harvard-style computer.</td>
<td>71</td>
</tr>
<tr>
<td>3.3</td>
<td>Computing the sum 10 + 20, as executed on a Harvard-style computer.</td>
<td>71</td>
</tr>
<tr>
<td>3.4</td>
<td>Computing the sum 10 + 20, as executed on a Harvard-style computer.</td>
<td>71</td>
</tr>
<tr>
<td>4.1</td>
<td>Computing the sum 10 + 20, as executed on a Harvard-style computer.</td>
<td>71</td>
</tr>
<tr>
<td>4.2</td>
<td>Computing the sum 10 + 20, as executed on a Harvard-style computer.</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>Computing the sum 10 + 20, as executed on a stored program computer.</td>
<td>71</td>
</tr>
<tr>
<td>4.4</td>
<td>Computing the sum 10 + 20, as executed on a stored program computer.</td>
<td>71</td>
</tr>
<tr>
<td>4.5</td>
<td>Computing the sum 10 + 20, as executed on a stored program computer.</td>
<td>71</td>
</tr>
<tr>
<td>4.6</td>
<td>Computing the sum 10 + 20, as executed on a stored program computer.</td>
<td>71</td>
</tr>
<tr>
<td>4.7</td>
<td>Computing the sum 10 + 20, as executed on a stored program computer.</td>
<td>71</td>
</tr>
<tr>
<td>4.8</td>
<td>Computing the sum 10 + 20, as executed on a stored program computer.</td>
<td>71</td>
</tr>
<tr>
<td>4.9</td>
<td>Computing the sum 10 + 20, as executed on a stored program computer.</td>
<td>71</td>
</tr>
<tr>
<td>5.1</td>
<td>A moth found by operations of the Harvard Mark 2; the “bug” was trapped within the computer and caused it to malfunction (public domain image, source: <a href="http://en.wikipedia.org/wiki/File:H96566k.jpg">http://en.wikipedia.org/wiki/File:H96566k.jpg</a>).</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>A table describing the printable ASCII character set.</td>
<td>86</td>
</tr>
<tr>
<td>5.3</td>
<td>Algorithms to compute the length of a string at address ( x ), represented using a P-string (left-hand side) or C-string (right-hand side) data structure.</td>
<td>87</td>
</tr>
<tr>
<td>5.4</td>
<td>Algorithms to convert a string at address ( x ), represented using a P-string (left-hand side) or C-string (right-hand side) data structure, into upper-case.</td>
<td>91</td>
</tr>
<tr>
<td>5.5</td>
<td>Algorithms to match one string at address ( x ) against another at address ( y ), both represented using a P-string (left-hand side) or C-string (right-hand side) data structure.</td>
<td>93</td>
</tr>
</tbody>
</table>
5.6 Algorithms to concatenate (i.e., join) one string at address $y$ onto the end of another at address $x$, both represented using a P-string (left-hand side) or C-string (right-hand side) data structure. 97

6.1 Two approaches to traversal of vertices in a graph. ............................................. 106
6.2 A simple, concrete web-graph that captures the link structure between, and content of each web-page; each of the $n = 6$ highly artificial web-pages is a short HTML file, which in combination provide structure (i.e., links between the web-pages) and content (i.e., some words, in this case names of fruit). ................................................................. 107
6.3 A simple 7-vertex tree used to explain the behaviour of BFS and DFS algorithms. ........ 107
6.4 Two simple, abstract web-graphs (derived from Figure 6.2) that capture the link structure between, but not content of each web-page. .................................................. 108

7.1 The message that uncovered the Babington Plot: masquerading as part of a message from Mary Stewart, the postscript asks Babington to reveal the names of the conspirators using the broken cipher (public domain image, source: http://en.wikipedia.org/wiki/File:Babington_postscript.jpg) ................................................................. 128
7.2 An analogue clock face showing the time three o’clock ............................................... 132
7.3 An example of the “dancing men” used as a cipher in The Adventure of the Dancing Men (public domain image, source: http://en.wikipedia.org/wiki/File:Dancing_men.png) ................................................. 135

8.1 Simplified Enigma machine ......................................................................................... 146

9.1 Scenes from Press Your Luck, 1984. ........................................................................ 160

10.1 Two analogue clock faces, one standard 12-hour version and a slightly odd 13-hour alternative. 179
10.2 Some algorithms for modular arithmetic. ................................................................. 185

11.1 An example rasterised image in two representations. ............................................... 205
11.2 Some example vector image transformations. .......................................................... 208
11.3 Scenes from You Only Live Twice, 1967 © United Artists; the microdot text translates as “photograph taken by female American tourist from coastal vessel; the woman has been liquidated as a routine precaution” . ................................................................. 211
11.4 Four images, each created by altering the pixels from an original image. .................. 212
11.5 The result of injecting a 6-character ASCII message into the LSBs of an eight pixel image. 213
11.6 A “real” microdot embedded into a full stop. .......................................................... 214

12.1 Scenes from WarGames, 1983 © MGM Pictures. ..................................................... 218
12.2 An algorithm to test whether the guess $G$ matches some password $P$ (where both $G$ and $P$ are strings).[center] ................................................................. 220
12.3 Four example countermeasures to harden MATCH-PWD against side-channel attack. .... 223
12.4 Two algorithms to test whether the guess $G$ matches some PIN $P$ (where both $G$ and $P$ are numbers). ................................................................. 225
12.5 Demonstrating viability of the SIM card physical attack. .......................................... 226

13.1 Two algorithms used as a running example. ............................................................. 234
13.2 Attempt #1 at implementing MAIN and ADD. ......................................................... 235
13.3 An altered MAIN that now invokes ADD twice rather than once. ............................. 236
13.4 Two algorithms to compute the $n$-th Fibonacci number. ......................................... 236
13.5 The Droste effect whereby an image contains a smaller instance of itself (public domain image, source: http://en.wikipedia.org/wiki/File:Droste.jpg) ................................................................. 237
13.6 Attempt #2 at implementing MAIN and ADD. ......................................................... 243
13.7 Attempt #3 at implementing MAIN and ADD. ......................................................... 245
13.8 Attempt #4 at implementing MAIN and ADD. ......................................................... 247
13.9 Attempt #5 at implementing MAIN and ADD. ......................................................... 249
13.10 Two algorithms used to represent the web-server subjected to a buffer overflow attack. 251
13.11 A program that implements the WEB-SERVER and Process-Req algorithms. .......... 252

14.1 The behaviour of a small direct-mapped cache when a sequence of loads from addresses 1, 34, 35, 36, 37, 1, 38, 39 and 40 is performed. ................................................................. 265
14.2 An example cipher, somewhat similar to AES, which acts as the target of an example cache-based side-channel attack. ................................................................. 270
14.3 The original example cipher specialised and unrolled for \( n = 2 \) and \( r = 2 \) to yield simpler descriptions, and now using the ET-Tab and DT-Tab tables rather than the ET and DT functions.

14.4 ET-Tab, a tabular description of ET. ................................................................. 271

14.5 DT-Tab, a tabular description of DT. ................................................................. 272

14.6 A diagrammatic description of \( \text{Enc} \) and \( \text{Dec} \) invoked using \( n = 2 \) and \( r = 2 \); the left-hand portion represents the encryption of the plaintext message \( P \) using the key \( K \), while the right-hand portion represents decryption of the resulting ciphertext \( C \). .................................................. 273

14.7 A partial (i.e., omitting most of the ET-Tab table) implementation of the \( \text{Enc} \) algorithm using the example computer. ................................................................. 275

14.8 Graphs showing variation in the execution time caused by different plaintexts of the form \( P = (f, p) \) for fixed \( f \) and variable \( p \). ................................................................. 279

A.1 A rough, and in places subjective comparison (using only a few criteria) between GUI- and CLI-based user interfaces. ................................................................. 289

A.2 An example BASH session within a UNIX-based terminal window. ...................... 290

B.1 Three scenarios describing use of \( \text{nc} \) to send from a source host to a target host. ........ 329

B.2 Example output from \( \text{nc} \) when used as a simple web-browser to load the web-page index. html from www.cs.bris.ac.uk. ................................................................. 331
CHAPTER

1

INTRODUCTION

Computer Science is a diverse subject: it covers topics that range from the design of computer hardware through to programming that hardware so it can do useful things, and encompasses applications in fields such as Science, Engineering and the Arts. As a result, a modern Computer Science degree will often see students study topics that could be described as applied Mathematics (e.g., cryptography), applied Physics (e.g., quantum computing), Psychology (e.g., human-computer interaction), Philosophy (e.g., artificial intelligence), Biology (e.g., bio-informatics) and Linguistics (e.g., speech synthesis); such a list could go on and on.

On one hand, this diversity means trying to capture what Computer Science is can be quite hard and the subject can be misunderstood as a result. A prime example is that ICT, a subject often taught in schools, can give a false impression of what Computer Science would be about at University: if ICT gives the impression that a Computer Science degree would be about using a word processor for example, the reality is it relates more closely to creating the word processor itself. The knock-on effect of this potential misunderstanding is that recruiting students to do Computer Science degrees is much harder work than you might think.

But on the other hand, the same diversity is tremendously exciting. Bjarne Stroustrup, inventor of the C++ programming language, is quoted as stating that “our civilisation runs on software” and therefore, by implication, the Computer Scientists who create it do likewise. From the perspective of other fields this might seem like a wild claim; for example, an Engineer might argue various wonders of the ancient world were realised without assistance from computers! However, the increasing ubiquity of computing in our lives somewhat justifies the statement. Beyond software running on desktop and laptop computers, we are increasingly dependent on huge numbers of devices that have permeated society in less obvious ways. Examples include software running on embedded computers within consumer products such as automobiles and televisions, and software running on chip-and-pin credit cards. All of these examples form systems on which we routinely depend. From domestic tasks to communication, from banking to travel: Computer Science underpins almost everything we do, even if you cannot see it on the surface.

Beyond the challenge this presents, it has tangible advantages in terms of employment: Computer Science graduates are regularly snapped up by industries as diverse as finance, media and aerospace for example. The reason is obvious: of course they are highly skilled in the use and creation of technology, but more importantly they have been trained to understand and solve problems, and to challenge what is possible. These skills in particular turn out to be the “gold dust” that distinguishes graduates in Computer Science from other subjects; they also represent the main reason why such a high proportion of those graduates become entrepreneurs by developing their own ideas into their own businesses.

So our goal here is to answer questions about, and encourage you to be interested in, the subject of Computer Science. More specifically, we want to answer questions such as “what is Computer Science” and “why should I study it at University”. Of course, to some extent the answers will differ for everyone; different topics within the subject excite different people for example. But rather than give a very brief overview of many topics, our approach is to showcase specific topics all roughly relating to one theme: information security. This is a slightly subjective choice, but it turns out to allow a very natural connection between lots of things you are already familiar with (e.g., how compact discs work, or the problem of
computer viruses) and theory in Computer Science that can help explain them. In each case, we try to demonstrate how the topic relates to other subjects (e.g., Mathematics) and how it allows us to solve real problems. You can think of each one as a “mini” lecture course: if you find one or more of them interesting, the chances are you would find a Computer Science degree interesting as well.

1.1 Intended audience

This book attempts a careful balancing act between two intended types of reader, who have related but somewhat different needs. Clearly the book can still be useful if you fall outside this remit, but by focusing on these types we can make some assumptions about what you already know and hence the level of detail presented.

Students  Our primary audience are students looking for an introduction to Computer Science. Two examples stand out:

1. A student studying Mathematics (perhaps ICT or even Computer Science) in later years of secondary or further education.

   In this case our goal is to give a taste of what Computer Science is about, and to provide a connection to your existing studies. Either way, we view you as a target to recruit into a Computer Science degree at University!

2. A student in early years of a Computer Science degree at University, or taking flavours of such a degree from within another subject (e.g., as an optional or elective course).

   In this case our goal is to (re)introduce topics and challenges you may already be faced with; offering a new, and hopefully accessible perspective can help understand basic concepts and motivate further study by illustrating concrete applications.

Teachers  Our secondary audience are teachers. Anecdotally at least, we have found two use-cases that seem important:

1. Where there is a lack of specialist staff, non-specialists or early career staff members are often tasked with teaching Computer Science. In some cases, for example with content in Mathematics, there is clear synergy or even overlap. Either way however, this task can be very challenging.

   In this case the book can be used more or less as a student would. Although you might absorb the material quicker or more easily, it still offers an good introduction to the subject as a whole.

2. Even with specialist staff, it can be hard to find and/or develop suitable material; there are an increasing number of good resources online, but still few that focus on fundamentals of the subject and hence form a link to University-level study.

   In this case, the book can act as a useful way to enrich existing lesson plans, or as further reading where appropriate. The tasks embedded in the material, in particular, offer potential points for discussion or work outside the classroom.

1.2 Overview of content

Some books are intended to be read from front-to-back in one go; others are like reference books, where you can dive into a specific part if or when you need to. This book sits somewhere between these types. It is comprised of various parts as explained below, each with specific goals and hence a suggested approach to reading the associated material.

1.2.1 Core material

The first part is concerned with the fundamentals of Computer Science, and outlines concepts used variously elsewhere in the book. As a result, each Chapter in this part should ideally be read in order, and before starting any other part:
Chapter 2 introduces the idea that numbers can be represented in different ways, and uses this to discuss the concepts of data compression, and error detection and correction. The example context is CDs and DVDs. Both store vast amounts of data and are fairly robust to damage; the question is, how do they do this?

Chapter 3 is a brief introduction to the study of algorithms, i.e., formal sets of directions which describe how to perform a task. The idea is to demonstrate that algorithms are fundamental tools in Computer Science, but no more complicated than a set of driving directions or cookery instructions. By studying how algorithms behave, the Chapter shows that we can compare them against each other and select the best one for a particular task.

Chapter 4 uses the example of computer viruses to show how computers actually work, i.e., how they are able to execute the programs (or software) we use every day. The idea is to show that there is no magic involved: even modern computers are based on fairly simple principles which everyone can understand.

Chapter 5 highlights the role of data structures, the natural companion to algorithms. By using the example of strings (which are sequences of characters) the Chapter shows why data structures are needed, and how their design can have a profound influence on algorithms that operate on them.

Chapter 6 deals with one of the most ubiquitous and influential information systems available on the Internet: Google web-search. Using only fairly introductory Mathematics, it explains how the system seems able to “understand” the web and hence produce such good results for a given search query.

The second and third parts focus on cryptography and information security in more detail. After completing the first part, the idea is that each Chapter in the second and third parts can be read more or less independently from the rest:

Chapter 7 gives a brief introduction to what we now regard as historical schemes for encrypting messages. The goal is to demonstrate the two sides of cryptography: the constructive side where new schemes are designed and used, and the destructive side where said schemes are attacked and broken. By using simple programs available on every UNIX-based computer, the Chapter shows how historical cryptanalysts were able to decrypt messages their sender thought were secure.

Chapter 8 overviews an important and famous story within cryptography: how the Enigma machine was designed and used by Germany during WW2, and how Allied cryptographers subsequently broke the security it provided. This story is pivotal within WW2; breaking the Enigma machine is usually credited as being fundamentally important to the resulting Allied victory.

Chapter 9 discusses the idea of randomness: what does random even mean, and when can we describe a number as random? The idea is that random numbers often play a crucial role in cryptography, and using just some simple experiments one can demonstrate the difference between “good” and “bad” randomness.

Chapter 10 shifts the focus from history to the present day. By giving a more complete overview of a specific area in Mathematics (i.e., the idea of modular arithmetic) it describes two modern cryptographic schemes that more or less all of us rely on every day.

Chapter 11 introduces the concept of steganography which relates to hiding secret data within non-secret data. This allows a discussion of how computers represent and process images (e.g., those taken with a digital camera) and how simple steganographic techniques can embed secret messages in them.

Chapter 12 approaches the idea of security from a different perspective than the description in Chapter 10. As well as reasoning in theory about how secure a system is, we also need to worry about whether the physical system leaks information or not. The example scenario is guessing passwords: can you break into a computer without having to try every possible password, i.e., use any leaked information to help you out?

The final, fourth part contains what we term advanced material. The idea is to really challenge yourself: that does not mean the content is inaccessible, but it does mean that a solid understanding of the previous parts is required before even thinking about this part. Each Chapter takes a similar approach. The first half introduces a fairly challenging concept in Computer Science, and the second half relates that concept to cryptography and information security using what you have learnt:
Chapter 13 outlines a fairly technical example of how programming mistakes (or at least oversights) can lead to security problems. The idea is to get a rough idea of how so-called buffer overflow attacks work, relating such attacks to what we already know about how computers execute programs.

Chapter 14 shows how over the years, improvements to the speed at which memory can be accessed have been central to improving the overall speed of computers. However, this often mean some memory accesses are faster than others. The idea is to show how such improvements are possible, and how they can leak information and hence permit an interesting (although quite technical) method of attack.

1.2.2 Supplementary material

In addition to electronic copies of each core Chapter, various supplementary material is available online at http://www.cs.bris.ac.uk/home/page/teaching/wics.html

This includes additional Chapters extending the range of topics covered, plus Appendices providing extra introduction and explanation for topics we think might need it.

Perhaps the single most important instance is an Appendix supporting the use of BASH within examples and tasks. If you have no experience with BASH, which is common, the examples can be confusing. As such, we have written a short tutorial that includes a high-level overview of BASH itself, plus lower-level explanations of every command used in the book. The best approach is arguably to use it for reference as you read through each Chapter: whenever you encounter something unfamiliar or confusing, take some time to look through the Appendix which should provide an explanation.

1.2.3 Embedded tasks

To ensure the content is as practically oriented as possible, various tasks are embedded into the material alongside the fixed examples. These fall into various categories:

Implement (task #1)

These tasks represent implementation challenges, usually with a clear or fixed answer or outcome. In some cases the task might act as a prompt to reproduce or extend an example; in other cases the task might ask you to design something, e.g., an algorithm based on an information description of something in the material.

Research (task #2)

These tasks outline topics that represent a good next step on from the material presented: you are interested in the material but want to learn more, or about a specific aspect in more depth, these give some suggestions of what to look at. They often present open-ended challenges or questions, and as a result often make good discussion points. Either way, the idea is that you stop reading through the material, and attempt to solve the task yourself (rather than rely on the resources provided).

Some tasks of this type will be harder than others, but none are designed to represent a significant amount of work: if you get stuck on one, there is no problem with just skipping it and moving on.

1.2.4 Notation

Throughout the book we have tried to make the notation used as simple and familiar as possible. On the other hand some notation is inevitable: we need a way to express sets and sequences for instance.

1.2.4.1 Ranges

When we write \( a \ldots b \) for a starting point \( a \) and a finishing point \( b \), we are describing a range that includes all numbers between (and including) \( a \) and \( b \). So writing \( 0 \ldots 7 \) is basically the same as writing \( 0, 1, 2, 3, 4, 5, 6, 7 \). If we say \( c \) is in the range \( 0 \ldots 7 \) we mean that

\[
0 \leq c \leq 7
\]

i.e., \( c \) is one of \( 0, 1, 2, 3, 4, 5, 6 \) and \( 7 \).
1.2.4.2 Sequences

We write a sequence of elements called $A$, which you can think of as like a list, as follows

$$A = \{0, 3, 1, 2\}.$$  

This sequence contains elements which are numbers, but it is important to keep in mind that elements can be any objects we want. For example we could write a sequence of characters such as

$$B = \langle 'a', 'b', 'c', 'd', 'e' \rangle.$$  

Either way, we know the size of $A$ and $B$, i.e., the number of elements they contain; in the case of $A$ we write this as $|A|$ so that $|A| = 4$ for instance.

In a sequence, the order of the elements is important, and we can refer to each one using an index. When we want to refer to the $i$-th element in $A$ for example (where $i$ is the index) we write $A_i$. Reading the elements left-to-right, within $A$ we have that $A_0 = 0$, $A_1 = 3$, $A_2 = 1$ and $A_3 = 2$. Note that we count from zero, so the first element is $A_0$, and that referring to the element $A_4$ is invalid (because there is no element in $A$ with index 4). Using $\perp$ to mean invalid, we write $A_4 = \perp$ to make this more obvious.

Sometimes it makes sense to save space by not writing all the elements in a given sequence. For example we might rewrite $B$ as

$$B = \langle 'a', 'b', \ldots, 'e' \rangle,$$

where the continuation dots written as $\ldots$ represent elements ‘$c$’ and ‘$d$’ which have been left out: we assume whoever reads the sequence can fill in the $\ldots$ part appropriately. This means it should always be clear and unambiguous what $\ldots$ means. This way of writing $B$ still means we know what $|B|$ is, and also that $B_5 = \perp$ for example. Another example is the sequence $C$ written as

$$C = \langle 0, 3, 1, 2, \ldots \rangle.$$  

When we used continuation dots in $B$, there was a well defined start and end to the sequence so they were just a short-hand to describe elements we did not want to write down. However, with $C$ the continuation dots now represent elements either we do not know, or do not matter: since there is no end to the sequence we cannot necessarily fill in the $\ldots$ part appropriately as before. This also means we might not know what $|C|$ is, or whether $C_4 = \perp$ or not.

It is possible to join together, or concatenate, two sequences. For example, imagine we start with two 4-element sequences

$$D = \langle 0, 1, 2, 3 \rangle,$$
$$E = \langle 4, 5, 6, 7 \rangle,$$

and want to join them together; we would write

$$F = D \parallel E = \langle 0, 1, 2, 3 \rangle \parallel \langle 4, 5, 6, 7 \rangle = \langle 0, 1, 2, 3, 4, 5, 6, 7 \rangle.$$  

Notice that the result $F$ is an 8-element sequence, where $F_{0..3}$ are those from $D$ and $F_{4..7}$ are those from $E$.

1.2.4.3 Sets

The concept of a set and the theory behind such structures is fundamental to Mathematics. A set is an unordered collection of elements; as with a sequence, the elements can be anything you want. We can write a set called $A$ by listing the elements between a pair of braces as follows

$$A = \{2, 3, 4, 5, 6, 7, 8\}.$$  

This set contains the whole numbers between two and eight inclusive. The size of a set is the number of elements it contains. For the set $A$ this is written $|A|$, so we have that $|A| = 7$. If the element $a$ is in the set $A$, we say $a$ is a member of $A$ or write

$$a \in A.$$  

We know for example that $2 \in A$ but $9 \notin A$, i.e., 2 is a member of the set $A$, but 9 is not. Unlike a sequence, the ordering of the elements in a set does not matter, only their membership or non-membership. This means we cannot refer to elements in $A$ as $A_i$. However, if we define another set

$$B = \{8, 7, 6, 5, 4, 3, 2\},$$

we can be safe in the knowledge that $A = B$. Note that elements cannot occur in a set more than once.
As with sequences, it sometimes makes sense to save space by not writing all the elements in a set. For example we might rewrite the set \( A \) as
\[
A = \{2,3,\ldots,7,8\}.
\]
Sometimes we might want to write a set with unknown size such as
\[
C = \{2,4,6,8,\ldots\}.
\]
This set is infinite in size in the sense there is no end: it represents all even whole numbers starting at two and continuing to infinity. In this case, the continuation dots are a necessity; if we did not use them, we could not write down the set at all.

1.3 Frequently Asked Questions (FAQs)

I have a question/comment/complaint for you. Any (positive or negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way. To get in contact, email

page@cs.bris.ac.uk

or

nigel@cs.bris.ac.uk

We are not perfect, so mistakes are of course possible (although hopefully rare). Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

If programming really is your sole interest, you might prefer

S.S. Skiena and M.A. Revilla.

*Programming Challenges: The Programming Contest Training Manual.*


which offers a range of programming challenges; the excellent online resource

http://projecteuler.net/

is similar, although with greater emphasis on problems grounded in Mathematics.
But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it is possible to study programming as a topic in itself, we are more interested in what can be achieved using programs: we treat programming itself as another tool.

Are there any other things like this I can read? There are many books about specific topics in Computer Science, but somewhat fewer which overview the subject itself. Amongst these, some excellent examples are the following:


There are of course innumerable web-site, blog and wiki style resources online. Some structured examples include the CS4FN (or “Computer Science for fun”) series from Queen Mary, University of London, UK

http://www.dcs.qmul.ac.uk/cs4fn/

and Computer Science Unplugged series from the University of Canterbury, New Zealand

http://csunplugged.org/

the latter of which now also offers downloadable and printable books ideal for use in earlier stages of school.

1.4 Acknowledgements

This book was typeset with \LaTeX, originally developed by Leslie Lamport and based on \TeX by Donald Knuth; among the numerous packages used, some important examples include `adjustbox` by Martin Scharrer, `caption` and `subcaption` by Axel Sommerfeldt, `algorithm2e` by Christophe Fiorio, `listings` by Carsten Heinz, PGF and TikZ by Till Tantau, and pxfonts by Young Ryu. The embedded examples make heavy use of the BASH shell by Brian Fox, and numerous individual commands developed by members of the GNU project housed at

http://www.gnu.org

Throughout the book, images from sources other than the authors have been carefully reproduced under permissive licenses only; each image of this type notes both the source and license in question. The one caveat are a limited number of images and still frames included in Chapter 2, Chapter 9, Chapter 11, and Chapter 12. The non-commercial, educational nature of this book means we have included them based on fair use (or fair dealing); their copyright clearly remains with the original holder.

We owe a general debt of gratitude to everyone who has offered feedback, ideas or help with production and publication of this book. We have, in particular, benefited massively from tireless proof reading by Valentina Banciu, David Bernhard, Jake Longo Galea, Simon Hoerder, Daniel Martin, Luke Mather, Jim Page, and Carolyn Whitnall; any remaining mistakes are, of course, our own doing. We would like to acknowledge the support of the EPSRC (specifically via grant EP/H001689/1), whose model for outreach and public engagement was instrumental in allowing development of the material. Springer have published selected Chapters in
while *also* allowing us to keep electronic versions online. Any royalties from this published version are
donated to the Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

We thank Simon Rees and Wayne Wheeler (Springer) for making publication such a smooth process,
and additionally Simon Humphreys (CAS) and Jeremy Barlow (BCS) for dealing ably with non-standard
administrative overhead of the royalties arrangement.
Part I

Foundations
If you talk to most teenage mobile telephone users, they will be experts in data compression [3]. Sadly, money rather than Computer Science is their motivation. Short Message Service (SMS) text messages [17] cost the sender some money and, in addition, are limited to 160 characters per-message. Newer messaging services like Twitter derive their limits from SMS (140 characters is actually 160 characters minus 20 for any meta-data), but is less clear where the 160 character limit came from in the first place; it seems the designers just guessed this would be about right in 1985, and now we are stuck with it forever!

Although Twitter messaging is (currently) free, SMS turns out to be a relatively expensive means of communication. For example, Nigel Bannister, a lecturer at the University of Leicester, made a comparison used on the Channel 4 documentary called The Mobile Phone Rip-Off: at £0.05 per-message in 2008, transmitting 1 MB of data using SMS costs about £374.49 whereas transmitting the same amount from the Hubble Space Telescope [9] to earth only costs about £8.85!

But I digress. The point is that no matter what means of communication we choose, it makes sense to pack as much information into each message as we can. For example the message

"R U @ hm cuz I wnt 2 cm ovr"

is only 27 characters long and looks like gibberish [18]. To the trained eye, however, it easily translates into the message

"are you at home because I want to come over"

which is a massive 43 characters.

We have compressed the message: the first message is shorter than the second, and perhaps costs less as a result. Of course, there is a trade-off or balance between how much we save in terms of communication and how much work we do to compress and decompress the text: typically the more cryptic a text message, the harder it becomes for someone to understand it.

The idea of compressing data is not a new one, but is often hidden by new trends in technology. Only a generation ago for example, before the advent of high-speed broadband and wireless Internet connections, communication between computers was achieved using a MODEM [13]. The purpose of such a device was to convert digital data into analogue sounds that could be sent along a normal telephone line; this allowed computers to “talk” to each other. However, the speed at which they could talk was slow in comparison to the speed at which they did everything else. Early MODEMs could transmit data at less than 1 kB/s: I can remember transmitting the contents of 880 kB floppy disks to my friend using a MODEM, it literally took all day! People quickly realised that compressing the data first could help: if there was less data to send, it would take less time to send it and also reduce their telephone bill. Exactly the same story is true of storage. Again looking back only a generation ago, it would have been seen as quite extravagant to own a 50 MB...
Figure 2.1: Two adverts published in Byte magazine issues 11/88 and 2/89, from the late 1980s.
hard disk. Constrained by cost, people using even smaller hard disks than this realised that they could extend the limits of the space they did have by compressing files which were not in use. It was common practice, whenever they finished writing a document, to first save it onto the hard disk and then compress it so that it took up less space.

These days one can buy a 50 GB hard disk, so it might appear that the need for compression has disappeared: with large bandwidth and storage capacities, who really needs it? The problem is that people have a tendency to fill up whatever bandwidth or storage capacity is provided with new forms of data produced by new forms of application! As an example, consider the rise in use of digital photography: armed with digital cameras, people are now used to taking many photographs every time they do something or go somewhere. It is highly likely that the average person takes over 1000 pictures per-year; over a lifetime that is a lot of data to store!

So we can compress data and save money when we communicate or store it. The next problem is, when we receive or read the data how do we know that is what was meant? How do we know there were no errors that may have corrupted the data so that instead of

"R U @ hm cuz I wnt 2 cm ovr"

the recipient actually gets the even less intelligible

"Q T ? gl btx H vms 1 bl nuq"

i.e., each character is “off by one”. The answer lies in the two related techniques of error detection and error correction [5]. The idea is that we add some extra information to the data we communicate or store so that if there is an error it is at least apparent to us and, ideally, we can also correct it. Returning to reminiscing about MODEMs, the advantage of an error correction scheme should be apparent: computers used MODEMs to talk over noisy telephone lines. We have all used a telephone where there is a bad connection for example. Humans deal with this reasonably well because they can ask the person with whom they are talking to “say that again” if they do not understand something. Computers can do the same thing, but first they need to know that they do not understand what is being said; when the data they are communicating is simply numbers, how can a computer know that one number is right while another is wrong? Error detection and correction solve this problem and are the reasons why after a day spent sending the content of floppy disks via our MODEMs, my friend did not end up with 880 kB of nonsense data he could not use.

The goal of this Chapter is to investigate some schemes for data compression and error correction in a context which should be fairly familiar: we will consider data stored on a Compact Disk (CD) [2]. This is quite a neat example because when introduced in the early 1980s, the amount of data one could store on a CD and their resilience against damage were two of the major selling points. Both factors (plus some effective marketing) enabled the CD to replace previous technologies such as the cassette tape. The “official” CD specifications are naturally quite technical; our goal here is to give just a flavour of the underlying techniques using the CD as a motivating example.

### 2.1 A compact disk = a sequence of numbers

Roughly speaking, you can think of the content of a CD as being a long spiral track onto which tiny marks are etched: a pit is where a mark is made, a land is where no mark is made. The physical process by which a writer device performs the marking depends slightly on the CD type. But however it is written, the idea is that a reader device can inspect the surface of a CD and detect the occurrence of pits and lands. It is easy to imagine that instead of talking about pits and lands we could write down the content as a sequence such as

\[ A = \langle 0, 1, 0, 1 \rangle \]

where for the sake of argument, imagine a pit is represented by a 1 and a land is represented by a 0. Quite often it is convenient to interpret the CD content represented by \( A \) in different ways that suit whatever we are doing with it. To this end, we need to understand how numbers are represented by a computer.

#### 2.1.1 Decimal and binary representation

As humans, we are used to working with base-10 or decimal numbers because (mostly) we have ten fingers and toes; this means the set of valid decimal digits is \( \{0, 1, \ldots, 9\} \). Imagine we write down a decimal number such as 123. Hopefully you can believe this is sort of the same as writing the sequence

\[ B = \langle 3, 2, 1 \rangle \]
given that 3 is the first digit of 123, 2 is the second digit and so on; we are just reading the digits from left-to-right rather than from right-to-left. How do we know what 123 or \( B \) means? What is their value? In simple terms, we just weight each of the digits 1, 2 and 3 by a different amount and then add everything up. We can see for example that

\[
123 = 1 \cdot 100 + 2 \cdot 10 + 3 \cdot 1
\]

which we might say out loud as “one lot of hundred, two lots of ten and three units” or “one hundred and twenty three”. We could also write the same thing as

\[
123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0
\]

since any number raised to the power of zero is equal to one.

In our example, the sequence \( B \) consists of three elements; we can write this more formally by saying \( |B| = 3 \) meaning “the size of \( B \) is three”. The first element of the sequence is \( B_0 \) and clearly \( B_0 = 3 \); likewise for the second and third elements we have \( B_1 = 2 \) and \( B_2 = 1 \). It might seem odd naming the first element \( B_0 \) rather than \( B_1 \), but we almost always count from 0 rather than 1 in Computer Science. We can now rewrite

\[
123 = \sum_{i=0}^{3-1} B_i \cdot 10^i
\]

as the summation

\[
123 = \sum_{i=0}^{3-1} B_i \cdot b^i
\]

In words, the right-hand side means that for each index \( i \) between 0 and \( |B| - 1 \) (since \( |B| = 3 \) this means \( i = 0, i = 1 \) and \( i = 2 \) ) we add up terms that look like \( B_i \cdot b^i \) (i.e., the terms \( B_0 \cdot 10^0, B_1 \cdot 10^1 \) and \( B_2 \cdot 10^2 \)). This means we add up

\[
\begin{align*}
B_0 \cdot 10^0 &= 3 \cdot 10^0 = 3 \cdot 1 = 3 \\
B_1 \cdot 10^1 &= 2 \cdot 10^1 = 2 \cdot 10 = 20 \\
B_2 \cdot 10^2 &= 1 \cdot 10^2 = 1 \cdot 100 = 100
\end{align*}
\]

to make a total of 123 as expected. As a final step, we could abstract away the number 10 (which is called the base of our number system) and simply call it \( b \). This means that our number

\[
123 = \sum_{i=0}^{3-1} B_i \cdot b^i
\]

can be rewritten as

\[
123 = \sum_{i=0}^{3-1} B_i \cdot b^i
\]
A popular magic trick is based on binary representations of numbers: you might have seen the trick itself before, which is a common (presumably since it is inexpensive) prize inside Christmas crackers. The whole thing is based on 6 cards with numbers written on them:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>19</td>
<td>21</td>
<td>23</td>
<td>25</td>
<td>27</td>
<td>29</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>35</td>
<td>37</td>
<td>39</td>
<td>41</td>
<td>43</td>
<td>45</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>51</td>
<td>53</td>
<td>55</td>
<td>57</td>
<td>59</td>
<td>61</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>22</td>
<td>23</td>
<td>26</td>
<td>27</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>38</td>
<td>39</td>
<td>42</td>
<td>43</td>
<td>46</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>51</td>
<td>54</td>
<td>55</td>
<td>58</td>
<td>59</td>
<td>62</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To pull off the trick, we follow these steps:

1. Give the cards to your target and ask them to pick a number $x$ that appears on at least one card, but to keep it secret.

2. Now show them the cards one-by-one: each time, ask them whether $x$ appears the card or not. If they tell you $x$ does appear on a card then place it in a pile, otherwise discard it.

3. To “magically” guess the number chosen, just add up each top, left-hand number on the cards in your pile.
An aside: a magic trick based on binary numbers (part 2).

Why does this work? Basically, if we write a number in binary then we are expressing it as the sum of some terms that are each a power-of-two. You can see this by looking at some examples:

\[
\begin{align*}
1 &= 1 &= 2^0 \\
2 &= 2 &= 2^1 \\
3 &= 1 + 2 &= 2^0 + 2^1 \\
4 &= 4 &= 2^2 \\
5 &= 1 + 4 &= 2^0 + 2^2 \\
6 &= 2 + 4 &= 2^1 + 2^2 \\
7 &= 1 + 2 + 4 &= 2^0 + 2^1 + 2^2 \\
&\vdots & &\vdots
\end{align*}
\]

Notice that the top, left-hand number \( t \) on each card is a power-of-two; all the other numbers on a given card are those where \( t \) appears as a term when we express it in binary. Look at the first card for example: each of the numbers 1, 3, 5, 7 and so on include the term \( t = 2^0 = 1 \) when we express it in binary. Or, on the second card each of the numbers 2, 3, 6, 7 and so on include the term \( t = 2^1 = 2 \).

So given a pile of cards on which \( x \) appears, we recover it more or less in reverse. Imagine the target selects \( x = 35 \) for example. Look at the cards: if we ask the target to identify cards on which 35 appears, we get a pile with those whose top, left-hand numbers are 1, 2 and 32 ... when we add them up we clearly recover

\[
2^0 + 2^1 + 2^5 = 1 + 2 + 32 = 35.
\]

To a target with no understanding of binary, this of course looks far more like magic than Mathematics!

for \( b = 10 \). So to cut a long story short, it is reasonable to interpret the sequence \( B \) as the decimal number 123 if we want to do so: all we need know is that since \( B \) represents a decimal sequence, we need to set \( b = 10 \).

The neat outcome is that there are many other ways of representing 123. For example, suppose we use a different value for \( b \), say \( b = 2 \). Using \( b = 2 \) equates to working with base-2 or binary numbers; all this means is our weights and digit set from above change. We could now express the number 123 as the binary sequence

\[
C = \{1, 1, 0, 1, 1, 1, 1, 0\}.
\]

For \( b = 2 \), the set of valid binary digits is \([0, 1]\). The value of \( C \) is therefore given by

\[
\sum_{i=0}^{\left|C\right|-1} C_i \cdot 2^i
\]

as before, which since \( \left|C\right| = 8 \) means we add up the terms

\[
\begin{align*}
C_0 \cdot 2^0 &= 1 \cdot 2^0 &= 1 \cdot 1 &= 1 \\
C_1 \cdot 2^1 &= 1 \cdot 2^1 &= 1 \cdot 2 &= 2 \\
C_2 \cdot 2^2 &= 0 \cdot 2^2 &= 0 \cdot 4 &= 0 \\
C_3 \cdot 2^3 &= 1 \cdot 2^3 &= 1 \cdot 8 &= 8 \\
C_4 \cdot 2^4 &= 1 \cdot 2^4 &= 1 \cdot 16 &= 16 \\
C_5 \cdot 2^5 &= 1 \cdot 2^5 &= 1 \cdot 32 &= 32 \\
C_6 \cdot 2^6 &= 1 \cdot 2^6 &= 1 \cdot 64 &= 64 \\
C_7 \cdot 2^7 &= 0 \cdot 2^7 &= 0 \cdot 128 &= 0
\end{align*}
\]

to obtain the number 123 as before.

Now we can move away from the specific example of 123, and try to think about a general number \( x \). For a given base \( b \), we have the digit set \([0, 1, \ldots, b-1]\). Remember that for \( b = 10 \) and \( b = 2 \) this meant the sets \([0, 1, \ldots, 9]\) and \([0, 1]\). A given number \( x \) is written as a sequence of digits taken from the appropriate digit set, i.e., each \( i \)-th digit \( x_i \in [0, 1, \ldots, b-1] \). We can express the value of \( x \) using \( n \) base-\( b \) digits and the summation

\[
x = \sum_{i=0}^{n-1} x_i \cdot b^i.
\]
The key thing to realise is that it does not matter so much how we write down a number, as long as we take some care the value is not changed when we interpret what it means.

### 2.1.2 Decimal and binary notation

Amazingly there are not many jokes about Computer Science, but here are two:

1. There are only 10 types of people in the world: those who understand binary, and those who do not.


Whether or not you laughed at them, both jokes relate to what we have been discussing: in the first case there is an ambiguity between the number ten written in decimal and binary, and in the second between the number twenty five written in octal and decimal.

Still confused? Look at the first joke: it is saying that the literal 10 can be interpreted as binary as well as decimal, i.e., as $1 \cdot 2^1 + 0 \cdot 1 = 2$ in binary and $1 \cdot 10 + 0 \cdot 1 = 10$. So the two types of people are those who understand that 2 can be represented by 10, and those that do not. Now look at the second joke: this is a play on words in that “Oct” can mean “October” but also “octal” or base-8. Likewise “Dec” can mean “December” but also “decimal”. With this in mind, we see that

$$3 \cdot 8 + 1 \cdot 1 = 25 = 2 \cdot 10 + 5 \cdot 1.$$

i.e., 31 oct equals 25 Dec in the sense that 31 in base-8 equals 25 in base-10.

Put in context, we have already shown that the decimal sequence $B$ and the decimal number 123 are basically the same if we interpret $B$ in the right way. But there is a problem of ambiguity: if we follow the same reasoning, we would also say that the binary sequence $C$ and the number 01111011 are the same. But how do we know what base 01111011 is written down in? It could mean the decimal number 123 (i.e., one hundred and twenty three) if we interpret it using $b = 2$, or the decimal number 01111011 (i.e., one million, one hundred and eleven thousand and eleven) if we interpret it using $b = 10$!

To clear up this ambiguity where necessary, we write literal numbers with the base appended to them. For example $123_{(10)}$ is the number 123 written in base-10 whereas $01111011_{(2)}$ is the number 01111011 in base-2. We can now be clear, for example, that $123_{(10)} = 01111011_{(2)}$. If we write a sequence, we can do the same thing: $(3, 2, 1)_{(10)}$ makes it clear we are still basically talking about the number 123. So our two “jokes” in this notation become $10_{(2)} = 2_{(10)}$ and $31_{(8)} = 25_{(10)}$.

### 2.1.3 Grouping bits into bytes

Traditionally, we call a binary digit (whose value is 0 or 1) a bit. Returning to the CD content described as the sequence of bits called $A$, what we really had was a sequence which we could interpret as a single (potentially very large) number written in binary. Imagine we write a similar sequence

$$D = \langle 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0 \rangle.$$

One could take $D$ and write it to the CD surface directly, or instead we could write it in groups of bits. The second approach would be sort of like constructing and writing a new sequence, for example splitting the bits of $D$ into groups of four,

$$E = \langle \langle 1, 1, 0, 0 \rangle, \langle 0, 0, 0, 0 \rangle, \langle 1, 0, 0, 0 \rangle, \langle 1, 0, 1, 0 \rangle \rangle$$

or eight,

$$F = \langle \langle 1, 1, 0, 0, 0, 0, 0, 0 \rangle, \langle 1, 0, 0, 0, 1, 0, 1, 0 \rangle \rangle.$$

So $E$ has four elements (each of which is a sub-sequence of four elements from the original sequence), while $F$ has two elements (each of which is a sub-sequence of eight elements from the original sequence). We call a group of four bits a nybble and a group of eight bits a byte: $E$ is a sequence of nybbles and $F$ is a sequence of bytes. The thing is, if we write the content of $E$ or $F$ to the CD surface we get the same bits (and hence the same pits and lands) as if we write $D$: it just depends on how we group them together.

Armed with the knowledge we now have about representing numbers, we can also use a short-hand to write each group as a decimal number. For example

$$G = \langle 3_{(10)}, 81_{(10)} \rangle.$$
can be reasonably interpreted as the same sequence as \( F \), and hence \( D \), because

\[
\langle 1, 1, 0, 0, 0, 0, 0, 0 \rangle_{(2)} \equiv 1 \cdot 2 + 1 \cdot 1 = 3_{(10)}
\]

\[
\langle 1, 0, 0, 0, 1, 0, 1, 0 \rangle_{(2)} \equiv 1 \cdot 64 + 1 \cdot 16 + 1 \cdot 1 = 81_{(10)}
\]

As an exercise, look at the four groups of four bits in \( E \): see if you can work out the equivalent of \( G \) for this sequence, i.e., what would the same decimal short-hand look like?

The upshot of this is that we can describe the CD content in a variety of different ways on paper, even though when we talk about actually writing them onto the CD surface everything \textit{must} be a sequence of bits. All we need to be careful about is that we have a consistent procedure to convert between different ways of describing the CD content.

Implement (task #3)

Actually working through examples is really the only way to get to grips with this topic. Convince yourself you understand things so far by

- converting the decimal literal \( 31_{(10)} \) into binary, then
- converting the binary literal \( 01101111_{(2)} \) into decimal.

Research (task #4)

Another representation of integers used by Computer Scientists, often as a short-hand for binary, is base-16 or \textbf{hexadecimal}. Do some research on this representation, and try to explain it within the general framework used for decimal and binary above. Also, explain

1. how we can cope with the fact that we only have ten digits (i.e., 0 to 9) but hexadecimal needs 16, and
2. how and why it acts as the short-hand described.

## 2.2 Data compression

Even though it is intended as an amusing example, there \textit{are} serious points we can draw from our previous discussion of text messaging:

- The topic of data compression has a golden rule which, roughly speaking, says to replace long things with shorter things. We can clearly see this going on, for example we have used the short symbol “@” to represent the longer sequence of characters “at”.

- \textit{How} were we able to compress and decompress the text at all? Partly the answer is that we know what English words and sentences mean; we can \textbf{adapt} our compression scheme because of this knowledge. However, in many situations we are just given a long binary sequence with no knowledge of what it means: it could represent an image, or some text for instance. In such a case, we have no extra insight with which to adapt our compression method; we only have the raw bits to look at.

- The fact that we \textit{can} find a good scheme to compress the text is, in part, because the English language is quite \textbf{redundant}; this is a fancy way to say some things occur more often than others. So for example, if you look at this Chapter then you would see that the sequence of characters “compression” appears quite often indeed: if we apply the golden rule of data compression and replace “compression” with some short symbol, e.g., ‘@’, then we will make the document quite a bit shorter.

- The final thing to note is the difference between \textbf{lossless} and \textbf{lossy} compression. If we compress something with a lossless compression scheme and then decompress it, we always get back what we started with. However, with lossy compression this is not true: such a scheme throws away data thought not to matter in relation to the meaning. For example, if we had a sequence of ten spaces, throwing away nine of them still means the text is readable even though it is not the same.

You may have come across lossy and lossless compression when dealing with digital photographs. When storing or editing the resulting image files, one usually stores them in the \textbf{jpeg} or \textbf{jpg} format; this is a standard produced by the \textbf{Joint Photographic Experts Group (JPEG)} [11]. Such a file usually compresses the image, but it does so in a lossy manner: some information is thrown away. The advantage of this approach is that the file can be smaller: much of the information in the image is so detailed our eyes cannot see it, so any disadvantage is typically marginal.
There are plenty of good software tools for manipulating images, and many of them are free; a good example is The GNU Image Manipulation Program (GIMP) 

www.gimp.org

Using such a tool, load a photograph and then save various versions of it: if you use JPEG as the format, you should be able to alter the compression ratio used, and hence the quality. What is the smallest sized version (on disk) you can make? At what point does the image quality start to degrade past what you find acceptable? Does the answer to these questions change if you use a different photograph?

Armed with this knowledge we can be a bit more specific about how to treat the CD content: we want a non-adaptive, lossless compression scheme. Put more simply, we want to take some binary sequence $X$ and, without knowing anything about what it means, compress it into a shorter sequence $\bar{X}$ so that later we can recover the exact original $X$ if we want to. This will basically mean we can write more data onto the surface of our CD (i.e., longer films, larger files, more music or whatever), though we need to work harder to access it (i.e., decompress it first).

### 2.2.1 A run-length based approach

Imagine we wanted to write a decimal sequence

$$X = (255, 255, 255, 255, 255, 255, 255, 255)$$

onto a CD. The number 255 is repeated eight times in a row: we call this a run [16], each run has a subject (i.e., the thing that is repeated, in this case 255) and a length (i.e., how many times the subject is repeated, in this case 8). Of course, you might argue that this is a contrived example: how likely is it that 255 will be repeated again and again in real CD content? Actually, this happens more often than you would think. Going back to the example, imagine we wrote a digital photograph onto the CD where numbers in the sequence $X$ basically represent the colours in the image; if the image has a large block of a single colour then the colour value will repeat many times. Another example is English text; although it is uncommon to have a run of more than two identical characters in a word (for example “moon” is possible with a run of two ‘o’ characters), sometimes there is a long run of spaces to separate words or paragraphs.

For the sake of argument, imagine there is at least a fair chance of a run occurring from time to time: how can we compress a run when we find one? Think about a simple question: which is shorter, $X$ or a description of $X$ written as

“repeat 255 eight times”.

Probably it is hard to say since we know how to write numbers onto the CD, but not the description of $X$. So we need to invent a scheme that converts the description into numbers we can write to the CD; imagine we choose to represent a run description as three numbers where:

1. the first number (the escape code) tells us we have found a run description rather than a normal number and therefore need to take some special action (here we use the number 0 as the escape code),
2. the second number tells us the length of the run, and
3. the third number tells us the subject of the run.

Using this scheme, we can compress our sequence $X$ into the new sequence

$$\bar{X} = (0, 8, 255)$$

where 0 tells us this is a run, 8 tells us the run length is eight and 255 tells us the number to repeat eight times is 255. Compressing $X$ into $\bar{X}$ is a matter of scanning the original sequence, identifying runs and converting them into the corresponding description. To decompress $\bar{X}$ and recover $X$ we simply process elements of the compressed sequence one at a time: when we hit a 0 we know that we need to do something special (i.e., expand a run description specified by the next two numbers), otherwise we just have a normal number.

However, there are two problems. First, since our scheme for describing runs has a length of three, it does not really make sense to use it for runs of length less than three. Of course, runs of length one or
zero do not really make sense anyway, but we should not compress runs of length two because we would potentially be making the compressed sequence longer! To see this, consider the sequence

\[ Y = (255, 255). \]

Our compression scheme would turn this into

\[ \bar{Y} = (0, 2, 255), \]

which is longer than the original sequence! In short, we are relying on the original sequence containing long runs, the longer the better: if it does not, then using our scheme does not make sense.

Second, and more importantly, what happens if the original sequence contains a 0? For example, imagine we want to compress the sequence

\[ Z = (255, 255, 255, 255, 255, 255, 255, 0, 1, 2) \]

with our scheme; we would end up with

\[ \bar{Z} = (0, 8, 255, 0, 1, 2). \]

When we read \( \bar{Z} \) from the CD and try to decompress it, we would start off fine: we would read the 0, notice that we had found a run description with length 8 and subject 255, and expand it into eight copies of 255. Then there is a problem because we would read the next 0 and assume we would found another run with length 1 and subject 2 which is not what we meant at all. We would end up recovering the sequence

\[ Z' = (255, 255, 255, 255, 255, 255, 0, 1, 2), \]

which is not what was originally compressed.

To fix things, we need to be a bit more clever about the escape code. One approach is to compress a “real 0” into a run of length one and subject 0. With this alteration we would compress \( Z \) to obtain

\[ Z = (0, 8, 255, 0, 1, 0, 1, 2) \]

which will then decompress correctly. However, this is a bit wasteful due to two facts: first we know it does not make sense to have a run of length zero, and second if the run length \( \bar{Z} \) was zero there would be no point having a subject since repeating anything zero times gives the same result. So we could reserve a run length of zero to mean we want a real 0. Using this approach, we would compress \( Z \) to get

\[ \bar{Z} = (0, 8, 255, 0, 0, 1, 2). \]

We can still decompress this correctly because when we read the second 0 and (falsely) notice we have found a run description, we know we were mistaken because the next number we read (i.e., the supposed run length) is also 0: there is no need to read a run subject because we already know we meant to have a real 0. In other words, the sequence \( (0, 0) \) is an encoding of the actual element 0.

### 2.2.2 A dictionary-based approach

A run-length approach is fine if the original sequence has long runs in it, but what else can we do? One idea would be to take inspiration from our original example of text messaging. Abusing our notation for CD content for a moment, roughly what we want to do is compress the sequence

\[ X = ("are", "you", "at", "home", "because", "I", "want", "to", "come", "over"). \]

The way we did this originally (although we did not really explain how at the time) was to construct and use a dictionary that converts long words into short symbols [4]. For example, if we had the dictionary

\[ D = ("are", "at", "to") \]

then we could compress \( X \) into

\[ \bar{X} = (D_0, "you", D_1, "home", "because", D_2, "want", D_2, "come", "over"). \]

Essentially we have replaced words in \( X \) with references to entries in the dictionary. For example \( D_0 \) is a reference to the 0-th entry in the dictionary \( D \) so each time we see \( D_0 \) in \( \bar{X} \), we can expand it out into “are”.

If \( \bar{X} \) is shorter than \( X \), we could claim we have compressed the original sequence; if we choose longer words to include in the dictionary or words that occur often, we improve how much we compress by. The sanity of this approach is easy to see if we continue to ignore sequences of numbers and consider some real text from Project Gutenberg:
Among a huge number of potential examples, consider the text of The Merchant of Venice by Shakespeare. To analyse the frequency of words, we will combine some standard commands in a BASH terminal. First we fetch the text and save it as the file A.txt:

```
bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$
```

The `wget` command downloads the file containing The Merchant of Venice text from the URL http://www.gutenberg.org/dirs/etext97/1ws1810.txt using three options, namely

1. `-q` tells `wget` not print out any progress information,
2. `-U chrome` tells `wget` to masquerade as the Chrome web-browser so the download is not blocked, and
3. `-O A.txt` tells `wget` to save the output into a file called A.txt,

plus the URL http://www.gutenberg.org/dirs/etext97/1ws1810.txt

Once we have the text, we translate all characters to lower-case so our task is a little easier (i.e., we do not need to consider the upper-case characters as distinct), and save the result in B.txt. This is achieved using the following command pipeline

```
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$
```

where the output of `cat` (the contents of A.txt) is fed as input to `tr` which performs the translation for us; the output is then redirected into B.txt. In this case, the rule `[:upper:] [:lower:]` used by `tr` can be read as “take upper-case letters, translate them into lower-case equivalents”.

Now we need a way to count the number of occurrences of words. To do this we first use `tr` to convert all space characters into EOL characters and delete all punctuation characters; this basically takes the original file and converts it into a version where there is one word per-line. Finally we remove all punctuation and blank lines using `tr` and `grep`, and save the result as C.txt. In summary we execute the command pipeline:

```
bash$ cat B.txt | tr [:space:] '' | tr -d [:punct:] | grep -v ˆ$ > C.txt
bash$
```

To get the actual count, we first sort C.txt then use `uniq` to count the unique occurrences of each word; we sort the result and use `head` to give us the top 31 most used words:

```
bash$ cat C.txt | sort | uniq -c | sort -n -r | head -n 31 | paste -s
bash$
```

This might all seem a bit like magic if you are not used to BASH or the commands themselves. However, the result we get at the end should be more obviously close to what you would expect. For example the words used most are things like “the” and “and”. Working down the list we start to find some good candidates for the dictionary. For example “bassanio” is quite long and also used fairly often, so replacing this with a dictionary reference would be quite effective.

Of course, a largely similar approach is possible when we return to consider sequences of numbers we want to write onto the CD. Imagine we wanted to write a decimal sequence

\[ Y = \langle 1, 2, 3, 4, 5, 5, 5, 1, 2, 3, 4 \rangle \]

onto a CD. First we construct a dictionary, say

\[ D = \langle \langle 1, 2, 3, 4 \rangle, \langle 5, 5, 5, 5 \rangle \rangle, \]

and then compress Y to get

\[ \bar{Y} = \langle D_0, D_1, D_0 \rangle. \]
Already we can identify two problems. First, the text messages example was a special case since the dictionary was actually in the head of the person reading the SMS message: we did not need to include it with the message! Now things are different. To decompress \( Y \) and hence recover \( Y \), we need to write \( D \) to the CD as well because this is the only way to be clear what \( D_0 \) means. There is a trade-off as a result: the more elements we add to the dictionary, the more chance we have to compress parts of the sequence but also the larger the dictionary becomes. Since we need to write the dictionary onto the CD as well as the compressed sequence, identifying a small set of good candidates for dictionary entries is therefore important, otherwise including the dictionary will cancel out any advantage (in terms of storage space) we get from the compression process.

Second, \( D_0 \) is not a number so we cannot currently write it to the CD at all. To solve this, we can use a similar approach as when we looked at run-length encoding; imagine we choose to represent a reference to a dictionary entry as two numbers:

1. the first number (the escape code) tells us we have found a dictionary reference rather than a normal number and therefore need to take some special action (here we use the number 1 as the escape code),
2. the second number tells us the dictionary entry we are referring to.

Returning to the sequence \( Y \), given the dictionary \( D \) we could compress \( Y \) to give

\[
\hat{Y} = (1, 0, 1, 1, 1, 0)
\]

where, looking at the first two elements in \( \hat{Y} \), 1 tells us this is a dictionary reference, and 0 tells us the reference is to entry 0 (i.e., \( (1, 2, 3, 4) \)) for example.

Compressing \( Y \) into \( \hat{Y} \) is a matter of scanning the original sequence, identifying good candidates for the dictionary and converting them into references. Of course this is a bit harder than when we looked at text: given we are not working with words which we can identify based on spaces around them, how could we decide \( (1, 2, 3, 4) \) is a good candidate? This is beyond the scope of our description; essentially this is the clever part in any dictionary based approach. We would need to perform a scan of the original sequence to capture statistics about the content, and construct the dictionary before a second scan performed the actual compression.

To decompress \( \hat{Y} \) and recover \( Y \), we simply process elements of the compressed sequence one at a time: when we hit a 1 we know that we need to do something special (i.e., replace a reference with an element from the dictionary), otherwise we just have a normal number. Again, we would need to do the same thing as in the run-length encoding case to cope with the fact that the escape code 1 might occur in the original sequence; we need to be able to specify that we meant a “real 1” somehow.

### 2.3 Error correction

In the previous Section we looked at two simple schemes for data compression. But recall this was only one reason that CDs became popular; the other reason was that CDs are less prone to damage than previous media such as vinyl or cassette tape. Now we need to turn our attention to the detection and correction of errors. To illustrate the problem, imagine we start off with the short binary sequence

\[
X = (0, 1, 1, 1)
\]

which represents (potentially compressed) binary data we want to write onto a CD. Two types of error might occur:

1. There might be permanent errors which occur every time we try to read \( X \). For example, if we scratch the CD surface then every time we try to read \( X \) we get some other sequence because the pits and lands have been destroyed.
2. There might be transient errors that only occur occasionally when we try to read \( X \). For example, if the CD surface has dust on it then we might read \( X \) incorrectly sometimes and correctly other times when the dust is displaced.

To cope with this, we would like to construct a scheme with two properties: first we would like to know when an error has occurred, and second we would like to be able to correct an error once we know it has occurred. To do this we will add some redundant elements to \( X \) in order to produce a new sequence \( \hat{X} \) which is what we actually write onto the CD.
2.3.1 An error detection approach

To present an error detection scheme we need to introduce two functions:

1. The Hamming-Weight function [8] counts the number of elements in a binary sequence that are equal to one. So given that

   \[ X = \langle 0, 1, 1, 1 \rangle, \]

   for example, this means Hamming-Weight(X) = 3. We can compute this result by just adding all the elements together, i.e.,

   \[ \text{Hamming-Weight}(X) = \sum_{i=0}^{|X|-1} X_i \]

   so that

   \[ \text{Hamming-Weight}(\langle 0, 1, 1, 1 \rangle) = X_0 + X_1 + X_2 + X_3 = 0 + 1 + 1 + 1 = 3. \]

2. The Parity function tells us whether the Hamming-Weight function gives an odd or even result. The idea relies on the XOR (short for “exclusive-or”) function [6] that is written using the symbol \( \oplus \) and gives us a single output from two inputs:

   \[
   x \oplus y = \begin{cases} 
   0 & \text{if } x = 0, \ y = 0 \\
   1 & \text{if } x = 1, \ y = 0 \\
   1 & \text{if } x = 0, \ y = 1 \\
   0 & \text{if } x = 1, \ y = 1
   \end{cases}
   \]

   We compute the Parity function by simply XOR’ing all the elements together, i.e.,

   \[ \text{Parity}(X) = \bigoplus_{i=0}^{|X|-1} X_i \]

   so that

   \[ \text{Parity}(\langle 0, 1, 1, 1 \rangle) = X_0 \oplus X_1 \oplus X_2 \oplus X_3 = 0 \oplus 1 \oplus 1 \oplus 1 = 1. \]

   In other words, we take the Hamming-Weight function and replace each + with \( \oplus \). As a consequence, you could also think of the result as being given by

   \[ \text{Parity}(X) = \text{Hamming-Weight}(X) \mod 2. \]

Given a sequence \( X \), an even parity code [15] adds an extra element in order to allow detection of errors. Starting with \( X \), we compute the extra element

\[ P_0 = \text{Parity}(X) = X_0 \oplus X_1 \oplus X_2 \oplus X_3 \]

and then concatenate it to the end of \( X \) to give

\[ \hat{X} = X \parallel P_0. \]

A more concrete example probably makes this easier to understand. If we start with

\[ X = \langle 0, 1, 1, 1 \rangle \]

then since \( P_0 = \text{Parity}(X) = 1 \), the result is a new sequence

\[ \hat{X} = X \parallel (1) \]

\[ = \langle 0, 1, 1, 1 \rangle \parallel (1) \]

\[ = \langle 0, 1, 1, 1, 1 \rangle \]

which we can write onto the CD. Notice that because of our choices, we will always have \( \text{Parity}(\hat{X}) = 0 \).

Now imagine that sometime after writing \( \hat{X} \) to the CD, we try to read the sequence back again. Unfortunately there is an error: instead of getting what we expected, i.e., \( \hat{X} \), we get some other sequence \( \hat{X}' \) where the error has flipped one element from either 0 to 1 or from 1 to 0. For example, we might get

\[ \hat{X}' = \langle 0, 1, 0, 1, 1 \rangle \]

\[ = \langle 0, 1, 0, 1 \rangle \parallel (1) \]

\[ = X' \parallel (1) \]
How can we detect that the error occurred? Because of the way we added the extra element to \( X \), \( \text{Parity}(\hat{X}') \) should be zero but because of the error it is one: this mismatch shows that an error occurred. Put more simply, if we recompute

\[
P'_0 = X'_0 \oplus X'_1 \oplus X'_2 \oplus X'_3
\]

then the fact we get 0 as a result signals that an error occurred: this does not match the original \( P_0 \) we added. Notice that

- if we read \( \hat{X} \), i.e., there was no error, we can simply strip off the extra element and get back the \( X \) we wanted to read, but
- if we read \( \hat{X}' \), i.e., there was an error, we cannot tell where the error occurred or how to correct it, so we have to try to reread the CD and hope the error is transient and does not occur again.

This sounds great, but there is a problem: if more than one error occurs then we can be fooled into thinking that there was no error at all. For example, imagine that two elements are flipped and we read

\[
\hat{X}' = \langle 0,0,0,1,1 \rangle = \langle 0,0,0,1 \rangle \parallel \langle 1 \rangle = X' \parallel \langle 1 \rangle
\]

instead of \( \hat{X} \). To check if an error occurred, we compute

\[
P'_0 = X'_0 \oplus X'_1 \oplus X'_2 \oplus X'_3
\]

and are fooled because we get 1 as a result which matches what we were expecting; as a result we use \( \hat{X}' \) thinking it is correct. Clearly detection of more than one error is important in reality, and more complex schemes that can achieve this goal are possible. But instead we turn our attention to error correction: once we have detected an error, how can we fix it rather than reread the CD and hope for the best?

Throughout the above, we used an even parity code. Provided all the working out is consistent, however, there is no reason to prefer this approach over the alternative: an odd parity code. Reproduce the example above using an odd parity code.

### 2.3.2 An error correction approach

The Hamming code [7], named after inventor Richard Hamming, improves on simple error detection mechanisms by allowing the correction of errors once they are detected. This requires more than one element to be added to the sequence we start with. Hamming used the term \((n,m)\)-code to mean a scheme where there were \( n \) elements in the end result, of which \( m \) are the original elements and \( n - m \) are added afterwards. Phrased like this, you could view the original even parity code from above as an \((n,n-1)\)-code because there were \( n \) elements to start with, and we added \( n - (n-1) = 1 \) element.

We will concentrate on the specific example of a \((7,4)\)-code introduced in 1950: this starts with a sequence of four elements and adds three more to realise the error correction scheme. Starting with \( X \), we compute the three extra elements

\[
\begin{align*}
P_0 &= X_0 \oplus X_1 \oplus X_3 \\
P_1 &= X_0 \oplus X_2 \oplus X_3 \\
P_2 &= X_1 \oplus X_2 \oplus X_3
\end{align*}
\]

and then concatenate them onto the end of \( X \) to give

\[
\hat{X} = X \parallel \langle P_0, P_1, P_2 \rangle.
\]

A reasonable question to ask is why we would choose to compute \( P_0, P_1 \) and \( P_2 \) like this rather than in some other way? The choice is explained neatly by placing them into the Venn diagram [19] shown in Figure 2.3. The idea is that the three sets in the Venn diagram represent the three extra elements we have computed.
Each set “covers” a different combination of the elements from $X$: the upper (red) set represents $P_0$ and covers $X_0$, $X_1$ and $X_3$; the lower-left (green) set represents $P_1$ and covers $X_0$, $X_2$ and $X_3$; the lower-right (blue) set represents $P_2$ and covers $X_1$, $X_2$ and $X_3$. Notice, for example, that the element $X_0$ is only included in the equations for $P_0$ and $P_1$, so it is placed in the intersection of the $P_0$ and $P_1$ sets and hence covered only by those sets. As a result, if there is an error and $X_0$ is flipped then we know that $P_0$ and $P_1$ will be affected but $P_2$ will not.

Again, a more concrete example probably makes this easier to understand. If we start with

$$X = \langle 0, 1, 1, 1 \rangle$$

then since

$$P_0 = X_0 \oplus X_1 \oplus X_3$$
$$= 0 \oplus 1 \oplus 1$$
$$= 0$$
$$P_1 = X_0 \oplus X_2 \oplus X_3$$
$$= 0 \oplus 1 \oplus 1$$
$$= 0$$
$$P_2 = X_1 \oplus X_2 \oplus X_3$$
$$= 1 \oplus 1 \oplus 1$$
$$= 1$$

the result is a new sequence

$$\hat{X} = X \parallel \langle 0, 0, 1 \rangle$$
$$= \langle 0, 1, 1, 1 \rangle \parallel \langle 0, 0, 1 \rangle$$
$$= \langle 0, 1, 1, 1, 0, 0, 1 \rangle$$

which we can write onto the CD.

Now imagine that when we try to read $\hat{X}$ from the CD, there is again some error: instead of getting what we expected, i.e., $\hat{X}$, we get some other sequence $\hat{X}'$ where one element has been flipped. For example, we might get

$$\hat{X}' = \langle 1, 1, 1, 1, 0, 0, 1 \rangle$$
$$= \langle 1, 1, 1, 1 \rangle \parallel \langle 0, 0, 1 \rangle$$

If we recompute

$$P_0' = X'_0 \oplus X'_1 \oplus X'_3$$
$$= 1 \oplus 1 \oplus 1$$
$$= 1$$
$$P_1' = X'_0 \oplus X'_2 \oplus X'_3$$
$$= 1 \oplus 1 \oplus 1$$
$$= 1$$
$$P_2' = X'_1 \oplus X'_2 \oplus X'_3$$
$$= 1 \oplus 1 \oplus 1$$
$$= 1$$
and match these against the original $P_0$, $P_1$ and $P_2$ we added to $X$, this shows there is an error: $P'_0$ and $P'_1$ do not match up with $P_0$ and $P_1$, but $P'_2$ does match $P_2$. Put another way, since we assume only one error has occurred and we know that the values of $P'_0$ and $P'_1$ are wrong, this means that the error must occur in the upper and lower-left sets of our Venn diagram; in the same way, because $P'_3$ is correct, this also means that the error did not occur in the lower-right set. There is only one region of the Venn diagram which is covered by the upper and lower-left sets yet not by the lower-right set: this corresponds to $X_0$, so it must be $X'_0$ which is wrong.

Does this work for all possible errors? Given we originally added three extra elements, i.e., $P_0$, $P_1$ and $P_2$, we can deal with at most eight possibilities; these are described by the left-hand column in Table 2.4. Note that one possibility corresponds to no errors occurring, leaving seven others: intuitively this should seem the correct number since we have seven bits in which an error could occur. The issue now is to determine how we can translate the “differences” between $P'_0$, $P'_1$ and $P'_2$ and $P_0$, $P_1$ and $P_2$ into the position of an error as we did for the example above; the right-hand column in Table 2.4 describes the correct translation. In our example, the difference was $(1,1,0)$, i.e., $P'_0$ and $P'_1$ were wrong but $P_2$ was right: the corresponding entry in the table shows that the error was in $X'_0$.

### 2.4 Recasting error correction as matrix arithmetic

Another way of thinking about the $(7,4)$-code for error correction is that it simply reuses the even parity code that we used for error detection. In this context, we could say that

$$(X_0,X_1,X_3,P_0)$$

is a **codeword** for the even parity code on the data $(X_0,X_1,X_3)$, and likewise both

$$(X_0,X_2,X_3,P_1)$$

and

$$(X_1,X_2,X_3,P_2)$$

are code words on the data $(X_0,X_2,X_3)$ and $(X_1,X_2,X_3)$. The idea is to recast this in terms of **matrices**. If you have covered matrices before then the following should link the theory you already know to a real-world application; if not, we try to introduce the theory as we go, but you can skip to the next Chapter if you prefer.

Basically we would like to generalise and extend the $(7,4)$-code so far explained in a fairly informal way; perhaps to send more data per-codeword, or to correct more than one error. Doing this with a Venn diagram seems unattractive for two reasons

1. for a human, more complicated Venn diagrams (e.g., in more than two dimensions) are hard to draw and understand, and

2. on a computer, the concept of using Venn diagrams does not easily map onto the types of operation that are available.

To combat both problems, we recast and formalise the basic idea, then let Mathematics take care of achieving the generalisation [1].
2.4.1 An overview of vectors

A row vector is simply a sequence of elements, e.g.,

$$\vec{R} = (r_0, r_1, \ldots, r_{m-1})$$

where the number of elements is $m$; we call $m$ the dimension of $\vec{R}$. A column vector is similar, except the elements are written down in a column rather than a row, e.g.,

$$\vec{C} = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix}.$$ 

In this case the number of elements in $\vec{C}$ is $n$, so the dimension of $\vec{C}$ is $n$. Note that we use lower-case letters for the vector elements, so, for example, $r_i$ is the $i$-th element of $\vec{R}$; the reason for this is that we want to use $\vec{R}_j$ to denote the $j$-th separate vector in some set.

If we take two row vectors (or two column vectors) which have the same dimension, i.e., $n = m$, we can combine them together. For example, we can compute a vector addition, where the idea is to add together corresponding elements of the vectors:

$$(1, 2, 3) + (4, 5, 6) = (1 + 4, 2 + 5, 3 + 6) = (5, 7, 9).$$

Note we cannot add a row vector to a column vector: the operation is only valid when the two vectors have the same type and dimension. However, if we take a row vector and a column vector which have the same dimension, we can compute a vector multiplication (or dot product): the idea is to multiply together corresponding elements of the vectors, and add up all the results. For example

$$(1, 2, 3) \cdot \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32.$$ 

We can write a more general method

$$\vec{R} \cdot \vec{C} = (r_0, r_1, \ldots, r_{n-1}) \cdot \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix} = r_0 \cdot c_0 + r_1 \cdot c_1 + \cdots + r_{n-1} \cdot c_{n-1} = \sum_{i=0}^{n-1} r_i \cdot c_i,$$

which captures the same idea more formally. In words, the right-hand side means that for each index $i$ between 0 and $n - 1$ (i.e., $i = 0, 1$ and so on up to $i = n - 1$) we add up terms that look like $R_i \cdot C_i$ (i.e., the product of the $i$-th elements of $\vec{R}$ and $\vec{C}$). Note we cannot multiply a row vector and a row vector, or a column vector and a column vector: the operation is only valid when the two vectors have different types but the same dimension. Further, we always write the column vector on the right, i.e.,

$$\vec{R} \cdot \vec{C}$$

is valid, but

$$\vec{C} \cdot \vec{R}$$

is not. These restrictions can be annoying because sometimes we might want to put the column vector on the left, or multiply a row vector by another row vector. The solution is to use a vector transpose operation to translate a row vector into a column vector or vice versa. For example

$$(1, 2, 3)^T = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}^T = (1, 2, 3)$$

where the “T” superscript means “apply transpose to this vector”. 
2.4.2 An overview of matrices

Suppose we have \( n \) separate row vectors, each of dimension \( m \); that is, suppose we have \( \vec{R}_1, \vec{R}_2, \ldots, \vec{R}_n \), where each \( \vec{R}_i \) has dimension \( m \). If we write the row vectors above each other, we get a “table” or matrix with \( n \) rows and \( m \) columns, i.e., of dimension \( n \times m \). For example, if we take the \( n = 3 \) row vectors
\[
\vec{R}_1 = (1,2,3,4) \quad \vec{R}_2 = (5,6,7,8) \quad \vec{R}_3 = (9,0,1,2),
\]
each of which has dimension \( m = 4 \), and write them above each other, we get a matrix
\[
M = \begin{pmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 0 & 1 & 2
\end{pmatrix}
\]
of dimension \( 3 \times 4 \). Alternatively, we could construct the same matrix by taking \( m \) column vectors, each of dimension \( n \), and writing them next to each other. For example, if we take \( m = 4 \) column vectors
\[
\vec{C}_1 = \begin{pmatrix} 1 \\ 5 \\ 9 \end{pmatrix} \quad \vec{C}_2 = \begin{pmatrix} 2 \\ 6 \\ 0 \end{pmatrix} \quad \vec{C}_3 = \begin{pmatrix} 3 \\ 7 \\ 1 \end{pmatrix} \quad \vec{C}_4 = \begin{pmatrix} 4 \\ 8 \\ 2 \end{pmatrix},
\]
each of which has dimension \( n = 3 \) and write them next to each other, we again get
\[
M = \begin{pmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 0 & 1 & 2
\end{pmatrix}
\]
This means a given matrix can be considered either as a collection of row vectors or column vectors. The vector transpose operation we described previously can also be used to perform a matrix transpose, i.e., to translate a matrix of dimension \( n \times m \) into one of dimension \( m \times n \). Basically we just apply the vector transpose to everything: given the matrix above, we see that
\[
M^T = \begin{pmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 0 & 1 & 2
\end{pmatrix}^T = \begin{pmatrix}
1 & 5 & 9 \\
2 & 6 & 0 \\
3 & 7 & 1 \\
4 & 8 & 2
\end{pmatrix}
\]
because
\[
M = \begin{pmatrix}
\vec{R}_1^T \\
\vec{R}_2^T \\
\vec{R}_3^T
\end{pmatrix}
\]
so
\[
M^T = \begin{pmatrix}
\vec{R}_1^T \\
\vec{R}_2^T \\
\vec{R}_3^T
\end{pmatrix}^T = \begin{pmatrix}
(1,2,3,4)^T \\
(5,6,7,8)^T \\
(9,0,1,2)^T
\end{pmatrix}^T = \begin{pmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 0 & 1 & 2
\end{pmatrix}.
\]
Finally, the vector multiplication operation is also useful in the context of matrices since it allows us to define matrix-vector multiplication. Suppose we have a matrix \( M \) of dimension \( n \times m \). We compute matrix-vector multiplication in one of two ways depending on the type of the vector:

1. A row vector \( \vec{R} \) with dimension \( n \) can be multiplied on the left of the matrix to obtain another row vector of dimension \( m \). To achieve this, we consider \( M \) as a set of \( m \) column vectors and compute
\[
\vec{R} \cdot M = \vec{R} \cdot (\vec{C}_0, \vec{C}_1, \ldots, \vec{C}_{m-1}) = (\vec{R} \cdot \vec{C}_0, \vec{R} \cdot \vec{C}_1, \ldots, \vec{R} \cdot \vec{C}_{m-1}).
\]
Notice that there are \( m \) elements in the result, each one computed via the dot product of \( \vec{R} \) and one of the \( \vec{C}_i \) vectors.

2. A column vector \( \vec{C} \) with dimension \( m \) can be multiplied on the right of the matrix, to obtain another column vector of dimension \( n \). To achieve this, we consider \( M \) as a set of \( n \) row vectors and compute
\[
M \cdot \vec{C} = (\vec{R}_1, \vec{R}_2, \ldots, \vec{R}_n) \cdot \vec{C} = (\vec{R}_1 \cdot \vec{C}, \vec{R}_2 \cdot \vec{C}, \ldots, \vec{R}_n \cdot \vec{C}).
\]
Notice that there are \( n \) elements in the result, each one computed via the dot product of one of the \( \vec{R}_i \) vectors and \( \vec{C} \).
2.4.3 Addition and multiplication modulo 2

Recall the XOR function from our discussion of error correction; it gives a single output from two inputs:

\[ x \oplus y = \begin{cases} 
0 & \text{if } x = 0, y = 0 \\
1 & \text{if } x = 1, y = 0 \\
1 & \text{if } x = 0, y = 1 \\
0 & \text{if } x = 1, y = 1 
\end{cases} \]

Although it was useful, you might be forgiven for thinking that XOR is a little ad hoc: where does this particular form come from? Another way of thinking about XOR is that it computes a special type of addition. What we are doing is called modular arithmetic [14], and we will encounter more formal definitions later in Chapter 7 and Chapter 10. For now, it is enough to think of XOR as addition where we treat all even numbers as 0 and all odd numbers as 1. In more detail, the four cases above can be described as

\[ 0 + 0 = 0 \equiv 0 \pmod{2} \]
\[ 1 + 0 = 1 \equiv 1 \pmod{2} \]
\[ 0 + 1 = 1 \equiv 1 \pmod{2} \]
\[ 1 + 1 = 2 \equiv 0 \pmod{2} \]

Likewise, the AND function

\[ x \wedge y = \begin{cases} 
0 & \text{if } x = 0, y = 0 \\
0 & \text{if } x = 1, y = 0 \\
0 & \text{if } x = 0, y = 1 \\
1 & \text{if } x = 1, y = 1 
\end{cases} \]

can be described as modular multiplication because

\[ 0 \cdot 0 = 0 \equiv 0 \pmod{2} \]
\[ 1 \cdot 0 = 0 \equiv 0 \pmod{2} \]
\[ 0 \cdot 1 = 0 \equiv 0 \pmod{2} \]
\[ 1 \cdot 1 = 1 \equiv 1 \pmod{2} \]

In both cases, when we write \( x \equiv y \pmod{2} \) this shows the number \( x \) on the left is equivalent to the number \( y \) on the right if considered modulo 2. The idea is that we have translated what looked like ad hoc functions into a more Mathematical setting: the goal is that whereas we previously did all our arithmetic “normally” on vectors and matrices of numbers, now we can do it all “modulo 2” and work with vectors and matrices of bits instead.

2.4.4 Using matrices for error correction

Our original goal was to generalise and extend the (7,4)-code using the concept of matrices as a formal underpinning. We now know enough to do just that. First we need to translate the encoding step: consider

\[ G = \begin{pmatrix} 
1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 
\end{pmatrix} \]

which we call the generating matrix since it will be used to generate codewords from data; notice it has dimension 4 \( \times \) 7. Now suppose we have the data

\[ X = (0, 1, 1, 1) \]

which matches our original (7,4)-code example. To compute the codeword \( \hat{X} \), we first write the elements of \( X \) as a row vector and then multiply it on the right by the generating matrix \( G \): all operations on elements are performed modulo 2. For example

\[ \hat{X} = \vec{X} \cdot G = (0, 1, 1, 1) \cdot \begin{pmatrix} 
1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 
\end{pmatrix} = (0, 1, 1, 1, 0, 0, 1). \]
As an exercise, to make sure you understand how to multiply a row vector by a matrix, verify that the calculation above is correct: work out each element in the result long-hand. To get you started, here are the first two elements:

\[
\hat{X}_0 = 0 \cdot 1 + 1 \cdot 0 + 1 \cdot 0 + 1 \cdot 0 \pmod{2} = 0 \pmod{2}
\]

\[
\hat{X}_1 = 0 \cdot 0 + 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 0 \pmod{2} = 1 \pmod{2}
\]

Next we need to translate the decoding step. This demands a closer look at the generating matrix \( G \), considering it as two parts:

1. Consider a matrix of dimension \( d \times d \) whose elements are all 0 except those on the main diagonal \([12]\) which are 1. This is called an identity matrix \([10]\) and denoted by \( I_d \); for example

\[
I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}
\]

and

\[
I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\]

The first, left-hand \( 4 \times 4 \) part of \( G \) is an identity matrix of this type, i.e., the left-hand \( 4 \times 4 \) sub-matrix is \( I_4 \).

2. The second, right-hand \( 4 \times 3 \) part of \( G \) is less structured; we can call this sub-matrix \( A \) for short, i.e.,

\[
A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}
\]

Using the two parts we can write \( G \) as \( I_4 \parallel A \), i.e., \( G \) is \( I_4 \) with \( A \) concatenated onto the right-hand side of it. We now form a new \( 3 \times 7 \) parity check matrix as

\[
H = A^T \parallel I_3,
\]

i.e., \( H \) is \( A^T \) with \( I_3 \) concatenated onto the right-hand side of it. In our case this means

\[
H = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}
\]

The parity check matrix allows us to perform error detection and correction on the codeword \( \hat{X} \): we take the codeword and interpret it as a column vector, before computing

\[
\vec{S} = H \cdot \hat{X}
\]

which we call the syndrome. As before, all operations on elements are performed modulo 2. If all the elements in \( \vec{S} \) are 0 then no errors have occurred; if \( S \) is non-zero however, then it tells us where an error occurred just as before. For example, suppose we receive the correct codeword

\[
\hat{X} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}
\]
We compute

\[
H \cdot \hat{X} = \begin{pmatrix}
1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1
\end{pmatrix} \cdot \begin{pmatrix}
0 \\
1 \\
1 \\
0 \\
0 \\
0 \\
1
\end{pmatrix} = \begin{pmatrix}
0 \\
0 \\
0
\end{pmatrix}
\]

and, since the result is zero, deduce that no error occurred. However, if receive the codeword

\[
\hat{X}' = \begin{pmatrix}
1 \\
1 \\
1 \\
0 \\
0 \\
0 \\
1
\end{pmatrix}
\]

that has an error in the first element we compute

\[
H \cdot \hat{X}' = \begin{pmatrix}
1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1
\end{pmatrix} \cdot \begin{pmatrix}
1 \\
1 \\
1 \\
1 \\
0 \\
0 \\
1
\end{pmatrix} = \begin{pmatrix}
1 \\
1 \\
0
\end{pmatrix}
\]

The result we get is non-zero: it is actually equal to the first column vector of the parity check matrix, allowing us to detect and correct the error.

As an aside, we can recast the parity code (i.e., the error detection scheme) using the same approach. Looking back, the generating and parity check matrices

\[
G = \begin{pmatrix}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1
\end{pmatrix}
\]

and

\[
H = (1, 1, 1, 1)
\]

reproduce the same properties. As an exercise, work through the original parity check code example and convince yourself that the syndrome this produces can detect errors but not correct them.

### 2.4.5 Generalising the matrix-based (7, 4)-code

We have recast the original (7, 4)-code in terms of matrices, but how can we generalise it? That is, how can we use the same theory to construct an arbitrary \((n, m)\)-code? In general, for an \((n, m)\)-code we have

\[
G = I_m \parallel A
\]

and

\[
H = A^T \parallel I_{n-m}
\]

where we know what \(I_m\) and \(I_{n-m}\) look like, but not the \(m \times (n-m)\) matrix \(A\). So the generalisation problem becomes one of trying to find a matrix \(A\) which gives us the properties we want, and then a way of interpreting what the syndrome is telling us. This is far from trivial, but at least it gives us somewhere to start: certainly this seems more achievable than imagining what multi-dimensional Venn diagrams would look like!

The final paragraph leaves an obvious question open: exactly what properties \(A\) need to have, and how do we create such a matrix for given values of \(n\) and \(m\)? Do some research into this issue, and see if you can produce a working (3, 1)-code or larger (15, 11)-code.
BIBLIOGRAPHY

Imagine you are a student at the University of Bristol; lectures for the day are finished, and you fancy a drink to celebrate. How do you get from the Computer Science Department to the Student Union? Easy! Just ask Google Maps:

http://maps.google.com/?saddr='bristol+bs8+1ub'&daddr='bristol+bs8+1ln'

What comes back is a fancy map plus a list of directions which resemble the following:

1. Start at Bristol, BS8 1UB, UK.
2. Head west on B4051 toward A4018; continue to follow B4051.
3. Continue on A4018.
4. Continue on B3129; go through two roundabouts; destination will be on the left.
5. Finish at Bristol, BS8 1LN, UK.

The directions have some features worth discussing in more detail. Each line in the directions represents a static description of some active step we should perform while following them. The lines have to be followed, or processed, in order: we start at the first step and once that is complete, move on to the next one. If one of the steps is missed out for example, or we start at the third step instead of the first, the directions do not work. The directions are (fairly) unambiguous. It may help to know that the road B4051 is more usually called Park Street, but line #4 is not “go through some roundabouts”; we should go through exactly two roundabouts. This means it is always clear how to process each line (i.e., exactly what steps to perform): we never become confused because there is not enough information and we do not know what to do.

3.1 Algorithms

As a result of the features described above, our directions at least loosely satisfy the definition of an algorithm [1]: they are simply an abstract description of how to solve a problem. Of course in this case the problem is helping someone get from one place to another, but we might just as easily write an algorithm to solve other problems as well.
3.1.1 What is an algorithm?

It is unusual to enforce any strict rules about how we should write an algorithm down: as long as it makes sense to whoever is reading it, we can use whatever format we want. This often means using pseudo-code, literally a “sort of” program [11]. An algorithm can have inputs, which we name, and can provide some outputs; we say the algorithm takes (or is given) some arguments as input, and returns (or produces) a result as output. Based on this, imagine we write the following algorithm:

```
1 algorithm FERMAT-TEST(n) begin
2     Choose a random number a between 2 and n − 1 inclusive
3     Compute d = gcd(a, n), the greatest common divisor of a and n
4     If d ≠ 1 then return false as the result
5     Compute t = a^(n-1) (mod n)
6     If t = 1 then return true as the result, otherwise return false
7 end
```

This looks more formal, but there is no magic going on: the algorithm is simply a list of five lines we can process. Note that we have numbered each of the lines so we can refer to them. In line #1 we name the algorithm FERMAT-TEST and show that it accepts a single argument called n as input; in lines #2 to #6 we list the directions themselves. The lines #4 and #6 are a bit special since they can produce output from the algorithm.

We invoke (or use) FERMAT-TEST by giving a concrete value to each of the inputs. For example, if we write

```
FERMAT-TEST(221)
```

basically what we mean is “follow the algorithm FERMAT-TEST, but each time you see n substitute 221 instead”. Like the travel directions, invoking FERMAT-TEST means we perform some active step (or steps) for each line in the algorithm:

**Step #1** Choose a random number between 2 and 220 inclusive, e.g., a = 11.

**Step #2** Compute d = gcd(11, 221) = 1.

**Step #3** Since d = 1, carry on rather than returning a result.

**Step #4** Compute t = 11^221−1 (mod 221) = 81.

**Step #5** Since t ≠ 1, return false as the result.

After step #5 the algorithm terminates: from the input 221, we have computed the result false. What this result means of course depends on the purpose of the algorithm; the purpose of FERMAT-TEST is certainly less clear than the travel directions we started off with!

FERMAT-TEST is actually quite a famous algorithm [7] due to the French mathematician Pierre de Fermat. The purpose is to tell if n, the input, is a prime [12] number or not. Lines #2 to #4 ensure we select an a that is co-prime [5] to n; this means a and n have no common factors. If we were to pick an a such that a divided n, then clearly n cannot be prime, so this possibility is ruled out before we carry on. Having computed t, if the result we get after lines #5 to #6 is false then n definitely is not a prime number; it is composite. On the other hand, if the result is true then n might be a prime number. The more times we invoke FERMAT-TEST with a given n and get true as a result, the more confident we are that n is a prime number. FERMAT-TEST works quite well, except for the so-called Carmichael numbers [3] which trick the algorithm: they are composite, but FERMAT-TEST always returns true. The smallest Carmichael number is 561 = 3 · 11 · 17: whatever a it chooses, FERMAT-TEST will always compute t = 1 and return true.

Try this out for yourself: pick some example values of n, and work through the steps used by FERMAT-TEST to compute a result; you could even use a friend to generate random values of a to make sure you cannot cheat! Using the algorithm, see if you can identify some

1. other Carmichael numbers, or
2. Mersenne primes, which have the special form n = 2^k − 1 for an integer value k,

or, get a friend to give you an n (which they know is either prime or composite) and try to decide whether or not it is prime.
3.1.1 Different styles of structure

There are of course lots of ways we can write down the same algorithm. For example, where the inputs and outputs need more explanation (e.g., to explain their type or meaning), it is common to write it in a slightly more verbose but otherwise similar way:

```
1 algorithm Fermat-Test begin
   Input: An integer n
   Output: false if n is composite, otherwise true if n is probably prime
2   Choose a random number a between 2 and n − 1 inclusive
3   Compute d = gcd(a, n), the greatest common divisor of a and n
4   If d ≠ 1 then return false as the result
5   Compute t = a^{(n−1)} (mod n)
6   If t = 1 then return true as the result, otherwise return false
7 end
```

As more of a contrast, Figure 3.1 shows a totally different way to describe the same algorithm. You might find this more natural, or easier to read: this form is called a flow chart [9] and can be traced back to the early 1920s. Since then, flow charts have become a common way of describing “real life” algorithms such as troubleshooting instructions that tell you what to do when your television breaks down, or for capturing decision making procedures in organisations. The point is, both our written and flow chart versions of Fermat-Test describe exactly the same thing: they are both algorithms.

3.1.1.2 Different styles of notation

There is one final, minor issue remaining that we need to be careful about whatever style of algorithm we opt for. As an example, look at the original Fermat-Test algorithm again, specifically at lines #3 and #4: both make use of the = (or “equals”) symbol. In line #3 we mean “assign the value produced by computing gcd(a, n) to d” whereas in line #4 we mean “evaluate to true if and only if the value d does not equal one”. It is reasonable to argue that we might confuse the two meanings. For example, we might mistakenly interpret “compute t = a^{(n−1)} (mod n)” as “compute the value false” since the value t does not equal a^{(n−1)} (mod n); we have not assigned any value to t yet! To prevent this possible ambiguity, Computer Scientists often use

1. the = symbol to mean equality, and
2. the ← symbol to mean assignment.

As such, we should really rewrite the algorithm as follows:

```
1 algorithm Fermat-Test(n) begin
2   Choose a random number a between 2 and n − 1 inclusive
3   Compute d ← gcd(a, n), the greatest common divisor of a and n
4   If d ≠ 1 then return false as the result
5   Compute t ← a^{(n−1)} (mod n)
6   If t = 1 then return true as the result, otherwise return false
7 end
```

3.1.2 What is not an algorithm?

3.1.2.1 Algorithms ≠ functions

It is tempting to treat algorithms like Mathematical functions. After all, invoking an algorithm looks like the use of a Mathematical function; writing Sn(90) uses the function Sn to compute a result using the input 90 for example. Strictly speaking, the difference (or at least one difference) comes down to the idea of state.

For Mathematical functions, writing something like Sn(x) might give different results depending on x, but once we have chosen an x we get the same same result every time. For example Sn(90) always equals 1 so we are entitled to write Sn(90) = 1, or use 1 whenever we see Sn(90). But algorithms are different: they may depend on the context they are invoked in; there may be some state which effects how an algorithm behaves and hence the result it returns. Another way to say the same thing is that algorithms might have side effects which alter the state; in contrast, functions are pure in the sense they are totally self contained.
Choose a random number \( a \) between 2 and \( n - 1 \) inclusive.

Compute \( d = \gcd(a, n) \), the greatest common divisor of \( a \) and \( n \).

Is \( d = 1 \)?

If yes, return false as the result.

Compute \( t = a^{n-1} \mod n \).

Is \( t = 1 \)?

If yes, return false as the result.

Is \( d \neq 1 \)?

If yes, return true as the result.

Figure 3.1: A flow chart description of Fermat-Test.
Fermat-Test demonstrates this fact neatly: line #2 reads “choose a random number $a$”. Since this can be any number (as long as it is co-prime to $n$), it is perfectly possible we might choose a different one each time we invoke Fermat-Test. So the first time we invoke

Fermat-Test(221)

we might choose $a = 11$ in line #2 as above; we know the result in this case is false. However, imagine we invoke the algorithm a second time

Fermat-Test(221)

and choose $a = 47$ instead. This time, the result is true. So we have got a different result with the same input: the context (represented in this case by whatever we are choosing random numbers with) influences what result we get, not just the input.

3.1.2.2 Algorithms ≠ programs

It is tempting to treat algorithms like programs. If you have written any programs before, there is a good chance Fermat-Test has a similar look and feel. Usually when we talk about a program, we mean something which is intended to be executed by a real computer: a word processor, a web-browser, or something like that. This means a program should be written in a “machine readable” form: each step or action required by a program needs to be something a computer can actually do.

Within Fermat-Test for example, we are required to compute $a^n - 1$ (mod $n$) for some $a$ and $n$. Would most computers know how to do this? Probably not. The computer might know how to do multiplication, and we might be able to explain to it how to do exponentiation using another algorithm, but it is not reasonable that the computer will know how to do the computation itself.

So in contrast to a “machine readable” program, we have written the Fermat-Test algorithm in a far less restrictive “human readable” form. Whereas a program relates to a real computer, you can think of an algorithm as relating to some abstract or make believe uber-computer which does not have any similar restrictions. The algorithm needs to be implemented, by rewriting it in a programming language, before it is ready for execution by a real computer.

3.2 Algorithms for multiplication

Imagine you are asked to multiply one number $x$ by another number $y$. You might perform multiplication in your head without thinking about it, but more formally what does multiplication actually mean? If you met someone from a different planet who does not know what multiplication is, how could you describe to them the steps required to compute $x \cdot y$? Clearly the answer is to write an algorithm that they can follow.

As a starting point, notice that multiplication is just repeated addition. So for example

$$x \cdot y = x + x + \cdots + x + x,$$

which means if we select $y = 6$ then we obviously have

$$x \cdot 6 = x + x + x + x + x + x.$$

Based on this approach, we can write an algorithm that describes how to multiply $x$ by $y$. We just need a list of careful directions that someone could follow:

```
1 algorithm Multiply-Repeat(x, y) begin
2     t ← 0
3     for i from 1 upto y do
4         t ← t + x
5     end
6     return t
7 end
```

The basic idea of this algorithm is simple: add together $y$ copies of $x$, keeping track of the value we have accumulated so far in $t$. Of course you could argue the algorithm is much harder to read and understand because we have replaced this English description with something that looks much more formal. On the other hand, by doing this we have made things more precise. That is, there is no longer any room for someone to misinterpret the description if they know how each construct behaves.
The construct in line #2 is another example of the **assignment** we saw earlier: it assigns a value (on the right) to a name, or variable (on the left). When we write \( X \leftarrow Y \), the idea is to set the variable \( X \) to a value given by evaluating the expression \( Y \). In this case, the construct \( t \leftarrow 0 \) sets the variable \( t \) to the value 0: every time we evaluate \( t \) in some expression after this, we can substitute 0 until \( t \) is assigned a new value.

The construct that starts in line #3 is an example of a **loop**. When we write for \( X \) do \( Y \), the idea is to repeatedly process the block \( Y \) for values dictated by \( X \); we say that the construct **iterates** over \( Y \). Our loop is **bounded** because we know how many times we will process \( Y \) before we start.

In this case the block is represented by line #4, and hence \( t \leftarrow t + x \) is iterated over for values of \( i \) in the range \( 1 \ldots y \). So basically the loop does the same thing as copying out line #4 a total of \( y \) times, i.e.,

\[
\begin{align*}
  t & \leftarrow t + x \quad | i = 1 \\
  t & \leftarrow t + x \quad | i = 2 \\
  & \vdots \\
  t & \leftarrow t + x \quad | i = y \\
\end{align*}
\]

The construct in line #6 is an example of a **return**. When we write return \( X \), the idea is that we evaluate \( X \) and return this as the result of the algorithm. In this case, we return the value accumulated in \( t \) as the result.

Now that we know the meaning of each line, we can invoke the algorithm and perform the steps required to compute a result. Imagine we select \( x = 3 \) and \( y = 6 \) for example; the steps we perform would be something like the following:

**Step #1** Assign \( t \leftarrow 0 \).

**Step #2** Assign \( t \leftarrow t + x \), i.e., \( t \leftarrow 0 + 3 = 3 \).

**Step #3** Assign \( t \leftarrow t + x \), i.e., \( t \leftarrow 3 + 3 = 6 \).

**Step #4** Assign \( t \leftarrow t + x \), i.e., \( t \leftarrow 6 + 3 = 9 \).

**Step #5** Assign \( t \leftarrow t + x \), i.e., \( t \leftarrow 9 + 3 = 12 \).

**Step #6** Assign \( t \leftarrow t + x \), i.e., \( t \leftarrow 12 + 3 = 15 \).

**Step #7** Assign \( t \leftarrow t + x \), i.e., \( t \leftarrow 15 + 3 = 18 \).

**Step #8** Return \( t = 18 \).

After eight steps we reassuringly find the result is \( x \cdot y = 18 \). The key thing to realise is that fundamentally we are still just following directions: lines in our algorithm might be more formal than “go through two roundabouts”, but as long as we know what they mean we can carry out the corresponding steps just as easily.

Another way of looking at what multiplication means is to see that it simply adds another “weight” to the digits that describe \( y \). It might look odd, but imagine we wrote \( y \) out as an \( n \)-bit binary number, i.e., we write

\[
y = \sum_{i=0}^{n-1} y_i \cdot 2^i
\]

where clearly each \( y_i \in \{0,1\} \). Then, we could write

\[
x \cdot y = x \cdot \sum_{i=0}^{n-1} y_i \cdot 2^i = \sum_{i=0}^{n-1} y_i \cdot x \cdot 2^i.
\]

What we are doing is taking each weighted digit of \( y \), and adding a further weight \( x \) to the base which is used to express \( y \) in. Again, as an example, selecting \( y = 6_{(10)} = 110_{(2)} \) we find that we still get the result we would expect to

\[
\begin{align*}
y \cdot x & = y_0 \cdot x \cdot 2^0 + y_1 \cdot x \cdot 2^1 + y_2 \cdot x \cdot 2^2 \\
& = 0 \cdot x + 1 \cdot x \cdot 2^1 + 1 \cdot x \cdot 2^2 \\
& = 0 \cdot x + 2 \cdot x + 4 \cdot x \\
& = 6 \cdot x
\end{align*}
\]
So far so good. Except that this still looks unpleasant to actually compute. For example we keep having to compute those powers of two to weight the terms. Fortunately, a British mathematician called William Horner worked out a scheme to do this more neatly [10]. Sometimes this is termed Horner’s rule (or scheme): bracket the thing we started with in such a way that instead of having to compute the powers of two independently we sort of accumulate them as we go. This is best shown by example:

\[
y \cdot x = y_0 \cdot x + 2 \cdot ( y_1 \cdot x + 2 \cdot ( y_2 \cdot x + 2 \cdot (0 \cdot 0) )) \\
= 0 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 1 \cdot x + 0 \cdot 0 ) ) \\
= 0 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot ( 1 \cdot x ) ) \\
= 0 \cdot x + 2 \cdot ( 1 \cdot x + 2 \cdot x ) \\
= 0 \cdot x + 6 \cdot x \\
= 6 \cdot x
\]

In much the same way as above, we can write an algorithm that describes how to multiply \( x \) by \( y \) using this approach:

```plaintext
1 algorithm Multiply-Horner(x, y) begin
2   t ← 0
3   for i from |y| − 1 downto 0 do
4     t ← 2 \cdot t
5     if y_i = 1 then
6       t ← t + x
7     end
8   end
9   return t
10 end
```

This algorithm again accepts two inputs called \( x \) and \( y \), the two numbers we would like to multiply together, and again produces one output that gives us the result \( x \cdot y \). This time, the basic idea is to write \( y \) in binary, and process it from left-to-right one digit at a time. In terms of the bracketing, we work from inside outward, applying Horner’s rule and keeping track of an accumulated value called \( t \):

- The construct in line #2 is another assignment. We have already seen what this means: it sets the variable \( t \) to the value 0.
- The construct that starts in line #3 is another loop; this is a little different from the previous one we saw. The first difference is that the values of \( i \) go downward rather than upward: the block, now represented by lines #4 to #7, is iterated over for \( i \) in the range \(|y| − 1 \ldots 0\). That is fine though, we still just copy out lines #4 to #7 a total of \(|y| \) times making sure the right values of \( i \) are alongside each copy, i.e.,

\[
\left. \begin{array}{l}
   t ← 2 \cdot t \\
   \text{if } y_i = 1 \text{ then } t ← t + x \\
   \vdots \\
   t ← 2 \cdot t \\
   \text{if } y_i = 1 \text{ then } t ← t + x \\
   t ← 2 \cdot t \\
   \text{if } y_i = 1 \text{ then } t ← t + x \\
\end{array} \right\} i = |y| - 1 \\
\left. \begin{array}{l}
   \vdots \\
   t ← 2 \cdot t \\
   \text{if } y_i = 1 \text{ then } t ← t + x \\
   t ← 2 \cdot t \\
   \text{if } y_i = 1 \text{ then } t ← t + x \\
\end{array} \right\} |y| \text{ copies} \\
\left. \begin{array}{l}
   t ← 2 \cdot t \\
   \text{if } y_i = 1 \text{ then } t ← t + x \\
\end{array} \right\} i = 0
\]

The second difference is that the block actually uses \( i \) within it. That is fine as well: wherever we see an \( i \), we can substitute the right value for that iteration. For example, the last copy from above becomes

\[
t ← 2 \cdot t \\
\text{if } y_0 = 1 \text{ then } t ← t + x
\]

after we substitute in the value \( i = 0 \).
- The construct that starts starts on line #5 is an example of a condition; it forms part of the block iterated over by the loop. When we write if \( X \) then \( Y \), the idea is that we perform a test: if \( X \) evaluates to true then we process the block \( Y \), otherwise we skip it. In this case, we test the \( i \)-th bit of \( y \): if \( y_i = 1 \) then we add \( x \) to \( t \) in line #6, otherwise \( t \) remains unchanged.
• The construct in line #9 is another return. We have already seen what this means: it returns $t$ as the result.

This algorithm is slightly more complicated than the last one. However, in the same way as before we know what each line means so we can invoke the algorithm and carry out steps in order to compute a result. Imagine we again select $x = 3$ and $y = 6_{(10)} = 110_{(2)}$; the steps we perform would be something like the following:

**Step #1** Assign $t \leftarrow 0$.

**Step #2** Assign $t \leftarrow 2 \cdot t$, i.e., $t \leftarrow 2 \cdot 0 = 0$.

**Step #3** Since $y_2 = 1$, assign $t \leftarrow t + x$, i.e., $t \leftarrow 0 + 3 = 3$.

**Step #4** Assign $t \leftarrow 2 \cdot t$, i.e., $t \leftarrow 2 \cdot 3 = 6$.

**Step #5** Since $y_1 = 1$, assign $t \leftarrow t + x$, i.e., $t \leftarrow 6 + 3 = 9$.

**Step #6** Assign $t \leftarrow 2 \cdot t$, i.e., $t \leftarrow 2 \cdot 9 = 18$.

**Step #7** Since $y_0 = 0$, skip the assignment $t \leftarrow t + x$.

**Step #8** Return $t = 18$.

The algorithm has clearly computed the result in a different way (i.e., the steps themselves are different), but we still find that $x \cdot y = 18$ as expected.

---

Although reading and following invocations of existing algorithms is a good start, writing your own algorithms is really the only way to get to grips with this topic. In Chapter 2 we met the Hamming-Weight function: it counts the number of elements in a binary sequence that are equal to one. Write an algorithm that can compute Hamming-Weight($x$) for a suitable input sequence $x$; demonstrate how it does so by listing the steps (similar to above) for an example $x$.

### 3.3 Algorithms for exponentiation

So much for multiplication, what about the exponentiation we needed in Fermat-Test? It turns out that we can pull the same trick again. In the same was as we wrote multiplication as repeated addition, we can write exponentiation as repeated multiplication:

$$x^y = \underbrace{x \cdot x \cdot \ldots \cdot x \cdot x}_{y \text{ copies}}.$$ 

If we again select $y = 6$ then we obviously have

$$x^6 = x \cdot x \cdot x \cdot x \cdot x \cdot x.$$ 

The thing to notice is that there is a duality here: where there was an addition in our description of multiplication, that has become a multiplication in our description of exponentiation; where there was a multiplication, this has become an exponentiation. As such, we can adapt the Multiply-Repeat algorithm as follows:

```plaintext
1 algorithm Exponentiate-Repeat(x, y) begin
2   t \leftarrow 1
3   for i from 1 upto y do
4     t \leftarrow Multiply-Horner(t, x)
5   end
6 return t
7 end
```
One purpose of this is to highlight a subtle but fairly obvious fact: we are allowed to invoke one algorithm from within another one. In this case, we needed to multiply \( t \) by \( x \) in line #4; Multipl-y-Repeat and Multipl-y-Horner both compute the same result, so we could have used either of them here to do what we wanted. Either way, the idea is that half way through following the steps within the Exponentiate-Repeat algorithm, we stop for a while and follow steps from Multipl-y-Horner instead so as to compute the value we need. Again selecting \( x = 3 \) and \( y = 6 \), the steps we perform would be something like the following:

**Step #1** Assign \( t \leftarrow 1 \).

**Step #2** Invoke Multipl-y-Horner\((t, x)\), i.e., invoke Multipl-y-Horner\((1, 3)\),

- **Step #2.1** Assign \( t \leftarrow 0 \).
- **Step #2.2** Assign \( t \leftarrow 2 \cdot t \), i.e., \( t \leftarrow 2 \cdot 0 = 0 \).
- **Step #2.3** Since \( y_1 = 1 \), assign \( t \leftarrow t + x \), i.e., \( t \leftarrow 0 + 1 = 1 \).
- **Step #2.4** Assign \( t \leftarrow 2 \cdot t \), i.e., \( t \leftarrow 2 \cdot 1 = 2 \).
- **Step #2.5** Since \( y_0 = 1 \), assign \( t \leftarrow t + x \), i.e., \( t \leftarrow 2 + 1 = 3 \).
- **Step #2.6** Return \( t = 3 \).

then assign \( t \leftarrow 3 \).

**Step #3** Invoke Multipl-y-Horner\((t, x)\), i.e., invoke Multipl-y-Horner\((3, 3)\),

- **Step #3.1** Assign \( t \leftarrow 0 \).
- **Step #3.2** Assign \( t \leftarrow 2 \cdot t \), i.e., \( t \leftarrow 2 \cdot 0 = 0 \).
- **Step #3.3** Since \( y_1 = 1 \), assign \( t \leftarrow t + x \), i.e., \( t \leftarrow 0 + 3 = 3 \).
- **Step #3.4** Assign \( t \leftarrow 2 \cdot t \), i.e., \( t \leftarrow 2 \cdot 1 = 6 \).
- **Step #3.5** Since \( y_0 = 1 \), assign \( t \leftarrow t + x \), i.e., \( t \leftarrow 6 + 3 = 9 \).
- **Step #3.6** Return \( t = 9 \).

then assign \( t \leftarrow 9 \).

\[
\ldots
\]

**Step #8** Return \( t = 729 \).

Nothing has changed: we are still just following directions. We need to keep track of which algorithm we are following and ensure the names we give to variables do not get mixed up, but other than that things are not fundamentally more complicated.

Of course, we can also use Horner’s rule by replacing all the additions with multiplications, and all multiplications with exponentiations. Using the same example as previously, we would end up with

\[
\begin{align*}
x^y &= x^{y_0} \cdot (x^{y_1} \cdot (x^{y_2} \cdot ( (1)^2 )^2 )^2 )^2 \\
    &= x^0 \cdot (x^1 \cdot (x^1 \cdot ( (1)^2 )^2 )^2 )^2 \\
    &= x^0 \cdot (x^1 \cdot (x^1 \cdot 1 )^2 )^2 \\
    &= x^0 \cdot (x^1 \cdot (x^1^2 )^2 )^2 \\
    &= x^0 \cdot (x^1 \cdot x^2 )^2 \\
    &= x^0 \cdot (x^3 )^2 \\
    &= x^6 \\
\end{align*}
\]

Unsurprisingly, our method for multiplying one number by another has a dual which is able to exponentiate one number by another:

```plaintext
1 algorithm Exponentiate-Horner(x, y) begin
2   t ← 1
3   for i from |y| − 1 downto 0 do
4      t ← Multipl-y-Horner(t, t)
5      if y_i = 1 then
6         t ← Multipl-y-Horner(t, x)
7      end
8   end
9 return t
10 end
```


You should compare Exponentiate-Horner as given above, line-by-line, with Multiply-Horner given earlier. Notice that they use the same idea; their structure is the same, we simply changed all the additions to multiplications. Exponentiate-Horner is often called the square-and-multiply algorithm [6] since it performs a sequence of squaring (line #4) and multiplication (line #6): one squaring is performed for every bit of \( y \) whether it is equal to zero or one, and one multiplication for those bits of \( y \) which are equal to one.

\[
y = 6_{(10)} = 110_{(2)}
\]

and processed \( y_2 = 1 \) first, then \( y_1 = 1 \) then finally \( y_0 = 0 \). There is an alternative version of the algorithm that processes \( y \) the other way around, i.e., right-to-left. Do some research into this alternative: write down the algorithm, and convince yourself it will computes the same result using some examples. Can you think why the left-to-right version might be preferred to the right-to-left alternative, or vice versa?

### 3.4 Computational complexity

Imagine we have two algorithms that solve the same problem, but do so in different ways. This should not be hard, because we already have some suitable candidates: given an \( x \) and \( y \), Multiply-Repeat and Multiply-Horner compute the same result \( x \cdot y \) differently. So armed with these, or another example of your choice, here is a question: which one of the algorithms is the “best”? Actually, maybe we should go back a step: what does “best” even mean? Fastest? Shortest? Most attractive? All are reasonable measures, but imagine we select the first one and focus on selecting the algorithm which gives us a result in the least time.

Suppose we implemented the algorithms we would like to compare. This would give one way of comparing one against the other, we could simply time how long the corresponding programs take to execute on a computer. There are, however, many factors which might influence how long the execution of each program takes. For example:

- the skill of the programmer who implements the algorithm,
- the programming language used,
- the speed at which the computer can execute programs,
- the input, and
- the algorithm implemented by the program.

We know little or nothing about the first three factors, so have to focus on the latter two. Our goal is to introduce the subject of computational complexity [4]. This might sound scary, but is essentially about selectively ignoring detail: via a series of sane simplifications, each focusing on the most important, big picture issues, we can determine the quality of one algorithm compared to another.

#### 3.4.1 Step counting and dominant steps

As a guess at how long an algorithm would take to give a result if it were implemented and executed, we could count how many steps it takes. If you think about it, this makes perfect sense: the more steps the algorithm takes, the longer it will take to give result. But this would be quite a boring task if the algorithm had many steps. So the first simplification we make is to focus just on a small set of dominant steps, i.e., those steps we think are the most important. For algorithms that sort things, maybe the number of comparisons is the most important thing to count; for algorithms that process sequences of things, maybe the number of accesses to the sequence is the most important thing to count.

Since we are comparing algorithms that perform multiplication, it makes sense that we are interested mainly in arithmetic operations: the most important thing to count is the number of additions each algorithm uses to compute a result. Figure 3.3 shows how many additions each algorithm performs for a range of inputs (ignore the columns marked \( f(n) \) and \( g(n) \) for now). It only includes a limited sample of inputs,
Figure 3.2: A table showing values of $y$ and the number of bits in their binary representation.

<table>
<thead>
<tr>
<th>$y$</th>
<th>$\log_2 (y+1)$</th>
<th>$\lceil\log_2 (y+1)\rceil$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (10)</td>
<td>1.000</td>
<td>1</td>
</tr>
<tr>
<td>2 (10)</td>
<td>1.584</td>
<td>2</td>
</tr>
<tr>
<td>3 (10)</td>
<td>2.000</td>
<td>2</td>
</tr>
<tr>
<td>4 (10)</td>
<td>2.322</td>
<td>3</td>
</tr>
<tr>
<td>5 (10)</td>
<td>2.585</td>
<td>3</td>
</tr>
<tr>
<td>6 (10)</td>
<td>2.807</td>
<td>3</td>
</tr>
<tr>
<td>7 (10)</td>
<td>3.000</td>
<td>3</td>
</tr>
<tr>
<td>8 (10)</td>
<td>3.170</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 3.3: A table showing the number of additions performed by Multiply-Repeat and Multiply-Horner, and the values of the associated step counting functions.

<table>
<thead>
<tr>
<th>$y$</th>
<th>$\lceil\log_2 (y+1)\rceil$</th>
<th>Multiply-Repeat $f(n) = 2^n - 1$</th>
<th>Multiply-Horner $g(n) = 2 \cdot n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

but already we can identify two problems: first, the number of addition depends on $y$, and, second, it is not clear cut which algorithm uses the least number of additions. That is, sometimes Multiply-Repeat uses less, sometimes Multiply-Horner uses less. So in terms of answering our question, we are not really much better off than when we started.

3.4.2 Problem size and step counting functions

Usually we would like to make a general judgement about an algorithm which is independent of the input. To do this, we need to make another simplification: instead of thinking about the number of steps for a particular input $y$, we will shift things to consider a **problem size** $n$. The basic idea is that we try to write down a function for the number of steps an algorithm takes in terms of $n$, the problem size; this is a **step counting** function. Once we have such functions, we can forget about the algorithms themselves and simply compare the associated step counting functions with each other.

There is not really a general definition of what the problem size should mean; basically it is just a measure of how hard the problem is. For example, if we have an algorithm that sorts a sequence of numbers, the length of the sequence makes a good problem size: if $n$ is larger, the problem is harder in the sense that the algorithm has to do more work. In our case, suppose we are looking at $n$-bit numbers: each input $x$ and $y$ has $n$ binary digits. If $n$ is larger, the problem is harder in the sense that the algorithm has to do more work: multiplying together large numbers is harder than multiplying small numbers together. We can describe $n$ in terms of $y$ as

$$n = \lceil\log_2 (y+1)\rceil.$$  

That is, we add one to $y$ then it takes the logarithm to base two, and then round the result up to the nearest integer (this is called the **ceiling** function). Figure 3.2 details some results.

Consider Multiply-Repeat to start with: if we have $n$-bit inputs, how many additions will the algorithm perform? Looking at the algorithm again, we can see that the best case is when the inputs are really small: that way, we are clearly going to perform the least additions. The smallest number we can write in $n$ bits is always going to be 0; this means we perform 0 additions. What about the worst case? This would be represented by the largest input we can write using $n$ bits; this is $2^n - 1$. If we set $n = 2$ for example, the largest number we can write is three: the next largest number, i.e., four, needs $n = 3$. In the worst case then, we can write the step counting function

$$f(n) = 2^n - 1.$$
to describe how many additions we would do. If \( n = 2 \), we are talking about 2-bit numbers and \( f(n) = 3 \) indicates that we would do at most three additions.

Now consider Multiply-Horner. If we have \( n \)-bit inputs, the loop in the algorithm iterates \( n \) times: we perform one iteration for each bit of the input \( y \). This means we do \( n \) additions as a result of line \#4. In the best case, all the bits of the input are zero (i.e., each \( y_i = 0 \)) so we never process line \#6 and do no more additions; in the worst case all the bits are one (i.e., each \( y_i = 1 \)) and we always process line \#6 and do \( n \) more additions. So in the worst case our step counting function will be

\[
g(n) = 2 \cdot n.
\]

Choosing \( n = 2 \) again, \( g(n) = 4 \) indicates that we would do four additions.

Figure 3.3 shows the results of the step counting functions along side the actual number of steps performed by the algorithms. Clearly the two are not the same; the key thing to realise is that the actual number of steps is always less than or equal to the step counting function for the corresponding algorithm. This is because we developed each step counting function in relation to the worst case behaviour for a given input size, not the actual behaviour for that specific input.

### 3.4.3 Ignoring small problems and minor terms

The next simplification we make is to assume \( n \) is always large. In a way, this is obviously sensible: if \( n \) were small then nobody cares what algorithm we choose, they will all be fast enough. The main result of making such an assumption is that we can make our step counting functions simpler. For example, we currently have the function

\[
f(n) = 2^n - 1.
\]

If \( n \) is going to be large, who cares about the 1 term? This will be incidental in comparison to the \( 2^n \) term. Imagine we select \( n = 8 \) for example: \( 2^8 = 256 \) and \( 2^8 - 1 = 255 \) are close enough that we may as well just treat them as the same and rewrite the function as

\[
f(n) = 2^n.
\]

What about the other step counting function? Consider the following sequence of step counting functions:

\[
\langle 1 \cdot n, \ 2 \cdot n, \ 3 \cdot n, \ 4 \cdot n, \ 5 \cdot n, \ 6 \cdot n, \ \ldots \rangle.
\]

Assuming \( n \) is positive, if we read the sequence from left-to-right each function is greater than the last one. For example \( 3 \cdot n \) will always be greater than \( 2 \cdot n \), no matter what \( n \) is. This means the number of steps would grow as we read from left-to-right: the associated algorithms would be slower and slower.

The natural end to this sequence is the point where one of the functions is \( n \cdot n = n^2 \). This is basically saying that \( n^2 \) is greater than all the functions in our sequence: if we select any constant \( c < n \), then \( c \cdot n \) is going to be less than \( n^2 \). As a result, another simplification we can make is to treat any step counting function that looks like \( c \cdot n \) as \( n \) instead. This means we can rewrite

\[
g(n) = 2 \cdot n
\]
as

\[
g(n) = n.
\]

On one hand, this seems mad: an algorithm whose step counting function is \( 200 \cdot n \) will take 100 times more steps than one whose step counting function is \( 2 \cdot n \). So how can we rationally treat them as the same? Well, consider running an algorithm on two different computers: one is about ten years old, and one is very modern. The same algorithm will probably run 100 times slower on the old computer than it does on the new one: a ten year old computer is likely to be 100 times slower than a new one. This comparison is nothing to do with the algorithm in the sense that the computers are what makes the difference. So whether we take the step counting function as being \( 2 \cdot n \) or \( 100 \cdot n \) does not matter, the constant will eventually be one if we wait for a computer which is fast enough. So in comparing algorithms we always ignore constants terms, thus treat both \( 2 \cdot n \) and \( 100 \cdot n \) as the function \( n \). Actually writing

\[
100 \cdot n = n
\]

looks a bit odd, so instead we use the **big-O notation** [2] and we write the above non-equation as the equation

\[
100 \cdot n = O(n)
\]
Figure 3.4: A graph illustrating the growth of several functions as the problem size is increased.
which basically says that if \( n \) is big enough, then the function on the left behaves at worst like the function inside the big-O (having ignored any constants). The phrase “at worst” is crucial; we can write \( n = O(n^2) \) since \( n \) is at worst \( n^2 \), but this is slightly lazy since we also know that \( n = O(n) \).

The big-O notation is one of the most important concepts, and also items of Mathematical notation, in Computer Science. It is worth getting used to it now, so consider the following nine examples:

\[
\begin{align*}
2n + 1 &= O(n) \\
\frac{\log n}{n^3} &= O(n) \\
\log n^3 &= O(n) \\
2^n + 3n &= O(2^n) \\
2^n \log n &= O(2^n) \\
(\log n)^3 &= O((\log n)^4) \\
6n^2 + 3n + 1 &= O(6n^2) \\
6n^2 + 3n + 1 &= O(n^2 + n) \\
(\log n)^3 &= O((\log n)^3)
\end{align*}
\]

One of them is wrong, can you work out which one (and the reason why)?

In turns out big-O is a short-hand for big-Omicron, if you want to be exact about the Greek symbol used. Within this context, using other Greek symbols allows us make other statements about a function: there is a whole family, including \( o \) (or little-O), \( \omega \) (or little-Omega), \( \Omega \) (or big-Omega), and \( \Theta \) (or big-Theta) for instance. Do some research into what these mean, and why we might use them instead of big-O in specific situations.

### 3.4.4 Studying the growth of functions

Have a look at Figure 3.4. It plots the growth of a few simple step counting functions as the problem size \( n \) grows; actually, it only uses very small values of \( n \) because some of the functions fly off the top quite quickly. What we are trying to do is visualise how the functions behave, relating the shape of the plot for a given step counting function to the number of steps taken by the associated algorithm.

1. The plot for \( h(n) = 1 \) is flat: it does not matter what the problem size is, there is no growth in the function. This is great! No matter how large the problem is, the step counting function says the algorithm will take a constant number of steps.

2. The next plot is for \( h(n) = \log_2 n \). Unlike \( h(n) = 1 \), this function grows as \( n \) grows; the larger we make the problem size, the more steps the algorithm takes. On the other hand, \( h(n) = \log_2 n \) grows quite slowly. We can see, from the trajectory that the plot is taking, that even for large values of \( n \), \( h(n) = \log_2 n \) will not be too large (in comparison to some of the other functions).

3. The plot for \( h(n) = n \) is not so great. This time, the growth is constant: if the problems size is \( n \), the algorithm takes \( n \) steps. If we compare the trajectory of the function \( h(n) = n \) with \( h(n) = \log_2 n \), \( h(n) = n \) is clearly going to be much larger than \( h(n) = \log_2 n \) in the long run.

4. Then things take a turn for the worse. The functions \( h(n) = n^2 \) and \( h(n) = 2^n \) grow really quickly compared to the others. The function \( h(n) = 2^n \) shows what we call exponential growth: looking at the almost vertical trajectory of \( h(n) = 2^n \), it will be enormous even for fairly small values of \( n \). So for large values of \( n \), the algorithm will take so many steps to produce a result that it is probably not worth even invoking it!

Just so this hits home, notice that the function \( h(n) = n \) is equal to 100 if \( n = 100 \): if the problem size is 100, the algorithm takes 100 steps. On the other hand, the function \( h(n) = 2^n \) is equal to

\[
1267650600228229401496703205376
\]

if \( n = 100 \). That is a lot of steps!

Now, if we put all this evidence together we can try to answer our original question: given a problem size \( n \), the number of steps taken by \textsc{Multiply-Repeat} can be approximated by the function \( f(n) = 2^n \); for \textsc{Multiply-Horner} the number of steps is approximately the function \( g(n) = n \). We have seen that \( f \) grows so quickly that even with quite small \( n \), it is fair to say it will always be larger than \( g \). This implies \textsc{Multiply-Horner} gives us a result in the least time, and is therefore the “best” algorithm. Putting this into context helps show the value of our analysis: in real computers, 32-bit numbers are common. If we wanted to multiply such numbers together \textsc{Multiply-Repeat} would take approximately 4294967296 steps in the worst case; if we selected \textsc{Multiply-Horner} instead, we only need approximately 32 steps. Hopefully it is clear that even though we have made simplifications along the way, they would have to be wrong on
a monumental scale to make our selection the incorrect one! So to cut a long story short, computational complexity has allowed us to reason about and compare algorithms in a meaningful, theoretical way: this is a very powerful tool if used for the right job.
BIBLIOGRAPHY

CHAPTER 4

PLAYING HIDE-AND-SEEK WITH
VIRUS SCANNERS

A computer virus [5] is a program that attempts to infect another host file (often another program) by replicating itself (e.g., appending a copy of itself to the host). This is analogous to how Biological viruses attach themselves to host cells. The idea is that when the host file is loaded or executed, the virus is activated: at this point the virus can deploy a payload of some kind, which often has malicious intent. Although a definition for computer viruses was yet to be written in 1971, it represents the point in time when instances of programs we retrospectively call viruses started to appear. Creeper [7], written by Bob Thomas, is often cited as the first: it targeted PDP [19] computers connected to the Advanced Research Projects Agency Network (ARPANET) [3], propagating itself via the telephone system. Once resident it displayed the annoying but otherwise non-destructive message “I’m the creeper, catch me if you can” to users, but seemingly did not replicate itself (at least intentionally): once the payload was deployed, it simply moved on to the next computer. Perhaps more (in)famous though, is the 1988 Morris Worm [17] written by Robert Morris. Although it again lacked a malign payload, the worm infected computers multiple times, slowing down their normal operation to the point where roughly 10% of computers connected to the Internet [14] were claimed to have been left non-operational.

A rich published history [24] of computer viruses now exists; since much of it is legally dubious, more undoubtedly remains unpublished. Both Creeper and the Morris Worm represented an experimental era in this historical timeline, with honest security researchers exploring what was and was not possible; neither was intentionally malicious, with Creeper being explicitly described as an experimental self-replicating program. Beyond the technological aspects, the Morris Worm is also notorious because of the subsequent prosecution of Morris and treatment of information, network and computer security as an important emerging threat: governments, rather than just programmers and researchers, started to take notice. Today of course, viruses (and malware [16] more generally) have become big business for the wider computer underground. Not content with sending you annoying messages, a modern virus might delete or encrypt your files (holding the owners to ransom, so they must pay before the content can be read again), steal information [15], or even attempt to actively destroy physical infrastructure. Stuxnet [23], for instance, represents what is widely viewed as a “cyber-weapon” developed and deployed by some (unnamed) government against uranium enrichment plants in Iran.

In many cases therefore, (cyber-)crime increasingly does pay. The result is an arms race between the people who write viruses, and the people who write anti-virus software. Again drawing on Biology (and the field of virotherapy), the first example of anti-virus was arguably a second virus called Reaper written

---

1 Strictly speaking, a virus is a program that propagates itself from file to file on one computer, but typically requires an external stimulus to propagate between computers (e.g., a user carrying infected files on a USB stick from one computer to another); the requirement for a host file to infect means the virus is typically not a stand-alone program. This contrasts with a worm, which propagates from computer to computer itself, acting as a stand-alone program without the need to infect a host file. A specific example might include aspects of both, so a precise classification is often difficult; we largely ignore the issue, using the term virus as an imprecise but convenient catch-all.
to stamp out the spread of updates to Creeper (which, in contrast to the original, did replicate themselves). More generally however, before we can disable a virus we need to detect whether a given host file is infected or not. Interestingly we can prove this is a good business to be in by showing that it is impossible to write a perfect virus detector algorithm. Imagine for a moment we could do this: DETECT represents a perfect virus detector, so we are sure $\text{DETECT}(x) = \text{true}$ whenever $x$ is a virus, and that $\text{DETECT}(x) = \text{false}$ in all other cases. Now imagine there is a virus called Virus. It is clever: it incorporates a copy of DETECT, the virus detector algorithm, inside itself. A simple description of how Virus works is as follows:

```
1 algorithm Virus begin
2     if DETECT(Virus) = true then
3         Behave like a non-virus, i.e., a normal program
4     else
5         Behave like a virus, i.e., do something “bad”
6 end
```

So when Virus is executed, it checks whether DETECT says it is a virus. If $\text{DETECT}(\text{Virus}) = \text{true}$, i.e., DETECT thinks Virus is a virus, then Virus does nothing. Hence, in this case Virus does not behave like a virus and so DETECT was wrong. If $\text{DETECT}(\text{Virus})$ returns $\text{false}$, i.e., DETECT does not think Virus is a virus, then Virus does something “bad” such as deleting all your files. In this case Virus definitely does behave like a virus, and so DETECT is wrong again. Thus, if we assume a perfect virus scanner DETECT exists then we get a contradiction; this means DETECT cannot exist. Although this proof relates to writing perfect anti-virus software, it is in fact a special case of the famous Halting problem [10].

Despite the fact that we cannot write perfect anti-virus software, it is still worth looking at how we might at least try to detect $V$; a common tool in this context is a virus scanner [2]. Most work in roughly the same way: the idea is that for each virus, one tries to create a unique signature that identifies files containing the virus. The scanner inspects each file in turn, and tries to match the file content with each signature. If there are no matches, then the scanner concludes that there is no virus in the file and moves on to the next one. We can imagine both the signature and the file being sequences of numbers called $S$ and $F$. There are lots of efficient ways to search $F$, the easiest of which is to simply take each $S$ and match it up against the content of $F$ one element at a time:

```
S = ⟨ 3 , 5 , 2 , 1 ⟩
F = ⟨ 6 , 5 , 2 , 1 , 4 , 0 , 3 , 5 , 2 , 1 , ... ⟩
```

In the $i$-th step, we compare $S$ against the sub-sequence of $F$ starting at $F_i$. We continue this process until either we run out of content, or there is a match between $S$ and a particular region of $F$. In the latter case, we have found an occurrence of $S$ in $F$ and identified the file as containing the virus. This approach relies on the fact that the people who sold you the virus scanner send you a new signature whenever a new virus is detected somewhere; this is basically why you need to update your virus scanner regularly. A given signature might not be perfect, and might sound the alarm for some files that do not contain a virus. For example, the signature $⟨3,5,2,1⟩$ might be a valid part of some uninfected data file, but would still cause the scanner to flag it as infected.

In this Chapter we will play the part of the bad guy, and imagine we want to write a virus. The goal is to explain how our virus can avoid detection by virus scanners (or at least the one described above), and yet still execute some payload. The approach we use is to have the virus change itself during execution so as to fool the virus scanner. This sort of self-modifying program is made possible by the way that modern computers execute programs. So before we start talking about viruses, we will start by looking at what programs are and how computers execute them.

### 4.1 Computers and programs

The questions “what is a computer” and “what is computation” are at the core of Computer Science; eminent scholars in the field have spent, and still spend, lots of time thinking about and producing new results in
this area. If you pick up a textbook on the subject, the topic of Turing Machines (TMs) [25] is often used as a starting point. Alan Turing [1], some might say the father of Computer Science, introduced TMs as a theoretical tool to reason about computers and computation: we still use them for the same tasks today. But TMs and their use are a topic in their own right, and one not quite aligned to what we want here. Specifically, we would like to answer some more concrete, more practical questions instead. So, consider that you probably sit in front of and use a physical (rather than theoretical) computer every day: we would like to know “what is that computer” and more importantly “how does it compute things”?

4.1.1 A theoretical computer

A computer is basically just a machine used to process steps, which we call instructions, collected together into programs. This should sound familiar: the computer is doing something similar to what we do when we step through an algorithm. In a sense, the only thing that makes a computer remarkable is that it executes the instructions in a program much faster than we can process algorithms, and more or less without error. Just like we can process any reasonably written algorithm, a computer can execute any program. This is a neat feature: we do not need one computer to send emails and a different one to view web-pages, we just need one general-purpose computer that can do more or less anything when provided with an appropriate program.

To design a computer, we need to write down an algorithm that describes how to process programs (which are simply algorithms). In a very rough sense, as the user of a computer you “see” the result as the combination of an operating system (e.g., UNIX) and a Central Processing Unit (CPU) [4], or processor, that is the main hardware component within the computer. Basically the processor is the thing that actually executes programs, and the operating systems acts as an assistant by loading the programs from disk, allowing you to select which program to execute and so on. Of course we then need to build the hardware somehow, but we will worry about that later; basically what we want as a starting point is an algorithm for processing algorithms. Here is a first attempt:

1. Write a program as a sequence of instructions called \( P \); start executing the sequence from the first element, i.e., \( P_0 \).
2. Fetch the next instruction from the program and call it \( IR \).
3. If \( IR = \perp \) (i.e., we have run out of instructions to execute) or if \( IR = \text{HALT} \) then halt the computer, otherwise execute \( IR \) (i.e., perform the operation it specifies).
4. Repeat from line #2.

Look at this a little more closely. The sequence of instructions we call \( P \) is a program for the computer; if it makes things easier, think of it as a word processor or web-browser or something. Each element of \( P \) is an instruction, and each instruction is taken from an instruction set of possibilities that the computer understands. Lines #2 to #4 form a loop. During each iteration we first fetch the next instruction from \( P \). We call the instruction \( IR \) because in real computers, it is held in the Instruction Register (IR). Once we have \( IR \) to hand, we set about performing whatever operation is specified; then if we encounter a special instruction called \( \text{HALT} \) we stop execution, otherwise we carry out the whole loop again.

Hopefully this seems sensible. It should do, because as we have tried to motivate, it more or less models the same thing you would do if you were processing the steps of an algorithm. An example makes the whole idea easier to understand: imagine we want to execute a short program that computes \( 10 + 20 \). First we write down the program, e.g.,

\[
P = \langle A \leftarrow 10, A \leftarrow A + 20, \text{HALT} \rangle,
\]

and then process the remaining steps of our algorithm to execute it

**Step #1** Fetch \( IR = A \leftarrow 10 \) from the sequence.

**Step #2** Since \( IR \neq \perp \) and \( IR \neq \text{HALT} \), execute \( A \leftarrow 10 \), i.e., set \( A \) to 10.

**Step #3** Fetch \( IR = A \leftarrow A + 20 \) from the sequence.

**Step #4** Since \( IR \neq \perp \) and \( IR \neq \text{HALT} \), execute \( A \leftarrow A + 20 \), i.e., set \( A \) to 30.

**Step #5** Fetch \( IR = \text{HALT} \) from the sequence.

**Step #6** Since \( IR \neq \perp \) but \( IR = \text{HALT} \), halt the computer.
after which we have the result \( 10 + 20 = 30 \) held in \( A \). Hopefully the underlying point is clear. Specifically, bar the issue with building the hardware itself there is no magic behind the scenes: this really is how a computer executes a program.

### 4.1.2 A real, Harvard-style computer

It turns out that the hardware components we would need to build a computer like the one described above relate well to those you would find in a simple pocket calculator. For example:

A pocket calculator:

- Has an accumulator (i.e., the current value).
- Has some memory (accessed via the \( M+ \) and \( MR \) buttons) that can store values.
- Has some device to perform arithmetic (i.e., to add together numbers).
- Has input and output peripherals (e.g., keypad, LCD screen).
- Responds to simple commands or instructions from the user; for example the user can supply things to perform arithmetic on (i.e., numbers) and commands to perform arithmetic (e.g., do an addition).

A computer:

- Has many accumulators (often called registers).
- Has potentially many levels and large amounts of memory (often called RAM).
- Has an Arithmetic and Logic Unit (ALU) to perform arithmetic.
- Has input and output peripherals (e.g., keyboard, mouse, hard disk, monitor).
- Executes sequences of simple instructions called programs; each instruction consists of some operands (i.e., the things to operate on) and an opcode (i.e., the operation to perform).

In fact, early computers essentially were very large versions of what today we would call a calculator. They were used to perform repetitive computation, such as computing tables of \( \sin \) and \( \cos \) for people to use. Computers of this era typically relied on

1. a paper tape [20], where instructions from the program were stored as patterns of holes in the paper, and
2. some memory, where data being processed by the computer was stored.

We could expand on both technologies, but their detail is not really that important. Instead we can model them using two sequences called \( TAPE \) and \( MEM \). When we write \( MEM_i \), we are accessing the \( i \)-th address in memory, whereas \( TAPE_j \) represents the \( j \)-th row on the continuous ream of paper tape.

So imagine we want to build an example computer of this type. We assume it has a tape and a memory as described above, and single accumulator called \( A \). Each element of \( TAPE \) holds an instruction from the program, while each element of \( MEM \) and the accumulator \( A \) can hold numbers; to make things simpler, we will write each number using decimal. The computer can understand a limited set of instructions:

- \( NOP \), i.e., do nothing.
- \( HALT \), i.e., halt or stop execution.
- \( A \leftarrow n \), i.e., load the number \( n \) into the accumulator \( A \).
- \( MEM_n \leftarrow A \), i.e., store the number in the accumulator \( A \) into address \( n \) of the memory.
- \( A \leftarrow MEM_n \), i.e., load the number from address \( n \) in memory into the accumulator \( A \).
- \( A \leftarrow A + MEM_n \), i.e., add the number in address \( n \) of the memory to the accumulator \( A \) and store the result back in the accumulator.
- \( A \leftarrow A - MEM_n \), i.e., subtract the number in address \( n \) of the memory from the accumulator \( A \) and store the result back in the accumulator.
- \( A \leftarrow A \oplus MEM_n \), i.e., XOR the number in address \( n \) of the memory with the accumulator \( A \) and store the result back in the accumulator.
One can view things on the right-hand side of the assignment symbol ← as being read from, and those on the left-hand side as being written to by an instruction. So for example, an instruction

\[ A \leftarrow 10 \]

reads the number 10 and writes it into A. This also implies that memory access can be written in the same way; for example

\[ MEM_{64} \leftarrow A \]

reads the number in A and writes it into memory at address sixty four.

It is important to notice that a given program for our example computer can only include instructions from this instruction set. It simply does not know how to execute any other type. For example, we could not feed it the Fermat-Test algorithm from Chapter 3 because it uses operations not included in the instruction set. Even so, we can start to write useful programs. Consider a similar example to the one we looked at previously: we want to add together two numbers held in \( MEM_4 \) and \( MEM_5 \) (where say \( MEM_4 = 10 \) and \( MEM_5 = 20 \)), and then store the result into \( MEM_6 \). The program consists of the four instructions

\[ \begin{align*}
A & \leftarrow MEM_4 \\
A & \leftarrow A + MEM_5 \\
MEM_6 & \leftarrow A \\
HALT & 
\end{align*} \]

The method used by the computer to execute a program like this is very similar to our first attempt above:

1. Encode a program \( P \) onto paper tape; load this tape into the tape reader and start the computer.
2. Using the tape reader, fetch the next instruction in the program and call it \( IR \).
3. If \( IR = \bot \) (i.e., we have run out of instructions to execute) or if \( IR = HALT \) then halt the computer, otherwise execute \( IR \) (i.e., perform the operation it specifies).
4. Repeat from line 2.

We can describe each step during execution of the program by detailing the state of the computer, e.g., what values are held by \( MEM \), \( TAPE \) and \( A \); Figure 4.1 and Figure 4.2 do just this. The execution shows that after step \#9, the computer halts and we end up with the result of the addition, i.e., 30, stored in \( MEM_6 \); not exactly a word processor or web-browser, but quite an achievement by the standards of the 1940s!

It is important to remember that the program is free to alter memory content, but has no means of altering the tape which houses the instructions. That is, once we start the computer, we cannot change the program: our example computer views instructions and the data as fundamentally different things. This is now termed a Harvard architecture [11] after the Automatic Sequence Controlled Calculator (ASCC) designed by Howard Aiken and built by IBM; the installation at Harvard University, delivered in around 1944, was nicknamed the “Mark 1” [12]. The crucial thing to take away is that there is still no magic involved: as shown in Figure 5.1, the Harvard Mark 1 was a real, physical computer built on exactly the principles as above.

### 4.1.3 A real, von Neumann-style computer

The Harvard style view of computers changed radically when John von Neumann [26] documented the concept of a stored program architecture in around 1945. The basic idea is that instructions and data are actually the same thing. Think about it: what does 1 mean? The meaning of 1 completely depends on the context it is placed in, and how we interpret it: if we interpret it as a number it means the integer one, if we interpret it as an instruction it could mean “perform an addition”. So basically, as long as we have an encoding from numbers to meaning then we can store both instructions and data as numbers in memory. For our purposes, the encoding does not matter too much; imagine we continue to represent everything as decimal numbers:

- \( 00nnnn \) means NOP.
- \( 10nnnn \) means HALT.
- \( 20nnnn \) means \( A \leftarrow n \).
- \( 21nnnn \) means \( MEM_n \leftarrow A \).
(a) Step #1: Load the tape into the tape reader and start the computer.

(b) Step #2: Fetch the next instruction IR = A ← MEM_4 from the tape.

(c) Step #3: Execute the instruction A ← MEM_4, i.e., set A to MEM_4 = 10.

(d) Step #4: Fetch the next instruction IR = A ← A + MEM_5 from the tape.

(e) Step #5: Execute the instruction A ← A + MEM_5, i.e., set A to A + MEM_5 = 30.

(f) Step #6: Fetch the next instruction IR = MEM_6 ← A from the tape.

Figure 4.1: Computing the sum 10 + 20, as executed on a Harvard-style computer.
CPU
state = execute
IR = MEM₆ ← A
A = 30

(a) Step #7: Execute the instruction MEM₆ ← A, i.e., set MEM₆ to A = 30.

CPU
state = fetch
IR = HALT
A = 30

(b) Step #8: Fetch the next instruction IR = HALT from the tape.

CPU
state = execute
IR = HALT
A = 30

(c) Step #9: Execute the instruction HALT, i.e., stop execution.

Figure 4.2: Computing the sum 10 + 20, as executed on a Harvard-style computer.

Figure 4.4: Two operators at the ENIAC control panel (public domain image, source: US Army Photo http://ftp.arl.army.mil/ftp/historic-computers/gif/eniac7.gif).

Figure 4.5: Original caption notes that “replacing a bad [vacuum] tube meant checking among ENIACs 19,000 possibilities”; a daunting task! (public domain image, source: US Army Photo http://ftp.arl.army.mil/ftp/historic-computers/gif/eniac3.gif).
• \(22nnnn\) means \(A \leftarrow \text{MEM}_n\).
• \(30nnnn\) means \(A \leftarrow A + \text{MEM}_n\).
• \(31nnnn\) means \(A \leftarrow A - \text{MEM}_n\).
• \(32nnnn\) means \(A \leftarrow A \oplus \text{MEM}_n\).
• \(40nnnn\) means \(\text{PC} \leftarrow n\).
• \(41nnnn\) means \(\text{PC} \leftarrow n\) iff. \(A = 0\).
• \(42nnnn\) means \(\text{PC} \leftarrow n\) iff. \(A \neq 0\).

The entries on the left-hand side perhaps need some explanation. Take the entry for \(20nnnn\): this is a six digit decimal number where the left-most two digits are 2 and 0, and the right-most four can be any digits in the range \(0, 1, \ldots, 9\). On the right-hand side, we replace \(nnnn\) by a single number \(n\) so it will be in the range \(0, 1, \ldots, 9999\). For example, the decimal number 300005, viewed as an encoded instruction 300005, means \(A \leftarrow A + \text{MEM}_5\); we match 0005 on the left-hand side of the table, and turn it into 5 on the right-hand side. The 30 part is the opcode, and the 0005 part is the operand; the 30 part tells the computer what operation to perform, while the 0005 part tells it what to perform the operation on.

So if we store both instructions and data in memory, how do we know which instruction to execute next? Basically, we need an extra accumulator, which we call the Program Counter (PC), to keep track of where to fetch the instruction from: it holds a number just like \(A\) does, but the number in PC will be used to keep track of where the next instruction is in memory. With this in mind, we also need to slightly alter the way programs are executed:

1. Encode a program \(P\) onto paper tape; load this tape into memory using the tape reader, zero the program counter PC and start the computer.
2. From the address in PC, fetch the next instruction in the program and call it IR.
3. Increment PC so it points to the next instruction.
4. If \(IR = \bot\) (i.e., we have run out of instructions to execute), or if \(IR = \text{HALT}\) then halt the computer, otherwise execute IR (i.e., perform the operation it specifies).
5. Repeat from line 2.

Notice that we have included a new step to update the value of PC (by adding one to it). This is analogous to ensuring we fetch the instruction from the next row of the tape in our previous design. Now reconsider the example program we looked at before, and how it is encoded:

\[
\begin{align*}
A & \leftarrow \text{MEM}_4 \quad \leftarrow \quad 220004 \\
A & \leftarrow A + \text{MEM}_5 \quad \leftarrow \quad 300005 \\
\text{MEM}_6 & \leftarrow A \quad \leftarrow \quad 210006 \\
\text{HALT} & \quad \leftarrow \quad 100000
\end{align*}
\]

Execution using our new computer proceeds in more or less the same way, except that now it holds both data and instructions in MEM rather than having the instructions on a dedicated tape. As before, one can imagine drawing the state of the computer at each step in the execution; we do this in Figure 4.6 to Figure 4.8. Again, after step #13, the computer halts and we end up with the result of the addition, i.e., 30, stored in MEM_6.

Although reading and following executions of existing programs is a good start, writing your own programs is really the only way to get to grips with this topic.

See if you can write a program for the von Neumann computer that evaluates the expression

\[x \cdot (y + z)\]

where \(x, y\) and \(z\) are stored in memory at addresses of your choice; since the computer has no multiplication instruction, this demands some careful thought! Demonstrate that the program works by listing the steps (similar to above) for example \(x, y\) and \(z\).
CPU

CPU state = reset
PC = 0
IR = 
A = 0

(a) Step #1: Load the tape into memory, set PC = 0 and start the computer.

MEM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>220004</td>
<td>A ← MEM4</td>
</tr>
<tr>
<td>1</td>
<td>300005</td>
<td>A ← A + MEM5</td>
</tr>
<tr>
<td>2</td>
<td>210006</td>
<td>MEM6 ← A</td>
</tr>
<tr>
<td>3</td>
<td>100000</td>
<td>HALT</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>NOP</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>NOP</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>NOP</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>NOP</td>
</tr>
</tbody>
</table>

(b) Step #2: Fetch the next instruction IR = 220004 from PC = 0.

CPU

CPU state = fetch
PC = 1
IR = 220004
A = 0

MEM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>220004</td>
<td>A ← MEM4</td>
</tr>
<tr>
<td>1</td>
<td>300005</td>
<td>A ← A + MEM5</td>
</tr>
<tr>
<td>2</td>
<td>210006</td>
<td>MEM6 ← A</td>
</tr>
<tr>
<td>3</td>
<td>100000</td>
<td>HALT</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>NOP</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>NOP</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>NOP</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>NOP</td>
</tr>
</tbody>
</table>

(c) Step #3: Decode IR = 220004 into A ← MEM4, and set PC to PC + 1 = 1.

CPU

CPU state = execute
PC = 1
IR = 220004
A = 10

MEM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>220004</td>
<td>A ← MEM4</td>
</tr>
<tr>
<td>1</td>
<td>300005</td>
<td>A ← A + MEM5</td>
</tr>
<tr>
<td>2</td>
<td>210006</td>
<td>MEM6 ← A</td>
</tr>
<tr>
<td>3</td>
<td>100000</td>
<td>HALT</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>NOP</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>NOP</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>NOP</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>NOP</td>
</tr>
</tbody>
</table>

(d) Step #4: Execute the instruction A ← MEM4, i.e., set A to MEM4 = 10.

CPU

CPU state = fetch
PC = 1
IR = 300005
A = 10

MEM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>220004</td>
<td>A ← MEM4</td>
</tr>
<tr>
<td>1</td>
<td>300005</td>
<td>A ← A + MEM5</td>
</tr>
<tr>
<td>2</td>
<td>210006</td>
<td>MEM6 ← A</td>
</tr>
<tr>
<td>3</td>
<td>100000</td>
<td>HALT</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>NOP</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>NOP</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>NOP</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>NOP</td>
</tr>
</tbody>
</table>

(e) Step #5: Fetch the next instruction IR = 300005 from PC = 1.

CPU

CPU state = decode
PC = 1
IR = 300005
A = 10

MEM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>220004</td>
<td>A ← MEM4</td>
</tr>
<tr>
<td>1</td>
<td>300005</td>
<td>A ← A + MEM5</td>
</tr>
<tr>
<td>2</td>
<td>210006</td>
<td>MEM6 ← A</td>
</tr>
<tr>
<td>3</td>
<td>100000</td>
<td>HALT</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>NOP</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>NOP</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>NOP</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>NOP</td>
</tr>
</tbody>
</table>

(f) Step #6: Decode IR = 300005 into A ← A + MEM5, and set PC to PC + 1 = 2.

Figure 4.6: Computing the sum 10 + 20, as executed on a stored program computer.
CPU

state = execute
PC = 2
IR = 300005
A = 30

(a) Step #7: Execute the instruction \( A \leftarrow A + \text{MEM}_5 \), i.e., set \( A \) to \( A + \text{MEM}_5 = 30 \).

CPU

state = fetch
PC = 2
IR = 210006
A = 30

(b) Step #8: Fetch the next instruction \( IR = 210006 \) from \( PC = 2 \).

CPU

state = decode
PC = 3
IR = 210006
A = 30

(c) Step #9: Decode \( IR = 210006 \) into \( \text{MEM}_6 \leftarrow A \), and set \( PC \) to \( PC + 1 = 3 \).

CPU

state = execute
PC = 3
IR = 210006
A = 30

(d) Step #10: Execute the instruction \( \text{MEM}_6 \leftarrow A \), i.e., set \( \text{MEM}_6 \) to \( A = 30 \).

CPU

state = fetch
PC = 3
IR = 100000
A = 30

(e) Step #11: Fetch the next instruction \( IR = 100000 \) from \( PC = 3 \).

CPU

state = decode
PC = 4
IR = 100000
A = 30

(f) Step #12: Decode \( IR = 100000 \) into \( \text{HALT} \), and set \( PC \) to \( PC + 1 = 4 \).

Figure 4.7: Computing the sum \( 10 + 20 \), as executed on a stored program computer.
CPU state = execute
PC = 4
IR = 100000
= HALT
A = 30

MEM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>220004</td>
<td>A ← MEM₁</td>
</tr>
<tr>
<td>1</td>
<td>300005</td>
<td>A ← A + MEM₅</td>
</tr>
<tr>
<td>2</td>
<td>210006</td>
<td>MEM₆ ← A</td>
</tr>
<tr>
<td>3</td>
<td>100000</td>
<td>HALT</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>NOP</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>NOP</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
<td>NOP</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>NOP</td>
</tr>
</tbody>
</table>

(a) Step #13: Execute the instruction HALT, i.e., stop execution.

Figure 4.8: Computing the sum $10 + 20$, as executed on a stored program computer.

The current instruction set contains three rough classes of instruction: some perform load and store from and to memory, some perform arithmetic operations, and some perform updates to PC. Depending on what we use the computer for, other instructions might be useful of course: assuming you have a free choice, write the encoding and meaning for some other instructions you deem useful.

Why choose these instructions in particular? For instance, what might you use them for? There is a limit to how many new instructions we can add: can you explain one reason why?

### 4.2 Harvard versus von Neumann computers

With such a simple example program, it might seem as if the two styles of computer are more or less the same, i.e., neither has a massive advantage over the other. For example, they both compute $10 + 20 = 30$ at the end of the day. But, and this is a big but, there are (at least) two subtle and key differences between them.

#### 4.2.1 Beyond straight-line programs

With the Harvard-style design, we were forced to write instructions onto the tape and have them executed in the same order. There was, for example, no mechanism to “skip over” a given instruction or “jump” execution to an instruction based on a result we computed. Clearly this limits the sorts of program we could write. The good news is that the von Neumann-style upgrade removes this restriction by allowing PC to be altered by the program rather than just by the computer. By simply setting the value of PC we can direct execution to any instruction; we no longer have to write programs which are straight-line.

As a simple example, imagine we alter our example program by replacing the HALT instruction with one that alters the value of PC:

$$A ← MEM₁ \quad \mapsto \quad 220004$$
$$A ← A + MEM₅ \quad \mapsto \quad 300005$$
$$MEM₆ ← A \quad \mapsto \quad 210006$$
$$PC ← 0 \quad \mapsto \quad 400000$$

When we execute the last instruction, it sets the value of PC back to zero. Put more simply, it starts executing the program again right from the start. Writing down the steps of execution only, rather than drawing the state at each step, we now get:

**Step #1:** Load the tape into memory, set $PC = 0$ and start the computer.

**Step #2:** Fetch the next instruction $IR = 220004$ from $PC = 0$.

**Step #3:** Decode $IR = 220004$ into $A ← MEM₁$, and set $PC$ to $PC + 1 = 1$.

**Step #4:** Execute the instruction $A ← MEM₁$, i.e., set $A$ to $MEM₁ = 10$.

**Step #5:** Fetch the next instruction $IR = 300005$ from $PC = 1$.

**Step #6:** Decode $IR = 300005$ into $A ← A + MEM₅$, and set $PC$ to $PC + 1 = 2$.

**Step #7:** Execute the instruction $A ← A + MEM₅$, i.e., set $A$ to $A + MEM₅ = 30$. 

76
Figure 4.9: A moth found by operations of the Harvard Mark 2; the “bug” was trapped within the computer and caused it to malfunction (public domain image, source: http://en.wikipedia.org/wiki/File:H96566k.jpg).

Step #8: Fetch the next instruction $IR = 210006$ from $PC = 2$.

Step #9: Decode $IR = 210006$ into $MEM_6 ← A$, and set $PC$ to $PC + 1 = 3$.

Step #10: Execute the instruction $MEM_6 ← A$, i.e., set $MEM_6$ to $A = 30$.

Step #11: Fetch the next instruction $IR = 400000$ from $PC = 3$.

Step #12: Decode $IR = 400000$ into $PC ← 0$, and set $PC$ to $PC + 1 = 4$.

Step #13: Execute the instruction $PC ← 0$, i.e., set $PC$ to 0.

Step #14: Fetch the next instruction $IR = 220004$ from $PC = 0$.

Step #15: Decode $IR = 220004$ into $A ← MEM_4$ and set $PC$ to $PC + 1 = 1$.

Step #16: Execute the instruction $A ← MEM_4$, i.e., set $A$ to $MEM_4 = 10$.

Step #17: ...

Clearly the program never finishes in the sense that we never execute a HALT instruction. What we have constructed is an infinite loop; in this situation the computer will seem to “freeze” since it ends up executing the same instructions over and over again forever [13].

Although the occurrence of an infinite loop is normally deemed a problem, can you think of an example where it might be required?

Even so, you might argue it would be useful to detect infinite loops so the computer can be manually halted. How might you approach doing so? For example, what changes to the computer might be necessary? Is your approach always guaranteed to work?

4.2.2 Toward self-modifying programs

With the Harvard-style design, we were prevented from altering the tape content after we started execution. Put another way, programming happens strictly before program execution. However, with the von Neumann-style design instructions and data are the same thing so we can alter instructions during execution just as easily as data.
What does this mean in practise? Imagine we take the example program but introduce a bug [22].

A bug is a programming error: it is a mistake in a program that causes it to malfunction, behaving in a different way than we might have intended. The term bug (and debug [8], that relates to fixing the mistake) have well-discussed histories; a nice story relates them to a real bug (a moth) that famously short-circuited the Harvard Mark 2 computer! The bug here is that instead of storing the addition result in MEM₆, we mistakenly store it in MEM₃:

\[
\begin{align*}
A & \leftarrow MEM₄ & \mapsto & 220004 \\
A & \leftarrow A + MEM₅ & \mapsto & 300005 \\
MEM₃ & \leftarrow A & \mapsto & 210003 \\
HALT & & \mapsto & 100000
\end{align*}
\]

Again writing down the steps of execution only, we obtain:

**Step #1**: Load the tape into memory, set PC = 0 and start the computer.

**Step #2**: Fetch the next instruction IR = 220004 from PC = 0.

**Step #3**: Decode IR = 220004 into A ← MEM₄, and set PC to PC + 1 = 1.

**Step #4**: Execute the instruction A ← MEM₄, i.e., set A to MEM₄ = 10.

**Step #5**: Fetch the next instruction IR = 300005 from PC = 1.

**Step #6**: Decode IR = 300005 into A ← A + MEM₅, and set PC to PC + 1 = 2.

**Step #7**: Execute the instruction A ← A + MEM₅, i.e., set A to A + MEM₅ = 30.

**Step #8**: Fetch the next instruction IR = 210003 from PC = 2.

**Step #9**: Decode IR = 210003 into MEM₃ ← A, and set PC to PC + 1 = 3.

**Step #10**: Execute the instruction MEM₃ ← A, i.e., set MEM₃ to A = 30.

**Step #11**: Fetch the next instruction IR = 000030 from PC = 3.

**Step #12**: Decode IR = 000030 into NOP, and set PC to PC + 1 = 4.

**Step #13**: Execute the instruction NOP, i.e., do nothing.

**Step #14**: Fetch the next instruction IR = 000010 from PC = 4.

**Step #15**: Decode IR = 000010 into NOP, and set PC to PC + 1 = 5.

**Step #16**: Execute the instruction NOP, i.e., do nothing.

**Step #17**: ...

Step #11 is now different: something weird has happened because initially we had a HALT instruction in MEM₃, but when we come to fetch it, it has changed! The reason is obvious if we look a few steps back. In step #10 we execute the instruction which should store the result of our addition. But remember the bug: instead of storing the result in MEM₆, we store the result, i.e., 30, in MEM₃. Of course, MEM₃ is part of the program, so as things progress we end up trying to execute the number 30 we previously stored. This behaviour is allowed because data and instructions are the same thing now. So we fetch the data value 30 and try to interpret it as an instruction according to our encoding.

To cut a long story short, the program has modified itself [21] by altering instructions relating to the same program as is being executed! In this case, the self-modification can only be a bad thing since without a HALT instruction the computer will never stop executing the program. More than likely it will “crash” somehow [6]. More generally however, the idea is that in other cases we can put self-modification to constructive use, i.e., have a program change instructions so that something useful happens as a result.
4.3 A self-modifying virus

Finally we can talk about our virus which, you will remember, aims to change itself to avoid detection by the virus scanner. Imagine our example computer includes a single instruction that, when executed, causes something bad to happen; maybe the computer self-destructs or something. For the sake of argument, we will say that whenever the computer executes the extra instruction 111111 this represents the payload of our virus. In other words, in our encoding 111111 means payload, and when payload is executed the virus has won.

It is not too hard to imagine that when we infect a file called F, the virus payload will then appear somewhere within it. If this is the case, the virus scanner can easily detect the virus by using the single element signature \( S = \langle 111111 \rangle \) to scan F as we discussed at the beginning of the Chapter:

\[
\begin{align*}
F &= \langle \ldots, \ldots, \ldots, \ldots, 111111, \ldots \rangle \\
S &= \langle 111111 \rangle \\
 &\quad \iff \text{no match} \\
 &\quad \iff \text{no match} \\
 &\quad \iff \text{no match} \\
 &\quad \iff \text{match}
\end{align*}
\]

With this in mind, the virus writer has to somehow hide the virus payload so the virus scanner cannot detect it. To do this, we will use the properties of the XOR (short for “exclusive-or”) function [9] we met in Chapter 2.

4.3.1 Using XOR to mask numbers

XOR can be used to mask (or hide) the value of a number. Imagine we have a bit \( x \) and we do not want anyone to know it; select a bit \( k \) and compute \( y = x \oplus k \). If we give \( y \) to someone, can they tell us what \( x \) is?

Well, if \( y = 0 \) then that could have been produced by us having \( x = 0 \) and selecting \( k = 0 \) or having \( x = 1 \) and selecting \( k = 1 \) because

\[
y = 0 = x \oplus k = \begin{cases} 0 \oplus 0 & \text{if } x = 0 \text{ and } k = 0 \\ 1 \oplus 1 & \text{if } x = 1 \text{ and } k = 1 \end{cases}
\]

Similarly, if \( y = 1 \) then that could have been produced by us having \( x = 0 \) and selecting \( k = 1 \) or having \( x = 1 \) and selecting \( k = 0 \) because

\[
y = 1 = x \oplus k = \begin{cases} 0 \oplus 1 & \text{if } x = 0 \text{ and } k = 1 \\ 1 \oplus 0 & \text{if } x = 1 \text{ and } k = 0 \end{cases}
\]

The point is, if we give someone \( y \) they can not tell us \( x \) for sure unless they also know \( k \). Since we know \( k \), we can easily tell what \( x \) is since \( y \oplus k = x \oplus k \oplus k = x \). You can think of this as a limited form of encryption, except we call \( k \) the mask (rather than the key) and say \( x \) has been masked by \( k \) to produce \( y \).

The problem is that XOR is a Boolean function: the inputs and output have to be either 0 or 1. To apply it to the decimal numbers in \( MEM \), we first apply what we learnt in Chapter 2 to convert between decimal and binary. For example

\[
310000_{(10)} = 010010111011110000_{(2)} \\
329975_{(10)} = 01010000100011110111_{(2)}
\]

Now we can compute

\[
010010111011110000_{(2)} \oplus 01010000100011110111_{(2)} = 000110110000001111_{(2)} = 111111_{(10)}
\]

simply by applying XOR to the corresponding bits. Likewise, since

\[
111111_{(10)} = 000110110000001111_{(2)} \\
329975_{(10)} = 01010000100011110111_{(2)}
\]

we can compute

\[
000110110000001111_{(2)} \oplus 01010000100011110111_{(2)} = 010010111011110000_{(2)} = 310000_{(10)}
\]

More simply, in terms of our example above we start with \( x = 310000 \) and select \( k = 329975 \). Then, if we compute \( x \oplus k = 310000 \oplus 329975 \) we get \( y = 111111 \). We know \( k \), so if we compute \( y \oplus k = 111111 \oplus 329975 \) then we get back \( x = 310000 \).
4.3.2 A virus that masks the payload

Now for the clever part: notice that

- 310000 corresponds to the encoded instruction 310000 which means \( A \leftarrow A - \text{MEM}_0 \),
- 329975 corresponds to the encoded instruction 329975 which means \( A \leftarrow A \oplus \text{MEM}_{9975} \), and
- 111111 corresponds to the encoded instruction 111111 which means payload.

So basically

\[
111111 = 310000 \oplus 329975
\]

means we can build the payload instruction by simply applying \( \oplus \) to two innocent looking instructions. Our strategy for hiding the virus payload should now be obvious: we mask the payload instruction within \( F \) which we load into memory and start executing. While the program is executing, we unmask the payload instruction using self-modification, and then execute it. There is a fancy name for viruses like this: they are called polymorphic [18]. The first time a virus was seen using the technique was around 1990 when a virus called 1260 (referring to the length) was discovered.

Confused? Understanding why it works might be a puzzle, but either way the end result is the following program and the corresponding encoding:

\[
\begin{align*}
A & \leftarrow \text{MEM}_3 \quad \mapsto \quad 220003 \\
A & \leftarrow A \oplus \text{MEM}_5 \quad \mapsto \quad 320005 \\
\text{MEM}_3 & \leftarrow A \quad \mapsto \quad 210003 \\
A & \leftarrow A - \text{MEM}_0 \quad \mapsto \quad 310000 \\
\text{HALT} & \quad \mapsto \quad 100000 \\
A & \leftarrow A \oplus \text{MEM}_{9975} \quad \mapsto \quad 329975
\end{align*}
\]

Just by eye-balling the encoded program we can see that the virus payload does not appear, so if we use the virus scanner on it we fail to detect a virus:

\[
\begin{array}{c}
F = \langle \ldots, 220003, 320005, 210003, 310000, 100000, 329975, \ldots \rangle \\
S = \langle 111111, 111111, 111111, 111111, 111111, \ldots \rangle \\
\end{array}
\]

\[
\begin{array}{c}
\quad \mapsto \text{no match} \\
\quad \mapsto \text{no match} \\
\quad \mapsto \text{no match} \\
\quad \mapsto \text{no match} \\
\quad \mapsto \text{no match}
\end{array}
\]

But so what? If the program does not contain the virus payload, nothing bad can happen right?! Unfortunately not. Have a look at what happens when we execute the program:

**Step #1:** Load the tape into memory, set \( PC = 0 \) and start the computer.

**Step #2:** Fetch the next instruction \( IR = 220003 \) from \( PC = 0 \).

**Step #3:** Decode \( IR = 220003 \) into \( A \leftarrow \text{MEM}_3 \), and set \( PC \) to \( PC + 1 = 1 \).

**Step #4:** Execute the instruction \( A \leftarrow \text{MEM}_3 \), i.e., set \( A \) to \( \text{MEM}_3 = 310000 \).

**Step #5:** Fetch the next instruction \( IR = 320005 \) from \( PC = 1 \).

**Step #6:** Decode \( IR = 320005 \) into \( A \leftarrow A \oplus \text{MEM}_5 \), and set \( PC \) to \( PC + 1 = 2 \).

**Step #7:** Execute the instruction \( A \leftarrow A \oplus \text{MEM}_5 \), i.e., set \( A \) to \( A \oplus \text{MEM}_5 = 111111 \).

**Step #8:** Fetch the next instruction \( IR = 210003 \) from \( PC = 2 \).

**Step #9:** Decode \( IR = 210003 \) into \( \text{MEM}_3 \leftarrow A \), and set \( PC \) to \( PC + 1 = 3 \).

**Step #10:** Execute the instruction \( \text{MEM}_3 \leftarrow A \), i.e., set \( \text{MEM}_3 \) to \( A = 111111 \).

**Step #11:** Fetch the next instruction \( IR = 111111 \) from \( PC = 3 \).

**Step #12:** Decode \( IR = 111111 \) into payload, and set \( PC \) to \( PC + 1 = 4 \).
Step #13: Execute the instruction **payload**, i.e., do something bad.

Step #14: ...

Step #13 executes **payload** somehow, so the virus payload has been triggered even though it did not appear in \( F \). Look at what happens step-by-step:

- Step #4 loads the masked virus payload, which looks like a \( A \leftarrow A - MEM_0 \) instruction, from \( MEM_3 \) and into \( A \).
- Step #7 unmaps it using an XOR instruction that applies the mask value \( k \) stored in \( MEM_5 \) just like we saw above.
- At this point we have \( A = 111111 \) which is then stored back into \( MEM_3 \) by step #10.
- Step #13 executes the unmasked instruction from \( MEM_3 \), at which point the payload is triggered and the virus wins.

---

The current instruction set includes instructions to load from and store into fixed addresses. For instance \( MEM_n \leftarrow A \) stores \( A \) at the address \( n \) which is fixed when we write the program. For computers such as ENIAC, self-modifying programs were often used in a positive way to extend this ability to variable addresses; put simply, this allows access to \( MEM_i \) for some \( i \) chosen during execution.

Imagine a sequence of numbers \( X = \langle X_0, X_1, \ldots, X_{m-1} \rangle \) is stored at a known address in memory, and we want to compute the sum

\[
\sum_{i=0}^{m} X_i
\]

If the sequence has 10 elements and starts at address 100, say, one way to do this would be

\[
\begin{align*}
A & \leftarrow MEM_{100+0} \\
A & \leftarrow A + MEM_{100+1} \\
& \vdots \\
A & \leftarrow A + MEM_{100+9}
\end{align*}
\]

where clearly 100 + 1 = 101 is fixed, so we can write an appropriate instruction.

But what if \( m \) is really large, or only known once the program starts to execute? In these cases, we might prefer to write a loop: based on an \( m \) also stored in memory somewhere (at address 99 say), use the concept of self-modifying programs to compute the same result by looping through each \( i \) (like the loop in Chapter 3) and hence accumulating each \( X_i \).

---

4.3.3 Preventing the virus without a virus scanner

The obvious question is, if the virus scanner we have does not work in this case then what *can* we do to stop the virus? In reality, this is quite an open question. On real computers we still do not have a definitive, general solution against viruses and malware. However, in the particular case of our virus we can think of a few possibilities:

- We could monitor the program during execution and prevent the virus payload being executed. So, for example, execution of every instruction would have to include an extra step saying “if \( IR \) is the virus payload then stop execution”. But of course in reality there is not just one instruction that does something bad: more usually a subtle combination of many instructions will represent the payload, so this approach would not work. It also has the drawback of reducing the speed at which we can execute all programs, which might be quite unpopular.

- We could try to be more clever at coming up with the signature that identifies this sort of virus. We cannot use the masked payload 310000 because that is a valid instruction which basically any program might include! We cannot use the mask 329975 because this could be anything: the virus is free to choose any mask it wants (although obviously this alters the masked payload). One option could be
to try and identify the pattern of self-modifying code. For example, if the virus scanner knew the
meaning of instructions it could perhaps analyse the program and work out what is really going on.
Clearly this means the virus scanner needs to be much more sophisticated and depends on the fact
that writing similar code in a different way is not possible.

- We could alter the computer so that writing self-modifying programs is impossible. Modern computers
include a scheme which roughly works by adding an extra “tag” to each element of MEM. If the tag
is 0, this indicates that the element can be written to but not executed as an instruction; if the flag is 1,
this means the element can be executed as an instruction but not written to. In terms of our example
virus, this would mean the memory looks like:

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>220003</td>
<td>$A \leftarrow MEM_3$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>320005</td>
<td>$A \leftarrow A \oplus MEM_5$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>210003</td>
<td>$MEM_3 \leftarrow A$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>310000</td>
<td>$A \leftarrow A - MEM_0$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>100000</td>
<td>HALT</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>329975</td>
<td>$A \leftarrow A \oplus MEM_{975}$</td>
</tr>
</tbody>
</table>

So basically, we could not execute the virus because $MEM_3$ could not be written to once the program
was loaded. Put another way, the instruction $MEM_3 \leftarrow A$ would cause the computer to halt due to a
violation of the rules. The drawback is that there are reasonable uses for self-modifying programs; by
implementing this solution we prevent those from being executed as well. Plus, we have to make our
memory, and the mechanism that accesses it, more complicated and therefore more costly.
BIBLIOGRAPHY


Recall that in Chapter 4 we described how computers only understand a fixed set of instructions that can operate only on fixed types of operand; at the lowest level, instructions typically only operate on operands which are numbers. So, for example, most computers understand how to add together numbers: in this case we would say that a number is a native type of data. However, it is not common for a computer to understand types of data more complicated than this. For example a computer does not have a native understanding of emails; there is no instruction that tells it to “search for the text X in the email Y”. In order to write a program that deals with non-native types of data like this, we need to decide on a representation for it using a data structure.

The design of data structures and the algorithms that perform operations on them is a fundamental subject that underpins almost every aspect of Computer Science. Essentially, a data structure allows us to look at numbers and interpret them as something else. Sometimes the data structure is simply a description of how numbers should be arranged so we can interpret them correctly; sometimes we add extra information or meta-data which allows us to inspect and alter the data structure. Interested in computer graphics? You need data structures (e.g., representations of images, or points in 3D space) and algorithms to operate on them (e.g., change the colour of an image, or rotate a 3D scene). Interested in computer networks? You need data structures (e.g., representations of connectivity between computers on the network) and algorithms to operate on them (e.g., find a path from computer X to computer Y). Interested in computer security? You need data structures (e.g., representations of very large numbers) and algorithms to operate on them (e.g., add together X and Y modulo Z). In all cases, efficiency is key: if we use a more efficient data structure, we might reduce the time taken to perform a given operation, or the amount of memory required to store the data.

Our focus here is on strings. The term string crops up in a lot of places, but generically means a sequence of things. For example, we string together a sequence of words to make a sentence, protein is made from a string of many amino acids and so on. In Computer Science, a string is a sequence of characters and one of the most fundamental data structures after the native types of data which a computer can operate on. If you think about it, an email is just a string of characters, the files created by a word processor could be thought of as similar strings, the messages a computer displays so it can communicate with the user are also strings, and so on. The aim is to show that even with something that seems so simple, there are many options for useful data structures and that our choice has a major impact on the algorithms used to manipulate them.

5.1 String data structures

As we saw in Chapter 4, the memory of a computer is where it stores data in the long term. The computer loads the data into an accumulator to operate on it, and stores the result back in memory afterwards. We modelled memory using a sequence called MEM and stored decimal numbers in each element. Since MEM is large, it is convenient to avoid writing out the whole sequence. To allow this, we will write the addresses
of elements above the elements themselves. For example, imagine we have

\[ i = \ldots, 3, 4, 5, 6, 7, \ldots \]
\[ MEM = \langle \ldots, 104, 101, 108, 108, 111, \ldots \rangle \]

If \( MEM \) has \( n \) elements in total, we have missed out the elements at addresses 0…2 and those at addresses 8…\( n - 1 \). We are only interested in addresses #3, #4, #5, #6 and #7 which have the values 104, 101, 108 and 111. Our goal is to represent strings within \( MEM \) but to do this, we need to solve two problems: first how to represent characters, and second how to represent strings.

### 5.1.1 Problem #1: representing characters

The first problem is that \( MEM \) can only store numbers, and the computer it exists within can only really process numbers. To allow it to deal with characters, we need some way to represent them numerically. Basically we just need to translate from one to the other. More specifically, we need two functions: \( O_{\text{ASCII}}(x) \) which takes a character \( x \) and gives us back the corresponding numerical representation, and \( C_{\text{ASCII}}(y) \) which takes a numerical representation \( y \) and gives back the corresponding character. But how should the functions work? Fortunately, people have thought about this problem for us and provided standards we can use. One of the oldest is the American Standard Code for Information Interchange (ASCII) [1], pronounced “ass key”.

ASCII has a rich history, but was developed to permit communication between early teleprinter devices. These were like a combination of a typewriter and a telephone, and were able to communicate text to each other before innovations such as the fax machine. Later, but long before monitors and graphics cards existed, similar devices allowed users to send input to early computers and receive output from them. Figure 5.2 shows the 128-entry ASCII table which tells us how characters are represented as numbers. Of the entries, 95 are printable characters we can instantly recognise (including \( \text{SPC} \) which is short for “space”). There are also 33 others which represent non-printable control characters: originally, these would have been used to control the teleprinter rather than to have it print something. For example, the \( \text{CR} \) and \( \text{LF} \) characters (short for “carriage return” and “line feed”) would combine to move the print head onto the next line; we still use these characters to mark the end of lines in text files. Other control characters also play a role in modern computers. For example, the \( \text{BEL} \) (short for “bell”) characters play a “ding” sound when printed to most UNIX terminals, we have keyboards with keys that relate to \( \text{DEL} \) and \( \text{ESC} \) (short for “delete” and “escape”) and so on.
<table>
<thead>
<tr>
<th><em>y</em></th>
<th><em>Chr(y)</em></th>
<th><em>y</em></th>
<th><em>Chr(y)</em></th>
<th><em>y</em></th>
<th><em>Chr(y)</em></th>
<th><em>y</em></th>
<th><em>Chr(y)</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord(x)</td>
<td></td>
<td>Ord(x)</td>
<td></td>
<td>Ord(x)</td>
<td></td>
<td>Ord(x)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td><em>NULL</em></td>
<td>1</td>
<td><em>SOH</em></td>
<td>2</td>
<td><em>STX</em></td>
<td>3</td>
<td><em>ETX</em></td>
</tr>
<tr>
<td>4</td>
<td><em>EOT</em></td>
<td>5</td>
<td><em>ENQ</em></td>
<td>6</td>
<td><em>ACK</em></td>
<td>7</td>
<td><em>BEL</em></td>
</tr>
<tr>
<td>8</td>
<td><em>BS</em></td>
<td>9</td>
<td><em>HT</em></td>
<td>10</td>
<td><em>LF</em></td>
<td>11</td>
<td><em>VT</em></td>
</tr>
<tr>
<td>12</td>
<td><em>FF</em></td>
<td>13</td>
<td><em>CR</em></td>
<td>14</td>
<td><em>SO</em></td>
<td>15</td>
<td><em>SI</em></td>
</tr>
<tr>
<td>16</td>
<td><em>DLE</em></td>
<td>17</td>
<td><em>DC1</em></td>
<td>18</td>
<td><em>DC2</em></td>
<td>19</td>
<td><em>DC3</em></td>
</tr>
<tr>
<td>20</td>
<td><em>DC4</em></td>
<td>21</td>
<td><em>NAK</em></td>
<td>22</td>
<td><em>SYN</em></td>
<td>23</td>
<td><em>ETB</em></td>
</tr>
<tr>
<td>24</td>
<td><em>CAN</em></td>
<td>25</td>
<td><em>EM</em></td>
<td>26</td>
<td><em>SUB</em></td>
<td>27</td>
<td><em>ESC</em></td>
</tr>
<tr>
<td>28</td>
<td><em>FS</em></td>
<td>29</td>
<td><em>GS</em></td>
<td>30</td>
<td><em>RS</em></td>
<td>31</td>
<td><em>US</em></td>
</tr>
<tr>
<td>32</td>
<td><em>SPC</em></td>
<td>33</td>
<td>!</td>
<td>34</td>
<td>&quot;</td>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>§</td>
<td>37</td>
<td>%</td>
<td>38</td>
<td>&amp;</td>
<td>39</td>
<td>'</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
<td>41</td>
<td>)</td>
<td>42</td>
<td>&quot;</td>
<td>43</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
<td>45</td>
<td>-</td>
<td>46</td>
<td>.</td>
<td>47</td>
<td>/</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
<td>49</td>
<td>1</td>
<td>50</td>
<td>2</td>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
<td>53</td>
<td>5</td>
<td>54</td>
<td>6</td>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>57</td>
<td>9</td>
<td>58</td>
<td>:)</td>
<td>59</td>
<td>;</td>
</tr>
<tr>
<td>60</td>
<td>&lt;</td>
<td>61</td>
<td>=</td>
<td>62</td>
<td>&gt;</td>
<td>63</td>
<td>?</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
<td>65</td>
<td>A</td>
<td>66</td>
<td>B</td>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
<td>69</td>
<td>E</td>
<td>70</td>
<td>F</td>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>H</td>
<td>73</td>
<td>I</td>
<td>74</td>
<td>J</td>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>76</td>
<td>L</td>
<td>77</td>
<td>M</td>
<td>78</td>
<td>N</td>
<td>79</td>
<td>0</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
<td>81</td>
<td>Q</td>
<td>82</td>
<td>R</td>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
<td>85</td>
<td>U</td>
<td>86</td>
<td>V</td>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>88</td>
<td>X</td>
<td>89</td>
<td>Y</td>
<td>90</td>
<td>Z</td>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>92</td>
<td>`</td>
<td>93</td>
<td>]</td>
<td>94</td>
<td>^</td>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td>96</td>
<td>a</td>
<td>97</td>
<td>b</td>
<td>98</td>
<td>c</td>
<td>99</td>
<td>d</td>
</tr>
<tr>
<td>100</td>
<td>D</td>
<td>101</td>
<td>E</td>
<td>102</td>
<td>f</td>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>104</td>
<td>h</td>
<td>105</td>
<td>i</td>
<td>106</td>
<td>j</td>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>108</td>
<td>l</td>
<td>109</td>
<td>m</td>
<td>110</td>
<td>n</td>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>112</td>
<td>p</td>
<td>113</td>
<td>q</td>
<td>114</td>
<td>r</td>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
<td>117</td>
<td>u</td>
<td>118</td>
<td>v</td>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>120</td>
<td>x</td>
<td>121</td>
<td>y</td>
<td>122</td>
<td>z</td>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>124</td>
<td></td>
<td></td>
<td>125</td>
<td>}</td>
<td>126</td>
<td>&quot;</td>
<td>127</td>
</tr>
</tbody>
</table>

**Figure 5.2:** A table describing the printable ASCII character set.
You can test this out by issuing the command

```
echo -e '\x07'
```

in a BASH terminal. It asks echo to display the ASCII character $07_{(16)} = 7_{(10)} = \text{BEL}$ on the terminal, which should mean a sound is produced (or perhaps a visual indicator if the sound is disabled). The -e option is quite important, because it means that \x07 is interpreted as an ASCII code rather than a normal string. Try this with some other examples. For instance, what would you expect

```
echo -e 'hello\x08world'
```

to produce?

Since there are 128 entries in the table, ASCII characters can be and are represented by bytes as 8-bit numbers. However, notice that $2^7 = 128$ and $2^8 = 256$ so in fact we could represent 256 characters: essentially one of the bits is not used by the ASCII encoding. Specific computer systems sometimes use the unused bit to permit use of an “extended” ASCII table with 256 entries; the extra characters in this table can be used for some special purpose. For example, foreign language characters are often defined in this range (e.g., é or ø). However, the original use of the unused bit was as an error detection mechanism: more specifically, it represents the parity bit from the even parity code in Chapter 2.

Another use for extended ASCII characters is ASCII art [2], popular in an era when plain text was the only viable display medium. Examples archived at

http://www.textfiles.com/

were often used by Bulletin Board Systems (BBSs) [4], which used the telephone system and MODEMs to form an early form of computer network: text was used almost exclusively to compensate for the relatively slow speed of data transfer. An excellent documentary at

http://www.bbsdocumentary.com/

chronicles the technology and personalities involved.

There are now various ASCII art editors online, such as

http://www.asciigraffiti.com/

Forget GIMP, which we used in Chapter 2, or other modern image manipulation software: take a trip into the past, and see how hard it is to draw images using text!

Given the table, we can see, for example, that \texttt{CHR}(104) = 'h', i.e., if we see the number 104 then this represents the character 'h'. Conversely we have that \texttt{ORD}('h') = 104. Although in a sense any consistent translation between characters and numbers like this would do, ASCII has some useful properties. Look specifically at the alphabetic characters:

- Imagine we want to test if one character $x$ is alphabetically before some other $y$. The way the ASCII translation is specified, we can simply compare their numeric representation. If we find \texttt{ORD}(x) < \texttt{ORD}(y) then the character $x$ is before the character $y$ in the alphabet. For example 'a' is before 'c' because

\[
\text{ORD('a')} = 97 < 99 = \text{ORD('c')}.\]

- Imagine we want to convert a character $x$ from lower-case into upper-case. The lower-case characters are represented numerically as the contiguous range 97...122; the upper-case characters as the contiguous range 65...90. So we can covert from lower-case into upper-case simply by subtracting 32. For example

\[
\text{CHR(ORD('a') - 32)} = 'A'.\]

Armed with the ASCII table and the new translation method, we can read new meaning into the contents of MEM. As well as writing the addresses of elements above the elements, we can write the ASCII translation of each element (i.e., the character that each numeric element represents) below this, on an extra line. For
example:

\[
\begin{align*}
  i &= \ldots, 3, 4, 5, 6, 7, \\
  MEM &= \langle \ldots, 104, 101, 108, 108, 111, \ldots \rangle \\
  \text{Chr}(MEM_i) &= \ldots, 'h', 'e', 'l', 'l', 'o', \ldots
\end{align*}
\]

We can now see that addresses 3...7 hold the string “hello” if we interpret the memory content as ASCII characters. Just to show that this really is how things work, consider this short experiment using BASH:

```
bash$ cat > A.txt
hello
bash$ cat A.txt | od -Ad -tu1 | cut -c 9-
104 101 108 108 111 10 ...
bash$
```

First we create a file called A.txt by typing the characters ‘h’, ‘e’, ‘l’, ‘l’ and ‘o’ followed by return (which you cannot see). The second use of cat prints the file to the terminal as ASCII characters as one might expect. However, using the od command we can inspect the numerical representation. In particular using the option -t with format u1 instructs od to format the content as a sequence of unsigned decimal integers in the range 0...255. As a result we see a total of six numbers, namely 104, 101, 108, 108 and 111 representing characters ‘h’, ‘e’, ‘l’, ‘l’ and ‘o’ as in MEM, and the number 10 which represents the character LF (which occurs as a result of our pressing return after “hello” when creating A.txt).

5.1.2 Problem #2: representing strings

The second problem is that so far we do not really have a way to know how long a string is, i.e., how many characters it contains. What we have been doing is looking at the memory content

\[
\begin{align*}
  i &= \ldots, 3, 4, 5, 6, 7, \\
  MEM &= \langle \ldots, 104, 101, 108, 108, 111, \ldots \rangle \\
  \text{Chr}(MEM_i) &= \ldots, 'h', 'e', 'l', 'l', 'o', \ldots
\end{align*}
\]

then seeing the characters ‘h’, ‘e’, ‘l’, ‘l’ and ‘o’ and assuming that they are the string “hello”. But how do we know the length of the string starting at address #3? It could be possible that addresses higher than #7 hold more characters so that we actually have a longer string than we thought. For example, maybe MEM₈ = 46 which means there is a full stop character there:

\[
\begin{align*}
  i &= \ldots, 3, 4, 5, 6, 7, 8, \\
  MEM &= \langle \ldots, 104, 101, 108, 108, 111, 46, \ldots \rangle \\
  \text{Chr}(MEM_i) &= \ldots, 'h', 'e', 'l', 'l', 'o', ' ', \ldots
\end{align*}
\]

How do we know if the full stop is actually part of the string starting at address #3, or if it is just some unrelated value which is there by chance? That is, how do we know we really mean the five character string “hello” rather than the six character string “hello.”?

Given some address x which tells us where in memory a string starts, our goal is to add structure to the data so we know the length (and therefore what the intended content is). One way is to form a data structure by embedding some extra information in MEM to tell us how long the string is; there are at least two schemes we could consider:

1. We could embed the string length, which is a number, as the first element:

\[
\begin{align*}
  i &= \ldots, 3, 4, 5, 6, 7, 8, \\
  MEM &= \langle \ldots, 5, 104, 101, 108, 108, 111, \ldots \rangle \\
  \text{Chr}(MEM_i) &= \ldots, ENQ, 'h', 'e', 'l', 'l', 'o', \ldots
\end{align*}
\]

We still have a string starting at address #3 but instead of interpreting the first element as a character, we interpret it as the length of the string. Since MEM₃ = 5 we know this string has five characters in it, and that they will be in addresses 4...8.

2. We could embed a character, which will mark the end of the string, as the last element:

\[
\begin{align*}
  i &= \ldots, 3, 4, 5, 6, 7, 8, \\
  MEM &= \langle \ldots, 104, 101, 108, 108, 111, 0, \ldots \rangle \\
  \text{Chr}(MEM_i) &= \ldots, 'h', 'e', 'l', 'l', 'o', 'NUL', \ldots
\end{align*}
\]
Again we have a string starting at address #3, but now we can be sure where it ends because \( \text{MEM}_8 = 0 \). This is not a character as such, but the end of string marker or terminator which enables us to determine that the length is \( 8 - 3 = 5 \) characters.

The reason these two schemes are interesting is because they are adopted by the Pascal [6] and C [5] programming languages respectively; because they do not really have good names\(^1\) we will refer to them as the P-string and C-string schemes. No, there is no G-string or equivalent before you ask.

Neither the P-string or C-string scheme is right or wrong, but do give particular advantages and disadvantages in each case:

**P-string:**
- The string is represented using one more element than the content suggests; if there are \( n \) characters, we need \( n + 1 \) elements because one is used to store the length.
- Since the length needs to fit into one element of \( \text{MEM} \), there is an upper limit on the lengths of string we can represent. We can remove this restriction by using more elements, but things become more tricky and we use more space.
- We have direct access to the length of the string, since we simply have to load it from the first element.

**C-string:**
- The string is represented using one more element than the content suggests; if there are \( n \) characters, we need \( n + 1 \) elements because one is used to store the NUL terminator.
- The actual string content cannot contain a NUL character, otherwise we would interpret this as the terminator, i.e., the end of the string. Although NUL is not printable, the fact we cannot use it can be a disadvantage in some circumstances.
- There is no real restriction on the string length, but we do not have direct access to that length. To find the length of a string, we need to search for the NUL terminator.

---

Research (task #19)

These are just two data structures that could be used to represent strings. Still more exist: can you think of at least one more possibility? Explain how this third data structure works using an example, and contrast it with the C-string and P-string approaches by constructing a list of advantages and disadvantages.

### 5.2 String algorithms

The idea of introducing these two data structures is to demonstrate some more algorithms; unlike the algorithms in Chapter 3 that computed an arithmetic result, these ones operate on and manipulate instances of the string data structures. The real goal is to point out something subtle about the algorithms and data structures involved, but it will take a while to get there.

Keep in mind that we can reuse the ideas we developed in Chapter 3 to compare algorithms against each other. In this case, a good definition of the problem size \( n \) is the length of the strings are dealing with.

#### 5.2.1 strlen: finding the length of a string

About the most simple task one might think of in the context of strings is computing their length. Given a string starting at address \( x \), the idea is to return the number of characters in that string. The C programming language includes a standard function called `strlen` which represents an implementation of this idea.

**5.2.1.1 P-string version**

The P-string version is shown in Figure 5.3a. Imagine the memory content is initially

\[
\begin{array}{c}
i = \ldots, 3, 4, 5, 6, 7, 8, \ldots \\
\text{MEM} = \langle \ldots, 5, 104, 101, 108, 108, 111, \ldots \rangle \\
\text{CHR}(\text{MEM}_i) = \langle \ldots, \text{ENQ}, 'h', 'e', 'l', 'l', 'o', \ldots \rangle 
\end{array}
\]

\(^1\)The second option is sometimes termed ASCIIZ which sort of reads as “zero-terminated ASCII”.

---

[90]
algorithm P-STRING-LENGTH(x) begin
  i ← 0
  while MEM_{i+1} ≠ 0 do
    i ← i + 1
  end
  return i
end

algorithm C-STRING-LENGTH(x) begin
  i ← 0
  while MEM_{i+1} ≠ 0 do
    i ← i + 1
  end
  return i
end

(a) P-string version.

(b) C-string version.

Figure 5.3: Algorithms to compute the length of a string at address x, represented using a P-string (left-hand side) or C-string (right-hand side) data structure.

and we invoke the algorithm using P-STRING-LENGTH(3). The only input is the string address, so in this case we want to find the length of a string at address 3; the corresponding steps performed by the algorithm are as follows:

Step #1 Return MEM_5 = 5.

This could not be simpler! Remember the string length is embedded as the first element, so the algorithm computes the required length in one step by simply loading it. Because of this, the algorithm always takes just one step; describing it using big-O notation we say it is $O(1)$ because no matter what the input size is, it takes a constant number of steps to give us a result.

5.2.1.2 C-string version

The C-string version is shown in Figure 5.3b. Imagine the memory content is initially

\[
\begin{align*}
i &= \ldots, 3, 4, 5, 6, 7, 8, \ldots \\
MEM &= \langle \ldots, 104, 101, 108, 108, 111, 0, \ldots \rangle \\
\text{Chr}(MEM_i) &= \ldots, 'h', 'e', 'Y', 'I', 'o', \text{NUL}, \ldots
\end{align*}
\]

which is just the C-string version of the P-string one we used above. If we invoke the algorithm with C-STRING-LENGTH(3) in a similar way, the corresponding steps performed by the algorithm are now:

Step #1 Assign $i \leftarrow 0$.

Step #2 Since $MEM_{3+0} \neq 0$, assign $i \leftarrow i + 1$, i.e., $i \leftarrow 1$.

Step #3 Since $MEM_{3+1} \neq 0$, assign $i \leftarrow i + 1$, i.e., $i \leftarrow 2$.

Step #4 Since $MEM_{3+2} \neq 0$, assign $i \leftarrow i + 1$, i.e., $i \leftarrow 3$.

Step #5 Since $MEM_{3+3} \neq 0$, assign $i \leftarrow i + 1$, i.e., $i \leftarrow 4$.

Step #6 Since $MEM_{3+4} \neq 0$, assign $i \leftarrow i + 1$, i.e., $i \leftarrow 5$.

Step #7 Return $i = 5$.

This time things are a bit more complicated. So what is going on? We have used a slightly different loop construct than in Chapter 3 for this algorithm. When we write while $X$ do $Y$, the idea is that we repeatedly process the block $Y$ until $X$ evaluates to false. So we perform a test: if $X$ evaluates to true then we process the block $Y$, and then start again, otherwise we do not bother and exit the loop. This sort of loop is unbounded because we do not know how many times we will process $Y$ before we start.

Here, the test $X$ is “have we found the string terminator”. We keep adding one to a counter called $i$ until $MEM_{i+1} = 0$ at which point we know that $i$ should give the length of the string. So how many steps does the algorithm take? The answer is that if we add one to $i$ for each character in the string, then we must take at least $n$ steps for an $n$-character string. Using big-O notation we say it is $O(n)$.

So the P-string approach is much better when we want to compute the length of a string: an algorithm which has complexity $O(1)$ will always be better than one which has complexity $O(n)$, assuming $n$ is large enough. In this case we do not even need $n$ to be large since for all strings the P-string method will be better: there is only one string for which the two methods will take exactly the same number of steps. As an exercise, can you work out which string this is?
5.2.2 \texttt{toupper:} converting a string to upper-case

Computing the length of a string \textit{accesses} the string content but does not \textit{alter} it. The next task we look at is altering the string starting at address $x$ so that the content consists of upper-case characters only: we take each lower-case character in the string and convert it into the upper-case equivalent. The C programming language does not include a standard function which represents quite this idea, but it does have a function \texttt{toupper} that turns lower-case characters (rather than strings) into the upper-case equivalent.

5.2.2.1 P-string version

The P-string version is shown in Figure 5.4a. Imagine the memory content is initially

\begin{align*}
i &= \ldots, 3, 4, 5, 6, 7, 8, \ldots \\
MEM &= \langle \ldots, 5, 104, 101, 108, 108, 111, \ldots \rangle \\
\text{Chr}(MEM_i) &= \ldots, \text{ENQ}, \text{‘h’}, \text{‘e’}, \text{‘l’}, \text{‘o’}, \ldots
\end{align*}

and we invoke the algorithm using \texttt{P-STRING-ToUpper}(3). The only input is the string address, so in this case we want to convert the string at address 3 into upper-case. The corresponding steps performed by the algorithm are

\begin{enumerate}
  \item \textbf{Step #1} Assign $n = \text{P-STRING-LENGTH}(3) = 5$.
  \item \textbf{Step #2} Assign $t = MEM_{t+1+0} = 104$.
  \item \textbf{Step #3} Since $97 \leq t \leq 122$, store the result, i.e., $MEM_{t+1+0} = 104 - 32 = 72$.
  \item \textbf{Step #4} Assign $t = MEM_{t+1+1} = 101$.
  \item \textbf{Step #5} Since $97 \leq t \leq 122$, store the result, i.e., $MEM_{t+1+1} = 101 - 32 = 69$.
  \item \textbf{Step #6} Assign $t = MEM_{t+1+2} = 108$.
  \item \textbf{Step #7} Since $97 \leq t \leq 122$, store the result, i.e., $MEM_{t+1+2} = 108 - 32 = 76$.
  \item \textbf{Step #8} Assign $t = MEM_{t+1+3} = 108$.
  \item \textbf{Step #9} Since $97 \leq t \leq 122$, store the result, i.e., $MEM_{t+1+3} = 108 - 32 = 76$.
  \item \textbf{Step #10} Assign $t = MEM_{t+1+4} = 111$.
  \item \textbf{Step #11} Since $97 \leq t \leq 122$, store the result, i.e., $MEM_{t+1+4} = 111 - 32 = 79$.
  \item \textbf{Step #12} Return.
\end{enumerate}

Of course this time the algorithm actually stores new content in memory rather than just reading existing content from it! After the algorithm has finished, the content is described by:

\begin{align*}
i &= \ldots, 3, 4, 5, 6, 7, 8, \ldots \\
MEM &= \langle \ldots, 5, 72, 72, 72, 72, 72, \ldots \rangle \\
\text{Chr}(MEM_i) &= \ldots, \text{ENQ}, \text{‘H’}, \text{‘E’}, \text{‘L’}, \text{‘O’}, \ldots
\end{align*}

How did it end up this way? In the example where we were finding the length of a string, the complication was a new type of loop to deal with; here the complication is that by invoking \texttt{P-STRING-ToUpper}, we invoke \texttt{P-STRING-LENGTH} as well during the first step. From then on, it is fairly easy to see what is going on. We perform a series of loads from memory to retrieve the “current” character at $MEM_{t+1+i}$, and if the character we load is in the range 97 ... 122 (i.e., it is a lower-case character) we subtract 32 from it and store it back at the same place.

Working out how to express this in big-O notation is not hard. The main loop in lines #3 to #8 clearly takes $2 \cdot n$ steps since, for each character, we need one step to load it and one step to test and assign it if applicable. But to be fair, we also need to include the number of steps that it took to compute the string length via \texttt{P-STRING-LENGTH}. In a sense we just add the number of steps together, so informally we might write $O(1) + O(n) = O(1 + n)$. Of course $O(n)$ dominates $O(1)$ so using the same sort of simplification we did in Chapter 3, this turns into $O(n)$.
The C-string version of the function C-S-Toupper is shown in Figure 5.4b.

Algorithm C-String-ToUpper(x) begin
1. Let n = C-StringLength(x).
2. For i from 0 to n - 1 do:
   a. Let t = MEM_{i}, if 97 ≤ t ≤ 122 then MEM_{i} = t - 32.
3. Return n.

(a) P-string version.

(b) C-string version.

Figure 5.4: Algorithms to convert a string at address x, represented using a P-string (left-hand side) or C-string (right-hand side) data structure, into upper-case.

5.2.2.2 C-string version

The C-string version is shown in Figure 5.4b. Imagine the memory content is initially

\[ i = \ldots, 3, 4, 5, 6, 7, 8, \ldots \]
\[ MEM = \langle \ldots, 104, 101, 108, 108, 111, 0, \ldots \rangle \]
\[ \text{Chr}(MEM_i) = \ldots, 'h', 'e', 'l', 'o', 'O', \ldots \]

which, again, is just the C-string version of the P-string one we used above. If we invoke the algorithm with C-String-ToUpper(3) again, the corresponding steps performed by the algorithm are:

**Step #1** Assign \( n = \text{C-StringLength}(3) = 5 \).

**Step #2** Assign \( t = \text{MEM}_{3,0} = 104 \).

**Step #3** Since \( 97 \leq t \leq 122 \), store the result, i.e., \( \text{MEM}_{3,0} = 104 - 32 = 72 \).

**Step #4** Assign \( t = \text{MEM}_{3,1} = 101 \).

**Step #5** Since \( 97 \leq t \leq 122 \), store the result, i.e., \( \text{MEM}_{3,1} = 101 - 32 = 69 \).

**Step #6** Assign \( t = \text{MEM}_{3,2} = 108 \).

**Step #7** Since \( 97 \leq t \leq 122 \), store the result, i.e., \( \text{MEM}_{3,2} = 108 - 32 = 76 \).

**Step #8** Assign \( t = \text{MEM}_{3,3} = 108 \).

**Step #9** Since \( 97 \leq t \leq 122 \), store the result, i.e., \( \text{MEM}_{3,3} = 108 - 32 = 76 \).

**Step #10** Assign \( t = \text{MEM}_{3,4} = 111 \).

**Step #11** Since \( 97 \leq t \leq 122 \), store the result, i.e., \( \text{MEM}_{3,4} = 111 - 32 = 79 \).

**Step #12** Return.

which alter the memory content to read

\[ i = \ldots, 3, 4, 5, 6, 7, 8, \ldots \]
\[ MEM = \langle \ldots, 72, 69, 76, 76, 79, 0, \ldots \rangle \]
\[ \text{Chr}(MEM_i) = \ldots, 'H', 'E', 'L', 'O', 'O', \ldots \]

afterwards. In the example where we were finding the length of a string, the steps taken by the two algorithms were radically different; here they are more similar. They are more or less the same in fact, bar the first step which obviously needs to use C-String-Length instead of P-String-Length. Since C-StringLength takes \( O(n) \) steps, the function C-String-ToUpper takes \( O(n) + O(n) = O(2 \cdot n) \) steps. The constant can be ignored if we again tolerate the simplifications described in Chapter 3, and we end up with \( O(n) \).
5.2.3 *strcmp*: testing if one string is the same as another

Next we look at the task of string comparison, or string matching. The idea is that we take two strings, one starting at address \(x\) and one at address \(y\), and try to determine whether they are the same or not. The C programming language includes a standard function called `strcmp` which represents an implementation of this idea.

5.2.3.1 P-string version

The P-string version is shown in Figure 5.5a. Imagine the memory content is initially

\[
\begin{align*}
  i & = \ldots, 3, 4, 5, 6, 7, 8, \\
  MEM & = \langle \ldots, 5, 104, 101, 108, 108, 111, \ldots \rangle \\
  \text{Chr}(MEM) & = \ldots, 'h', 'e', 'l', 'l', 'o', \\
  i & = \langle 9, 10, 11, 12, 13, 14, \ldots \rangle \\
  MEM & = \langle 5, 104, 101, 76, 76, 111, \ldots \rangle \\
  \text{Chr}(MEM) & = \langle 'h', 'e', 'l', 'l', 'o', \ldots \rangle
\end{align*}
\]

Notice that we have got some more content: in fact so much more we have now split the memory content onto two lines. This is, in part, because this algorithm takes two arguments representing the addresses of the strings we want to compare. Using P-STRING-MATCH(3, 9) to invoke the algorithm means we want to compare the strings at addresses 3 and 9, with the corresponding steps performed as follows:

**Step #1** Assign \(n = \text{P-STRING-LENGTH}(3) = 5\).

**Step #2** Assign \(m = \text{P-STRING-LENGTH}(9) = 5\).

**Step #3** Since \(n = m\), continue.

**Step #4** Since \(MEM_{3+1+0} = MEM_{9+1+0} = 104\), continue.

**Step #5** Since \(MEM_{3+1+1} = MEM_{9+1+1} = 101\), continue.

**Step #6** Since \(MEM_{3+1+2} \neq MEM_{9+1+2}\), return false.

The strings are similar, but not the same. In particular, the ‘l’ characters in the string at address #3 (i.e., the characters at addresses #6 and #7) are lower-case, but those in the string at address #9 (i.e., the characters at addresses #12 and #13) are upper-case. This is reflected in the fact that the algorithm returned false.

To understand the reason it gives the right answer we need to take a closer look at what happens at each step. The first thing that happens is a condition, namely if the number of characters in the first string is not the same as the number of characters in the second string then the strings cannot be the same, and the algorithm returns false as the result. But that does not happen here, since both strings have five characters in them so we carry on, and proceed to check each character. To do this, we use a loop which iterates over a block for values of \(i\) in the range \(0 \ldots n - 1\). By this point we know \(n = m\) so there is no danger of the \(i\)-th character of either string being “out of bounds” (i.e., beyond their actual length). For each value of \(i\), we test the \(i\)-th characters of the two strings against each other. If they are not equal, then clearly the two strings are not equal and we can return false as the result. For \(i = 0\) and \(i = 1\), the characters in both strings are ‘h’ and ‘e’ so the algorithm continues. When we hit \(i = 3\), however, the characters are ‘l’ and ‘l’ which are not equal; the algorithm notices this, concludes that the strings do not match, then returns false as the result.

If, on the other hand, it had got all the way through the strings and found that they all matched it would return true instead.

As you might have expected, there is an extra complication here. The number of steps the algorithm takes depends on the strings themselves, i.e., their content and length, but it is not too hard to describe the number of steps in big-O notation. The two invocations of P-STRING-LENGTH take \(O(1) + O(1)\) steps which is still \(O(1)\), and then in the worst case we have to test all \(n\) characters of the strings so the main loop takes \(O(n)\). The total number of steps is therefore \(O(1) + O(n)\) which simplifies to \(O(n)\).
1 \textbf{algorithm} P-\textit{String}-\textit{Match}(x, y) \begin{align*}
2 & n \leftarrow \text{P-\textit{String}-\textit{Length}}(x) \\
3 & m \leftarrow \text{P-\textit{String}-\textit{Length}}(y) \\
4 & \text{if } n \neq m \text{ then} \\
5 & \quad \text{return false} \\
6 & \end{align*}
7 for \(i\) from 0 \text{upto} \(n - 1\) do
8 \quad \text{if } \text{MEM}_{x+i} \neq \text{MEM}_{y+i} \text{ then}
9 \quad \quad \text{return false}
10 \quad \end{align*}
11 end
12 return true
13 end

\begin{align*}
1 \textbf{algorithm} \ C-\textit{String}-\textit{Match}(x, y) \begin{align*}
2 & n \leftarrow \text{C-\textit{String}-\textit{Length}}(x) \\
3 & m \leftarrow \text{C-\textit{String}-\textit{Length}}(y) \\
4 & \text{if } n \neq m \text{ then} \\
5 & \quad \text{return false} \\
6 & \end{align*}
7 for \(i\) from 0 \text{upto} \(n - 1\) do
8 \quad \text{if } \text{MEM}_{x+i} \neq \text{MEM}_{y+i} \text{ then}
9 \quad \quad \text{return false}
10 \quad \end{align*}
11 end
12 return true
13 end

(a) P-string version. \hspace{1cm} (b) C-string version.

\textbf{Figure 5.5:} Algorithms to match one string at address \(x\) against another at address \(y\), both represented using a P-string (left-hand side) or C-string (right-hand side) data structure.

\subsection*{5.2.3.2 C-string version}

The C-string version is shown in Figure 5.5b. Imagine the memory content is initially

\begin{align*}
\text{i} & = \ldots, 3, 4, 5, 6, 7, 8, \\
\text{MEM} & = \langle \ldots, 104, 101, 108, 108, 111, 0, \rangle \\
\text{Chr}(\text{MEM}_i) & = \ldots, \text{‘h’}, \text{‘e’}, \text{‘l’}, \text{‘l’}, \text{‘o’}, \text{NUL},
\end{align*}

which, again, is just the C-string version of the P-string one we used above. Invoking the algorithm with \(\text{C-\textit{String}-\textit{Match}}(3, 9)\) gives basically the same steps as last time, namely

\textbf{Step # 1} Assign \(n = \text{C-\textit{String}-\textit{Length}}(3) = 5\).

\textbf{Step # 2} Assign \(m = \text{C-\textit{String}-\textit{Length}}(9) = 5\).

\textbf{Step # 3} Since \(n = m\), continue.

\textbf{Step # 4} Since \(\text{MEM}_{3+0} = \text{MEM}_{9+0} = 104\), continue.

\textbf{Step # 5} Since \(\text{MEM}_{3+1} = \text{MEM}_{9+1} = 101\), continue.

\textbf{Step # 6} Since \(\text{MEM}_{3+2} \neq \text{MEM}_{9+2}\), return \textbf{false}.

This time we use \(\text{C-\textit{String}-\textit{Length}}\) instead of \(\text{P-\textit{String}-\textit{Length}}\), but the result is still \textbf{false} as you would expect. The two initial invocations of \(\text{C-\textit{String}-\textit{Length}}\) plus the main algorithm take a grand total of \(O(n) + O(n) + O(n) = O(3 \cdot n)\) steps, but of course we again ignore the constant and simplify this to \(O(n)\).

\subsection*{5.2.4 \texttt{strcat}: concatenating two strings together}

The final task we look at is the key one we have been building up to, as you might have guessed. The idea is to take one string (the source string) starting at address \(y\) and concatenate it with, or join it onto, another one (the target string) starting at address \(x\); the source string remains unaltered.
5.2.4.1 P-string version

The P-string version is shown in Figure 5.6a. Imagine the memory content is initially

\[ i = \ldots, 3, 4, 5, 6, 7, 8, \]
\[ MEM = \langle \ldots, 5, 104, 101, 108, 108, 111, \rangle \]
\[ \text{Chr}(MEM_i) = \ldots, \text{ENQ}, \ 'h', \ 'e', \ 'l', \ 'o', \ 'a', \]

and we invoke the algorithm using P-STRING-CONCAT(9, 3). This means we specify that the target string starts at address #9, and the source string starts at address #3; both strings are initially “hello”. The algorithm itself is quite simple. Basically, after computing the lengths of the source and target string, it uses a loop to copy each character of the source string to the corresponding address after the target string ends. In short, the character at address \( y + 1 + i \) in the source string is copied to address \( x + 1 + i + n \) in the target string. The steps performed by the algorithm in this case are

**Step #1** Assign \( n = \text{P-STRING-LENGTH}(9) = 5 \).

**Step #2** Assign \( m = \text{P-STRING-LENGTH}(3) = 5 \).

**Step #3** Assign \( \text{MEM}_{9+1+0+5} = \text{MEM}_{3+1+0} = 104 \).

**Step #4** Assign \( \text{MEM}_{9+1+1+5} = \text{MEM}_{3+1+1} = 101 \).

**Step #5** Assign \( \text{MEM}_{9+1+2+5} = \text{MEM}_{3+1+2} = 108 \).

**Step #6** Assign \( \text{MEM}_{9+1+3+5} = \text{MEM}_{3+1+3} = 108 \).

**Step #7** Assign \( \text{MEM}_{9+1+4+5} = \text{MEM}_{3+1+4} = 111 \).

**Step #8** Assign \( \text{MEM}_{9} = 5 + 5 = 10 \).

**Step #9** Return.

which alter the memory content to read

\[ i = \ldots, 3, 4, 5, 6, 7, 8, \]
\[ MEM = \langle \ldots, 5, 104, 101, 108, 108, 111, \rangle \]
\[ \text{Chr}(MEM_i) = \ldots, \text{ENQ}, \ 'h', \ 'e', \ 'l', \ 'o', \ 'a', \]

\[ i = 9, 10, 11, 12, 13, 14, \]
\[ MEM = \langle 10, 104, 101, 108, 108, 111, \rangle \]
\[ \text{Chr}(MEM_i) = \text{ENQ}, \ 'h', \ 'e', \ 'l', \ 'o', \]

\[ i = 15, 16, 17, 18, 19, \]
\[ MEM = \langle 104, 101, 108, 108, 111, \rangle \]
\[ \text{Chr}(MEM_i) = \ 'h', \ 'e', \ 'l', \ 'o', \]

We have split the content across three lines now to make it fit, but the point is that if you check the addresses and the content you find that the source string at address #3 is still “hello” but the target string at address #9 is now “hellohello”, i.e., the original source and target joined together.

In terms of a big-O description of P-STRING-CONCAT, the two invocations of P-STRING-LENGTH take \( O(1) + O(1) \) steps which is still \( O(1) \), and then we perform \( n \) steps inside the loop that copies the characters. This gives a total of \( O(1) + O(1) + O(n) \) which of course we simplify to \( O(n) \).
The only real difference is the last step, which instead of setting the length of the target string to \( n + m \), i.e., the sum of the lengths of the source and target, it “moves” the terminator character to the correct position at the new end of the target. The memory content is altered to read

\[
\begin{align*}
  \text{MEM} &= \langle \ldots, 104, 101, 108, 108, 111, 0, \ldots \rangle \\
  \text{Chr}(\text{MEM}_i) &= \ldots, 'h', 'e', 'l', 'o', 'NUL', \\
  i &= 15, 16, 17, 18, 19, \ldots \\
  \text{MEM} &= \langle 101, 108, 108, 111, 0, \ldots \rangle \\
  \text{Chr}(\text{MEM}_i) &= 'e', 'l', 'o', 'NUL', \ldots
\end{align*}
\]

which, again, is just the C-string version of the P-string one we used above. Invoking the algorithm with \text{C-STRING-CONCAT}(9, 3) gives basically the same steps as last time, i.e.,

\[\text{Step } #1 \text{ Assign } n = \text{C-STRING-LENGTH}(9) = 5.\]

\[\text{Step } #2 \text{ Assign } m = \text{C-STRING-LENGTH}(3) = 5.\]

\[\text{Step } #3 \text{ Assign } \text{MEM}_{9+0+5} = \text{MEM}_3 + 0 = 104.\]

\[\text{Step } #4 \text{ Assign } \text{MEM}_{9+1+5} = \text{MEM}_3 + 1 = 101.\]

\[\text{Step } #5 \text{ Assign } \text{MEM}_{9+2+5} = \text{MEM}_3 + 2 = 108.\]

\[\text{Step } #6 \text{ Assign } \text{MEM}_{9+3+5} = \text{MEM}_3 + 3 = 108.\]

\[\text{Step } #7 \text{ Assign } \text{MEM}_{9+4+5} = \text{MEM}_3 + 4 = 111.\]

\[\text{Step } #8 \text{ Assign } \text{MEM}_{19} = 0.\]

\[\text{Step } #9 \text{ Return.}\]

The only real difference is the last step, which instead of setting the length of the target string to \( n + m \), i.e., the sum of the lengths of the source and target, it “moves” the terminator character to the correct position at the new end of the target. The memory content is altered to read

\[
\begin{align*}
  \text{MEM} &= \langle \ldots, 3, 4, 5, 6, 7, 8, \ldots \rangle \\
  \text{Chr}(\text{MEM}_i) &= \ldots, 'h', 'e', 'l', 'o', 'NUL', \\
  i &= 9, 10, 11, 12, 13, 14, \ldots \\
  \text{MEM} &= \langle 104, 101, 108, 108, 111, 0, \ldots \rangle \\
  \text{Chr}(\text{MEM}_i) &= 'h', 'e', 'l', 'o', 'NUL', \ldots
\end{align*}
\]

\[\text{Step } #1 \text{ Assign } n = \text{P-STRING-LENGTH}(9) = 5.\]

\[\text{Step } #2 \text{ Assign } m = \text{P-STRING-LENGTH}(3) = 5.\]

\[\text{Step } #3 \text{ Assign } \text{MEM}_{9+1+5} = \text{MEM}_3 + 0 = 104.\]

\[\text{Step } #4 \text{ Assign } \text{MEM}_{9+2+5} = \text{MEM}_3 + 1 = 101.\]

\[\text{Step } #5 \text{ Assign } \text{MEM}_{9+3+5} = \text{MEM}_3 + 2 = 108.\]

\[\text{Step } #6 \text{ Assign } \text{MEM}_{9+4+5} = \text{MEM}_3 + 3 = 108.\]

\[\text{Step } #7 \text{ Assign } \text{MEM}_{9+5+5} = \text{MEM}_3 + 4 = 111.\]

\[\text{Step } #8 \text{ Assign } \text{MEM}_{19} = 0.\]

\[\text{Step } #9 \text{ Return.}\]
so that again we find the source string at address #3 is still “hello” but the target string at address #9 is now “hellohello”, i.e., the original source and target joined together. Again, the big-O notation for C-STRING-CONCAT is simple. We end up with $O(n) + O(n) + O(n)$ which we simplify to $O(n)$.

You can probably think of lots of other useful operations we could perform on a string, but two examples are described below. In each case, your challenge is to write two algorithms that apply the operation using the C-string and P-string data structures respectively.

1. C has a standard function called strchr that takes a character $c$ and the address of a string $x$ as input: the function returns how far the first instance of the character is (i.e., the offset) from the start of the string. For example, if the character is ‘l’ and the string is “hello” we expect the result to be 2: the first instance of ‘l’ is 2 characters from the start of “hello”.

What happens if the character does not occur in the string: what useful value could the algorithm return in this case?

2. C has no standard function called strrev, but imagine it takes the address of a string $x$ as input and reverses the order of the characters. For example, the string “hello” would become “olleh”.

### 5.2.5 Problem #3: repeated concatenation

After all that, here is the problem that justifies what we have been doing. Have a look again at the last task of concatenating strings together and imagine we try to concatenate $m$ strings together, accumulating the result in the string at address $x$. Using P-STRING-CONCAT as an example, we would end up with something like this:

```
P-STRING-CONCAT(x, “foo”) ↦ x = “foo”
P-STRING-CONCAT(x, “bar”) ↦ x = “foobar”
P-STRING-CONCAT(x, “baz”) ↦ x = “foobaz”
```

That is, after the first invocation we would have appended “foo” to the empty string to get “foo”, and after the second we would have appended “bar” to “foo” to get “foobar” and so on.

The thing to notice is that $x$ gets longer and longer, but each string we concatenate to it remains short (say $n$ characters). So how does this alter the number of steps taken by each invocation of P-STRING-CONCAT? Well, the number of steps in the main loop is determined by the number of characters in the source string, so this does not change. What about the number of steps required before the main loop to compute the length of the strings? P-STRING-LENGTH is $O(1)$, so no matter how long $x$ is it will take a constant number of steps to give us a result. Even if we invoke P-STRING-LENGTH $m$ times like this, the end result is still $O(n)$.

What about the C-string alternative? Again starting off with $x$ as the empty string, we would get the same result, i.e.,

```
C-STRING-CONCAT(x, “foo”) ↦ x = “foo”
C-STRING-CONCAT(x, “bar”) ↦ x = “foobar”
C-STRING-CONCAT(x, “baz”) ↦ x = “foobaz”
```

but now the behaviour is a bit different. Of course, the number of steps is still determined by the number of characters in the source string and this still does not change. But now, as $x$ gets longer and longer C-STRING-LENGTH takes more and more steps to give us a result. Remember, it is $O(n)$ and $n$ is getting larger and larger with each invocation. This might not look that bad, because if we invoke C-STRING-CONCAT four times then the total number of steps will be

$$O(n) + O(2n) + O(3n) + O(4n) = O(10n),$$

but we simplify this by ignoring the constants to get $O(n)$. But what happens if we invoke C-STRING-CONCAT $m$ times? Now we get the sum

$$O(n) + O(2n) + \cdots + O(m \cdot n) = O\left(\sum_{i=1}^{m} i \cdot n\right)$$

which simplifies first to

$$O((m \cdot (m+1)/2) \cdot n)$$

because $\sum_{i=1}^{m} i = m(m+1)/2$ and then to

$$O(n \cdot m^2)$$
because $O(m(m + 1)/2) = O(m^2)$. So the upshot is that if we invoke C-STRING-CONCAT $m$ times like this, the end result is more like $O(m^2)$, since now we treat the value $n$ as the constant. Yikes! Remember Chapter 3? $O(m^2)$ is bad. An eloquent analogy of this problem is offered by famed writer and programmer Joel Spolsky [8]:

Shlemiel gets a job as a street painter, painting the dotted lines down the middle of the road. On the first day he takes a can of paint out to the road and finishes 300 yards of the road. “That’s pretty good!” says his boss, “you’re a fast worker!” and pays him a kopeck. The next day Shlemiel only gets 150 yards done. “Well, that’s not nearly as good as yesterday, but you’re still a fast worker. 150 yards is respectable,” and pays him a kopeck. The next day Shlemiel paints 30 yards of the road. “Only 30!” shouts his boss. “That’s unacceptable! On the first day you did ten times that much work! What’s going on?” “I can’t help it,” says Shlemiel. “Every day I get farther and farther away from the paint can!”

The C-string strcat algorithm is poor old Shlemiel in this analogy: each time we try to concatenate one of our $m$ strings, we need to place it further and further away from the end of the start of $x$.

Hopefully you can appreciate the problem now that we have spelled it out in such gruesome detail, but why does it matter? In short, understanding problems like this highlights the fact that C hides low-level detail of the data structure from us; this is even more true of languages such as Java. On one hand the abstraction this offers is great news because we do not have to worry about the data structure so much: the programming language takes care of it all for us automatically. But on the other hand, only by understanding low-level details can one hope to write efficient high-level programs! This is an issue that, in the opinion of many people, plagues modern Computer Science. We have built great tools to abstract away detail so we can construct wondrous hardware and software artefacts, but without an understanding of the fundamentals, one is always at a disadvantage.

C is not a bad language, and the C-string data structure is not the wrong choice: remember each of the C-string and P-string approaches have advantages and disadvantages. Therefore, it is interesting to look at the reason why the designers of C, Brian Kernighan and Dennis Ritchie, chose C-string rather than the P-string alternative in the 1970s. Two historical drivers are important. First, memory was at a real premium at the time C was developed; when dealing with large strings, the advantage of having a single string terminator character rather than a larger length field is tangible. It might seem amazing now, but saving even an extra byte of memory here and there could have been important then. Second, the PDP [7] range of computers used as early development platforms for C already used C-string type strings, and had some instructions that could deal with such strings quite efficiently. Hindsight is a wonderful thing; maybe they would have made a different choice looking back, maybe not. But it should be clear that by understanding low-level details, one at least has the opportunity to learn from the benefit of hindsight, and potentially to design better data structures and algorithms as a result. After all, data structures get much more complicated than strings so the cost of not understanding the details is potentially much greater as well.

Lots of instruction sets include instructions that were once useful, but now seem slightly odd; these legacy instructions often remain to ensure backward compatibility, i.e., to make sure old programs still work.

The x86 instruction set used by Intel includes support for something called **Binary Coded Decimal (BCD)** [3]. Find out about the instructions available for BCD; what do you think the original motivation for including them was? Why do you think they are or are not still useful now?
BIBLIOGRAPHY

CHAPTER

6

DEMYSTIFYING WEB-SEARCH: THE MATHEMATICS OF PAGERANK

Ask yourself a question: other web-search engines [39] exist of course, but how often do you use Google via

https://www.google.com/

to search for something? In fact, if you have a Google account the answer is available at

https://history.google.com/

unless you turned this feature off. For me it was around 100 times a day, although interestingly it changes a lot depending on what day it is. Another one: how often does the set of results produced fail to include what you were searching for? This is harder to answer precisely, but my guess would be not that often overall. Even accepting that it might fail sometimes, if you stop to think about it this is really amazing: versus only a generation ago, it seems fair to say that the ability to access so much information with such ease and accuracy has changed the world we live in fundamentally.

Although online advertising underpins the Google business model (a large proportion of income stemming from the AdWords [2] and AdSense [1] systems), from a technology perspective their web-search system remains a core interest. To support it, Google store and process a lot of information (some estimates cite upward of 50 billion web-sites alone, on top of which they deal with images etc.) and deal with a lot of search queries (40% of the current 7 billion world population make use of the Internet, so if each of them perform 100 Google searches a day we are potentially talking about a huge volume of queries at any given point in time). This means they clearly need a lot of computing power to keep pace with demand. Beyond this however, and given all the challenges involved, how are they able to provide such high-quality results?

A full answer is obviously quite complicated, and has also changed over time to meet new challenges and capitalise on new opportunities. Even so, the central concepts depend on applying fundamental Computer Science to solve real, practical challenges; better still, they can be explained using Mathematical techniques you are already (somewhat) familiar with. These concepts represent our focus in this Chapter: our aim is to explain how Google produces results for your search queries, using graph theory and probability theory behind the scenes.¹

¹The Chapter assumes you have at least some exposure to these topics, and focuses on explaining how they are used. However, we include a number of fairly lengthy introductions in case you need a refresher or even a place to start learning about them from scratch.
6.1 PageRank: the essence of Google web-search

6.1.1 What actually is web-search?

First, a recap of things you probably already know. The World Wide Web (WWW) [41] (or web) was conceived in a 1989 proposal by Tim Berners-Lee: the idea was to develop a universal mechanism to view (and edit) documents, with embedded connections that would facilitate browsing through them. Overviewing the rich body of previous and related work is beyond our scope, but the proposal basically aimed to implement what was an established concept called hypertext [13]; one famous example is the so-called “mother of all demos” in 1969 by Doug Engelbart [32], which included use of the first mouse to navigate hypertext documents in NLS [22]. This aim was achieved when Berners-Lee and a group of collaborators eventually developed

1. the HTML [14] language for describing document content, which we now know as web-pages, and the embedded hyperlinks [11];
2. the HTTP [15] protocol used to communicate such content to and from client (or web-browser) and web-server software, plus
3. the first actual implementations of such software that could be used to form a working system.

The result was advertised to the Internet at large in 1991, and the rest, as they say, is history: various aspects of the original system have matured and improved, but either way it now represents a ubiquitous presence in modern life.

If you think of the web as a massive database of information stored within web-pages, web-search can be described as an information retrieval problem [17]: given the database of web-pages, represented by a set $V$, we want to find the sub-set $R \subset V$ so each $x \in R$ matches a search query $q$. Exactly what constitutes a match depends on the type of search queries allowed [40] of course, so deciding whether a given web-page matches or not is a challenge in itself. To make things simple however, imagine $q$ is just a single word: $R$ might then be the set of web-pages which contain it somewhere.

6.1.2 Web-search before Google

Depending on how old you are, the birth of the web in 1991 might already seem like a long time ago. But systems for web-search did already exist between then and 1998, when Google was founded as a company. Understanding of this history is important, because it explains the technical context into which Google web-search was born a few years earlier (based on a 1996 research project at Stanford University): like most good products it solved a problem, so by understanding the problem means we can better appreciate the solution.

6.1.2.1 Web-directories

Once a significant number of web-pages were available, the most obvious way to locate one of interest was initially a web-directory [36]. Probably the first example, dated roughly 1992, was a list of web-server maintained by Tim Berners-Lee at CERN (when there were few enough to do so easily); contemporary examples include the WWW Virtual Library [42], also started by Berners-Lee, at

http://www.vlib.org/

and the Open Directory Project (ODP) [24] at

http://www.dmoz.org/

The basic idea is to classify web-pages within a giant table of contents. Much like the table of contents in a book is divided into Chapters and/or Sections, the web-directory is arranged into a hierarchy of categories with all web-pages about a given topic listed under the same category. Imagine some user wants to find web-pages related to some topic $q$. To do so, they start at the top, least specific level of the hierarchy and

---

3 A copy of the post is preserved at http://groups.google.com/groups?selm=6487%40cernvax.cern.ch.
4 A copy of the web-page is preserved at http://www.w3.org/History/19921103-hypertext/hypertext/DataSources/WWW/Servers.html.
move level-by-level through more specific categories until they get to a set $R$ of matches for $q$. Imagine $q = “star wars”$ for instance: using ODP, they might navigate through

“Top” → “Arts” → “Movies” → “Titles” → “S” → “Star Wars Movies”

until eventually finding (lots of) relevant matches.

6.1.2.2 Web-search engines

A web-search engine (or search engine) offers a different way to resolve search queries, basically forming $R$ by testing $q$ against every web-page on behalf of the user. This avoids the need to maintain the hierarchy of categories, and also automates the process so that users no longer need to manually control the search.

A lot of web-search engines have existed, and their history is interesting as a topic in itself. Although it oversimplifies the range of approaches employed somewhat, we can think of them as using three stages:

1. web-crawling [35],
2. web-indexing [38], then
3. search query resolution.

The first and second stages automatically collect and summarise information about web-pages, using the content to form a web-index used by the third stage to produce a set of results. The volume of information and relative lack of storage capacity initially limited the information collected to specific features of a web-pages (e.g., within so-called meta tags [20], or headings and titles), but this was quickly expanded to full-text search (meaning search of any web-page content).

Think of it like this: the index of a book is basically a map from words (usually those deemed important in some sense), to pages on which they appear. The index is prepared before the book is printed by analysing the content of each pagem, then if you want to know which page includes word $x$, you look-up $x$ in the index and it tells you. The concept of web-indexing is exactly the same, except we want a mapping from words to web-pages. Now, we want to know which web-pages match the search query $q$, so we look-up $q$ in the web-index and use the associated web-pages as $R$.

The system as a whole operates in two phases [23]: preparation of the web-index is performed offline (meaning before anyone uses the system), while search query resolution is performed online (meaning during use of the system). It is important the web-index can be prepared offline, because the web-crawling process will take a long time for the entire web. By pre-computing [26] it before anyone uses the system, the actual searches queries can still be resolved quickly because the bulk of the work has already been completed.

6.1.3 Web-search after Google

Arguably, a good web-directory will produce high-quality results provided users can navigate the hierarchy to resolve their query. Why? Ignoring the issue of deciding on a category for web-page $x$, a quality control system decides whether or not $x$ is good enough to include at all: this decision might be resolved in two parts, by a submitter first suggesting the web-page then an editor making a final decision about inclusion. You could think of this as roughly analogous to them voting for the web-page. However, there are a number of problems, including

- the actual decisions may be subjective and/or hard to articulate precisely, so it is hard to deal with issues such as bias,
- it is difficult to cope with many web-pages, since the workload involved in doing so is relatively high, and
- it sort of assumes the web-pages stay the same, because otherwise the process would have to be repeated again and again, which of course further adds to the workload.

You could also argue that web-search engines solve some of these problems through automation, since the workload can be borne by a computer rather than a human. However, it also seems fair to say they just shift rather than solve the underlying problem: the maintainer of a given web-search engine has an easier task than for a web-directory, but now the user is faced with the challenge of enforcing quality control. They must filter the good results from potentially lots of bad ones.
PageRank [25] is, roughly speaking, the solution employed by Google. Although it represents an important aspect of their producing such high-quality results, and therefore the success of the company, the concept itself is very easy to grasp: instead of humans voting for web-pages, the web-pages will vote for each other. Think of each link from web-page \( x \) to \( y \) as a vote, in the sense that \( x \) indicates \( y \) has something important (or at least relevant) on it: \( y \) is deemed important if many other web-pages link to it, and even more so if those web-pages are themselves important. By considering the structure of the web (i.e., the links between web-pages) alongside their content, a PageRank-based web-search engine can solve the quality control problem: it allows us to automatically sort the results, and display the good, important ones before the bad, unimportant ones.

Strictly speaking, PageRank is just a component within the wider Google web-search system: the term may be used to describe either the measure of importance assigned to each web-page, or the algorithm used to compute such values. So given our intuition about what it is, how does it work? This is really three questions in one. In reality, we need to answer the following:

1. how do we collect information to inform our analysis of web-page importance, which is basically the same web-crawling challenge that any other web-search engine faces,
2. how do we formalise the concept of importance, so we can compute actual PageRank values for each web-page, and,
3. once we have those values, how do we use them while resolving a given search query?

These questions are addressed, in order, by the following Sections.

6.2 Using graph theory to model and explore the web

As already discussed, the goal of web-crawling is to automatically collect and summarise information about web-pages: we want a computer to do this rather than a human because there are a lot of web-pages to process. The idea is to develop an algorithm that controls the web-crawler software, i.e., determines exactly how it automatically browses the web. This task can be made a lot easier by using graph theory, including various existing algorithms, as a starting point: we model the real web as a directed graph

\[
G = (V, E)
\]
called the web-graph [37]: each vertex \( v \in V \) represents a web-page, and each edge \((u, v) \in E\) represents a link from web-page \( u \) to web-page \( v \). A (very small) example is illustrated by Figure 6.2, which includes \( n = 6 \) web-pages identified by URLs such as \( a.html \), and links such as from \( a.html \) to \( d.html \). This might not seem like a massive step; the web-graph is certainly a natural, and fairly obvious way to model the real web. Crucially though, this added formalism allows us to develop and reason about algorithms that process the web-graph.

6.2.1 Graph traversal

Given a graph \( G = (V, E) \) as input, graph traversal [10] is fairly simple problem to explain: starting at some vertex \( s \), the goal is simply to move along edges until all are vertices have been visited. If you translate “vertex” into “web-page” and “visit” into “extract and summarise the content”, this should seem reasonably close to what we want.

Algorithm 6.1a and Algorithm 6.1b detail two classic ways to solve this problem, called Breadth-First Search (BFS) [3] and Depth-First Search (DFS) [7] respectively. Each algorithm starts at \( s \), and then visits vertices one at a time by traversing edges from those already visited. To keep track of this process, a sequence \( Q \) of vertices yet to visit (called the worklist), and a set \( D \) of vertices already visited are maintained. The intuition is as follows:

- The loop in lines #3 to #12 processes vertices until the worklist is empty; line #4 removes the first vertex \( u \) from \( Q \) and then processes it.
- The loop in lines #5 to #9 process each edge \((u, v)\) from \( u \) to some other vertex \( v \): if \( v \) has not been visited already (i.e., \( v \notin D \)), then it is added to both \( Q \) and \( D \). Checking \( D \) first is important, because it allows cycles [5] to be avoided.

5 Why search? This terminology relates to how BFS and DFS are often used, namely to search for a target vertex within the graph: once the target vertex \( v \) is visited, the traversal usually stops rather than continuing to visit all vertices.
algorithm BFS($V, E, s$) begin
  Q ← $\langle s \rangle$, $D ← \{s\}$
  while $|Q| \neq 0$ do
    Remove first element $u$ from $Q$
    foreach $v$ such that $(u, v) \in E$ do
      if $v \notin D$ then
        Append $v$ to $Q$
        Add $v$ to $D$
      end
    end
    Visit and process $u$
  end
  return
end

(a) An algorithm for Breadth-First Search (BFS).

algorithm DFS($V, E, s$) begin
  Q ← $\langle s \rangle$, $D ← \{s\}$
  while $|Q| \neq 0$ do
    Remove first element $u$ from $Q$
    foreach $v$ such that $(u, v) \in E$ do
      if $v \notin D$ then
        Prepend $v$ to $Q$
        Add $v$ to $D$
      end
    end
    Visit and process $u$
  end
  return
end

(b) An algorithm for Depth-First Search (DFS).

Figure 6.1: Two approaches to traversal of vertices in a graph.
Figure 6.2: A simple, concrete web-graph that captures the link structure between, and content of each web-page; each of the \( n = 6 \) highly artificial web-pages is a short HTML file, which in combination provide structure (i.e., links between the web-pages) and content (i.e., some words, in this case names of fruit).

Figure 6.3: A simple 7-vertex tree used to explain the behaviour of BFS and DFS algorithms.
(a) Before processing to deal with sink web-page f.

(b) After processing to deal with sink web-page f; artificially added links from f shown as dashed lines.

Figure 6.4: Two simple, abstract web-graphs (derived from Figure 6.2) that capture the link structure between, but not content of each web-page.
An aside: a quick introduction to graphs and graph theory.

In graph theory \[9\], a graph \[8\] is used to describe a set of abstract objects and relationships between them. Formally, we specify a graph using two sets:

1. The set \( V \) is called the set of vertices (or nodes): these are basically labels that identify the abstract objects we are interested in.
2. The set \( E \) is called the set of edges: each edge is a pair of vertices from \( V \), i.e., \((u, v)\) where \( u, v \in V \), which specify a relationship between the associated objects.

As a result, we often write \( G = (V, E) \) to show that a particular graph \( G \) is specified by particular sets \( V \) and \( E \). Consider an example where \( V = \{a, b, c, d\} \) \( E = \{(a, b), (b, d), (d, c), (d, d), (c, a), (c, b)\} \)

From this, we can see there are four vertices labelled \( a, b, c \) and \( d \); there are six edges between them in total, meaning for example that \( a \) and \( b \) are connected and so on. We often visualise the structure of a graph by drawing each of the vertices, and linking them with lines to depict the edges.

Beyond this general definition, several particular types of graph are useful:

**Undirected versus directed edges** In an undirected graph, each edge simply connects two vertices: if we have \((u, v)\) for instance, this means the same as \((v, u)\) in that both specify an edge between \( u \) and \( v \). In a directed graph however, each edge carries some more information. Directed edges still connect vertices, but the order matters. For example, \((u, v)\) is an edge from \( u \) to \( v \) (but not from \( v \) to \( u \)) whereas \((v, u)\) is an edge from \( v \) to \( u \) (but not from \( u \) to \( v \)). It is common to visualise this using an arrow head on the edge.

**Unweighted versus weighted edges** In an unweighted graph, the edges carry no information other than the relationship they specify; this alone is enough for many applications. In a weighted graph however, we associate extra information called a weight with each edge: an edge between some \( u \) and \( v \) would look like \((u, v, w)\) where \( w \) is the weight, which is usually just a number.

There are numerous properties we can define given some graph \( G \), but an important one is the degree of a given vertex \( u \in V \): given two functions

\[
\begin{align*}
\text{into}(u) & = \{v \mid v \in V, (v, u) \in E\} \\
\text{from}(u) & = \{v \mid v \in V, (u, v) \in E\}
\end{align*}
\]

that produce the set of all vertices \( v \) where there is an edge from \( v \) to \( u \) or \( u \) to \( v \) respectively, the degree of \( u \) is \( \text{deg}(u) = |\text{from}(u) \cup \text{into}(u)| \) meaning the total number of edges that connect it to any other vertex.

A path is a sequence of edges in \( E \) which connect a sequence of vertices in \( V \): in a sense, it connects the first vertex in the sequence to the last vertex by moving along the intermediate edges. Within our example, the path \( P = \langle(a, b), (b, d), (d, c)\rangle \) connects \( a \) to \( c \), for instance; a path like this with no repeated vertices is deemed to be simple.
http://www.cs.bris.ac.uk

University of Bristol, Department of Computer Science

- Finally, line #11 visits vertex $u$, processing it in whatever way is appropriate.

Given a worklist $Q = \langle Q_0, Q_1, \ldots, Q_{n-1} \rangle$, one or two of the steps might need some more detailed explanation:

- In line #7 of Algorithm 6.1a, where we need to append $v$ to $Q$, we update $Q$ to be $Q \parallel \langle \langle v \rangle \rangle = \langle Q_0, Q_1, \ldots, Q_{n-1}, v \rangle$ so that $v$ becomes the new last element.
- In line #7 of Algorithm 6.1b, where we need to prepend $v$ to $Q$, we update $Q$ to be $Q = \langle v, Q_0, Q_1, \ldots, Q_{n-1} \rangle$ so that $v$ becomes the new first element.
- In line #4 of either algorithm, where we need to remove the first element from $Q$, we just take $Q_0$ (i.e., the first element) as $u$ then update $Q$ to be $(Q_1, \ldots, Q_{n-1})$ (i.e., all the other elements).

Beyond this, the behaviour of both algorithms is better explained with an example. Using a tree [34], a special type of undirected graph in which any two vertices are connected by one simple path, means their differing behaviours are easier to explain: consider a 7-vertex tree described formally via

$$
V = \{a, b, c, d, e, f, g\} \\
E = \{(a, b), (a, c), (b, d), (b, e), (c, f), (c, g)\}
$$

or visually in Figure 6.3. If we invoke the BFS algorithm using $a$ (the so-called root of the tree) as a starting point, it proceeds as follows:

**Step #1** Set $Q = \langle a \rangle$ and $D = \{a\}$.

**Step #2** Since $|Q| \neq 0$, perform the next loop iteration for $v \in \{b, c\}$.

  **Step #2.1** Set $u = a$ and $Q = \langle \rangle$.
  **Step #2.2** Since $b \notin D$, set $Q = \langle b \rangle$ and $D = \{a, b\}$.
  **Step #2.3** Since $c \notin D$, set $Q = \langle b, c \rangle$ and $D = \{a, b, c\}$.
  **Step #2.4** Visit $a$.

**Step #3** Since $|Q| \neq 0$, perform the next loop iteration for $v \in \{d, e\}$.

  **Step #3.1** Set $u = b$ and $Q = \langle c \rangle$.
  **Step #3.2** Since $d \notin D$, set $Q = \langle c, d \rangle$ and $D = \{a, b, c, d\}$.
  **Step #3.3** Since $e \notin D$, set $Q = \langle c, d, e \rangle$ and $D = \{a, b, c, d, e\}$.
  **Step #3.4** Visit $b$.

**Step #4** Since $|Q| \neq 0$, perform the next loop iteration for $v \in \{f, g\}$.

  **Step #4.1** Set $u = c$ and $Q = \langle d, e \rangle$.
  **Step #4.2** Since $f \notin D$, set $Q = \langle d, e, f \rangle$ and $D = \{a, b, c, d, e, f\}$.
  **Step #4.3** Since $g \notin D$, set $Q = \langle d, e, f, g \rangle$ and $D = \{a, b, c, d, e, f, g\}$.
  **Step #4.4** Visit $c$.

**Step #5** Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

  **Step #5.1** Set $u = d$ and $Q = \langle e, f, g \rangle$.
  **Step #5.2** Visit $c$.

**Step #6** Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

  **Step #6.1** Set $u = e$ and $Q = \langle f, g \rangle$.
  **Step #6.2** Visit $e$.

**Step #7** Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

  **Step #7.1** Set $u = f$ and $Q = \langle g \rangle$.
  **Step #7.2** Visit $f$.

**Step #8** Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.
Step #8.1 Set \( u = g \) and \( Q = \emptyset \).
Step #8.2 Visit \( g \).

Step #9 Since \( |Q| = 0 \), stop the loop.

Step #10 Return.

Doing the same with the DFS algorithm produces a different behaviour:

Step #1 Set \( Q = \langle a \rangle \) and \( D = \{ a \} \).

Step #2 Since \( |Q| \neq 0 \), perform the next loop iteration for \( v \in \{ b, c \} \).
   
   Step #2.1 Set \( u = a \) and \( Q = \emptyset \).
   
   Step #2.2 Since \( b \notin D \), set \( Q = \langle b \rangle \) and \( D = \{ a, b \} \).
   
   Step #2.3 Since \( c \notin D \), set \( Q = \langle c, b \rangle \) and \( D = \{ a, b, c \} \).
   
   Step #2.4 Visit \( a \).

Step #3 Since \( |Q| \neq 0 \), perform the next loop iteration for \( v \in \{ f, g \} \).
   
   Step #3.1 Set \( u = c \) and \( Q = \langle b \rangle \).
   
   Step #3.2 Since \( f \notin D \), set \( Q = \langle f, b \rangle \) and \( D = \{ a, b, c, f \} \).
   
   Step #3.3 Since \( g \notin D \), set \( Q = \langle g, f, b \rangle \) and \( D = \{ a, b, c, f, g \} \).
   
   Step #3.4 Visit \( c \).

Step #4 Since \( |Q| \neq 0 \), perform the next loop iteration for \( v \in \emptyset \).
   
   Step #4.1 Set \( u = g \) and \( Q = \langle f, b \rangle \).
   
   Step #4.2 Visit \( c \).

Step #5 Since \( |Q| \neq 0 \), perform the next loop iteration for \( v \in \emptyset \).
   
   Step #5.1 Set \( u = f \) and \( Q = \langle b \rangle \).
   
   Step #5.2 Visit \( f \).

Step #6 Since \( |Q| \neq 0 \), perform the next loop iteration for \( v \in \{ d, e \} \).
   
   Step #6.1 Set \( u = b \) and \( Q = \emptyset \).
   
   Step #6.2 Since \( d \notin D \), set \( Q = \langle d \rangle \) and \( D = \{ a, b, c, f, g, d \} \).
   
   Step #6.3 Since \( e \notin D \), set \( Q = \langle e, d \rangle \) and \( D = \{ a, b, c, f, g, d, e \} \).
   
   Step #6.4 Visit \( c \).

Step #7 Since \( |Q| \neq 0 \), perform the next loop iteration for \( v \in \emptyset \).
   
   Step #7.1 Set \( u = e \) and \( Q = \langle d \rangle \).
   
   Step #7.2 Visit \( e \).

Step #8 Since \( |Q| \neq 0 \), perform the next loop iteration for \( v \in \emptyset \).
   
   Step #8.1 Set \( u = d \) and \( Q = \emptyset \).
   
   Step #8.2 Visit \( d \).

Step #9 Since \( |Q| = 0 \), stop the loop.

Step #10 Return.
To summarise, both BFS and DFS satisfy the goal of visiting all vertices but differ in the order they do so. You can think of this tree as three levels, where the top level contains a, the middle level contains b and c, and the bottom level contains d, e, f and g. BFS visits each vertex in the current level before moving onto a lower level; the order of traversal is

\[ (a, b, c, d, e, f, g) \]

which sort of moves across the tree then down. DFS on the other hand visits each vertex in the lower level before the current one; the order of traversal is therefore

\[ (a, c, g, f, b, e, d) \]

which moves down the tree then across.

Using a tree is just a way to make the behaviours easier to explain however: we can of course invoke the same algorithms on a more general graph. If we take the web-graph in Figure 6.2 for example, the order BFS visits the web-pages is

\[ \langle \text{a.html, c.html, d.html, b.html, e.html, f.html} \rangle \]

whereas for DFS this changes to

\[ \langle \text{a.html, d.html, f.html, e.html, b.html, c.html} \rangle \]

Both BFS or DFS traverse the web-graph, so provided they act in the right way when visiting each web-page they (more or less) solve the original problem of automatically browsing the web.

6.2.2 Graph exploration

"More or less solves" is a hint: a subtle problem exists with using standard graph traversal. Remember that the web-crawler does not have access to the web-graph as input. The whole point is to automatically browse the web and collect information about web-pages, so if we already have the web-graph there would be no point!

Really solving this problem means translating BFS or DFS into graph exploration algorithms, but fortunately the difference is minor. Obviously we cannot have G as an input to the algorithm any longer; in a sense, the web-graph is now an output from exploration, not the input to traversal. As a result, line #5 cannot check for edges \((u, v) \in E\) because \(E\) is unknown. Instead, we need to take \(u\) and discover edges from it to other, potentially unknown web-pages. The form of web-pages makes this easy: HTML is a mark-up language [19], meaning as well as the actual content we see, each web-page contains extra information (the so-called mark-up). This mark-up will includes specifications of which web-page a given link is to, so if we parse the web-page content for a.html in Figure 6.2 for example, we might extract the edges \((\text{a.html, c.html})\) and \((\text{a.html, d.html})\) from the links &lt;a href=’c.html’&gt;link to c&lt;/a&gt; and &lt;a href=’d.html’&gt;link to d&lt;/a&gt; which are embedded in it.

So the idea is that we invoke the web-crawler on a starting point, for example a Top Level Domain (TLD) [33] such as \(s = \text{www.bbc.co.uk}\), and let it explore links from there to construct G. Once finished (e.g., when it runs out of web-pages to explore) we might invoke it on another starting point and merge together the results (e.g., to increase the chance of visiting every web-page), or just use G as is for whatever purpose we had in mind.

Research (task #22)

In BFS and DFS, the way \(v\) is added to the worklist \(Q\) in line #7 is the central difference; formally, \(Q\) is used as a queue in BFS and as a stack in DFS. Find out more about these data structures.

Research (task #23)

By using a so-called robots.txt file [29], a web-server can signal that certain web-pages should be ignored by a web-crawler. Find out about this mechanism: when and why do you think it makes sense to use it?
In line #11 of Algorithm 6.1a and Algorithm 6.1b, we need to visit and process a web-page \( u \). What this actually means depends on the task at hand of course: for some web-search engines this could just mean extracting and summarising the content of \( u \), noting that \( a.html \) contains the words “apple” and “orange” for instance.

Since PageRank needs the actual web-graph structure (i.e., the links between web-pages) as input, the processing basically needs to form and eventually return \( G = (V, E) \) by collecting the vertices and edges. Write an graph exploration algorithm to do this, using either Algorithm 6.1a or Algorithm 6.1b as a basis; demonstrate how it works using the web-graph in Figure 6.2.

### 6.3 Using probability theory to model web-browsing

Section 6.2 showed a suitable web-crawler can automatically generate a web-graph \( G = (V, E) \) for us, where the edges in \( E \) capture the link structure required to reason about each web-page \( x \in V \). The example in Figure 6.2 is quite detailed however: once we have \( G \), it is easier to consider Figure 6.4a instead. This is the same graph (e.g., \( a.html \) is just renamed \( a \)) but any unnecessary detail such as the web-page content is removed.

Given a \( G \) as input, our next challenge is computation of a PageRank value for each web-page. So where do we start? As stated, this challenge is enormously vague: we lack a formal definition of what we should compute, how we should try to compute it, or even what the correct answer looks like! To make reasoning about the problem easier, Google use the following model: imagine the web is comprised of \( n \) web-pages, which a user browses indefinitely. At each step, this user randomly causes one of two events to occur:

1. with probability \( 1 - p \), a random web-page is loaded from anywhere on the web (i.e., it thinks of a random web-page and types the URL into the web-browser address bar), or

2. with probability \( p \), it clicks on and hence follows a random link on the current web-page to some other web-page.

This is called the random surfer model, and amounts to performing a random walk [28] on the web-graph. Although this might not be how people really behave, it allows us to translate the vague English description into something more concrete: assuming we accept the random surfer model, the PageRank of a web-page is equivalent to the probability of visiting it. Think about it: if web-page \( x \) is more important then there will be more links to it, and as a result the probability is higher that at some point the random surfer visits \( x \) by following such a link (plus the probability it visits it at random of course). We can stress this by keeping in mind

\[
\text{"the PageRank of } x \text{" } \equiv \text{ Pr}[x]
\]

where the right-hand side means the probability that web-page \( x \) is visited; this in turn is equivalent to the number of votes accumulated by \( x \) if you prefer the voting analogy. It also allows a sanity check later when we come to compute the actual values: since we are dealing with probabilities, the sum of \( \text{Pr}[x] \) for all \( x \in V \) should be 1.

This all becomes more concrete still by capturing the description as a formula:

\[
\text{Pr}[x] = \frac{1-p}{n} + p \cdot \left( \sum_{y \in \text{into}(x)} \frac{\text{Pr}[y]}{\text{from}(y)} \right)
\]

There are two terms because there are two events possible within the random surfer model; each term computes the associated probability, which we then add together because we want the probability of either one event or the other occurring as the result. Although they might look cryptic, both terms are just translations of the English description of random surfer behaviour into Mathematics. For the first term, this is easy to see: if there are \( n \) web-pages in total and we load a random one with probability \( 1 - p \) then we end up on \( x \) with a probability of \( \frac{1-p}{n} \). The second is more difficult however. The term itself is \( p \) multiplied by

\[
\sum_{y \in \text{into}(x)} \frac{\text{Pr}[y]}{\text{from}(y)}
\]
or, in English, the probability of this event occurring multiplied by the combined probabilities of arriving at $x$ having followed a link from another web-page $y$. Think about it again using the voting analogy: a web-page $y$ has a number of votes to cast (i.e., $\Pr[y]$), so gives each each web-page $x$ that it links to a number of votes in proportion to the total number of links it contains (i.e., divides $\Pr[y]$ by the number of outgoing links from $y$, namely $\text{from}(y)$). We are interested in $x$ of course, so the summation basically deals with all web-pages $y$ that link to $x$ (i.e., all $y \in \text{into}(x)$), forming the sum of their votes cast for $x$ per the above. This means two things:

1. if $y$ has many votes to cast, the number given to $x$ will be larger (since the numerator $\Pr[y]$ will be larger) than if it had few, and

2. if $y$ votes for fewer web-pages by linking to them, the proportion given to $x$ will be larger (since the denominator $\text{from}(y)$ will be smaller) than if it votes for many.

By replacing votes with probabilities we get the desired result: there is a higher probability the random surfer visits $x$ by following a link if other web-pages link to it, and even more so if those web-pages have a high probability of being visited themselves.

Although PageRank is probably the most famous, other similar examples exist within a family of related techniques. Two such examples are

- the impact factor [16] used to gauge how important an academic publication is, and
- the Hyperlink-Induced Topic Search (HITS) algorithm [12] that deals with web-page ranking.

It is often important to see how techniques relate or build on each other: do some research into the above, and compare them with PageRank in terms of their approach, features and so on.

### 6.3.1 Sanitising the web-graph to avoid a subtle problems

Two special types of web-page can occur in a web-graph, namely

1. **source web-pages**, which have no incoming links (i.e., $\text{into}(x) = \emptyset$) so will never be *visited* by following links, and

2. **sink web-pages**, which have no outgoing links (i.e., $\text{from}(x) = \emptyset$) so will never be *exited* by following links.

For the first case, the only potential problem might be that our web-crawler fails to visit it. In terms of computing PageRank values, there is no issue: it is just deemed unimportant according to the PageRank metric, due to the lack of links to it. The second case is problematic however. There are two ways to think about why this is the case:

1. The random surfer model says one of two events will occur: either the random surfer loads a random web-page or follows a random link. If the random surfer visits a sink web-page however, the first event cannot occur because there are no links to follow. Intuitively this is a problem because it means the probabilistic model of behaviour we rely on breaks down for sink web-pages.

2. In the corresponding formula, each web-page can be seen as voting for others by distributing the votes allocated to it via the links it contains. A sink web-page votes for no other web-page however, even though they might vote for it. Numerically this is a problem because a sink web-page accumulates votes like a sort of black hole. That is, votes flow in but never comes out again; this skews the results produced for non-sink web-pages because there is a fixed number of votes in total.

As a result, it is common to treat sink nodes differently. Various ways to do this have been proposed, but the easiest to justify is as follows: for each sink web-page $x$ identified, we add an artificial edge of the form $(x, y)$ to every other vertex $y \in V$. By doing so, the random surfer is happy again because either event can occur once it visits $x$; it also ensures fairness to non-sink web-pages, because the PageRank accumulated by $x$ is now shared evenly among all other web-pages.

Consider Figure 6.4a: web-page $f$ satisfies the criteria for being a sink, since $\text{from}(f) = \emptyset$. To cope, we might amend the web-graph to produce Figure 6.4b where the dashed edges have been added artificially;
these ensure \( f \) is linked to every other web-page, in this case \( a, b, c, d \) and \( e \). We use this altered web-graph rather than the original from here on.

Research (task #26)
The “link sink web-page \( x \) to all other web-pages” strategy described is not the only option: various other strategies have also been proposed. Find out about at least one other strategy, then compare and contrast each option using a list of advantages and disadvantages.

Implement (task #27)
Write an algorithm that takes a web-graph as input, and applies a strategy (whether the one described, or discovered in Task 26) for dealing with sink web-pages; the output should be the amended web-graph ready for use during computation of PageRank values.

Research (task #28)
If you consider the real web rather than the examples presented, the value of \( n \) is large and hence many web-pages will be identified as sinks. If you implement the strategy described for dealing with them exactly, a problem starts to emerge. What do you think this problem could be, and, with reference to your algorithm in Task 27 for instance, how could it be avoided?

6.3.2 A Mathematical approach to computing PageRank

6.3.2.1 Rewriting the formula to avoid cycles

Now armed with a web-graph and a formula to compute \( \text{Pr}[x] \) for each web-page, you could be forgiven for thinking that we are done. There is a subtle but important problem lurking behind the scenes however. Look again at

\[
\text{Pr}[x] = \frac{1 - p}{n} + p \cdot \left( \sum_{\text{into}(x)} \frac{\text{Pr}[y]}{|\text{from}(y)|} \right),
\]

keeping in mind the goal is to compute the left-hand side. How? There is no hidden trick: to get the left-hand side, we just evaluate the right-hand side as with any equality. But the right-hand side includes \( \text{Pr}[y] \), so how do we compute this? Use the formula again! Now we have a chicken-and-egg style problem: we cannot compute \( \text{Pr}[x] \) until we already know \( \text{Pr}[y] \) for all \( y \in V \), which is then a cyclic argument (literally, because the problem stems from cycles in the web-graph).

Fortunately, we can resolve this using some slightly more advanced probability theory. The basic idea is that the random surfer model implicitly includes a notion of time: the random surfer browses web-pages indefinitely, but it does so step-by-step. Instead of thinking of \( \text{Pr}[x] \), as we have done so far, we make an alteration by letting \( \text{Pr}[x](t) \) denote the probability that the random surfer visits web-page \( x \) in step (or at time) \( t \). If we count the steps as starting at \( t = 0 \), then

\[
\text{Pr}[a(0)] = \text{Pr}[b(0)] = \text{Pr}[c(0)] = \text{Pr}[d(0)] = \text{Pr}[e(0)] = \frac{1}{6}.
\]

Hopefully this makes sense: we have to start somewhere, and since there are \( n = 6 \) web-pages then each one has probability \( \frac{1}{n} = \frac{1}{6} \) of being the starting point. What now? Well, at step \( t = 1 \) one or other of the events occurs and the random surfer visits another web-page. Imagine the random surfer starts by visiting web-page \( b \) at step \( t = 0 \) for example; what is the probability it visits web-page \( d \) at step \( t + 1 \)? We can answer this by looking at how probable the two events are in this case:

1. It might visit \( d \) with probability \( 1 - p \), by loading the web-page at random. There is a \( \frac{1}{n} \) probability of visiting any specific web-page of the \( n \) in total, so a \( \frac{1}{6} \) probability of visiting \( d \). This means a \( \frac{1 - p}{6} \) probability overall.

2. It might visit \( d \) with probability \( p \), by following a random link. There are \( |\text{from}(y)| \) links from any given web-page \( y \), so \( |\text{from}(b)| = |\{a, d, e\}| = 3 \) from \( b \) specifically. Only one of those links is to \( d \) though, so the probability of following that one specifically is \( \frac{1}{3} \). This means a \( \frac{p}{3} \) probability overall.
An aside: root finding with the iterative Newton-Raphson method.

A good example to illustrate the concept of using iterative methods is the Newton-Raphson [21] method for computing the roots of a function \( \delta \), i.e., an \( x \) such that

\[
\delta(x) = 0.
\]

If you think about the case of finding the square root of an integer \( y \), this amounts to finding an \( x \) such that \( x^2 = y \). Skipping a lot of theory that explains why, we do so using the relationship

\[
x_{i+1} = x_i - \frac{\delta(x_i)}{\delta'(x_i)}
\]

where \( \delta' \) is the derivative of \( \delta \); here, since \( \delta(x) = x^2 - y = 0 \), we know \( \delta'(x) = 2 \cdot x \). The important thing is that the left-hand side shows how to compute the \( (i+1) \)-th element in a sequence of progressively more accurate solutions, given the \( i \)-th such element on the right-hand side. For example, imagine we have \( y = 4321 \) and want to compute \( x = \sqrt{y} \). First we make a guess at \( x \), say \( x_0 = 100 \), and then iterate use of the recurrence to produce the following sequence:

\[
\begin{align*}
x_1 &= x_0 - \frac{\delta(x_0)}{\delta'(x_0)} = 100.000 - \frac{5679.00}{200.00} = 71.605 \\
x_2 &= x_1 - \frac{\delta(x_1)}{\delta'(x_1)} = 71.605 - \frac{1020.276}{143.210} = 65.974 \\
x_3 &= x_2 - \frac{\delta(x_2)}{\delta'(x_2)} = 65.974 - \frac{31.697}{131.949} = 65.734 \\
x_4 &= x_3 - \frac{\delta(x_3)}{\delta'(x_3)} = 65.734 - \frac{0.057}{131.468} = 65.734 \\
x_5 &= x_4 - \frac{\delta(x_4)}{\delta'(x_4)} = 65.734 - \frac{0.000}{131.468} = 65.734 \\
&\vdots & \vdots \\x_i &= \vdots
\end{align*}
\]

Eventually, the changes in our solution get very small and we can say they converge. We need a termination criteria to detect this, but in our example this is easy since we can check whether \( \text{abs}(x_i^2 - y) \) is small enough to consider that \( x_i \) as correct: here we might say that because

\[
\text{abs}(65.734^2 - 4321) = \text{abs}(4320.958 - 4321) = 0.041
\]

is small enough, \( x = x_5 = 65.734 \approx \sqrt{4321} \) is an acceptable solution.
Putting this together, we can write
\[
\Pr[d^{(t+1)} \mid b^{(0)}] = \frac{1 - p}{6} + \frac{p}{3}
\]
where the left-hand side means the probability of visiting web-page \(d\) in step \(t + 1\) having visited web-page \(b\) in step \(t\); this is an example of conditional probability [4].

Of course this is only one way we might visit \(d\). To be complete, we need to take into account that we could follow a link from any web-page to it. We can follow the same reasoning as above, and find the following:
\[
\begin{align*}
\Pr[d^{(t+1)} \mid a^{(0)}] &= \frac{1 - p}{6} + \frac{p}{2} \\
\Pr[d^{(t+1)} \mid b^{(0)}] &= \frac{1 - p}{6} + \frac{p}{3} \\
\Pr[d^{(t+1)} \mid c^{(0)}] &= \frac{1 - p}{6} + \frac{p}{4} \\
\Pr[d^{(t+1)} \mid d^{(0)}] &= \frac{1 - p}{6} + 0 \\
\Pr[d^{(t+1)} \mid e^{(0)}] &= \frac{1 - p}{6} + 0 \\
\Pr[d^{(t+1)}] &= \frac{1 - p}{6} + \frac{p}{5}
\end{align*}
\]

What we want though, is what is the probability of visiting web-page \(d\) in step \(t + 1\) outright not as part of some condition. All this means is we combine all possible ways of visiting \(d\) per the above. Keeping in mind that the sum of \(\Pr[y^{(t)}]\) for all \(y \in V\) must be 1 since these are probabilities, we end up with the following:
\[
\Pr[d^{(t+1)}] = \Pr[d^{(t+1)} \mid a^{(0)}] \cdot \Pr[a^{(0)}] + \Pr[d^{(t+1)} \mid b^{(0)}] \cdot \Pr[b^{(0)}] + \Pr[d^{(t+1)} \mid c^{(0)}] \cdot \Pr[c^{(0)}] + \Pr[d^{(t+1)} \mid d^{(0)}] \cdot \Pr[d^{(0)}] + \Pr[d^{(t+1)} \mid e^{(0)}] \cdot \Pr[e^{(0)}] + \Pr[d^{(t+1)} \mid f^{(0)}] \cdot \Pr[f^{(0)}]
\]
\[
= \left(\frac{1 - p}{6} + \frac{p}{2}\right) \cdot \Pr[a^{(0)}] + \left(\frac{1 - p}{6} + \frac{p}{3}\right) \cdot \Pr[b^{(0)}] + \left(\frac{1 - p}{6} + \frac{p}{4}\right) \cdot \Pr[c^{(0)}] + \frac{1 - p}{6} \cdot \Pr[d^{(0)}] + \left(\frac{1 - p}{6} + 0\right) \cdot \Pr[e^{(0)}] + \left(\frac{1 - p}{6} + \frac{p}{5}\right) \cdot \Pr[f^{(0)}]
\]
\[
= \frac{1 - p}{6} + p \cdot \left(\frac{\Pr[e^{(0)}]}{2} + \frac{\Pr[f^{(0)}]}{3} + \frac{\Pr[f^{(0)}]}{1} + \frac{\Pr[f^{(0)}]}{5}\right)
\]

Each term on the right-hand side multiplies the probability of visiting some web-page \(x\) in step \(t + 1\) having visited another one \(y\) at step \(t\), i.e.,
\[
\Pr[x^{(t+1)} \mid y^{(0)}]
\]
with the probability of actually having visited \(y\) at step \(t\), i.e.,
\[
\Pr[y^{(0)}]
\]
All such terms are added together, because we want to know the probability of any one of them occurring. Doing a similar thing for each web-page, we end up with
\[
\begin{align*}
\Pr[a^{(t+1)}] &= \frac{1 - p}{6} + p \cdot \left(\frac{\Pr[e^{(0)}]}{3} + \frac{\Pr[f^{(0)}]}{2}\right) \\
\Pr[b^{(t+1)}] &= \frac{1 - p}{6} + p \cdot \left(\frac{\Pr[e^{(0)}]}{3} + \frac{\Pr[f^{(0)}]}{5}\right) \\
\Pr[c^{(t+1)}] &= \frac{1 - p}{6} + p \cdot \left(\frac{\Pr[e^{(0)}]}{2} + \frac{\Pr[f^{(0)}]}{5}\right) \\
\Pr[d^{(t+1)}] &= \frac{1 - p}{6} + p \cdot \left(\frac{\Pr[e^{(0)}]}{2} + \frac{\Pr[e^{(0)}]}{3} + \frac{\Pr[e^{(0)}]}{1} + \frac{\Pr[e^{(0)}]}{5}\right) \\
\Pr[e^{(t+1)}] &= \frac{1 - p}{6} + p \cdot \left(\frac{\Pr[e^{(0)}]}{3} + \frac{\Pr[f^{(0)}]}{3} + \frac{\Pr[f^{(0)}]}{5}\right) \\
\Pr[f^{(t+1)}] &= \frac{1 - p}{6} + p \cdot \left(\frac{\Pr[e^{(0)}]}{3} + \frac{\Pr[e^{(0)}]}{1}\right)
\end{align*}
\]
or, generalising this for any given web-page \(x\),
\[
\Pr[x^{(t+1)}] = \frac{1 - p}{n} + p \cdot \left(\sum_{y \in \text{into}(x)} \Pr[y^{(0)}] \right)
\]
Unsurprisingly, this looks a like the formula we started with. The crucial difference is that on the left-hand side we now refer to step \(t + 1\) (values relating to which are unknown), whereas on the right-hand side we only refer to step \(t\) (values relating to which we know). That might not seem like a big difference, but is solves our chicken-and-egg problem: finally, we can compute actual PageRank values.
6.3.2.2 Using an iterative method to compute results

To do so, we draw on use of iterative methods [18] elsewhere in Mathematics. The underlying idea is that instead of computing a solution directly, we compute a sequence of approximate solutions each of which is more accurate (i.e., closer to the real solution) than the last: we start with an approximation $x_0$, then use a function $\delta$ to iteratively compute

$$x_{i+1} = \delta(x_i)$$

forming the next, $(i+1)$-th approximation from the last, $i$-th one. At some $i$-th step the process will converge, meaning $x_i$ is then accurate enough to accept as the solution. This general description should seem similar to what we developed above. Remember that we want to compute

$$Pr[x^{(i+1)}] = \frac{1 - p}{n} + p \cdot \left( \frac{\sum_{\text{from}(y)} Pr[y^{(0)}]}{|\text{from}(y)|} \right),$$

so, following the above, $\delta$ is basically just the PageRank formula with a right-hand side relating to the last, $t$-th approximation and a left-hand side relating to the next, $(t+1)$-th approximation. To adopt the same approach however, we need to resolve one minor difference: in the general description we are only interested in one $x$, whereas we want a solution for each web-pages in the web-graph, i.e., all $x \in V$ or $n$ solutions in total. To cope, we combine all $n$ formula into one using the matrix form of what is then a system of $n$ linear equations [31]:

$$\begin{pmatrix}
Pr[a^{(i+1)}] \\
Pr[b^{(i+1)}] \\
Pr[c^{(i+1)}] \\
Pr[d^{(i+1)}] \\
Pr[e^{(i+1)}] \\
Pr[f^{(i+1)}]
\end{pmatrix} = \left( \frac{1 - p}{n} \right) \cdot 
\begin{pmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
1
\end{pmatrix} + p \cdot 
\begin{pmatrix}
0 & \frac{1}{3} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{3} & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{3} & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{1}{3} \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix} \cdot 
\begin{pmatrix}
Pr[a^{(0)}] \\
Pr[b^{(0)}] \\
Pr[c^{(0)}] \\
Pr[d^{(0)}] \\
Pr[e^{(0)}] \\
Pr[f^{(0)}]
\end{pmatrix}$$

This could be simplified into

$$\bar{x}^{(i+1)} = \delta(\bar{x}^{(i)}) = \bar{b} + A \cdot \bar{x}^{(i)}$$

where $\bar{x}$ is now a column vector capturing the PageRank values for all web-pages rather than just one, while $A$ and $\bar{b}$ are a constant column vector and a matrix respectively (the latter of which is derived from link structure in the web-graph). We already informally decided that an appropriate initial approximation would be

$$\bar{x}^{(0)} = \begin{pmatrix}
\frac{1}{n} \\
\frac{1}{n} \\
\vdots \\
\frac{1}{n}
\end{pmatrix}$$

i.e., an $n$-element column vector whose elements are all $\frac{1}{n}$. So, using this we can proceed to iteratively compute

$$\begin{align*}
\bar{x}^{(1)} &= \delta(\bar{x}^{(0)}) = \bar{b} + A \cdot \bar{x}^{(0)} \\
\bar{x}^{(2)} &= \delta(\bar{x}^{(1)}) = \bar{b} + A \cdot (\bar{b} + A \cdot \bar{x}^{(0)}) \\
&= \bar{b} + A \cdot \bar{x}^{(0)} \\
\vdots \\
\bar{x}^{(l)} &= \delta(\bar{x}^{(l-1)}) = \left( \sum_{i=0}^{l-1} A^i \right) \cdot \bar{b} + A^l \cdot \bar{x}^{(0)}
\end{align*}$$

until at some $l$-th step the process converges, meaning $\bar{x}^{(l)}$ is then accurate enough to accept as the solution.

If you consider the real web rather than the examples presented, the value of $n$ is large and hence the value of $Pr[x]$ for any given $x$ will be very small; numerical precision [27] becomes a problem. How can this problem be resolved?
6.3.2.3 Selecting the probability \( p \)

The only remaining question is what value we should choose for \( p \), the probability which controls the random surfer model: a larger \( p \) means there is a higher probability the random surfer opts to follow a random link, whereas a smaller \( p \) means a higher probability it loads a random web-page. It is quite important to find a balance, because taken to an extreme,

- if \( p \) is too large then the random surfer might never encounter poorly connected web-pages at all (since it is less likely to load them at random), whereas
- if \( p \) is too small then we sort of ignore the web-graph structure (since it is less likely to follow any given link), which of course contradicts the original aim.

A little more formally, \( p \) also influences how quickly the iterative method will arrive at a solution: a larger \( p \) places more emphasis on following links, meaning their influence will spread more quickly, and vice versa. In their original research paper describing a prototype Google web-search system, Sergey Brin and Larry Page quote \( p = 0.85 \); it is less clear what Google use now, but we will stick with this as a reasonable guess.

6.3.2.4 Concrete PageRank values for the example web-graph

Now, finally, we can actually compute the PageRank values themselves. Recall that we now have a formula

\[
\vec{x}^{(t+1)} = \delta(\vec{x}^{(t)}) = \vec{b} + A \cdot \vec{x}^{(t)}
\]

which allows an iterative method of computing a solution; each part of the formula is now specified, in the sense that our example web-graph shown in Figure 6.4b tells us that \( n = 6 \) and

\[
A = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]

Likewise, we know that

\[
\vec{b} = \begin{pmatrix}
\frac{1}{4} \\
\frac{1}{4} \\
\frac{1}{4} \\
\frac{1}{4} \\
\frac{1}{6} \\
\frac{1}{6}
\end{pmatrix}
\]

and

\[
\vec{x}^{(0)} = \begin{pmatrix}
\frac{1}{6} \\
\frac{1}{6} \\
\frac{1}{6} \\
\frac{1}{6} \\
\frac{1}{6} \\
\frac{1}{6}
\end{pmatrix}
\]

We therefore produce the following sequence of approximate but concrete solutions:

<table>
<thead>
<tr>
<th>( \vec{x}^{(0)} )</th>
<th>( \vec{x}^{(1)} )</th>
<th>( \vec{x}^{(2)} )</th>
<th>( \vec{x}^{(3)} )</th>
<th>( \vec{x}^{(4)} )</th>
<th>( \vec{x}^{(5)} )</th>
<th>( \vec{x}^{(6)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Pr[a^{(0)}] )</td>
<td>0.166 0.101</td>
<td>0.090 0.124</td>
<td>0.108 0.133</td>
<td>0.104 0.130</td>
<td>0.104 0.134</td>
<td>0.104 0.138</td>
</tr>
<tr>
<td>( \Pr[b^{(0)}] )</td>
<td>0.166 0.101</td>
<td>0.100 0.133</td>
<td>0.133 0.133</td>
<td>0.134 0.134</td>
<td>0.134 0.134</td>
<td>0.134 0.134</td>
</tr>
<tr>
<td>( \Pr[c^{(0)}] )</td>
<td>0.166 0.124</td>
<td>0.104 0.133</td>
<td>0.133 0.133</td>
<td>0.134 0.134</td>
<td>0.134 0.134</td>
<td>0.134 0.134</td>
</tr>
<tr>
<td>( \Pr[d^{(0)}] )</td>
<td>0.166 0.313</td>
<td>0.238 0.238</td>
<td>0.238 0.238</td>
<td>0.238 0.238</td>
<td>0.238 0.238</td>
<td>0.238 0.238</td>
</tr>
<tr>
<td>( \Pr[e^{(0)}] )</td>
<td>0.166 0.148</td>
<td>0.179 0.176</td>
<td>0.171 0.171</td>
<td>0.171 0.171</td>
<td>0.171 0.171</td>
<td>0.171 0.171</td>
</tr>
<tr>
<td>( \Pr[f^{(0)}] )</td>
<td>0.166 0.214</td>
<td>0.239 0.244</td>
<td>0.241 0.241</td>
<td>0.243 0.243</td>
<td>0.243 0.243</td>
<td>0.243 0.243</td>
</tr>
</tbody>
</table>

We could continue of course, but the values in iteration 4 versus those in iteration 5 already show little change. Using this as a termination criteria, we take \( \vec{x}^{(5)} \) as our solution: these are the probabilities the

---

6 This also explains why in various descriptions, including the original research paper, \( p \) is termed a damping factor (or damping ratio); this term stems from description of a similar feature of physical systems [6].
random surfer will visit each web-page in our web-graph. For instance, it visits \( a \) with probability 0.104, i.e., about 10 percent of the time.

The results presented above relate to Figure 6.4b, i.e., the web-graph that has been amended to cope with any sink web-pages (in this case just \( f \)). Try to

1. reproduce these results yourself, then
2. do the same thing with Figure 6.4a, the original web-graph.

Compare the results with each other: what do you notice, and how do you explain each identifiable difference?

6.4 Putting it all together: using PageRank to produce web-search results

As noted, PageRank is only one component in the Google web-search system: it forms part of the final stage outlined in Section 6.1.2.2 by ranking (or sorting) the set of results produced for some search query. To wrap-up and meet the challenge of explaining how the system works, it makes sense to look at how and where PageRank fits in. To start with, in the offline phase, we

1. use a web-crawler to build a web-graph and summary of content for each web-page, then
2. compute PageRank values for each web-page in the web-graph.

For our limited example, the outcome of this phase can be summarised as follows

\[
\begin{align*}
\{\text{“apple”, “orange”}\} \in a & \quad \Pr[a] = 0.104 \\
\{\text{“apple”, “banana”}\} \in b & \quad \Pr[b] = 0.134 \\
\{\text{“banana”, “orange”}\} \in c & \quad \Pr[c] = 0.110 \\
\{\text{“apple”, “banana”}\} \in d & \quad \Pr[d] = 0.244 \\
\{\text{“apple”, “pear”}\} \in e & \quad \Pr[e] = 0.171 \\
\{\text{“banana”, “pear”}\} \in f & \quad \Pr[f] = 0.238
\end{align*}
\]

in the sense that web-page \( a \) is viewed as containing content relating to the words “apple” and “orange”, and has a PageRank of 0.104. Now, imagine a user gives us a search query \( q \) during the online phase. We need to

1. match web-pages with the query in order to build \( R \), an initial set of results, then
2. take the web-pages in \( R \), and sort them according to their PageRank values to produce the results actually presented for the user.

The first step depends a lot on the type of queries we want to allow, but given the simple form of content our web-pages house, imagine a simple form of query that is just a single word: we want the set of results to give us the web-pages which are most relevant, ranked by their PageRank-decided importance. Consider the following for example:

- If \( q = \text{“orange”} \), the initial set of results is \( R = \{a, c\} \). Given
  \[
  \begin{align*}
  \Pr[a] & = 0.104 \\
  \Pr[c] & = 0.110
  \end{align*}
  \]
  the results are displayed with web-page \( c \) first, then \( a \).

- If \( q = \text{“apple”} \), the initial set of results is \( R = \{a, b, d, e\} \). Given
  \[
  \begin{align*}
  \Pr[a] & = 0.104 \\
  \Pr[b] & = 0.134 \\
  \Pr[d] & = 0.244 \\
  \Pr[e] & = 0.171
  \end{align*}
  \]
  the results are displayed with web-page \( d \) first, then \( c, b \) and finally \( a \).
Even with such simple web-pages and queries, you see the PageRank concept in action. For instance, \( a, b, d \) and \( e \) are all relevant for the query “apple”. Both \( a \) and \( b \) have links to \( d \): each link source can be viewed as attesting to the importance of the link target, meaning \( d \) ends up with a (relatively) high PageRank value and is displayed first.

**Search Engine Optimisation (SEO)** [30] is the art of designing web-pages so they appear in web-search results more often and/or with a higher ranking than normal; this might be used in a marketing strategy, where the overall goal is that users visit the web-page more often.

Various acceptable and unacceptable SEO methods exist: do some research into both sides of this arms race, i.e.,

1. how web-page owners might try to inflate their PageRank and hence get ranked higher in a set of results, and
2. how Google identify and eliminate unfair SEO practices which skew their results and, arguably, devalue PageRank.

Given a small example, it is hard to see the value PageRank gives in general. There are two related take-away points from this Chapter. First, once \( n \) grows large enough, a huge number of web-pages will always be deemed relevant for any given query. Put another way, without PageRank we would almost be back to square one: there would be so many web-pages like \( a, b \) and \( e \) that match, without PageRank to help us we would be tasked with picking out \( d \) by hand. Second, given the benefit PageRank provides, you may have previously thought of it as complex or even magic in some way (if you knew it existed at all). In reality however, we have explained more or less the entire thing by taking a 2-step strategy:

1. model various problems in a way we can more easily understand and reason about, and
2. apply fundamental but fairly simple Mathematics (i.e., probability and graph theory, and iterative methods) and Computer Science (i.e., algorithms) to produce solutions.

Although the techniques themselves might not be applicable to all problems, the general strategy is. This makes PageRank a great example, and advert, for Computer Science as a whole.
BIBLIOGRAPHY


Part II

Cryptography
CHAPTER 7

USING SHORT PROGRAMS TO MAKE
AND BREAK HISTORICAL CIPHERS

It might seem hard to imagine, but in early 1587 Mary Stewart (or Mary Queen of Scots) [6] was sitting in a jail cell, most likely cursing the subject of cryptography. Around a year or so beforehand, Mary was imprisoned in Chartley Hall as a result of her increasingly tense relationship with the then Queen, Elizabeth I. Having been placed under close observation, Mary was only able to communicate with her allies using messages smuggled in and out of the jail inside beer barrels. However, to prevent the messages being used against her should they be discovered, Mary used a system of encoding that substituted characters and common words with a variety of symbols. This gave Mary enough confidence that, while still in jail, she instigated a plot to overthrow Elizabeth: what we now know as the Babington Plot is named after her chief conspirator, Anthony Babington. Unbeknown to them, messages sent between Mary and Babington were being intercepted by a double agent, then analysed by an espionage team established by Sir Francis Walsingham. The messages were processed by Thomas Phelippes, who carefully copied their content before resealing and sending them to their intended recipient. Phelippes eventually worked out the encoding system used, and the plot was uncovered when Phelippes took a real message from Mary and added a forged postscript that asked Babington for the names of the conspirators. The resulting proof of conspiracy led to the arrest of Babington and subsequent execution of Mary.

Historical significance aside, the same underlying issue in this story has reoccurred again and again. In 2006, for instance, the famed Mafia boss Bernardo Provenzano was captured for much the same reason as Mary Stewart was executed. Provenzano’s encoded notes, including orders to his henchmen and so on, were decoded by police who were then able to capture him. The issue, basically, is that while both Stewart and Provenzano clearly understood the need for secrecy, they lacked the range of formal techniques offered by modern cryptography.

Almost all terms used above can be translated into a modern setting. We still talk about encoding messages, but now call this encryption, a technique that is used to ensure the secrecy of messages. An unencrypted message is called a plaintext, whereas an encrypted message is called a ciphertext. In the context of Mary Stewart, a plaintext was formed from characters of the English alphabet while a ciphertext was formed from an alphabet of abstract symbols. We still use the term alphabet to describe the symbols used to form plaintext and ciphertext, but they are more likely to be sequences of bits or bytes that can be processed by a computer. Put another way, we still think about a sender and a receiver who are communicating messages, but the message is more likely to be an electronic file communicated over a network such as the Internet than written on paper. As a result, either the sender and/or receiver might also be a computer rather than a human. Finally, interception and attempted decryption of messages also have an analogy in communication as we use it today. We call the party trying to do this an attacker or adversary, while the art of trying to decrypt a message that should be kept secret would be termed cryptanalysis. Whereas the encoding methods used by Stewart and Provenzano were eventually broken via cryptanalysis of some sort, the design of modern encryption schemes is intended to resist even the most capable attacker.
Figure 7.1: The message that uncovered the Babington Plot: masquerading as part of a message from Mary Stewart, the postscript asks Babington to reveal the names of the conspirators using the broken cipher (public domain image, source: http://en.wikipedia.org/wiki/File:Babington_postscript.jpg).
As with any topic that has evolved in this way, study of basic techniques can still offer insight into modern practice. Our aim here is to look at two types of historical cipher (i.e., methods of encryption) in a very practical way. In each case we describe how the cipher works, how it can be broken via cryptanalysis, and how both aspects can be reproduced using single-line (or at least very short) BASH commands.

7.1 Shift ciphers

Apparently, Julius Caesar knew about cryptography. Chronicling the life of the Roman leader in De Vita Caesarum, Divus Iulius, Suetonius wrote:

If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others.

This should sound familiar: Caesar was doing something similar to Mary Stewart in the sense that he was translating characters in a plaintext message into other characters to form a ciphertext message. We use the name shift cipher to describe the method of translation used by Caesar. ROT13, a modern day equivalent of the same method, is still used to hide solutions to puzzles in newspapers and so on.

7.1.1 Encryption and decryption

7.1.1.1 3-place shifts

We can describe the method used by Caesar using two functions

Encryption works by examining each character of the plaintext message in turn. For example, where we see ‘a’ and want to encrypt it, we use E_\text{Exc}(‘a’) to look-up the result ‘d’. Or, in the other direction, if we want to decrypt ‘d’ then we use D_\text{Dec}(‘d’) to look-up the result ‘a’. The term shift cipher comes from the fact that what we are actually doing is shifting the alphabet around: in this case the shifting moves characters by three places.

To demonstrate the process on a larger example, we need something to act as plaintext. As in Chapter 2, we use text downloaded from Project Gutenberg http://www.gutenberg.org/

There are numerous worthy examples we could use, but opt for The Merchant of Venice by Shakespeare. Using ‘△’ to make it clear where the spaces are, encrypting the plaintext

‘t’ ‘h’ ‘e’ ‘△’ ‘m’ ‘e’ ‘r’ ‘c’ ‘h’ ‘a’
‘n’ ‘t’ ‘△’ ‘o’ ‘f’ ‘△’ ‘v’ ‘e’ ‘n’ ‘i’
‘c’ ‘h’

yields the ciphertext

‘w’ ‘k’ ‘h’ ‘△’ ‘p’ ‘h’ ‘u’ ‘f’ ‘k’ ‘d’
‘q’ ‘w’ ‘△’ ‘r’ ‘i’ ‘△’ ‘y’ ‘h’ ‘q’ ‘l’
‘f’ ‘h’

One can imagine the cipher being like a big codebook; to encrypt or decrypt messages, Caesar probably employed a trusted slave to apply the translation using tables in the codebook for reference. Obviously this is very tedious, so an important question to resolve (given we lack a slave, but on the other hand
have computers) is whether encryption and decryption might be automated. One way would be to write a
dedicated program for the task; since we want to focus on the concepts rather than teach programming,
we will instead try to automate the process using **BASH** commands. First we need some plaintext to encrypt.
The text for *The Merchant of Venice* will do fine: we save it as the file `A.txt` before translating all characters
to lower-case so that our job is made a little easier (since we no longer need to consider the upper-case
characters as distinct):

```
bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$
```

Although we aim to process the whole file, this is tricky to demonstrate because of the length. Therefore,
we focus on a seven line extract starting at line #274:

```
bash$ cat B.txt | tail -n +274 | head -n 7
antonio. in sooth, i know not why i am so sad.
  it wearies me; you say it wearies you;
  but how i caught it, found it, or came by it,
  what stuff 'tis made of, whereof it is born,
  i am to learn;
  and such a want-wit sadness makes of me
  that i have much ado to know myself.
bash$
```

To encrypt and decrypt we employ the **tr** command, which we already saw in Chapter 2. Recall that **tr**
reads lines of input, translates characters in those lines based on rules supplied by the user, and writes the
result as output. As before, the rule is given by two sequences: all instances of a given character in the first
sequence are translated into the corresponding character in the second sequence. This is easy to see with a
simple example. Imagine we want to translate from `⟨a, b, c⟩` into `⟨g, h, i⟩`:

```
bash$ cat | tr [a-c] [g-i]
abcdef
ghidef
bash$
```

The input “abcdef” is typed by the user; the first three characters (‘a’, ‘b’ and ‘c’) match those in the first
sequence and are thus translated by **tr** into the corresponding characters in the second sequence (‘g’, ‘h’
and ‘i’). Notice that the next three characters (‘d’, ‘e’ and ‘f’) do not match any in the first sequence so
are passed through unaltered. Using this technique, we can encrypt and decrypt files using a 3-place shift
cipher as follows:

```
bash$ cat B.txt | tr [a-cd-z] [d-za-c] > C.txt
bash$ cat C.txt | tr [d-za-c] [a-cd-z] > D.txt
bash$
```

In the first command we start with `B.txt` (the original file turned into lower-case), feed this to **tr**, and
direct the output into the file `C.txt` which represents the ciphertext. The rule for translation is given by the
sequences `[a-cd-z]`, which are short-hand for

\[ ⟨'a', 'b', 'c', 'd',..., 'w', 'x', 'y', 'z') \]

i.e., the standard alphabet, and `[d-za-c]`, meaning

\[ ⟨'d', 'e', 'f', 'g',..., 'z', 'a', 'b', 'c') \]

In other words, the first command reads input and translates ‘a’ into ‘d’, ‘b’ into ‘e’, ‘c’ into ‘f’, ‘d’ into ‘g’
and so on. In the second command we reverse the process by starting with `C.txt` (the ciphertext), feed this
to **tr** (where the sets for translation are reversed), and direct the output into the file `D.txt`. Inspecting the
relevant lines in the encrypted and decrypted files shows the result:

```
bash$ cat C.txt | tail -n +274 | head -n 7
dqwrqlr. lq vrrwk, l nqrz gwr zkb l dp vz vdg.
  lw zhdulhv ph; brx vdb lw zhdulhv brx;
exx krz i fdujkw lw, izxq lw, ru fdpb eb lw,
  zkdw vwxii 'wlv pddq ri, zkhubri lw lv eruq,
  l dp wzvduq;
  dqg pxfsk d zdgw-zlw vdgqhvv pdnhv ri ph
  wkw i kdyb pxfsk dgr wr nqrz pbvhoi.
bash$ cat D.txt | tail -n +274 | head -n 7
antonio. in sooth, i know not why i am so sad.
  it wearies me; you say it wearies you;
  but how i caught it, found it, or came by it,
  what stuff 'tis made of, whereof it is born,
  i am to learn;
  and such a want-wit sadness makes of me
  that i have much ado to know myself.
bash$
```
Just by looking at the text, we can see the 3-place shift cipher at work. For example, the ‘a’ of “antonio” in B.txt has become a ‘d’ in C.txt. The file D.txt which is the decryption should match the original file B.txt which was encrypted. Although the decrypted extract looks the same as the original, we can prove that the whole file is the same using the diff command to perform a file comparison:

```
bash$ diff B.txt D.txt
bash$ echo ${?}
0
bash$
```

The lack of output (and exit code, as printed by the echo command) indicate there are no differences, i.e., the encryption and subsequent decryption were successful.

```bash
Implement
(task #32)
```

This is a simple task: take any other plaintext of your choice (e.g., a text file you wrote, or another one from Project Gutenberg) and reproduce the steps above to encrypt then decrypt it.

### 7.1.1.2 \( k \)-place shifts

At the moment, the Enc and Dec functions must be kept secret: since they are fixed to performing 3-place shifts, if the attacker can work out their behaviour he can decrypt all messages encrypted with them. This is often called “security through obscurity” and is frowned upon in modern cryptography. More usually, we try to build schemes whose security relies only on the secrecy of some key rather than the actual method of encryption. This philosophy was articulated by Auguste Kerckhoffs in the late 1800s, and is often called the **Kerckhoffs Principle** [5].

```bash
Research
(task #33)
```

Kerckhoffs actually cited six design principles for ciphers. Find out what the others are. Based on how we communicate today compared with 1883, do you think all the principles are still relevant? If any are no longer relevant, what has changed to make this so? Are there any new principles you could add to the list?

Fortunately, we can easily generalise Enc and Dec by adding a key parameter called \( k \) that allows more general \( k \)-place shifts; the resulting functions are written Enc\(_k\) and Dec\(_k\). To describe the generalised functions easily, we first need a way to convert characters to numbers. We could use ASCII, as in Chapter 2, but to make things easier we will use the functions

\[
\text{Ord}(x) = \begin{cases} 
0 & \text{if } x = 'a' \\
1 & \text{if } x = 'b' \\
2 & \text{if } x = 'c' \\
3 & \text{if } x = 'd' \\
\vdots & \\
22 & \text{if } x = 'w' \\
23 & \text{if } x = 'x' \\
24 & \text{if } x = 'y' \\
25 & \text{if } x = 'z'
\end{cases}
\quad \text{CHR}(x) = \begin{cases} 
'a' & \text{if } x = 0 \\
'b' & \text{if } x = 1 \\
'c' & \text{if } x = 2 \\
'd' & \text{if } x = 3 \\
\vdots & \\
'w' & \text{if } x = 22 \\
'x' & \text{if } x = 23 \\
'y' & \text{if } x = 24 \\
'z' & \text{if } x = 25
\end{cases}
\]

where \text{Ord}(x) takes a character \( x \) and returns the associated number, and \text{CHR}(x) does the reverse by taking a number \( x \) and returning the associated character.

To describe what is going on, it is easier to view the scheme as another application of the **modular arithmetic** [7] we first touched on in Chapter 2. This topic appears frequently in cryptography, so it makes sense to describe it more fully now. Fortunately, we have a nice analogy to help: we are doing what is sometimes called clock arithmetic. Imagine someone says to you “the time is now ten o’clock; I will meet you in four hours”, what time do they mean? You might say “two o’clock” instinctively, or maybe getting this answer by looking at a clock face such as Figure 7.2 and moving clockwise 4 hours starting at 10 in order to get to 2. More formally we write this as

\[(10 + 4) \equiv 2 \pmod{12}\]

so that mod can be read to mean “remainder after division”: 14 and 2 are equivalent modulo 12 because 14 divided by 12 gives the remainder 2. Of course, we might alter our diagram so it describes a 24-hour clock
An analogue clock face showing the time three o’clock.

face instead; this would illustrate that the equivalence we are discussing “wraps around”. We can see this easily by looking at a number line detailing $x$ and $x \pmod{12}$:

$x = \cdots -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 \cdots$

$x \pmod{12} = \cdots 10 11 0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 \cdots$

Notice that as you would expect $13 \equiv 1 \pmod{12}$, i.e., the time 13 : 00 means one o’clock, and also that two useful facts pop out of looking at the number line:

1. Taking any $x$ and adding 12 is the same as adding 0 because $12 \equiv 0 \pmod{12}$. For example $1 + 12 \equiv 1 \pmod{12}$.

2. If $x$ is negative, $x \pmod{12}$ is given by $12 + x$. For example $-2 \pmod{12}$ is given by $12 + (-2) = 10$.

To answer the question “the time is now four o’clock, what time was it six hours ago?” we know $4 - 6 = -2$ and hence $-2 \equiv 10 \pmod{12}$.

While talking about clock faces, 12 is the natural modulus to use. In general however, we can select (more or less) any integer modulus we like. What links this fact to the way we have performed encryption and decryption so far, is that both $E_{k}(x)$ and $D_{k}(x)$ can be described using modular arithmetic by setting the modulus to 26. More specifically, we can write

$E_{k}(x) = \text{CHR}([\text{ORD}(x) + k] \pmod{26})$

$D_{k}(x) = \text{CHR}([\text{ORD}(x) - k] \pmod{26})$

This is starting to get a bit complicated, so it makes sense to look at what is going on in more detail. Say we select $k = 3$ to mimic our original functions, and want to encrypt the plaintext ‘a’:

1. First turn ‘a’ into a number using $\text{ORD}('a') = 0$.

2. Next add $k$ to get $0 + 3 = 3$, and then reduce it modulo 26 to get $3 \pmod{26} = 3$.

3. Finally, translate the number back into a character to get the result $\text{CHR}(3) = 'd'$.

In short, we have encrypted ‘a’ into the ciphertext ‘d’. What about decryption? Essentially the same technique applies, meaning we can decrypt ‘d’ as follows:

1. First turn ‘d’ into a number using $\text{ORD}('d') = 3$.

2. Next subtract $k$ to get $3 - 3 = 0$, and then reduce it modulo 26 to get $0 \pmod{26} = 0$.

3. Finally, translate the number back into a character to get the result $\text{CHR}(0) = 'a'$.

Notice that by opting to include the mod 26 operation we have ensured that when adding or subtracting $k$ to a number, the result will “wrap around” to the start or end of the alphabet when turning the number back into a character. We can see this more clearly with another example, this time we encrypt the plaintext ‘x’:
1. First turn ‘x’ into a number using \( \text{Ord}(‘x’) = 23 \).

2. Next add \( k \) to get \( 23 + 3 = 26 \), and then reduce it modulo 26 to get \( 26 \mod 26 = 0 \).

3. Finally, translate the number back into a character to get the result \( \text{chr}(0) = ‘a’ \).

The corresponding decryption of the ciphertext ‘a’ is as follows:

1. First turn ‘a’ into a number using \( \text{Ord}(‘a’) = 0 \).

2. Next subtract \( k \) to get \( 0 - 3 = -3 \), and then reduce it modulo 26 to get \( -3 \mod 26 = 23 \).

3. Finally, translate the number back into a character to get the result \( \text{chr}(23) = ‘x’ \).

One can view all this as a mechanism to generate new codebooks. Whereas before Caesar just had one codebook, he can now generate a new one for each value of \( k \). Either way, we now have a situation where only \( k \) need be kept secret. An attacker might know the method of encryption but without the value of \( k \), he cannot decrypt messages. Better still, we are free to select a different \( k \) for each message so that even if an attacker recovers the key for one message, he still would not necessarily know the key for another message.

### 7.1.2 Cryptanalysis

We already fixed one problem with the initial 3-place shift cipher by making it into a general \( k \)-place version; this gave us a cipher that at least made some attempt to comply with the Kerckhoffs Principle. You might have guessed, however, that this is not really enough, and that the cipher is still easy to cryptanalyse:

- Although we could have included space as a character in the plaintext alphabet, we did not do so. As such, the word structure of the plaintext is retained in the ciphertext when we encrypt it. We can identify words in both simply by spotting where the space characters are.
- A given character is translated the same way every time it occurs. For example if we encrypt one ‘a’ in the plaintext into a ‘d’ in the ciphertext, we know that all occurrences of ‘a’ will encrypt to ‘d’.
- Even though we are free to select \( k \), there are not that many choices. Selecting \( k = 27 \) and shifting the alphabet by 27 places, for example, is the same as \( k = 1 \) because of the “wrap around” effect. So basically there are only 26 possible keys. In fact with \( k = 0 \) the ciphertext is the same as the plaintext, so actually there are probably more like 25 useful keys.

How can we use these features to cryptanalyse the cipher, for example to decrypt an encrypted message we have intercepted? When presented with the ciphertext we might first employ a brute-force attack [1]. This means that we just try all the keys. Although laborious, there are only 26 of them so if the key is worth finding it could be worthwhile. Caesar would probably just have 26 slaves try one key each in order to speed things up. The problem is, how do we know when we have got the right key? In our case, we have been assuming that the underlying plaintext is English so as soon as a trial decryption yields text which forms English words we know we have got the right one.

Imagine either we do not have the time to perform the brute-force attack or are just lazy and want a short cut. The next thing we could do is apply a technique called frequency analysis [4]. The basic idea is that for a language like English, single characters and combinations of characters occur with varying frequencies. In fact, given a large enough sample, the frequencies are characteristic of any text written in that same language: this means, for example, we can make a good guess if the text is English (versus German say), or predict the frequency with which characters occur in one text given frequencies in another (if they are written in the same language).

By retrieving a fresh copy of The Merchant of Venice

bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$

we can use some further BASH commands to demonstrate this:

bash$ cat B.txt | fold -c -w 1 | grep [[:alpha:]] | sort | uniq -c | paste -s
7789 a 1721 b 2427 c 3748 d 1170 e
2174 f 1724 g 6024 h 7623 i 256 j
799 k 4227 l 2781 m 6580 n 8816 o
1474 p 60 q 6032 r 6395 s 8733 t
3166 u 1602 v 2284 w 180 x 2671 y
117 z
bash$
The last command in particular needs some explanation. The first part of the command pipeline feeds the file B.txt (the original text turned into lower-case) into fold which splits each line into one character per-line. Since this is the first time we have used fold, a simpler example might be helpful:

```
bash$ cat | fold -c -w 1
abcd
a
b
c
d
bash$
```

Notice that the single line input of four characters typed by the user has been split into four lines each of one character. As used originally, the output of fold is then filtered to leave only alphanumeric characters (or letters and numbers), then sorted using sort and fed to uniq to count how many duplicates appear (i.e., how many times a given character exists). Finally we format the output nicely using paste, and after all that effort, hope something useful came out! The interesting thing is that some occur much more frequently than others. For example ‘e’ is the most used by some distance, followed by ‘o’, ‘t’, ‘a’ and so on. You can view the result above as being somewhat indicative of English in general. Okay, it is Shakespearean English but more or less the same frequencies occur in modern text as well, bar the odd “ye olde” or two.

Implement (task #34)

The claim above is that our character frequencies are indicative of the language. Test this claim by performing the same type of analysis on a text file written in something other than English; Project Gutenberg offers a number of French books for instance.

So imagine someone hands us some ciphertext and challenges us to tell them the key it was encrypted with. To simulate this, we will retrieve a copy of A Midsummer Night’s Dream again by Shakespeare:

```
bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1710.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$
```

and imagine that someone takes the plaintext B.txt, selects a k and then encrypts B.txt to give the ciphertext C.txt. Our task is to determine the unknown k given only C.txt. We call this scenario a ciphertext only attack [3] since we are given (rather than choose) the ciphertext, and also do not get the corresponding plaintext. We do not present the file C.txt, but you could obtain your own version by encrypting some text and then following the same analysis we do; the results you obtain may be slightly different, but the general method should still work.

We first examine what happens if we apply the same frequency analysis to the ciphertext (rather than the plaintext as above). All the cipher does is shift around the alphabet, basically just rearranging the table of frequencies. Employing the method as above, we get a different set of numbers; this should not be surprising since this was a different plaintext originally:

```
bash$ cat C.txt | fold -c -w 1 | grep [:[:alpha:]] | sort | uniq -c | paste -s
5195 a 6673 b 1411 c 121 d 5114 e
5237 f 7679 g 2763 h 783 i 1886 j
169 k 2176 l 16 m 5984 n 1310 o
1751 p 3173 q 10178 r 1528 s 1372 t
5835 u 5560 v 87 w 1684 x 3674 y
2544 z
bash$
```

However, the crucial thing to notice is that the relative frequency of the characters in the new results should match those in the old results. For example, in this case ‘r’ is the most used by some distance, followed by ‘g’, ‘b’, ‘n’ and so on. If we consider ‘r’, ‘g’, ‘b’ and ‘n’ to be the only reasonable way one could have encrypted ‘e’ then we narrow the range of possible keys to $k = 13$, $k = 2$, $k = 23$ or $k = 9$.

Consider some more evidence in the shape of the eleven line extract of ciphertext starting at line #959:

```
bash$ cat C.txt | tail -n +959 | head -n 11
boreba. v cenl gur tivr vz gur. v sabj n max jurer gur jyvq gulsr obyfj.
jurer bkyvcf naq gur abqvat jvbyrg tehjf.
dhqr bire-pnabcvqy jugu yfhpbhf jbbqoar,
jvgu sfjrrg zhsx-ebfrf, naq jugu rtyngapvar:
gurer fyrrcf gynavn fbzrgvz bs gur avtug,
yhvy'q va gurfr sybjref jugu qnarpf naq qvyytug;
naq gurer gur fanx guebjf ure ranzrry'q fava,
jrrq jvqr rabhtu gb jenc n snvel va;
naq jugu gur whvpr bs guvf v'yy fjernx ure rlf.
```

Consider some more evidence in the eleven line extract of ciphertext starting at line #959:
Figure 7.3: An example of the “dancing men” used as a cipher in The Adventure of the Dancing Men (public domain image, source: http://en.wikipedia.org/wiki/File:Dancing_men.png).

naq znxr ure shyy bs ungcrshy snagnfvrfl.

bash$

On the second line of the output, we find a 1-character word ‘v’. Not many of these exist in English, so basically we know either ‘a’ or ‘i’ must encrypt to ‘v’ which suggests $k = 21$ or $k = 13$. By this point, $k = 13$ is looking like a good choice: we could now try to decrypt the ciphertext using this key, and see what we get. Selecting $k = 13$ means translating ‘a’ to ‘n’ and so on, which means we just need to select the right sequences for tr and we are done:

bash$

On the second line of the output, we find a 1-character word ‘v’. Not many of these exist in English, so basically we know either ‘a’ or ‘i’ must encrypt to ‘v’ which suggests $k = 21$ or $k = 13$. By this point, $k = 13$ is looking like a good choice: we could now try to decrypt the ciphertext using this key, and see what we get. Selecting $k = 13$ means translating ‘a’ to ‘n’ and so on, which means we just need to select the right sequences for tr and we are done:

bash$

Even if we did not have diff to confirm the result as follows

bash$

it would be fairly bad luck to have selected the wrong key and get perfect Shakespearean as output, so we can conclude $k = 13$ was the right key. If we had intercepted C.txt from Caesar then we could decrypt his messages and get the jump on him the next time he invaded our country.

7.2 Substitution ciphers

After generalising the 3-place shift cipher into a $k$-place version and still failing to produce something which can secure our messages, the next step is something called a substitution cipher [11]. In The Adventure of the Dancing Men, Sir Arthur Conan Doyle had his character Sherlock Holmes, the arch detective, encounter a cipher of this type. Holmes and Watson are confronted with a number of pictures of dancing men:

> These hieroglyphics have evidently a meaning. If it is a purely arbitrary one, it may be impossible for us to solve it. If, on the other hand, it is systematic, I have no doubt that we shall get to the bottom of it.

Of course, the pictures are symbols which encode a message; Holmes and Watson eventually decode the messages and solve yet another case [12]. So if substitution ciphers are important enough for the great Sherlock Holmes to worry about, then they are good enough for us as well.

7.2.1 Encryption and decryption

Imagine we have a sequence or list of characters

$$A = \langle 'a', 'b', 'c', 'd' \rangle.$$

Given such a sequence, the concept of a permutation [9] is central: if we permute the elements in a source sequence, we basically reorder them to produce a target sequence. This means that each element occurs once, but there is some translation from the source to the target sequence. We can describe an example permutation $P$ as follows:

$$P(X) = \langle X_1, X_2, X_3, X_0 \rangle.$$
Put more simply, if we apply $P$ to some source sequence $X$ then the target sequence we get back has $X_1$ as the 0-th element, $X_2$ as the 1-st element, $X_3$ as the 2-nd element and $X_0$ as the 3-rd element. Applying $P$ to the sequence $A$ above therefore gives us

$$P(A) = \langle 'b', 'c', 'd', 'a' \rangle.$$  

Since $P$ is simply reordering the elements, we could write down the inverse permutation $P^{-1}$ which performs translation in the opposite direction. In this case

$$P^{-1}(X) = (X_2, X_0, X_1, X_3),$$

which means that

$$P^{-1}(A) = \langle 'd', 'a', 'b', 'c' \rangle$$

and, more importantly, that

$$P^{-1}(P(A)) = \langle 'a', 'b', 'c', 'd' \rangle = A.$$  

The basic idea is that the key for a substitution cipher is a permutation; we still write $k$ as the key just for continuity, so you can think of it as a name (or index) that identifies the permutation we use among all those available. The permutation tells us how to translate characters from a source sequence (i.e., the plaintext alphabet) into characters in a target sequence (i.e., the ciphertext alphabet). Of course, one can view the shift cipher as a particular form of permutation. However, the fact that the permutation is of such a particular form makes the cipher weak. We already saw that there are only 26 possible keys, which is far from ideal: a substitution cipher generalises the idea, relaxing the need for a particular form of permutation and allowing any permutation at all.

How many possible keys would there be using this generalised approach? We can get the answer by looking at a more general question: say we have an $n$-element source sequence, how many different permutations of those elements are there? The answer is $n$ factorial or

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1.$$  

Why is this the case? We start with $n$ elements in the source sequence, so there are $n$ choices for the first element in the target sequence. When we remove one of those choices, there are $n-1$ choices left for the second element in the target sequence, $n-2$ choices left for the third element and so on. So given we have $n = 26$ possible characters in our plaintext and ciphertext alphabets there are a total of

$$26! = 403291461126605653584000000$$

possible keys for the substitution cipher. This is now too big to allow searching for the key by brute-force: we need to think a bit harder if we want to break this scheme.

Each key specifies a different permutation; you can think of this as each $k$ specifying a different, secret pair of encryption and decryption functions. Consider an example and imagine that some $k$ specifies the functions:

$$\begin{align*} 
\text{ENC}_k(x) = \{ & \ 'z' \text{ if } x = 'a' \\ & \ 'y' \text{ if } x = 'b' \\ & \ 'x' \text{ if } x = 'c' \\ & \ 'w' \text{ if } x = 'd' \\ & \ldots \\ & \ 'd' \text{ if } x = 'w' \\ & \ 'c' \text{ if } x = 'x' \\ & \ 'b' \text{ if } x = 'y' \\ & \ 'a' \text{ if } x = 'z' \} \\
\text{DEC}_k(x) = \{ & \ 'a' \text{ if } x = 'z' \\ & \ 'b' \text{ if } x = 'y' \\ & \ 'c' \text{ if } x = 'x' \\ & \ 'd' \text{ if } x = 'w' \\ & \ldots \\ & \ 'w' \text{ if } x = 'd' \\ & \ 'x' \text{ if } x = 'c' \\ & \ 'y' \text{ if } x = 'b' \\ & \ 'z' \text{ if } x = 'a' \} 
\end{align*}$$

Although these functions are not complete (so they can fit on a page), the general idea acts as a mechanism to generate a codebook; as was the case of the $k$-place shift cipher, security is based on knowledge of $k$ rather than the actual method of encryption.

Automating this encryption method is almost as simple as for the shift cipher. We again fetch the text of *The Merchant of Venice* text and save it as `A.txt` before translating all characters to lower-case:

```bash
bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$
```

Next we set up two strings, which essentially define source and target sequences, and hence the permutation we want to use. This means we can again use `tr` to perform the encryption and decryption:
bash$ S='abcdefghijklmnopqrstuvwxyz'
bash$ T='zyxwvutsrqponmlkjihgfedcba'
bash$ cat B.txt | tr ${S} ${T} > C.txt
bash$ cat C.txt | tr ${T} ${S} > D.txt
bash$

What does this mean? Essentially what we are saying is that the \( i \)-th character in the source sequence \( S \) should be translated into the \( i \)-th character in target sequence \( T \) so, for example, in this case an ‘a’ encrypts to a ‘z’.

As before, we can focus on a seven line extract starting at line \#274 to show the process is working as expected:

bash$ cat C.txt | tail -n +274 | head -n 7
zmglmrl. rm hllgs , r pmld mlg dsb r zn hl bzw.
rg dvzirvh nv ; blf hzb rg dvzirvh blf;
yf g sid r xzf tsg rg , uf mwg rg , li zxnv yb rg.
dxg bpqfuu 'gph nzew lu , dxvlu r g rh ylim ,
r zn gl ovzim ;
zwm hfx z dzmg-drzg hzwvwhh nzpvh lu nv
gszg r szev nxz suicide gl pmld nhbouu.
bash$ cat D.txt | tail -n +274 | head -n 7
antonio. in sooth , i know not why i am so sad.
it wearies me; you say it wearies you;
but how i caught it, found it, or came by it,
what stuff 'tis made of, whereof it is born ,
i am to learn;
and such a want-wit sadness makes of me
that i have much ado to know myself.
bash$

or use \texttt{diff} to show that the decrypted file is the same as the original plaintext:

bash$ diff B.txt D.txt
bash$ echo ${?}
0
bash$

### 7.2.2 Cryptanalysis

Since we have improved upon the shift cipher using the stronger permutation cipher, we need need to consider better forms of cryptanalysis to attack it: we cannot say it prevents the previous attack, so therefore is secure! In particular, we need to consider an improved form of frequency analysis that relies on further properties of language.

The concepts of \textbf{bigrams} and \textbf{trigrams} are special cases of something called an \textit{n-gram} \cite{8}. An \textit{n-gram} is a sub-sequence of length \( n \) taken from some other sequence; a bigram is the case where \( n = 2 \) and a trigram is the case where \( n = 3 \). Imagine we want to find all the bigrams of

\[
A = \langle 'a', 'b', 'c', 'd' \rangle.
\]

We can formulate a solution by saying we want all the sub-sequences which look like

\[
\langle A_i, A_{i+1} \rangle.
\]

That is, we want all the sub-sequences formed by taking the \( i \)-th element and the \( (i+1) \)-th element of \( A \). Of course we cannot select element three as the \( i \)-th element since the \( (i+1) \)-th element would not be valid, but apart from this the sub-sequences we can form are

\[
\langle 'a', 'b' \rangle, \langle 'b', 'c' \rangle, \langle 'c', 'd' \rangle.
\]

You can think of a text file as just a long string, so it is easy to imagine that we could work out all the bigrams within such a file. How can we do this in practical terms? Using only existing \texttt{BASH} commands demands a cunning approach; imagine we have a file called \texttt{A.txt} which has one character per-line:

bash$ cat > A.txt
a
b
c
d
bash$

Now imagine we take the file and paste it next to itself; we can achieve this easily using the \texttt{paste} command:
Believe it or not, we are almost there. All we need now do is “skew” the second column by one character. That is, if we skewed the column so that we started at ‘b’ rather than ‘a’ we would (more or less) have the bigrams one per-line. To perform the skewing, we use tail to take A.txt and copy the content starting at the second line. Pasting the result, which we call B.txt, alongside the original A.txt gives:

```
bash$ tail -n +2 A.txt > B.txt
bash$ paste -d ' ' A.txt B.txt
a b
b c
c d
d
```

Of course, we might want to eliminate the last line (this is the result of including an “invalid” index) but other than that we can build a list of all bigrams in A.txt. The case for trigrams is quite similar but we need to skew the original file by two characters and include that in our paste command as well. Suppose we apply this to *The Merchant of Venice*. First we retrieve the text and turn it into lower-case as usual; then we split the characters from B.txt into a file called E.txt where there is one character per-line and finally skew this file by one and two lines to get F.txt and G.txt:

```
bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$ tail -n +2 E.txt > F.txt
bash$ tail -n +3 E.txt > G.txt
bash$
```

Now we are ready to construct the bigrams and trigrams. Using paste as above we take the files E.txt, F.txt and G.txt and paste them into place next to each other. Then we use grep to throw away any invalid lines. We do this by specifying that we only want lines with two or three alphabetic characters on them (for the respective bigram and trigram case). Finally we remove the inter-character spacing using tr and get the bigrams and trigrams in H.txt and I.txt:

```
bash$ paste -d ' ' E.txt F.txt > H.txt
bash$ paste -d ' ' E.txt F.txt G.txt > I.txt
bash$
```

Next we can apply a similar approach to analysis of the bigram and trigram frequencies as we previously applied to single character frequencies. We take the input file, sort it using sort and feed the output to uniq to count how many duplicates exists (i.e., how many times a given bigram or trigram exists). Unlike single character frequencies where there were not many (since there are not many characters), there are a huge number of bigrams and trigrams: we feed the result through sort and tail to produce only the 20 most frequent:

```
bash$ cat H.txt | sort | uniq -c | sort -n -r | head -n 20 | paste -s
2922 th
1814 an
1592 er
1430 ou
1273 re
1238 or
1181 nd
1136 ha
990 en
974 at
829 es
814 to
789 me
786 ar
782 ve
692 he
688 th
672 ha
663 be
652 ze
649 er
```

If you think about it, the results are what we would expect: “th” is obviously going to occur more often than “tz” for example, and it should not be a surprise that three character words such as “the” and “and” are the most popular trigrams.

Now imagine we play the same game as before. Someone hands us some ciphertext and challenges us to tell them the key it was encrypted with. To simulate this, we first retrieve a fresh copy of *A Midsummer Night's Dream*:

```
bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1710.txt'
bash$ cat A.txt | tr [:upper:] [:lower:] > B.txt
bash$
```

138
and imagine someone takes the plaintext \( B.txt \), selects \( k \) then encrypts \( B.txt \) to give the ciphertext \( C.txt \): our task is again to determine \( k \) given only \( C.txt \). To avoid having to inspect the whole file, we will again focus on the eleven lines of the ciphertext starting at line #959:

```bash
bash$ cat C.txt | tail -n +959 | head -n 11
ylivyz. e xvmo tfii geri et ai.
e czqv m lnzq qfivi tfii qeqj tfosi lbyqu.
qfivi ybhesu mzj tfii zyjohn rehlyg qyvqy.
wsen yriv-kmxzeij qeqf bsykeusy qyvlezi.
qeqf uqif asuc-veyui, mzj qeqf lgbhezeteii;
273p tfii qeqf qezyui "tfii qeqf:"
shbb[e] qeqf tfii bbyqyv qeqf jmkjui mzj jibeqft:
mzj tfivi tfii ufsmi tfqyuqy siv izmaibb’j ucez.
qeqj qeqj izysgfy ty qeqm m hneve ez;
mzj tfivi tfii ufsmi tfqyuqy siv izmaibb’j ucez.
```

By now, we have a range of techniques available to us. The first step is to run a single character frequency analysis:

```bash
bash$ cat C.txt | fold -c -w 1 | grep [[:alpha:]] | sort | uniq -c | paste -s
2544 a 3674 b 678 c 87 d 5589 e
5835 f 1372 g 1528 h 10178 i 3173 j
1751 k 1310 l 5984 m 16 n 2176 o
169 p 1886 q 783 r 2763 s 7079 t
5237 u 5114 v 121 w 1411 x 6673 y
5195 z
```

and then to extract the most common bigrams and trigrams:

```bash
bash$ cat C.txt | fold -c -w 1 > E.txt
bash$ tail -n -2 E.txt > F.txt
bash$ tail -n -3 E.txt > G.txt
bash$ paste -d ' ' E.txt F.txt G.txt | tr -d ' ' | grep .. > H.txt
bash$ paste -d ' ' E.txt F.txt G.txt | tr -d ' ' | grep ... > I.txt
bash$ cat H.txt | sort | uniq -c | sort -n -r | head -n 20 | paste -s
2494 tf 1990 fi 1487 iv 1330 mz 1122 ys
1102 ez 1026 zj 913 vi 801 fm 797 li
787 yu 705 mt 677 bb 670 mv 649 et
bash$ cat I.txt | sort | uniq -c | sort -n -r | head -n 20 | paste -s
1186 tfi 797 mzl 595 fix 468 oyx 361 fsf
337 tfe 332 feu 308 eag 288 byv 275 mbb
273 ysv 248 tfm 242 etf 236 zyt 226 1vi
226 inv 224 get 219 fys 211 yri 282 ebb
bash$
```

Based on all this information we can start to make some guesses about how the ciphertext was produced:

- We still do not consider spaces during encryption, so the cipher still retains the word structure and we know that ‘△’ decrypts to ‘△’.
- Based on the single character frequency analysis we can be reasonably sure about at least the three most frequent characters and say that ‘i’ decrypts to ‘e’, ‘t’ decrypts to ‘t’ and ‘y’ decrypts to ‘o’.
- The bigram and trigram analysis confirms the guesses above because, for example, the most frequent trigram in the ciphertext is “tfi” and so if we match this against “the” (the most frequent trigram in some general text) we confirm the likelihood of ‘t’ decrypting to ‘t’. Based on further similar matching we can guess that ‘f’ decrypts to ‘h’, ‘m’ decrypts to ‘a’, ‘z’ decrypts to ‘n’ and ‘j’ decrypts to ‘d’.
- Given we already guessed ‘m’ decrypts to ‘a’, we can guess that ‘e’ decrypts to ‘i’ since on the second line we have a one letter word and we know it cannot decrypt to ‘a’.

Based on these initial guesses, and without too much effort, we can already be fairly confident about roughly a third of the key; we can start taking the ciphertext and performing a partial decryption. Considering just the first two lines of our example text:

```
y ’ ‘i ’ ‘i ’ ‘y ’ ‘z ’ ‘△ ’ ‘e ’ ‘△ ’
x ’ ‘v ’ ‘m ’ ‘o ’ ‘△ ’ ‘f ’ ‘i ’ ‘j ’ ‘△ ’
g ’ ‘e ’ ‘t ’ ‘i ’ ‘△ ’ ‘a ’ ‘q ’ ‘△ ’ ‘a ’ ‘i ’
’ ‘△ ’ ‘f ’ ‘i ’ ‘a ’ ‘q ’ ‘e ’ ‘b ’ ‘j ’ ‘△ ’
’ ‘△ ’ ‘t ’ ‘f ’ ‘o ’ ‘a ’ ‘i ’ ‘△ ’ ‘l ’ ‘b ’ ‘y ’ ‘q ’
’ ‘u ’ ‘,
```

```bash
bash$ cat C.txt | tail -n +959 | head -n 11
ylivyz. e xvmo tfii geri et ai.
```

By now, we have a range of techniques available to us. The first step is to run a single character frequency analysis:

```bash
bash$ cat C.txt | fold -c -w 1 | grep [[:alpha:]] | sort | uniq -c | paste -s
2544 a 3674 b 678 c 87 d 5589 e
5835 f 1372 g 1528 h 10178 i 3173 j
1751 k 1310 l 5984 m 16 n 2176 o
169 p 1886 q 783 r 2763 s 7079 t
5237 u 5114 v 121 w 1411 x 6673 y
5195 z
```

Based on all this information we can start to make some guesses about how the ciphertext was produced:

- We still do not consider spaces during encryption, so the cipher still retains the word structure and we know that ‘△’ decrypts to ‘△’.
- Based on the single character frequency analysis we can be reasonably sure about at least the three most frequent characters and say that ‘i’ decrypts to ‘e’, ‘t’ decrypts to ‘t’ and ‘y’ decrypts to ‘o’.
- The bigram and trigram analysis confirms the guesses above because, for example, the most frequent trigram in the ciphertext is “tfi” and so if we match this against “the” (the most frequent trigram in some general text) we confirm the likelihood of ‘t’ decrypting to ‘t’. Based on further similar matching we can guess that ‘f’ decrypts to ‘h’, ‘m’ decrypts to ‘a’, ‘z’ decrypts to ‘n’ and ‘j’ decrypts to ‘d’.
- Given we already guessed ‘m’ decrypts to ‘a’, we can guess that ‘e’ decrypts to ‘i’ since on the second line we have a one letter word and we know it cannot decrypt to ‘a’.

Based on these initial guesses, and without too much effort, we can already be fairly confident about roughly a third of the key; we can start taking the ciphertext and performing a partial decryption. Considering just the first two lines of our example text:

```
y ’ ‘i ’ ‘i ’ ‘v ’ ‘y ’ ‘z ’ ‘△ ’ ‘e ’ ‘△ ’
x ’ ‘v ’ ‘m ’ ‘o ’ ‘△ ’ ‘f ’ ‘i ’ ‘j ’ ‘△ ’
g ’ ‘e ’ ‘t ’ ‘i ’ ‘△ ’ ‘a ’ ‘q ’ ‘△ ’ ‘a ’ ‘i ’
’ ‘△ ’ ‘f ’ ‘i ’ ‘a ’ ‘q ’ ‘e ’ ‘b ’ ‘j ’ ‘△ ’
’ ‘△ ’ ‘t ’ ‘f ’ ‘o ’ ‘a ’ ‘i ’ ‘△ ’ ‘l ’ ‘b ’ ‘y ’ ‘q ’
’ ‘u ’ ‘,
```

```bash
bash$ cat C.txt | tail -n +959 | head -n 11
ylivyz. e xvmo tfii geri et ai.
```

By now, we have a range of techniques available to us. The first step is to run a single character frequency analysis:

```bash
bash$ cat C.txt | fold -c -w 1 | grep [[:alpha:]] | sort | uniq -c | paste -s
2544 a 3674 b 678 c 87 d 5589 e
5835 f 1372 g 1528 h 10178 i 3173 j
1751 k 1310 l 5984 m 16 n 2176 o
169 p 1886 q 783 r 2763 s 7079 t
5237 u 5114 v 121 w 1411 x 6673 y
5195 z
```

Based on all this information we can start to make some guesses about how the ciphertext was produced:

- We still do not consider spaces during encryption, so the cipher still retains the word structure and we know that ‘△’ decrypts to ‘△’.
- Based on the single character frequency analysis we can be reasonably sure about at least the three most frequent characters and say that ‘i’ decrypts to ‘e’, ‘t’ decrypts to ‘t’ and ‘y’ decrypts to ‘o’.
- The bigram and trigram analysis confirms the guesses above because, for example, the most frequent trigram in the ciphertext is “tfi” and so if we match this against “the” (the most frequent trigram in some general text) we confirm the likelihood of ‘t’ decrypting to ‘t’. Based on further similar matching we can guess that ‘f’ decrypts to ‘h’, ‘m’ decrypts to ‘a’, ‘z’ decrypts to ‘n’ and ‘j’ decrypts to ‘d’.
- Given we already guessed ‘m’ decrypts to ‘a’, we can guess that ‘e’ decrypts to ‘i’ since on the second line we have a one letter word and we know it cannot decrypt to ‘a’.

Based on these initial guesses, and without too much effort, we can already be fairly confident about roughly a third of the key; we can start taking the ciphertext and performing a partial decryption. Considering just the first two lines of our example text:

```
y ’ ‘i ’ ‘i ’ ‘v ’ ‘y ’ ‘z ’ ‘△ ’ ‘e ’ ‘△ ’
x ’ ‘v ’ ‘m ’ ‘o ’ ‘△ ’ ‘f ’ ‘i ’ ‘j ’ ‘△ ’
g ’ ‘e ’ ‘t ’ ‘i ’ ‘△ ’ ‘a ’ ‘q ’ ‘△ ’ ‘a ’ ‘i ’
’ ‘△ ’ ‘f ’ ‘i ’ ‘a ’ ‘q ’ ‘e ’ ‘b ’ ‘j ’ ‘△ ’
’ ‘△ ’ ‘t ’ ‘f ’ ‘o ’ ‘a ’ ‘i ’ ‘△ ’ ‘l ’ ‘b ’ ‘y ’ ‘q ’
’ ‘u ’ ‘,
```
we can already decrypt portions of it to read:

'o' 'l' 'e' 'v' 'o' 'n' '△' 'i' '△'
'x' 'v' 'a' 'o' '△' 't' 'h' 'e' 'e' '△'
'g' 'i' 'r' 'e' '△' 't' '△' 'a' 'e'
'△' 't' '△' 'c' 'n' 'o' 'q' '△' 'a' '△'
'i' 'a' 'n' 'c' '△' 'q' 'h' 'e' 'v' 'e'
'△' 't' 'h' 'e' '△' 'q' 'i' 'b' 'd' '△'
'i' 'h' 'o' 'a' 'e' '△' 'l' 'b' 'o' 'q'
'u' '△'

At this point we need to start working harder ... but we can still lean on some existing tools to help us. The basic idea is to start looking at the words which we know part of and narrow down the possibilities for the parts we do not know based on which real words fit the template. This is sort of like the process of filling in a crossword. For example, we can see two partially decrypted words “ae” and “thoe”. We know that ‘a’ probably decrypts to something which will make both of these examples real words. So first we can search the standard dictionary [13] file for all two letter words which end in ‘e’:

```
bash$ cat /usr/share/dict/words | grep -i ˆ.e$ | sort | uniq | paste -s
AE BE Be CE Ce DE Be EE FE Fe GE Ge HE He IE Je KE LE Le ME Me fe fe ge he ie le me ne oe pe
qe re se te we ye
bash$
```

This does not help much; the dictionary does not seem much good for this case! For example, the words “ve” and “qe” might be real, but do not really seem realistic possibilities for someone writing English. If we eliminate the words beginning with characters we are already confident about, this helps to reduce the possibilities; probably only “be”, “me”, “we” and “ye” remain. Now we can search for all five letter words that start with “th” and end with either “be”, “me”, “we” or “ye”:

```
bash$ cat /usr/share/dict/words | grep -i ˆth.be$ | sort | uniq | paste -s
Thebe thebe
bash$ cat /usr/share/dict/words | grep -i ˆth.me$ | sort | uniq | paste -s
theme thyme
bash$ cat /usr/share/dict/words | grep -i ˆth.we$ | sort | uniq | paste -s
bash$ cat /usr/share/dict/words | grep -i ˆth.ye$ | sort | uniq | paste -s
bash$
```

Although “thebe” might be a reasonable Shakespearean word, it seems more likely that ‘a’ decrypts to ‘m’ given the only other two words in the dictionary support this choice. Updating our partial decryption we get:

```
o' 'l' 'e' 'v' 'o' 'n' '△' 'i' '△'
x' 'v' 'a' 'o' '△' 't' 'h' 'e' 'e' '△'
g' 'i' 'r' 'e' '△' 't' '△' 'm' 'e'
'△' 't' '△' 'c' 'n' 'o' 'q' '△' 'a' '△'
i' 'a' 'n' 'c' '△' 'q' 'h' 'e' 'v' 'e'
'△' 't' 'h' 'e' '△' 'q' 'i' 'b' 'd' '△'
i 'h' 'o' 'm' 'e' '△' 'l' 'b' 'o' 'q'
'u' '△'
```

The process can continue in a similar way, using the frequency analysis to back up our guesses. Although we are working in a known ciphertext scenario, we can bend the rules a bit by considering some knowledge about the plaintext (actually we already did this by assuming it was English). Why is this not cheating? Imagine you get an encrypted email. In this case you know that there is a high chance that the start of the email includes the headers “to”, “from”, “subject” and so on, and this type of information can help conclude our search more quickly.

Fast-forwarding a little then, we would eventually recover the whole key and hence the method of substitution between plaintext and ciphertext characters. Using this, and in the same way as the shift cipher example, we can test if the result of a trial decryption looks reasonable:

```
bash: $='abcdefghijklmnopqrstuvwxyz'
bash: T='mlkjihgfedcbazxwvutsrqpon'
bash$ cat C.txt | tr "$T" "$S" > D.txt
140
```
Find someone to work with. One of you act as the sender of some secret plaintext message, and the other as the cryptanalyst:

1. the sender selects $k$, i.e., a permutation, and encrypts the plaintext message to produce a ciphertext, then
2. the cryptanalyst is given the ciphertext, and asked to recover $k$ (or at least the plaintext message, partial or otherwise).

Did it work? Several things can go wrong: can you think why the process might fail? For example, what assumptions do we make and therefore depend on being true?
BIBLIOGRAPHY

CHAPTER

THE ENIGMA OF GROUP THEORY

With the advent of the 1920s people saw the need for a mechanical encryption device. Taking a substitution cipher and then rotating it became seen as the ideal solution. This idea had actually been used previously in a number of manual ciphers, but using machines it was seen how this could be done more efficiently. The rotors could be implemented using wires and then encryption could be done mechanically using an electrical circuit. By rotating the rotor we obtain a new substitution cipher.

As an example, suppose the rotor used to produce the substitutions is given by

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
TMKGOYDSIPELUAVCRJWXZNHBQF
```

By this we mean that the plaintext letter A will encrypt to the ciphertext letter T, and so on. In the past a substitution cipher was only used on its own, thus the word EGG would encrypt to the word ODD. This turns out not to be a good idea because the plaintext letter, e.g. G, always encrypts to the same ciphertext letter, e.g. D. This fact allows a cipher based solely on a single substitution, a so-called substitution cipher, to be easily broken [9].

A simple way of increasing the security of a substitution cipher is to rotate the ciphertext alphabet by one letter every time we encrypt. So to encrypt the first letter we use the substitutions given by

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
TMKGOYDSIPELUAVCRJWXZNHBQF
```

However, to encrypt the second letter we rotate the rotor by one position and use the substitutions

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
MKGOYDSIPELUAVCRJWXZNHBQFT
```

To encrypt the third letter we use the substitutions

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
KGOYDSIPELUAVCRJWXZNHBQFTM
```

and so on. This the word EGG will become encrypted to ODS. This gives us a polyalphabetic substitution cipher with 26 alphabets.

The most famous of the rotor machines developed in the first half of the twentieth century was the Enigma machine used by the Germans in World War II [2]. We shall describe the most simple version of Enigma which only used three such rotors, chosen from the following set of five.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
EKMFLGQVZNTOWXUSPAIBRCJ
AJDKSIRUXBLHWTMCQGZNPYFVOE
BDFHJLCPRXTZVNYEIAGKMUSQO
ESOVPZJAYQUIHRXLNFTGKDCWMB
VZBRGITYUPSNDHXLAWMJQOFCK
```
Machines in use towards the end of the war had a larger number of rotors, chosen from a larger set. Note, the order of the rotors in the machine is important, so the number of ways of choosing the rotors is

\[ 5 \cdot 4 \cdot 3 = 60. \]

Each rotor had an initial starting position, and since there are 26 possible starting positions for each rotor, the total number of possible starting positions is \( 26^3 = 17,576 \).

The first rotor would step on the second rotor on each full iteration under the control of a ring hitting a notch, likewise the stepping of the third rotor was controlled by the second rotor. Both the rings were movable and their positions again formed part of the key, although only the notch and ring positions for the first two rotors were important. Hence, the number of ring positions was \( 26^2 = 676 \). The second rotor also had a kick associated to it making the cycle length of the three rotors equal to

\[ 26 \cdot 25 \cdot 26 = 16,900. \]

The effect of the moving rotors was that a given plaintext letter would encrypt to a different ciphertext letter on each press of the keyboard. Finally, a plug board was used to swap letters twice in each encryption and decryption operation. This increased the complexity and gave another possible \( 10^{14} \) keys.

The rotors used, their order, their starting positions, the ring positions and the plug board settings all made up the secret key. Hence, the total number of keys was then around \( 2^{75} \).

To make sure encryption and decryption were the same operation the message letter was passed through the plug board, then through the three rotors, and then through another fixed substitution called the reflector. After passing through the reflector the message letter was sent back through the three rotors and the plugboard again. The reflector was fixed in the machine, and was given by

```
ABCDEFHJKLMOVQRSTUWXYZ
YRUHQLDPXNGOKMIEBFZCWVJAT
```

The operation of a simplified Enigma machine is described in Fig. 8.1. By tracing the red lines one can see how the plaintext character A encrypts to the ciphertext character D. Notice that encryption and decryption can be performed by the machine being in the same positions. Now assume that rotor one moves on one step, so A now maps to D under rotor one, B to A, C to C and D to B. You should work out what happens with the example when we encrypt A again.

![Figure 8.1: Simplified Enigma machine](image)

The purpose of this note is to present a mathematical model of the Enigma machine and show how some basic facts about permutations allowed the Polish cryptographers to break the cipher.

### 8.1 Permutations

We want to find a mathematical description of the Enigma machine. To do this we will first need to introduce the concept of a permutation [7].

We let \( A \) be a finite set of size \( n \), we might as well assume that the set is given by \( A = \{1, 2, \ldots, n\} \). A function from \( A \) to \( A \) is said to be one-to-one if every element in \( A \) maps to exactly one element in \( A \); for example if \( A = \{1, 2, 3\} \), then we have the one-to-one function \( f(1) = 2, f(2) = 3 \) and \( f(3) = 1 \). Another name for such one-to-one functions from \( A \) to \( A \) is a permutation, since such a function permutes the elements in the set around.
This is a very cumbersome way to write a permutation. Mathematicians (being lazy people) have invented the following notation, the function $f$ above is written as

$$
\begin{pmatrix}
1 & 2 & 3 \\
2 & 3 & 1
\end{pmatrix}.
$$

What should be noted about this notation (which applies for arbitrary $n$) is that all the numbers between 1 and $n$ occur exactly once on each row. The first row is always given as the numbers 1 to $n$ in increasing order. Any such matrix with these properties represents a permutation, and all permutations can be represented by such a matrix.

The set of all permutations on a set of size $n$ is denoted by $S_n$. The above matrix notation allows us to see very easily that the size of the set $S_n$ is $n!$. To see this we notice that there are $n$ choices for the first element in the second row of the above matrix. Then there are $n-1$ choices for the second element in the second row and so on.

Suppose we define the permutations

$$
g = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \quad \text{and} \quad f = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}
$$

As permutations are nothing but functions we can compose them. Remembering that $g \circ f$ means apply the function $f$ and then apply the function $g$ we see that

$$
\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}
$$

means $1 \rightarrow 3 \rightarrow 1, 2 \rightarrow 2 \rightarrow 3$ and $3 \rightarrow 1 \rightarrow 2$. Hence, the result of composing the above two permutations is

$$
\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}.
$$

If $\sigma$ is a permutation on a set $A$ then we usually think of $\sigma$ acting on the set. So if $a \in A$ then we write

$$
a^\sigma
$$

for the action of $\sigma$ on the element $a$, i.e. this is another way of writing $\sigma(a)$. However, this can cause confusion when using the standard notation for function composition above. For example

$$
1^{\circ f} = g(f(1)) = 3
$$

so we are unable to read the permutation from left to right. However, if we use another notation say $\cdot$ to mean

$$
f \cdot g = g \circ f
$$

then we are able to read the expression from left to right, i.e.

$$
1^{f \cdot g} = g(f(1)).
$$

We shall call this operation multiplying permutations.

Mathematicians are, as we said, by nature lazy people and this notation we have introduced is still a little too much. For instance we always write down the numbers $1, \ldots, n$ in the top row of each matrix to represent a permutation. Also some columns are redundant, for instance the first column of the permutation

$$
\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}.
$$

We now introduce another notation for permutations which is concise and clear, which uses the concept of a cycle. By a cycle or $n$-cycle we mean the object $(x_1, \ldots, x_n)$ with distinct $x_i \in \mathbb{N} \setminus \{0\}$. This represents the permutation $f(x_1) = x_2, f(x_2) = x_3, \ldots, f(x_{n-1}) = x_n, f(x_n) = x_1$ and for $x \notin \{x_1, \ldots, x_n\}$ we have $f(x) = x$. For instance we have

$$
\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} = (1,2,3) = (2,3,1) = (3,1,2).
$$

Notice that a cycle is not a unique way of representing a permutation. As another example we have

$$
\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} = (1,3)(2) = (3,1)(2).
$$

147
The identity permutation is represented by (). Again, as mathematicians are lazy we always write \((1,3)(2) = (1,3)\). This can still lead to ambiguities as \((1,2)\) could represent a function
\[
[1,2] \rightarrow [1,2] \text{ or } [1,2,\ldots,n] \rightarrow [1,2,\ldots,n].
\]
Which function it represents is usually however clear from the context.

Two cycles \((x_1,\ldots,x_t)\) and \((y_1,\ldots,y_n)\) are called disjoint if \(\{x_1,\ldots,x_t\} \cap \{y_1,\ldots,y_n\} = \emptyset\). It is easy to show that if \(\sigma\) and \(\tau\) are two disjoint cycles then
\[
\sigma \cdot \tau = \tau \cdot \sigma.
\]
Note this is not true for cycles which are not disjoint, e.g.
\[
(1,2,3,4) \cdot (3,5) = (1,2,5,3,4) \neq (1,2,3,5,4) = (3,5) \cdot (1,2,3,4).
\]
This means that multiplying permutations does not follow the usual "commutative rule" of standard multiplication.

Our action of permutations on the underlying set can now be read easily from left to right,
\[
2^{(1,2,3,4) \cdot (3,5)} = 3^{(3,5)} = 5 = 2^{(1,2,5,3,4)},
\]
as the permutation \((1,2,3,4)\) maps 2 to 3 and the permutation \((3,5)\) maps 3 to 5. With this notation it is easily seen that computing the value of permutations becomes very simple, which is why the notation is used quite a lot.

What really makes disjoint cycles interesting is that every permutation can be written as a product of disjoint cycles. To see this let \(\sigma\) be a permutation on \([1,\ldots,n]\) and let \(\sigma_1\) denote the cycle
\[
(1,\sigma(1),\sigma(\sigma(1)),\ldots,\sigma(\ldots\sigma(1)\ldots)),
\]
where we keep applying \(\sigma\) until we get back to 1. We then take an element \(x\) of \([1,\ldots,n]\) such that \(\sigma_1(x) = x\) and consider the cycle \(\sigma_2\) given by
\[
(x,\sigma(x),\sigma(\sigma(x)),\ldots,\sigma(\ldots\sigma(x)\ldots)).
\]
We then take an element of \([1,\ldots,n]\) which is fixed by \(\sigma_1\) and \(\sigma_2\) to create a cycle \(\sigma_3\). We continue this way until we have used all elements of \([1,\ldots,n]\). The resulting cycles \(\sigma_1,\ldots,\sigma_t\) are obviously disjoint and their product is equal to the cycle \(\sigma\).

What is nice about the above argument is that it is constructive. Given a permutation we can follow the procedure in the argument above to obtain the permutation as a product of disjoint cycles. To see this consider the permutation
\[
\sigma = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
2 & 3 & 7 & 6 & 8 & 4 & 1 & 5 & 9
\end{pmatrix}.
\]
We have \(\sigma(1) = 2\), \(\sigma(2) = 3\), \(\sigma(3) = 7\) and \(\sigma(7) = 1\) so the first cycle is
\[
\sigma_1 = (1,2,3,7).
\]
The next element of \([1,\ldots,9]\) which we have not yet considered is 4. We have \(\sigma(4) = 6\) and \(\sigma(6) = 4\) so \(\sigma_2 = (4,6)\). Continuing in this way we find \(\sigma_3 = (5,8)\) and \(\sigma_4 = (9)\). Hence we have
\[
\sigma = (1,2,3,7)(4,6)(5,8)(9) = (1,2,3,7)(4,6)(5,8).
\]

In the rest of this note we present more details on the Enigma machine and some of the attacks which can be performed on it. However before presenting the machine itself we need to fix some notation which will be used throughout the note. In particular lower-case letters will denote variables, upper-case letters will denote "letters" (of the plaintext/ciphertext languages) and greek letters will denote permutations in \(S_{26}\) which we shall represent as permutations on the upper-case letters. Hence \(x\) can equal \(X\) and \(Y\), but \(X\) can only ever represent \(X\), whereas \(\chi\) could represent \((XY)\) or \((ABC)\).

Our convention of permutations acting on the right of elements of the underlying set, then in this context is exemplified by
\[
A^{(ABCD)(XY)} = B.
\]
We now collect some basic facts and theorems about permutations which we will need in the sequel.

**Fact 1** Two permutations \(\sigma\) and \(\tau\) which are conjugate, i.e. for which \(\sigma = \lambda \cdot \tau \cdot \lambda^{-1}\) for some permutation \(\lambda\), have the same cycle structure.
We define the support of a permutation to be the set of letters which are not fixed by the permutation. Hence, if \( \sigma \) acts on the set of letters \( L \), then as usual we denote by \( L^\sigma \) the set of fixed points and hence the support is given by \( L \setminus L^\sigma \).

**Fact 2** If two permutations, with the same support, consist only of disjoint transpositions then their product contains an even number of disjoint cycles of the same lengths.

**Fact 3** If a permutation with support an even number of symbols has an even number of disjoint cycles of the same lengths, then the permutation can be written as a product of two permutations each of which consists of disjoint transpositions.

In many places we need an algorithm to solve the following problem: Given \( \alpha_i, \beta_i \in S_{26} \), for \( i = 1, \ldots, m \) find \( \gamma \in S_{26} \) such that

\[
\alpha_i = \gamma^{-1} \cdot \beta_i \cdot \gamma \quad \text{for} \ i = 1, \ldots, m.
\]

Note, there could be many such solutions \( \gamma \), but in the situations we will apply it we expect there to be only a few. For example suppose we have one such equation with

\[
\begin{align*}
\alpha_1 &= (AFCNE)(BWXHUJOG)(DVIQZ)(KLMYTRPS), \\
\beta_1 &= (AEYSXUWJ)(BFXNO)(CDPKQ)(GHIYVLMRT)
\end{align*}
\]

We need to determine the structure of the permutation \( \gamma \) such that

\[
\alpha_1 = \gamma^{-1} \cdot \beta_1 \cdot \gamma.
\]

We first look at what should \( A \) map to under \( \gamma \). If \( A^\gamma = B \), then from \( \alpha_1 \) and \( \beta_1 \) we must have \( E^\gamma = W \), which in turn implies \( Y^\gamma = X \). Carrying on in this way via a pruned depth first search [1] we can determine a set of possible values for \( \gamma \). Such an algorithm is relatively simple to write down in a computer programming language, using a recursive procedure call [8]. It however of course been a bit of a pain to do this by hand, as one would need to in the 1930’s and 1940’s.

### 8.2 An equation for the Enigma

To aid our discussion in later sections we now describe the Enigma machine as a permutation equation. We first assume a canonical map between letters and the integers \( \{0, 1, \ldots, 25\} \) such that \( 0 = A, 1 = B, \) etc and we assume a standard three wheel Enigma machine.

The wheel which turns the fastest we shall call rotor one, whilst the one which turns the slowest we shall call rotor three. This means that when looking at a real machine rotor three is the left most rotor and rotor one is the right most rotor. This can cause confusion (especially when reading day/message settings), so please keep this in mind.

The basic permutations which make up the Enigma machine are as follows:

### 8.2.1 Choice of rotors

We assume that the three rotors are chosen from the following set of five rotors. We present these rotors in cycle notation, but they are the commonly labelled rotors \( I, II, III, IV \) and \( V \) used in the actual Enigma machines, which were given earlier. Each rotor also has a different notch position which controls how the stepping of one rotor drives the stepping of the others.

<table>
<thead>
<tr>
<th>Rotor</th>
<th>Permutation Representation</th>
<th>Notch Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>(AELTPHQXRU)(BKNW)(CMOY)(DFG)(IV)(JZ)</td>
<td>16 = Q</td>
</tr>
<tr>
<td>II</td>
<td>(B)(CDKLHUP)(ESZ)(FIXVYOMW)(GR)(NT)</td>
<td>4 = E</td>
</tr>
<tr>
<td>III</td>
<td>(ABDHEP)(CFVLMZOYQRWUKXSG)</td>
<td>21 = V</td>
</tr>
<tr>
<td>IV</td>
<td>(AEIIYWCOXMRFZSTGQNH)(DV)(KLU)</td>
<td>9 = J</td>
</tr>
<tr>
<td>V</td>
<td>(AVOLDRFUIQQ)(RZKSMNHYC)(EGTJ)</td>
<td>25 = Z</td>
</tr>
</tbody>
</table>
8.2.2 Reflector

There were a number of reflectors used in actual Enigma machines. In our description we shall use the reflector given earlier, which is often referred to as “Reflector B”. This reflector has representation via disjoint cycles as


8.2.3 An Enigma key

An Enigma key consists of the following information:

- A choice of rotors \( \rho_1, \rho_2, \rho_3 \) from the above choice of five possible rotors. Note, this choice of rotors affects the three notch positions, which we shall denote by \( n_1, n_2 \) and \( n_3 \). Also, as noted above, the rotor \( \rho_3 \) is placed in the left of the actual machine, whilst rotor \( \rho_1 \) is placed on the right. Hence, if in a German code book it says use rotors

\[ I, II, III, \]

this means in our notation that \( \rho_1 \) is selected to be rotor III, that \( \rho_2 \) is selected to be rotor II and \( \rho_3 \) is selected to be rotor I.

- One must also select the ring positions, which we shall denote by \( r_1, r_2 \) and \( r_3 \). In the actual machine these are letters, but we shall use our canonical numbering to represent these as integers in \([0, 1, \ldots, 25]\).

- The plugboard is simply a product of disjoint transpositions which we shall denote by the permutation \( \tau \). In what follows we shall denote a plug linking letter \( A \) with letter \( B \) by \( A \leftrightarrow B \).

- The starting rotor positions we shall denote by \( p_1, p_2 \) and \( p_3 \). These are the letters which can be seen through the windows on the top of the Enigma machine. Remember our numbering system is that the window on the left corresponds to \( p_2 \) and that on the right corresponds to \( p_1 \).

8.2.4 The encryption operation

We let \( \sigma \) denote the shift-up permutation given by

\[ \sigma = (ABCDEFGHIJKLMNOPQRSTUVWXYZ). \]

The stepping of the second and third rotor is probably the hardest part to grasp when first looking at an Enigma machine, however this has a relatively simple description when one looks at it in a mathematical manner.

Given the above description of the key we wish to deduce the permutation \( \epsilon_j \), which represents the encryption of the \( j \)th letter, for \( j = 0, 1, 2, \ldots \).

We first set

\[
\begin{align*}
m_1 &= n_1 - p_1 - 1 \pmod{26}, \\
m &= n_2 - p_2 - 1 \pmod{26}, \\
m_2 &= m_1 + 1 + 26m.
\end{align*}
\]

The values of \( m_1 \) and \( m_2 \) control the stepping of the second and the third rotors.

We let \( \lfloor x \rfloor \) denote the round towards zero function, i.e. \( \lfloor 1.9 \rfloor = 1 \) and \( \lfloor -1.9 \rfloor = -1 \). We now set, for encrypting letter \( j \),

\[
\begin{align*}
k_1 &= \lfloor (j - m_1 + 26)/26 \rfloor, \\
k_2 &= \lfloor (j - m_2 + 650)/650 \rfloor, \\
i_1 &= p_1 - r_1 + 1, \\
i_2 &= p_2 - r_2 + k_1 + k_2, \\
i_3 &= p_3 - r_3 + k_2.
\end{align*}
\]

Notice, how \( i_3 \) is stepped on every 650 = 26 · 25 iterations whilst \( i_2 \) is stepped on every 26 iterations and also stepped on an extra notch every 650 iterations. We can now present \( \epsilon_j \) as

\[
\epsilon_j = \tau \cdot (\sigma^{i_3+i} p_1^{-i} \sigma^{-h-i}) \cdot (\sigma^{i_2} p_2^{-i} \sigma^{-h-i}) \cdot (\sigma^{i_1} p_3^{-i} \sigma^{-h-i}) \cdot q \cdot (\sigma^{i_3} p_3^{-1} \sigma^{-i}) \cdot (\sigma^{i_2} p_2^{-1} \sigma^{-i}) \cdot (\sigma^{i_1} p_1^{-1} \sigma^{-i}) \cdot \tau.
\]

150
Note that the same equation/machine is used to encrypt the $j$th letter as is used to decrypt the $j$th letter. Hence we have 

$$
\epsilon_j^{-1} = \epsilon_j.
$$

Also note that Each $\epsilon_j$ consists of a product of disjoint transpositions.

Note, we shall always use $\gamma_j$ to represent the internal rotor part of the Enigma machine, hence 

$$
\epsilon_j = \tau \cdot \gamma_j \cdot \tau.
$$

### 8.3 The Polish work on Enigma

The polish mathematicians Jerzy Rozycki [5], Henryk Zygalski [4] and Marian Rejewski [6] were the first to find ways of analysing the Enigma machine. To understand their methods one must first understand how the Germans used the machine. On each day the machine was set up with a key, as above, which was chosen by looking up in a code book. Each subnet would have a different day key.

To encipher a message the sending operator decided on a message key. The message key would be a sequence of three letters, say $DHI$. The message key needs to be transported to the recipient. Using the day key, the message key would be enciphered twice. The double enciphering is to act as a form of error control. Hence, $DHI$ might be enciphered as $XHJKLM$. Note, that $D$ encrypts to $X$ and then $K$, this is a property of the Enigma machine.

The receiver would obtain $XHJKLM$ and then decrypt this to obtain $DHI$. Both operators would then move the wheels around to the positions $D$, $H$ and $I$, i.e. they would turn the wheels so that $D$ was in the leftmost window, $H$ in the middle one and $I$ in the rightmost window. Then the actual message would be enciphered.

For this example, in our notation, this would mean that the message key is equal to the day key, except that $p_1 = 8 = I$, $p_2 = 7 = H$ and $p_3 = 3 = D$.

However, life was even more difficult for the Poles as they did not even know the rotor wirings or the reflector values. Hence, they needed to break the machine without even having a description of the actual machine. They did have access to a non-military version of Enigma and deduced the basic structure. In this they were very lucky in that they deduced that the wiring between the plugboard and the right most rotor was in the order of the alphabet. If this were not the case there would have been some hidden permutation which would also have needed to be found.

Luckily the French cryptographer Gustave Bertrand [3] obtained from a German spy, Hans-Thilo Schmidt, two months worth of day keys. Thus, for two months of traffic the Poles had access to the day settings. From this information they needed to deduce the internal wirings of the Enigma machine.

Note, in the pre-war days the Germans only used three wheels out of a choice of three, hence the number of days keys is actually reduced by a factor of ten. This is, however, only a slight simplification (at least with modern technology).

Suppose we are given that the day setting is

<table>
<thead>
<tr>
<th>Rotors</th>
<th>Rings</th>
<th>Pos</th>
<th>Plugboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>III, II, I</td>
<td>TXC</td>
<td>EAZ</td>
<td>(AMTEBC)</td>
</tr>
</tbody>
</table>

We do not know what the actual rotors are at present, but we know that the one labelled rotor I will be placed in the rightmost slot (our label one). So we have 

$$
\begin{align*}
    r_1 &= 2, \\
    r_2 &= 23, \\
    r_3 &= 19, \\
    p_1 &= 25, \\
    p_2 &= 0, \\
    p_3 &= 4.
\end{align*}
$$

Suppose we intercept a set of messages which have the following headers, consisting of the encryption of the three letter rotor positions, followed by its encryption again, i.e. the first six letters of each message are equal to

---

UCWBLR ZSETEY SLVMQH SGIMVW PMRWWW
VNGCTP QDQPHS CBRVPV KSCJEA GSTGUE
DQLSNT HXYYHF GETGSU EELXJS OSQPEB
WISIIT TXFEHX ZAMTAM VEMCSM LQFPNI
LOIFMWW JXHJUZ PXYWFQ FAYQAF QJPOUI
EPILWW DOGSMP ADDSRT XLJXQK BKEAKY
...... ...... ...... ...... ...... ......
DDESRY Q3COUA JEZUSN MUXROQ SLPMQI
---
If we take the last one of these and look at it in more detail. We know that there are three underlying secret letters, say \( l_1, l_2 \) and \( l_3 \). We also know that

\[
\begin{align*}
l_1^{c_0} &= N, & l_2^{c_0} &= Y, & l_3^{c_0} &= I,
\end{align*}
\]

and

\[
\begin{align*}
l_1^{c_1} &= K, & l_2^{c_1} &= F, & l_3^{c_1} &= W.
\end{align*}
\]

Hence, given that \( \epsilon_j^{-1} = \epsilon_j \), we have

\[
\begin{align*}
N^{c_{00}} &= l_1^{c_0 \epsilon_0 c_0}, & l_1^{c_1 \epsilon_1} &= K, & Y^{c_0} &= F, & F^{c_0} &= W.
\end{align*}
\]

Continuing in this way we can compute a permutation representation of the three products as follows:

\[
\begin{align*}
\epsilon_0 \cdot \epsilon_3 &= (ADSMRNKJUB)(CV)(ELFQOPWIZT)(HY), \\
\epsilon_1 \cdot \epsilon_4 &= (BPWJUOMGV)(CLQNTRDFY)(ES)(HX), \\
\epsilon_2 \cdot \epsilon_5 &= (AC)(BDSTUEYFXQ)(GPIWRVHZNO)(JK).
\end{align*}
\]

From these we wish to deduce the values of \( \epsilon_0, \epsilon_1, \ldots, \epsilon_5 \). We will use the fact that \( \epsilon_j \) is a product of disjoint transpositions and Facts 2 and 3.

We take the first product and look at it in more detail. We take the sets of two cycles of equal degree and write them above one another, with the bottom one reversed in order, i.e.

\[
\begin{align*}
A & D S M R N K J U B & C V \\
T & Z I W P O Q F L E & Y H
\end{align*}
\]

We now run through all possible shifts of the bottom rows. Each shift gives us a possible value of \( \epsilon_0 \) and \( \epsilon_3 \). The value of \( \epsilon_0 \) is obtained from reading off the disjoint transpositions from the columns, the value of \( \epsilon_3 \) is obtained by reading off the transpositions from the “off diagonals”. For example with the above orientation we would have

\[
\begin{align*}
\end{align*}
\]

But we have \( 20 = 2 \cdot 10 \) such orientations, so there are 20 possible values for \( \epsilon_0 \) and \( \epsilon_3 \).

Now, to reduce this number we need to really on stupid operators. Various operators had a tendency to always select the same three letter message key. For example popular choices where \( QWE \) (the first letters on the keyboard). One operator used the letters of his girlfriend name, Cillie, hence such “cribs” (or guessed/known plaintexts in todays jargon) became known as “Cillies”. Note, for our analysis here we only need one Cillie for the day when we obtain the internal wiring of rotor I.

In our dummy example, suppose we guess (correctly) that the first message key is indeed \( QWE \). This means that \( UCWBLR \) is the encryption of \( QWE \) twice, this in turn tells us how to align our cycle of length 10 in the first permutation, as under \( \epsilon_0 \) the letter \( Q \) must encrypt to \( U \).

\[
\begin{align*}
A & D S M R N K J U B & C V \\
L & E T Z I W P O Q F
\end{align*}
\]

We can check that this is consistent as we see that \( Q \) under \( \epsilon_3 \) must then encrypt to \( B \). If we guessed one more such Cillies we can reduce the number of possibilities for \( \epsilon_1, \ldots, \epsilon_6 \). Assuming we carry on in this way we will finally deduce that

\[
\begin{align*}
\end{align*}
\]

We now need to use this information to deduce the value of \( \rho_i \), etc. So for the rest of this section we assume we know \( \epsilon_j \) for \( j = 0, \ldots, 5 \), and so we mark it in blue.
Recall that we have,
\[
\epsilon_j = \tau \cdot (\sigma^j \rho_1 \sigma^{-j}) \cdot (\sigma^j \rho_2 \sigma^{-j}) \cdot (\sigma^j \rho_3 \sigma^{-j}) \cdot \rho \cdot \\
\cdot (\sigma^j \rho_4 \sigma^{-j}) \cdot (\sigma^j \rho_5 \sigma^{-j}) \cdot (\sigma^j \rho_6 \sigma^{-j}) \cdot (\sigma^j \rho_7 \sigma^{-j}) \cdot \tau
\]

We now assume that no stepping of the second rotor occurs during the first six encryptions under the day setting. This occurs with quite high probability, namely 20/26 ≈ 0.77. If this assumption turns out to be false we will notice this in our later analysis and it will mean we can deduce something about the (unknown to us at this point) position of the notch on the first rotor.

Given that we know the day settings, so that we know $\tau$ and the values of $\ell_1, \ell_2$ and $\ell_3$ (since we are assuming $k_1 = k_2 = 0$ for $0 \leq j \leq 5$), we can write the above equation for $0 \leq j \leq 5$ as
\[
\lambda_j = \sigma^{j-1} \cdot \tau \cdot \epsilon_j \cdot \tau \cdot \sigma^{j+1} = \rho_1 \cdot \sigma^{-j} \cdot \gamma \cdot \sigma^{j+1} \cdot \rho_1^{-1}.
\]

Where $\lambda_j$ is now known and we wish to determine $\rho_1$ for some fixed but unknown value of $\gamma$. The permutation $\gamma$ is in fact equal to
\[
\gamma = (\sigma^0 \rho_2 \sigma^{-0}) \cdot (\sigma^0 \rho_3 \sigma^{-0}) \cdot \rho \cdot (\sigma^0 \rho_5 \sigma^{-0}) \cdot (\sigma^0 \rho_1 \sigma^{-0}).
\]

In our example we get the following values for $\lambda_j$,
\[
\lambda_0 = \langle AD \rangle BR(CQ)(EV)(FZ)(GP)(HM)(IN)(JK)(LU)(OS)(TW)(XY), \\
\lambda_1 = \langle AV \rangle BP(CZ)(DF)(ED)(GS)(HY)(JL)(KO)(MU)(NQ)(RW)(TX), \\
\lambda_2 = \langle AL \rangle BK(CN)(DZ)(EV)(FP)(GX)(HS)(HY)(JM)(OQ)(RU)(TW), \\
\lambda_3 = \langle AS \rangle BF(CZ)(DR)(EM)(GN)(HY)(IW)(JO)(KQ)(LX)(PV)(TU), \\
\lambda_4 = \langle AQ \rangle BK(CT)(DL)(EP)(FI)(GX)(HW)(JL)(MO)(NY)(RS)(VZ), \\
\]

We now form, for $j = 0, \ldots, 4$,
\[
\mu_j = \lambda_j \cdot \lambda_{j+1},
\]
\[
= \rho_1 \cdot \sigma^{-j} \cdot \gamma \cdot \sigma^{-1} \cdot \gamma \cdot \sigma^{j+1} \cdot \rho_1^{-1},
\]
\[
= \rho_1 \cdot \sigma^{-j} \cdot \delta \cdot \sigma^j \cdot \rho_1^{-1},
\]

where $\delta = \gamma \cdot \sigma^{-1} \cdot \gamma \cdot \sigma$ is unknown. Eliminating $\delta$ via $\delta = \sigma^{-1} \rho_1^{-1} \mu_{j-1} \rho_1 \sigma^{j+1}$ we find the following equations for $j = 1, \ldots, 4$,
\[
\mu_j = (\rho_1 \cdot \sigma^{-1} \cdot \rho_1^{-1}) \cdot \mu_{j-1} \cdot (\rho_1 \cdot \sigma \cdot \rho_1^{-1}),
\]
\[
= \alpha \cdot \mu_{j-1} \cdot \alpha^{-1},
\]

where $\alpha = \rho_1 \cdot \sigma^{-1} \cdot \rho_1^{-1}$. Hence, $\mu_j$ and $\mu_{j-1}$ are conjugate and so by Fact 1 have the same cycle structure. For our example we have
\[
\mu_0 = \langle AFCN \rangle (BWXH)(JO)(CV)(QZ)(KL)(MY)(TR)(PS), \\
\mu_1 = \langle AEYS \rangle WUJ(BFZNO)(CDPKQ)(GHIV)(LMR), \\
\mu_2 = \langle AXN \rangle RTI(BJ)(Q)(EP)(CGLS)(YWUD)(FVMOK), \\
\mu_3 = \langle ARLG \rangle YWFK(BIHNXDS)(CVEOL)(JMPZT), \\
\mu_4 = \langle AGYPM \rangle DIR(BHU)(TV)(CJ)(WKZ)(ENXQSFLO).
\]

At this point we can check whether our assumption of no-stepping, i.e. a constant value for the values of $\ell_2$ and $\ell_3$ is valid. If a step did occur in the second rotor then the above permutations would be unlikely to have the same cycle structure.

We need to determine the structure of the permutation $\alpha$, this is done by looking at the four equations simultaneously. We note that since $\sigma$ and $\alpha$ are conjugates, under $\rho_1$, we know that $\alpha$ has cycle structure of a single cycle of length 26.

In our example we only find one possible solution for $\alpha$, namely
\[
\alpha = \langle AGY \rangle WUJOQ(NI)(RX)(HT)(MK)(CE)(BZ)(VPFD).
\]
To solve for $\rho_1$ we need to find a permutation such that

$$\alpha = \rho_1 \cdot \sigma^{-1} \cdot \rho_1^{-1}.$$ 

We find there are 26 such solutions

$$(AELTPHQXRU)(BKNW)(CMOY)(DFG)(IV)(JZ)$$
$$(AFHRVJ)(BLU)(CNXSTQYDGEMPIW)(KOZ)$$
$$(AGFIXRWDHSLUCO)(BMQZLVK)(ENY)$$
$$(AHTSVLWEOBNZMRXUDITYF)(CPKQ)$$
$$(AIZN)(BQOC)(DJ)(EPLXVMSWFKRYGHLU)$$
$$(AJEQRZDOKSXFQGJ)(BPMTUFLYHV)$$
$$(AKTVOER)(BQDSZPNCSYI)(FMUG)(HW)$$
$$(AL)(BR)(CTWI)(DMVPOFN)(ESZQ)(GKUHXY)$$
$$(AMWJHYKVFQGLBS)(CUIDNETXZ)$$

These are the values of $\rho_1 \cdot \sigma^i$, for $i = 0, \ldots, 25$.

So with one days messages we can determine the value of $\rho_1$ up to multiplication by a power of $\sigma$. The Polish had access to two months such data and so were able to determine similar sets for $\rho_2$ and $\rho_3$ (as different rotor orders are used on different days). Note, at this point the Germans did not use a selection of three from five rotors.

If we select three representatives $\hat{\rho}_1$, $\hat{\rho}_2$ and $\hat{\rho}_3$, from the sets of possible rotors, then we have

$$\hat{\rho}_1 = \rho_1 \cdot \sigma^h,$$
$$\hat{\rho}_2 = \rho_2 \cdot \sigma^l,$$
$$\hat{\rho}_3 = \rho_3 \cdot \sigma^b.$$ 

However, we still do not know the value for the reflector $\varphi$, or the correct values of $h$, $l$ and $b$.

Any one of these versions of $\hat{\rho}_i$ will give a valid Enigma machine (with a different reflector). So it does not matter which choice we make for $\rho_i$.

### 8.4 Determining the day settings

Now having determined the internal wirings, given the set of two months of day settings obtained by Bertrand, the next task is to determine the actual key when the day settings are not available. At this stage we assume the Germans are still using the encrypt the message setting twice routine.

The essential trick here is to notice that if we write the cipher as

$$\epsilon_j = \tau \cdot \gamma_j \cdot \tau,$$

then

$$\epsilon_j \cdot \epsilon_{j+3} = \tau \cdot \gamma_j \cdot \gamma_{j+3} \cdot \tau.$$
So $\epsilon_j \cdot \epsilon_{j+3}$ is conjugate to $\gamma_j \cdot \gamma_{j+3}$ and so by Fact 1 they have the same cycle structure. More importantly the cycle structure does not depend on the plug board $\tau$.

Hence, if we can use the cycle structure to determine the rotor settings then we are only left with determining the plugboard settings. If we can determine the rotor settings then we know the values of $\gamma_j$ for $j = 1, \ldots, 6$, from the encrypted message keys we can compute $\epsilon_j$ for $j = 1, \ldots, 6$ as in the previous section.

Hence, determining the plugboard settings is then a question of solving one of our conjugacy problems again, for $\tau$. But this is easier than before as we have that $\tau$ must be a product of disjoint transpositions.

We have already discussed how to compute $\epsilon_j \cdot \epsilon_{j+3}$ from the encryption of the message keys. Hence, we simply compute these values and compare their cycle structures with those obtained by running through all possible

$$60 \cdot 26^3 \cdot 26^3 = 18,534,946,560$$

choices for the rotors, positions and ring settings. Note, that when this was done by the Poles in the 1930’s there was only a choice of the ordering of three rotors. The extra choice of rotors did not come in till a bit later. Hence, the total choice was 10 times less than this figure.

The above simplifies further if we assume that no stepping of the second and third rotor occurs during the calculation of the first six ciphertext characters. Recall this happens around 77 percent of the time. In such a situation the cycle structure depends only on the rotor order and the difference $p_i - r_i$ between the starting rotor position and the ring setting. Hence, we might as well assume that $r_1 = r_2 = r_3 = 0$ when computing all of the cycle structures. So, for 77 percent of the days our search amongst the cycle structures is then only among

$$60 \cdot 26^3 = 1,054,560$$

possible cycle structures. Whilst this may seem a lot to do without a modern computer the Polish cryptographers found various ingenious ways of making this tractable.

After the above procedure we have determined all values of the initial day setting bar $p$ and $r$, however we know the differences $p_i - r_i$. We also know for any given message the message key $p'_1, p'_2, p'_3$. Hence, in breaking the actual message we only require the solution for $r_1, r_2$, the value for $r_3$ is irrelevant as the third rotor never moves a fourth rotor. Most German messages started with the same two letter word followed by space (space was encoded by ‘X’). Hence, we only need to go through $26^2$ different positions to get the correct ring setting. Actually one goes through $26^2$ wheel positions with a fixed ring, and use the differences to infer the actual ring settings. Once, $r_i$ is determined from one message the value of $p_i$ can be determined for the day key and then all messages can be trivially broken.

Once the rotor positions had been found for the day, the next task was to compute the positions of the plugs. However, once the rotors are determined the Enigma machine becomes very much like a substitution cipher. It is not quite a substitution cipher, however the same cryptanalytic techniques can be applied. That is one determines the plugboard settings from the viewing the enough ciphertext and from the statistics of the underlying language, which in this case is German.
BIBLIOGRAPHY

GENERATION AND TESTING OF RANDOM NUMBERS

It might seem unlikely, but there are some really great stories about randomness [14]. The way Michael Larson used knowledge of the lack of randomness on the US game show Press Your Luck in 1984, for example, is so great that it warrants being made into a film of some sort [12]. Part of the game involved the players moving around an eighteen square board. The squares were either empty, contained a prize or contained the so-called Whammy character: landing on a prize square won you that prize, landing on the Whammy square lost all prizes won so far.

To make things exciting, the contents of the squares was updated every second or so in a “random” manner. Except it was not random at all. Larson video taped Press Your Luck episodes and played them back frame-by-frame. Then, by writing down the sequence of board states, he discovered that the board in fact cycled through just five simple patterns. Better still, during a given turn there were some squares that would never contain the Whammy. So armed with this knowledge, Larson reasoned that he could carry on playing without really gambling at all: provided he could remember the patterns and which turn he was on, he could always avoid the Whammy. Larson went on the game show and stayed on so long it had to be split into multiple episodes; the look on the presenters face as he consistently avoided the Whammy with seemingly steel-eyed bravery must have been priceless. Well, not exactly priceless: Larson walked off with $110,000 after lawyers from the TV station conceded that he had not cheated. You can still find video of the now legendary episodes on YouTube:

http://www.youtube.com/results?search_query=Press+Your+Luck

I hear a more mundane story much more often: my mother has a love/hate relationship with the UK National Lottery game Lotto [10]. The idea is that she picks six numbers between 1 and 49 and then every Saturday, a machine selects six numbers at random. If her numbers match the ones the machine picked, she wins something: typically the more numbers that match, the more money she gets. But invariably the numbers do not match and she is left to contemplate the injustice of it all. This takes the form of a ritual tirade against the machine: “it’s a fix, having 21 and 22 cannot be random”.

Examples like this beg some questions about what random numbers are and how we generate them. These are quite important questions because random numbers are used in lots of different areas of Computer Science. A good example is that the security of many cryptographic schemes relies on the fact that one can make random choices and choose random numbers. For example, it is common to assume that any key we choose is done so in a random way; if there was some way to predict how we selected it, the key would be more easily guessed and security more easily breached. Setting a password to “X4$ia0!I” is arguably better than “password” for example!
(a) At the start of the game Larson, with opponents Ed and Janie, ...

(b) ... is looking understandably relaxed.

(c) As well as avoiding the Whammy squares ...

(d) ... Larson hits high-value cash prizes ...

(e) ... $4000 in this case.

(f) Ed is understandably depressed by the way the game turns out ...

(g) ... and his inability to avoid the Whammy squares!

(h) Larson ends the game with $110,237 cash ...

(i) ... leaving the host lost for words.

Figure 9.1: Scenes from Press Your Luck, 1984.
9.1 What is randomness?

The term **entropy** is often used by scientists to describe disorder: if a physical system has high entropy it behaves in an unpredictable way. For example as you heat up a gas, predicting how it will behave starts to become harder than if it is in a more stable, cooled state. When we say something is random we mean more or less the same thing: the behaviour we observe follows no deterministic or predictable pattern. As a result we can not write an algorithm to describe it; instead we have to describe it in terms of **probability**, or chance.

Randomness is quite an abstract concept. To make things more concrete, we will talk exclusively about sequences of random numbers. Imagine we want to generate a sequence of such numbers, e.g.,

\[ X = (0, 1, 0, 1). \]

The idea is to use coin flips (or tosses) [3]: in order to generate each \( X_i \), we first throw the coin in the air then and inspect which side it lands on. A real coin has two sides, normally called the tails side and heads side, but we can make life easier by using the number 0 with tails and 1 with heads; we also rule out any freak occurrences such as the coin landing on anything other than one of the sides (e.g., on the edge). As a result, we can say each coin flip makes a random selection from the set \( S = \{0, 1\} \). Clearly we cannot write down an algorithm to describe how the coin behaves, so instead we use a **probability distribution**:

\[
P(x) = \begin{cases} 
\frac{1}{2} & \text{if } x = 0 \text{ i.e., the coin flip was a tail} \\
\frac{1}{2} & \text{if } x = 1 \text{ i.e., the coin flip was a head} 
\end{cases}
\]

Looking at \( P \), for a given output \( x \) we can say what the probability of selecting \( x \) from \( S \) is. In this case, we have a special name for the probability distribution: when the probability for each \( x \) is the same we say the distribution is **uniform**.

The subject of randomness is one where it is quite difficult to prove things categorically. More usually, we define randomness in terms of properties which we can measure. Typically we generate a sequence of random numbers and then use statistical tests of randomness [16] on the sequence. For example we might say that a sequence contains some feature that mean it is **not** random; we discuss two such features below.

9.1.1 Biased versus unbiased

Imagine we get a friend to flip a coin eight times, resulting in the sequence

\[ Y = (1, 1, 1, 1, 1, 1, 1, 0). \]

We would have to ask ourselves whether or not this friend is cheating somehow: intuitively, we might say that the coin is **biased**. On the other hand, this sequence is just as probable as any other. The probability of getting \( Y \) is

\[
\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2^8} = \frac{1}{256}
\]

which is the same as any other sequence of the same length. So given \( Y \) is just as likely to occur as any other sequence, why would we conclude that the coin is unfair?

In more general terms, if we select uniformly at random from a set of \( m \) numbers the probability of selecting a number \( x \) is

\[ P(x) = \frac{1}{m}. \]

If we repeat the selection \( n \) times, we would expect each of the numbers to appear roughly \( \frac{n}{m} \) times on average. If some number \( x \) is selected significantly more or less than \( \frac{n}{m} \) times, we could conclude that the selection process is not random at all: it is biased in favour or against \( x \) somehow. This means that there exists a number \( x \) with \( P(x) \neq \frac{1}{m} \), but we expected that for all \( x \) we would have \( P(x) = \frac{1}{m} \). This is another way of saying the selection process is biased, i.e., \( P \) is not uniform.

The “on average” part in the previous paragraph is important in the sense that we need \( n \) to be large before we start talking about average behaviour. For example if we flip a coin twice and get 1 twice, we cannot conclude that the coin is unfair because we do not have enough evidence. If we flip a coin eight hundred times and get 1 seven hundred times however, we can be more confident that something fishy is going on.

Looking again at the sequence \( Y \), we can start to see why our friend might be cheating. We are selecting from \( m = 2 \) numbers and have repeated the selection \( n = 8 \) times so we would expect each number to occur
\[ \frac{8}{2} = 4 \text{ times. But they do not: we get 1 seven times and 0 just once, so we could conclude that for this limited sample the coin is biased toward 1 and is therefore unfair. Rather than being uniform, based on having seen } Y \text{ we might say the behaviour of the coin is better described by} \]

\[
P(x) = \begin{cases} 
\frac{1}{8} & \text{if } x = 0 \\
\frac{7}{8} & \text{if } x = 1 
\end{cases}
\]

Maybe we should get some more trustworthy friends!

You can get some statistics about the UK National Lottery here

http://www.lottery.co.uk/statistics/

Based on the discussion above, i.e., explaining your answer in terms of Mathematics, probability in particular, is it biased or not?

### 9.1.2 Predictable versus unpredictable

Now imagine we get a different friend to flip another coin eight times, and that this time the resulting sequence is

\[ Z = \langle 1, 0, 1, 0, 1, 0, 1, 0 \rangle. \]

The probability of getting this sequence is still

\[
\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2^8} = \frac{1}{256}
\]

so again we cannot really infer anything from it occurring rather than some other sequence. Also, we can see that the coin is not biased in the same way the other one was: we get 1 four times and 0 four times so the coin is not biased toward either case. On the other hand, we might intuitively say the new coin is still unfair because there is a clear pattern: if a given flip of the coin gives 1, there seems a strong chance the next flip will give 0. If we believe the pattern will continue to hold, this means we can predict the next result with some confidence.

You can think of this in terms of **conditional probability** [2]: the result of a given coin flip should not depend on anything other than the probability distribution. It especially should not depend on the results from previous coin flips: imagine you flip the coin ten times and every time you get 1. We might say a 0 was “due”. But looking at our description of the behaviour, it clearly is not: the probability of getting a 0 on the eleventh flip is still \( \frac{1}{2} \) and the probability of getting a 1 is still \( \frac{1}{2} \). The tendency for people to ignore this is sometimes called the gambler’s fallacy [5]. In our case, we can see that the result of the \( i \)-th coin flip probably is dependent on the \( (i-1) \)-th coin flip. For example if \( Z_{i-1} = 1 \) then it is more probable that \( Z_i = 0 \) than \( Z_i = 1 \). Therefore we could conclude that the coin behaviour is not as well described by the probability distribution as expected.

There is another, perhaps neater way to think about this. Imagine we want to take \( Z \) and compress it by coming up with a shorter way to describe the sequence, just as we did in Chapter 2. For example, imagine we say that the symbol \( \star \) represents the sequence \( \langle 1, 0 \rangle \). We might then describe \( Z \) as the sequence

\[ \bar{Z} = \langle \star, \star, \star, \star \rangle. \]

Why does this work? Well, we know what \( \star \) “means” so we can take \( \bar{Z} \) and reconstruct \( Z \) by just replacing each occurrence of \( \star \) with \( \langle 1, 0 \rangle \). But the way we describe \( \bar{Z} \) is shorter than the way we describe \( Z \) (even if we include the definition of \( \star \)), so in some sense we have used the fact there is a “pattern” to compress the information. We have glossed over a lot of the detail, but there is a fancy name for this general concept: we call it Kolmogorov-Chaitin complexity [7]. Very roughly, this concept says that the more we can compress a sequence the less random it must be; equivalently, the more we can compress a sequence the more usable structure there must be in the sequence.

### 9.1.3 Random versus arbitrary

There used to be a joke about messages from computers that would prompt the user to “press any key” [1]; the joke was that users would find the space key and the escape key, but could not find the “any” key. It seems debatable whether this was ever actually funny, but looking at things more closely highlights an
important point: the message did not read “press a random key”, the choice of key does not matter so really what we mean is “press an arbitrary key”.

The difference between arbitrary and random is sometimes important however. For example, many files used within a UNIX-based operating system start with a magic number [9] which allows us to easily identify their type; if the file starts with the number 8993 this tells us that we can execute it for example. In a sense the choice of 8993 is arbitrary; there is no real reason to choose 8993 rather than say 1234 or 9999, we just needed “any” number. In this sort of situation, it is tempting to just select the numbers at random.

In cryptography, this approach can have some disadvantages. The most famous example is illustrated by the Data Encryption Standard (DES) [4] which was designed and standardised in the US in the late 1970s. DES is a block cipher; it encrypts messages. One of the components in the algorithm is a large table called the S-box whose contents is chosen carefully but somewhat arbitrarily. The problem is, there was no explanation of how the content was generated. People began to become suspicious that the content had been chosen in a special way so that, for example, the US government could decrypt their messages. In turned out that the US government had not selected the S-box content either randomly or arbitrarily. They had actually been selected to prevent an attack which, at the time, only the US governments cryptographers knew about. Even so, the lack of openness over the design choices for the S-box sparked a trend toward use of so-called “nothing up my sleeve” numbers [11]. This idea is used when we want “any” number, but one which we can prove has not been selected with special properties. For example, the number \(\pi\) might not be a good random number, but if all you need is an arbitrary number then it is probably a good choice. For example, people would be hard pushed to prove your choice of \(\pi\) was underhand: it cannot have been generated in some special way.

9.2 Real randomness

9.2.1 Generating randomness

When we talk about real random numbers, we typically mean numbers that come from some physical process. The idea is that we cannot control, or to some extent understand, how these processes might work. In other words, we cannot write down an algorithm that describes them. Radioactive decay is a good candidate. The idea is that an unstable atom will decay by emitting energy in the form of radiation; we know the average rate at which this might happen, but when exactly a given atom will decay is unpredictable. So we could generate random numbers by simply taking a measuring device such as a Geiger counter [6] and have it tell us whenever radioactive decay is detected.

Another candidate is atmospheric noise. In the same way as radioactive decay, it is difficult to predict the level and characteristics of the noise around us. Sampling such noise using even a basic radio or microphone can give quite effective results: Mads Haahr, a lecturer at Trinity College, Dublin has rigged up such a system to the Internet at

http://www.random.org/

The web-site offers a neat interface which we can easily use from BASH by employing the wget command. The idea is to issue a command that mimics what happens when we type a URL into the address bar of a web-browser:

```
bash$ wget -q -U chrome -O- 'http://www.random.org/integers/?num=100&min=0&max=255&col=5&base=10&format=plain&rnd=new'
52 131 189 14 3
118 123 153 46 88
286 77 42 72 171
194 229 113 19 153
55 244 69 46 220
117 9 181 82 82
5 64 164 172 250
119 26 1 87 58
169 231 44 194 182
28 218 243 124 193
148 129 177 71 249
178 262 181 25 47
```
Of course having a Geiger counter attached to every computer is not ideal, and neither is having to access a remote computer over the Internet every time we need to generate random numbers. Fortunately there are lots of devices already connected to your local computer which could do a similar job. Most operating systems have a mechanism for collecting together random events from such devices into what is called an entropy pool [17]. You can think of the entropy pool as a sequence of numbers; the idea is that each time a random event happens, we mix it into the entropy pool by adding some numerical representation of the event onto the end of the sequence. You could imagine that every time someone moves the mouse, the computer might add the mouse speed and direction to the sequence. Then, when someone or something needs to generate a random number, we take one from the start of the sequence.

UNIX systems commonly have two types of entropy pool which they let users access via the /dev/random and /dev/urandom files. Except the files are not really files at all: when we read from them, behind the scenes we are taking random numbers from an entropy pool. The difference between the two is that /dev/random will wait for enough entropy to exist before allowing us to read from it, while /dev/urandom will let us read whenever we want. You can see this as a choice between the quality of the numbers we read and the length of time required to read them. On one hand, if there have not been enough random events to fill the entropy pool it does not make sense to start taking numbers from it; the numbers we read from /dev/random should be more random. On the other hand, waiting until enough random events happen might take a long time; although they might be less random, we can read numbers /dev/urandom with less of a delay.

Again, we can easily read some numbers from /dev/urandom using BASH:

```bash
bash$ cat /dev/urandom | od -Ad -tu1 -w5 -N50 | cut -c 9-
```

That is quite a horrible looking command, so it makes sense to look at what is going on in more detail. Basically, we take /dev/urandom and pass it through a command called od which is controlled using a number of options: we tell od to give us fifty bytes of the input using -N50, to format the output in five columns using -w5, and to format the content as unsigned decimal integers in the range 0...255 inclusive using -t with the format u1. Finally we pass the output of od through cut to remove some information we do not need, i.e., to get just the random numbers.

9.2.2 Testing randomness

So we can generate random numbers, but can we apply similar statistical tests as those discussed earlier to show they are random? As an example, we will look at the tests for bias and predictability; we will compare the real random numbers against English text, which is not random at all. Again we will use Project Gutenberg as a source of text:

http://www.gutenberg.org/

For the sake of argument, we fetch the text for War and Peace by Tolstoy (which is quite large) and save it as the file A.txt. Using the du command we can see that it weighs in at roughly 3 MB:

```bash
bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/files/2600/2600.txt'
bash$ du -b A.txt
3226645 A.txt
bash$
```

Next we take the same amount of random data from /dev/urandom (using du and cut to provide the number of bytes as an option for head), then save it as the binary file B.bin as follows:
bash$ cat /dev/urandom | head -c `du -b A.txt | cut -f 1` > B.bin
bash$

Remember that testing for bias in A.txt and B.bin basically means testing that each possible number we could select has the same probability of selection. The test will be performed in two steps:

1. Take the input file and chop it up into decimal numbers in the range 0…255. The output will need some cleaning up, but the end result will essentially be a long sequence of decimal numbers (one per-line) that represents the file content.

2. Take the sequence and count how many times each number occurs in it.

There are two main things to notice in the output for A.txt:

First, some numbers do not appear at all; for example the output starts at 10 so clearly 0…9 appear zero times. Second, among the numbers that do appear, some appear much more often than others. For example the number 32 appears many times whereas the number 90 appears fewer times. The reason for this is simple. Since A.txt is an ASCII text file, the numbers will be biased toward those which relate to printable ASCII codes. Looking back to Chapter 5, we find that 32 is the ASCII code for SPC, the characters ‘a’…‘z’ have ASCII codes 97…122, and ASCII codes 10 and 13 produce a new line; unsurprisingly these all appear very often! What about B.bin?

bash$ cat B.bin | od -Ad -tu1 -w1 -v | cut -c 9- | grep [0-9]* > D.txt
bash$ cat D.txt | tr -d [:blank:] | grep -v ^$ | sort -n | uniq -c | paste -s

<table>
<thead>
<tr>
<th>Number</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>12539</td>
<td>6</td>
</tr>
<tr>
<td>12627</td>
<td>5</td>
</tr>
<tr>
<td>12669</td>
<td>10</td>
</tr>
<tr>
<td>12525</td>
<td>15</td>
</tr>
<tr>
<td>12438</td>
<td>20</td>
</tr>
<tr>
<td>12688</td>
<td>25</td>
</tr>
<tr>
<td>12650</td>
<td>30</td>
</tr>
<tr>
<td>12739</td>
<td>35</td>
</tr>
<tr>
<td>12628</td>
<td>40</td>
</tr>
<tr>
<td>12654</td>
<td>45</td>
</tr>
<tr>
<td>12454</td>
<td>50</td>
</tr>
<tr>
<td>12723</td>
<td>55</td>
</tr>
<tr>
<td>12565</td>
<td>60</td>
</tr>
<tr>
<td>12663</td>
<td>65</td>
</tr>
<tr>
<td>12663</td>
<td>70</td>
</tr>
<tr>
<td>12663</td>
<td>75</td>
</tr>
<tr>
<td>12663</td>
<td>80</td>
</tr>
<tr>
<td>12413</td>
<td>85</td>
</tr>
<tr>
<td>12663</td>
<td>90</td>
</tr>
<tr>
<td>12663</td>
<td>95</td>
</tr>
<tr>
<td>12569</td>
<td>100</td>
</tr>
<tr>
<td>12581</td>
<td>105</td>
</tr>
<tr>
<td>12559</td>
<td>110</td>
</tr>
<tr>
<td>12440</td>
<td>115</td>
</tr>
<tr>
<td>12546</td>
<td>120</td>
</tr>
<tr>
<td>12577</td>
<td>125</td>
</tr>
<tr>
<td>12521</td>
<td>130</td>
</tr>
<tr>
<td>12545</td>
<td>135</td>
</tr>
<tr>
<td>12513</td>
<td>140</td>
</tr>
<tr>
<td>12521</td>
<td>145</td>
</tr>
<tr>
<td>12513</td>
<td>150</td>
</tr>
<tr>
<td>12513</td>
<td>155</td>
</tr>
<tr>
<td>12513</td>
<td>160</td>
</tr>
<tr>
<td>12773</td>
<td>165</td>
</tr>
<tr>
<td>12576</td>
<td>170</td>
</tr>
<tr>
<td>12353</td>
<td>175</td>
</tr>
</tbody>
</table>

165
The difference is quite obvious: all numbers appear in the output and do so roughly 12600 times on average. There are a few cases which vary a little, but basically we can conclude that there is no bias in B.bin to the same extent there was in A.txt. We might use some further statistics to back this up, for example if we were to measure the standard deviation of B.bin it would not be very large.

Measuring predictability is a little more tricky. One thing we could do quite easily is to approximate the Kolmogorov-Chaitin complexity by seeing how easy it is to compress the file content. The command `bzip2` performs a full blown version of the compression ideas we discussed in Chapter 2. bzip2 processes an input file to identify patterns and replaces them with shorter symbols to produce a compressed output file. It needs to define what those symbols mean so it adds a dictionary to the start of the compressed file so that we can decompress it later if we want to. Using `bzip2` to compress A.txt and B.bin is simple; to be fair, we instruct `bzip2` to make the best effort it can at compressing the files rather than worrying about producing the result quickly:

```
bash$ bzip2 -c -9 A.txt > A.txt.bz2
bash$ bzip2 -c -9 B.bin > B.bin.bz2
bash$
```

We then inspect how big the resulting files are, again using the `du` command we used earlier:

```
bash$ du -b A.txt A.txt.bz2
 3226645 A.txt
  883666 A.txt.bz2
bash$ du -b B.bin B.bin.bz2
 3226645 B.bin
 3241465 B.bin.bz2
bash$
```

Inspecting the results confirms more or less what we expected: the *War and Peace* text A.txt was compressed into A.txt.bz2, which is roughly a quarter of the size. This is what we would expect since A.txt represents English text; for example the word “and” probably appears very often so we might imagine replacing each occurrence with a one character symbol. However, the randomness produced by `/dev/urandom` in B.bin is processed by bzip2 to produce the file B.bin.bz2. In this case the output of `/dev/urandom` is so hard to compress that the overhead of adding the dictionary to the compressed file has meant the end result is larger than what we started with!

Maybe `bzip2` is just not very good at compressing files (or at least this file? To test this, we can try the same thing with some other compression tools like gzip and get similar results:

```
bash$ gzip -c -9 A.txt > A.txt.gz
bash$ gzip -c -9 B.bin > B.bin.gz
bash$
```

```
bash$ du -b A.txt A.txt.gz
 3226645 A.txt
  1193969 A.txt.gz
bash$ du -b B.bin B.bin.gz
 3226645 B.bin
  3227164 B.bin.gz
bash$
```

Or perhaps using the zip compression method:

```
bash$ zip -q -9 A.zip A.txt
bash$ zip -q -9 B.zip B.bin
bash$
```

```
bash$ du -b A.zip A.zip
 3226645 A.txt
1194105 A.zip
bash$ du -b B.zip B.bin
 3226645 B.bin
 3227164 B.zip
bash$
```

```
bash$ du -b A.txt A.txt
 3226645 A.txt
883666 A.txt.bz2
bash$ du -b B.bin B.bin
 3226645 B.bin
3241465 B.bin.bz2
bash$
```

166
Of course, this still does not prove anything; perhaps bzip2, gzip and zip are all missing some obvious way to compress the files. But basically we conclude from our tests that A.txt is quite easy to compress whereas B.bin is at least harder to compress. This means the Kolmogorov-Chaitin complexity of A.txt is quite low: we can make a shorter description of A.txt, so it is not very random. On the other hand, for B.bin the Kolmogorov-Chaitin complexity is higher: it is not as easy to describe B.bin in a shorter way, so in a sense it is more random.

Using the commands described above for compressing files, set yourself a challenge: what is the

1. **largest** file you can create that can be compressed into the **smallest** size, and
2. **smallest** file you can create that can be compressed into the **largest** size.

Put more precisely, imagine the original and compressed files have sizes $x$ and $y$ bytes respectively: you are trying to maximise or minimise the compression ratio $x/y$. In each case, explain what your strategy for constructing the original file is, i.e., what features does it have that make it a good choice?

### 9.3 Fake randomness

#### 9.3.1 Generating randomness

The problem with real random numbers is that sometimes they are *too* random for our purposes. Yes, really: imagine we want to simulate a scientific experiment using a computer; this is a good idea if the real experiment would be impractical. For example, maybe we would like to experiment with things that expand very fast and generate large amounts of heat; usually we call these things explosions and doing real experiments can be quite hazardous! So instead, we simulate the explosion on a computer. On one hand it is quite possible we would want a source of random numbers to model parts of the experiment, for example randomness in atomic-level behaviour. On the other hand it would be a good idea if we could repeat the simulation and get the *same* results. Using real random numbers is not ideal because of the second reason: we cannot reproduce the real randomness so we would need to generate all the numbers, store them somewhere and then look them up if we needed them again.

A solution is to use **pseudo-random** numbers [13]. Whereas real randomness cannot be described using an algorithm, pseudo-randomness can. The idea is to think of the pseudo-random numbers we generate as a sequence called $R$. We start by specifying the first element in the sequence $R_0$, this is called the **seed**. Then, to generate the next element $R_1$ we apply the algorithm to compute it for us; we can perform this over and over again so that more generally if we have $R_i$, the $i$-th element in the sequence, we can generate the next element $R_{i+1}$. The thing we need to be very careful about is that the sequence we generate still passes the tests for randomness we looked at previously: although pseudo-random numbers are in a sense fake, they should still satisfy our definition of what randomness looks like. If we are successful, reproducing pseudo-random numbers becomes a matter of using the algorithm over and over again rather than storing a large amount of real random numbers: we have traded more computation for less storage.

A **Linear Congruence Generator (LCG)** is one way to generate pseudo-random numbers [8]. The algorithm we use to compute the next element of the sequence given the current element is

$$R_{i+1} = a \cdot R_i + c \pmod{p}.$$

This means that we take $R_i$, multiply it by some number $a$, and finally add another number $c$: the result $R_{i+1}$ is produced by using modular arithmetic where the modulus we are using is $p$. Imagine we select $a = 5$, $b = 1$, $p = 8$ and set the seed value $R_0 = 2$. We can apply the LCG equation to generate successive elements
in the sequence

\[
\begin{align*}
R_0 &= 2 \\
R_1 &= 5 \cdot R_0 + 1 \mod 8 = 11 \mod 8 = 3 \\
R_2 &= 5 \cdot R_1 + 1 \mod 8 = 16 \mod 8 = 0 \\
R_3 &= 5 \cdot R_2 + 1 \mod 8 = 1 \mod 8 = 1 \\
R_4 &= 5 \cdot R_3 + 1 \mod 8 = 6 \mod 8 = 6 \\
R_5 &= 5 \cdot R_4 + 1 \mod 8 = 31 \mod 8 = 7 \\
R_6 &= 5 \cdot R_5 + 1 \mod 8 = 36 \mod 8 = 4 \\
R_7 &= 5 \cdot R_6 + 1 \mod 8 = 21 \mod 8 = 5 \\
R_8 &= 5 \cdot R_7 + 1 \mod 8 = 26 \mod 8 = 2 \\
R_9 &= 5 \cdot R_8 + 1 \mod 8 = 11 \mod 8 = 3 \\
\vdots & \quad \vdots & \quad \vdots
\end{align*}
\]

Think back to our example application of pseudo-randomness where we wanted to use a computer to simulate something, and imagine we want to start the simulation half way through. Normally this would be tricky: we would first have to start our random number generator at the beginning using the seed element, and then repeatedly compute the next element until we got to the point we actually wanted to start. Depending on where this is, we might waste quite a bit of time just finding the \(R_i\) we want.

The LCG is just one example of a Pseudo-Random Number Generator (PRNG). Find out about at least one other type, including how it works; compare it with the LCG in terms of any advantages and disadvantages each might offer given a particular context or application.

The LCG has a nice property which allows us to avoid this wasted time by skipping ahead in the sequence. We already have a way to skip ahead by one element, namely

\[R_{i+1} = a \cdot R_i + c \pmod{p}.
\]

What if we take \(R_{i+1}\) and apply the equation again? We would skip ahead two elements, and the result would look a bit like

\[R_{i+2} = a \cdot S + c \pmod{p}.
\]

where \(S = a \cdot R_i + c\); writing things out fully gives a way to go straight from \(R_i\) to \(R_{i+2}\)

\[R_{i+2} = a \cdot (a \cdot R_i + c) + c \pmod{p}.
\]

Pulling the same trick again we can compute what the result of skipping ahead three elements would be

\[R_{i+3} = a \cdot T + c \pmod{p}.
\]

where \(T = a \cdot (a \cdot R_i + c) + c\); things are starting to get a bit of a mess, but we can again write things out to get straight from \(R_i\) to \(R_{i+3}\)

\[R_{i+3} = a \cdot (a \cdot (a \cdot R_i + c) + c) + c \pmod{p}.
\]

The aim of all this is not to give you a headache, but rather to show that there is a pattern emerging: if we start at \(R_i\) then to skip ahead \(k\) elements means that first we multiply \(R_i\) by \(a\) a total of \(k\) times, but also add \(c\) a total of \(k\) times as we go. The problem is, the multiplications by \(a\) and additions of \(c\) are sort of mixed up. We can unravel the mess by rewriting things a bit

\[
\begin{align*}
R_{i+3} &= a \cdot (a \cdot (a \cdot R_i + c) + c) + c \pmod{p} \\
&= a \cdot (a^2 \cdot R_i + a \cdot c + c) + c \pmod{p} \\
&= a^3 \cdot R_i + a^2 \cdot c + a \cdot c + c \pmod{p}
\end{align*}
\]

The important thing to notice from the above is that given \(R_i\) we just need to compute two values which we can write down as

\[
\begin{align*}
A(k) &= a^k \\
C(k) &= a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c
\end{align*}
\]

and then we can use them to compute

\[R_{i+k} = A(k) \cdot R_i + C(k) \pmod{p}.
\]
If we plug in the specific case of \( k = 3 \) the terms are as we expect, i.e.

\[
\begin{align*}
A(3) &= a^3 \\
C(3) &= a^2 \cdot c + a \cdot c + c
\end{align*}
\]

which gives

\[
R_{i+3} = a^3 \cdot R_i + a^2 \cdot c + a \cdot c + c.
\]

The great thing is, since \( a \) and \( c \) are constant values, so if we know \( k \) before we start then we can also compute \( A(k) \) and \( C(k) \) before we start. But what if we do not know \( k \) before we start? What is the best way to compute \( A(k) \) and \( C(k) \)? Certainly we want to do this in an inexpensive way, otherwise we might as well perform \( k \) steps, skipping ahead 1 element each time.

The \( A(k) \) part is quite easy, but the \( C(k) \) part looks much less pleasant. We have got an expression of the form

\[
a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c
\]

to compute. This looks bad because for a large value of \( k \) the number of additions and multiplications we need to do is also quite large. However, we are saved because this large expression is actually the same as the much nicer looking

\[
\frac{(a^k - 1) \cdot c}{a - 1}.
\]

To see why, multiply the first expression by \((a - 1)\), to obtain

\[
(a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a \cdot c + c) \cdot (a - 1) = (a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c) \cdot a
\]

\[
- (a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c)
\]

\[
= (a^k \cdot c + a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c)
\]

\[
- (a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c)
\]

\[
= a^k \cdot c - c
\]

\[
= (a^k - 1) \cdot c.
\]

Now dividing both sides by \((a - 1)\) we obtain

\[
a^{k-1} \cdot c + a^{k-2} \cdot c + \cdots + a^2 \cdot c + a \cdot c + c = \frac{(a^k - 1) \cdot c}{a - 1}.
\]

This is great news because we already have to compute \( a^k \) in order to get \( A(k) \), so given that \( a - 1 \) is also just a constant value we only need to do one subtraction, one multiplication and one division to get \( C(k) \) rather than the mass of multiplications and additions that we started with. In other words we have

\[
C(k) = \frac{(A(k) - 1) \cdot c}{a - 1}.
\]

Computing the sequence an LCG produces by hand is somewhat tedious, so how might we automate the process? One way would be to write a dedicated program for the task; since we want to focus on the concepts rather than teach programming, we will instead try to automate the process using only existing \texttt{BASH} commands. Our approach is to write a small \texttt{BASH} \texttt{script} [15] which will generate \( n \) elements of the sequence given \( R_0, a, c \) and \( p \). You can think of the script as creating a new command which we can use as follows:

```
bash$ ./P.sh n R a c p
```

In other words it will produce the output of the LCG for values of \( i \) in the range \( 1, 2, \ldots, n \), starting at position \( R_0 \). We create the script, which we call \texttt{P.sh}, as follows:

```bash
#!/bin/bash

R=$1

for ((i = 0; i < $1; i += 1)); do
    echo "$R"
    R=$(( ( ( $3 ) * $R ) + $4 ) % $5 )
done

bash$ chmod 775 P.sh
bash$
```
Note that cat is used to capture input from the user and save it into a file called P.sh. The permissions of P.sh are set using chmod so we can use it as a script (rather than just a normal file). For example, to replicate the sequence we looked at above, we might run P.sh as follows:

```
bash$ ./P.sh 100 5 1 8
```

where we read $\{1\}$ as 100 meaning $n = 100$, $\{2\}$ as 2 meaning $R_0 = 2$, $\{3\}$ as 5 meaning $a = 5$, $\{4\}$ as 1 meaning $c = 1$, and finally $\{5\}$ as 8 meaning $p = 8$.

Although overall it might seem unfamiliar, each step of the script is easy to explain: you can think of it as an algorithm, but written in a slightly different way. First we assign $R$ to $\{2\}$ which is the second option given to P.sh on the command line (i.e., the seed). Then we use a loop to iterate over a block of statements $\{1\}$ times, where $\{1\}$ is the first option given to P.sh on the command line (i.e., the value of $n$). The block does two things. First it writes the current value of $R$ (i.e., the value $R_i$) to standard output using the echo command, then updates $R$ to the next value (i.e., $R_{i+1}$) using the method we have already seen. Note that $\{3\}$, $\{4\}$ and $\{5\}$ are the third, fourth and fifth options given to P.sh on the command line (i.e., the values of $a$, $c$ and $p$).

### 9.3.2 Testing randomness

Hang on a second: that sequence we just generated looks like it just repeats over and over again! We call the number of elements before this repetition occurs the **period** of the sequence and in fact, the case where the period is equal to $p$ is the best we can hope for. Since we are computing elements modulo $p$ there are only $p$ possibilities, i.e., numbers in the range $0 \cdots p - 1$. So in our case, after the eighth element the sequence **must** repeat because the next element **must** be one we have already seen. It turns out that depending on the choices of $a$, $c$ and $p$ things can get even worse:

```
bash$ ./P.sh 100 2 0 1 8 | paste -s -d ''
```

The first four cases are a bit dumb: we select $a$ or $c$ as zero for example, we do not really update $R_i$ to get a new $R_{i+1}$ as we would like. The fifth and sixth cases are a bit more subtle however. In the fifth case the period is one: we always get 6 from the generator after the seed even though, at face value, our choices of $a$, $c$ and $p$ are not very special. In the sixth case a slightly better situation occurs: the period is four which is less disastrous, but also less than the value of ten which we would hope to get given the choice of $p$. Hopefully it is clear that **none** of these cases are attractive: if the sequence repeats itself after $p$ elements (or less) and we generate more than that many elements, then there will be a pattern that we can use to compress the sequence and it does not pass our tests for randomness any more. So in some sense, the smaller the period the less randomness there is for us to use.

We can use some rules of thumb to avoid these problems. An often used approach is to select the seed element $R_0$ as the current time and date; this ensures that we get a different starting point (more or less) every time we generate the sequence. We also need to select $p$ so that it is large enough that there would not be any (or at least very few) cycles. Finally, we should select $a$ and $c$ in such a way that the sequence

---

1 Using cat here is a bit awkward: we do so simply to show this example within the same BASH-based setting as the others. An easier way to create P.sh might of course be to use a text editor.
we generate is not trivially bad (like those above). Putting all this together we can run P.sh as follows for example:

```bash
bash$ cat > Q.sh
#!/bin/bash

R=${1}
p=${2}
r=${3}
r=$(( ( ( ${r} * ${p} ) + ${q} ) % 256 ))

bash$ chmod 775 Q.sh
bash$ ./P.sh 100
```

and get an output which reassuringly looks like nonsense!

Here we selected $p = 2^{31} - 1 = 214783647$ so we would expect there to be no repetitions in the sequence unless we generated a huge amount of output. But how can we be sure that what we have generated passes our tests for randomness? The easy answer to this is to actually run those tests. To do this we need to update our script a little. We create the new script, which we call Q.sh, as follows:

```bash
bash$ cat > Q.sh
#!/bin/bash

R=${1}
p=${2}
r=${3}
r=$(( ( ( ${r} * ${p} ) + ${q} ) % 256 ))

bash$ chmod 775 Q.sh
bash$
```

The new Q.sh script is used in exactly the same way as P.sh. There are two main differences however. Firstly, we do not write each $R_i$ exactly, but instead compute $r_i = R_i \mod 256$. So basically, the LCG is generating the sequence $R$ which looks like

$$\begin{align*}
R_0 &= 16807 \\
R_1 &= 16807 \cdot R_0 + 0 \mod 2147483647 = 7749875770000 \mod 2147483647 = 1754771624 \\
R_2 &= 16807 \cdot R_1 + 0 \mod 2147483647 = 2942446684568 \mod 2147483647 = 1053760317 \\
R_3 &= 16807 \cdot R_2 + 0 \mod 2147483647 = 17710549647819 \mod 2147483647 = 252011010 \\
R_4 &= 16807 \cdot R_3 + 0 \mod 2147483647 = 4235549045070 \mod 2147483647 = 711293186 \\
R_5 &= 16807 \cdot R_4 + 0 \mod 2147483647 = 11954704577102 \mod 2147483647 = 1810597900 \\
R_6 &= 16807 \cdot R_5 + 0 \mod 2147483647 = 30430718905300 \mod 2147483647 = 875627310 \\
R_7 &= 16807 \cdot R_6 + 0 \mod 2147483647 = 14716668199170 \mod 2147483647 = 2110249926 \\
R_8 &= 16807 \cdot R_7 + 0 \mod 2147483647 = 35466970506282 \mod 2147483647 = 1278076077 \\
R_9 &= 16807 \cdot R_8 + 0 \mod 2147483647 = 21480624626139 \mod 2147483647 = 1493188845 \\
\end{align*}$$

but we are actually using

$$\begin{align*}
r_0 &= R_0 \mod 256 = 461110000 \mod 256 = 240 \\
r_1 &= R_1 \mod 256 = 1754771624 \mod 256 = 168 \\
r_2 &= R_2 \mod 256 = 1053760317 \mod 256 = 61 \\
r_3 &= R_3 \mod 256 = 252011010 \mod 256 = 2 \\
r_4 &= R_4 \mod 256 = 711293186 \mod 256 = 2 \\
r_5 &= R_5 \mod 256 = 1810597900 \mod 256 = 12 \\
r_6 &= R_6 \mod 256 = 875627310 \mod 256 = 46 \\
r_7 &= R_7 \mod 256 = 2110249926 \mod 256 = 198 \\
r_8 &= R_8 \mod 256 = 1278076077 \mod 256 = 173 \\
r_9 &= R_9 \mod 256 = 1493188845 \mod 256 = 237 \\
\end{align*}$$

Secondly, we do not write each $r_i$ directly to standard output as text; instead we add some nasty looking printf commands that result in the script writing binary output. The idea behind both alterations is that
Q.sh generates output in the same format as /dev/urandom (i.e., binary values in the range 0...255) so we can use the same testing strategy.

Running Q.sh with the parameters above that we eventually decided on for P.sh, we can generate a binary file called C.bin which is the same size as our original tests:

```
bash$ ./Q.sh 3288738 `date +\%Y%m%d` 16807 0 2147483647 > C.bin
bash$
```

This takes quite a long time; BASH is not really intended for this sort of thing so our script is not exactly tuned for performance. Even so, it gives us a result eventually and we can perform the same analysis as before:

```
bash$ cat C.bin | od -Ad -tu1 -w1 -v | cut -c 9- | grep [0-9]* > D.txt
bash$
```

```
cat D.txt | tr -d [:blank:] | grep -v ^$ | sort -n | uniq -c | paste -s
12817 0 12889 1 12917 2 12732 3 12948 4 12817 5 12766 6 12713 7 12965 8 12877 9
12766 10 12789 11 12861 12 12862 13 12654 14 12950 15 12784 16 13027 17 12801 18
12887 19 12780 20 12721 21 12944 22 12921 23 12980 24 12825 25 12845 26 12963 27
13059 28 12769 29 12698 30 12944 31 12921 32 12754 33 12822 34 13063 35 12949 36
12859 37 12654 38 12950 39 12866 40 12734 41 12770 42 12578 43 12701 44 12709 45
13060 46 12726 47 12608 48 13075 49 12990 50 12935 51 12836 52 12903 53 13034 54
12744 55 12785 56 12768 57 11555 58 12675 59 12744 60 12831 61 12856 62 12807 63
13047 64 13157 65 12869 66 12857 67 12762 68 12899 69 12748 70 12759 71 12984 72
12836 73 12936 74 12794 75 12977 76 12914 77 12812 78 12863 79 12768 80 12893 81
12980 82 12836 83 12922 84 12980 85 12836 86 12893 87 12980 88 12836 89 12768 90
12893 91 12980 92 12836 93 12893 94 12980 95 12836 96 12980 97 12917 98 12757 99
```

This looks good: in a similar way to B.bin, the output of /dev/urandom, we can see that every number is appears in the output and do so roughly 12800 times on average. Again there are a few cases which vary a little, but again we can conclude that there is no significant bias in C.bin. What about the test for predictability? Running bzip2 again gives a positive result:

```
bash$ bzip2 -c -9 C.bin > C.bin.bz2
bash$
```

The randomness produced by the LCG in C.bin is processed by bzip2 to produce the file C.bin.bz2; again slightly larger than the original. An interesting thing to note is that this proves bzip2 is not a perfect compression algorithm: if it was perfect it would have detected that the file contained the output of our LCG, and then compressed the entire file into a description of the LCG (i.e., Q.sh), plus the requisite parameters.
We know that the output of the LCG is only pseudo-random so what can we conclude? The first thing to highlight repeats what we already said: we cannot really prove anything about randomness, we can just test for features. In a way, this perhaps hints that our tests are not good enough to capture the difference between real randomness and pseudo-randomness. The second thing is that we have to be pragmatic about what we actually want: the pseudo-random results look random, and we have shown that for our application their generation is perhaps more attractive than using real randomness. So in a way, who cares? They might not really be random but as long as we can test them and they are good enough for our purposes, then their use should not be viewed as invalid.
CHAPTER 10
SAFETY IN NUMBERS: MODERN CRYPTOGRAPHY FROM ANCIENT ARITHMETIC

Think back to the \( k \)-place shift encryption scheme that was introduced in Chapter 7, and consider two parties, a sender called Alice and a receiver called Bob, using it to communicate with each other. We might describe how Alice and Bob behave using a diagram:

\[
\begin{array}{ccc}
\text{Alice} & \text{Eve} & \text{Bob} \\
\hline
C & \leftarrow & \text{EN}_k(M) \\
\hline
\text{C} & \rightarrow & M \\
\hline
M & \leftarrow & \text{DE}_k(C)
\end{array}
\]

This description is a simple cryptographic protocol: it describes the computational steps each party performs and the communication steps that occur between them. Note that we could have named the parties Angharad and Bryn but, somewhat bizarrely, an entire menagerie of standard names are used within protocol descriptions of this sort [1]. Unless we use Alice and Bob, cryptographers get confused as to what their roles are! In this case, the protocol that Alice and Bob engage in has three simple steps read from top-to-bottom:

1. Alice encrypts a plaintext, e.g., \( M = 'a' \), using the key \( k = 3 \) to produce the ciphertext message
   \[
   C = 'd' = \text{EN}_k('a').
   \]

2. Alice communicates \( C \) to Bob.

3. Bob decrypts the ciphertext, e.g., \( C = 'd' \), using the key \( k = 3 \) to recover the plaintext message
   \[
   M = 'a' = \text{DE}_k('d').
   \]

There are some important features of this protocol that demand further discussion:

- The diagram shows a third party: Eve is a passive adversary who hopes to learn \( M \) (or \( k \)) just by observing what Alice and Bob communicate to each other (in this case \( C \)). If Eve is allowed to alter the communication rather than just observe it, she turns into Mallory the active, malign adversary. We saw why this could be a problem at the start of Chapter 7: Thomas Phelippines was basically doing the same thing with messages communicated between Mary Stewart and Anthony Babington.
• With the encryption scheme we have, both $M$ and $C$ are single characters. Formally, we have block cipher [2] where each block is one character in size. What if Alice wants to send a sequence of characters (i.e., a string) to Bob? This is easy to accommodate: we simply apply the encryption scheme independently to each of the characters (i.e., the blocks) in the sequence. That is, take the $i$-th block of the plaintext and encrypt it to form the $i$-th block of the ciphertext

$$C_i = \text{ENC}_k(M_i)$$

and vice versa

$$M_i = \text{DEK}_k(C_i).$$

• Both Alice and Bob need to know and use the same shared key. We call this type of encryption scheme a symmetric-key [24] block cipher because the use of $k$ is symmetric, i.e., the same for both Alice and Bob. In some circumstances this poses no problem at all, but in others it represents what we call the key distribution problem: given that Eve can see everything communicated between Alice and Bob, how can they decided on $k$ in the first place?

The way we use a block cipher is, roughly speaking, called a mode of operation [3]. For example, although

$$C_i = \text{ENC}_k(M_i)$$

and

$$M_i = \text{DEK}_k(C_i).$$

describe the Electronic Codebook (ECB) mode, various alternatives also exist. Find out about at least one alternative, plus when and why it might be preferred. Try to write equations, similar to those above, that describe how the mode produces a block of ciphertext from the corresponding block of plaintext (and vice versa).

The focus of this Chapter relates to the final point above, and two potential solutions more specifically: key agreement protocols [15] and public-key encryption schemes [21]. Both are interesting from a historical point of view, but also underpin a widely-used application of modern cryptography: the Secure Sockets Layer (SSL) [25] system that permits modern e-commerce to exist in a usable form. If you open a web-browser and load a secure web-site (whose URL will often start with https:// rather than http://), these technologies enable you to shop online without a real Eve finding out and using your credit card number!

Additionally, they both rely on a field of mathematics called number theory [17]. Many of the algorithms within this field have been studied for thousands of years; two of the most famous are Euclid’s algorithm [8] for computing the greatest common divisor of two numbers, and the binary exponentiation algorithm [10] that we saw in Chapter 3. Both are often studied as examples in Computer Science because, aside from being useful, they have a number of attractive properties. In particular they

1. are short and fairly easy to understand (or at least explain),
2. allow simple arguments and proofs about their correctness, and
3. allow relatively simple analysis of their computational complexity.

Our aim is to demonstrate these properties and to show how these ancient algorithms support the modern cryptographic protocols on which we now rely.

### 10.1 Modular arithmetic: the theory

Chapter 7 described modular arithmetic as being like the “clock arithmetic” you need when reading the time from an analogue clock face such as Figure 10.1a. Based on someone saying “the time is now ten o’clock; I will meet you in four hours”, our example question asked what time they mean? We reasoned that since

$$(10 + 4) = 14 \equiv 2 \pmod{12},$$

the answer is two o’clock. We started being more formal by saying that the $\equiv$ symbol means equivalent to, so writing

$$x \equiv y \pmod{N}.$$
Figure 10.1: Two analogue clock faces, one standard 12-hour version and a slightly odd 13-hour alternative.
means $x$ and $y$ give the same remainder after division by the **modulus $N$**. This is the same as saying there exists some $\kappa$ so that

$$x = y + \kappa \cdot N,$$

i.e., $x$ equals $y$ after some multiple of the modulus is added on. We can see this from our example: we can say

$$14 \equiv 2 \pmod{12}$$

precisely because for $\kappa = 1$

$$14 = 2 + 1 \cdot 12,$$

i.e., both 14 and 2 give the remainder 2 after division by 12.

### 10.1.1 Rules for modular addition

Although things look more complicated, many of the rules of addition and subtraction you already know still apply to modular arithmetic:

$$x + y \equiv y + x \pmod{N}$$

$$x + (y + z) \equiv (x + y) + z \pmod{N}$$

$$x + 0 \equiv x \pmod{N}$$

In the last rule, 0 is called an **additive identity**. More generally, an **identity function** [13] gives the same output as the input. For example,

$$f(x) = x$$

means $f$ is an identify function because whatever $x$ we give it as input, the output is $x$ as well. Here, you can think of the “add 0 function” as an identity function as well, i.e.,

$$f(x) = x + 0 \equiv x \pmod{N}.$$

A useful question to ask is whether for a given $x$ we can find $y$, the **additive inverse** of $x$, which produces

$$x + y \equiv 0 \pmod{N}.$$  

That is, can we find a $y$ which when added to $x$ produces the additive identity as a result? One way to answer this question is to just search through all possible values of $y$. If $x = 3$ and $N = 12$ for example, we can look at the following possibilities:

$$
\begin{align*}
3 + 0 & \equiv 3 \pmod{12} & 3 + 6 & \equiv 9 \pmod{12} \\
3 + 1 & \equiv 4 \pmod{12} & 3 + 7 & \equiv 10 \pmod{12} \\
3 + 2 & \equiv 5 \pmod{12} & 3 + 8 & \equiv 11 \pmod{12} \\
3 + 3 & \equiv 6 \pmod{12} & 3 + 9 & \equiv 0 \pmod{12} \\
3 + 4 & \equiv 7 \pmod{12} & 3 + 10 & \equiv 1 \pmod{12} \\
3 + 5 & \equiv 8 \pmod{12} & 3 + 11 & \equiv 2 \pmod{12}
\end{align*}
$$

The one we want is $y = 9$ which produces $3 + 9 \equiv 0 \pmod{12}$. But can we always find an additive inverse, no matter what $x$ and $N$ are? Ignoring modular arithmetic, the intuitive answer would be to set $y = -x$ meaning that $x + y = x - x = 0$. This also works for modular arithmetic. Remember from Chapter 7 that $-x \pmod{N} = N - x$, so setting $y = -x$ gives us

$$x - y \equiv x + (-y) \equiv x + (N - x) \equiv N \equiv 0 \pmod{N}.$$  

In our example above, this is demonstrated by

$$3 - 3 \equiv 3 + (-3) \equiv 3 + (12 - 3) \equiv 12 \equiv 0 \pmod{N},$$

which produces exactly what we wanted.
10.1.2 Rules for modular multiplication

In Chapter 7 we only used modular addition, but it should be no great surprise that we can consider other operations as well. We saw in Chapter 3 that multiplication is just repeated addition, so it follows that modular multiplication could be similar.

As another example, suppose you started a job at ten o’clock and worked in three shifts each of two hours. When you finished, the time would be given by

\[ 10 + (3 \cdot 2) = 16 \equiv 4 \pmod{12}, \]

i.e., you would have finished at four o’clock. Of course this is just the same as

\[ 10 + (2 + 2 + 2) = 16 \equiv 4 \pmod{12}. \]

We can again rely on existing rules for multiplication:

\[
\begin{align*}
x \cdot y &\equiv y \cdot x \pmod{N} \\
x \cdot (y \cdot z) &\equiv (x \cdot y) \cdot z \pmod{N} \\
x \cdot 1 &\equiv x \pmod{N}
\end{align*}
\]

This time 1 is called a **multiplicative identity**; the “multiply by 1 function” is another identity function. Since this resembles the additive identity case we saw previously, we can ask the same question: can we always find a \( y \) so that \( x \cdot y \equiv 1 \pmod{N} \)? The answer this time is no, not always. If \( x = 3 \) and \( N = 12 \), we show this by searching through all values of \( y \) again:

\[
\begin{align*}
3 \cdot 0 &\equiv 0 \pmod{12} & 3 \cdot 6 &\equiv 6 \pmod{12} \\
3 \cdot 1 &\equiv 3 \pmod{12} & 3 \cdot 7 &\equiv 9 \pmod{12} \\
3 \cdot 2 &\equiv 6 \pmod{12} & 3 \cdot 8 &\equiv 0 \pmod{12} \\
3 \cdot 3 &\equiv 9 \pmod{12} & 3 \cdot 9 &\equiv 3 \pmod{12} \\
3 \cdot 4 &\equiv 0 \pmod{12} & 3 \cdot 10 &\equiv 6 \pmod{12} \\
3 \cdot 5 &\equiv 3 \pmod{12} & 3 \cdot 11 &\equiv 9 \pmod{12}
\end{align*}
\]

None of the \( y \) gives us the result we want. Why?! Remember that finding such a \( y \) is the same as finding a \( \kappa \) which satisfies

\[ 3 \cdot y = 1 + 12 \cdot \kappa \]

or put another way,

\[ 3 \cdot y - 12 \cdot \kappa = 1. \]

The left-hand side of this equation is divisible by three, i.e., we can write it as

\[ 3 \cdot (y - 4 \cdot \kappa) = 1 \]

instead, whereas the right-hand side is not; this means that no such \( \kappa \) can exist. What about some other values of \( x \), say \( x = 7 \)? This time things work out

\[
\begin{align*}
7 \cdot 0 &\equiv 0 \pmod{12} & 7 \cdot 6 &\equiv 6 \pmod{12} \\
7 \cdot 1 &\equiv 7 \pmod{12} & 7 \cdot 7 &\equiv 1 \pmod{12} \\
7 \cdot 2 &\equiv 2 \pmod{12} & 7 \cdot 8 &\equiv 8 \pmod{12} \\
7 \cdot 3 &\equiv 9 \pmod{12} & 7 \cdot 9 &\equiv 3 \pmod{12} \\
7 \cdot 4 &\equiv 4 \pmod{12} & 7 \cdot 10 &\equiv 10 \pmod{12} \\
7 \cdot 5 &\equiv 11 \pmod{12} & 7 \cdot 11 &\equiv 5 \pmod{12}
\end{align*}
\]

and we find that for \( y = 7, x \cdot y \equiv 1 \pmod{12} \). The reason is that given the equation

\[ 7 \cdot y - 12 \cdot \kappa = 1 \]

we can write, using \( \kappa = 4 \),

\[ 7 \cdot 7 - 12 \cdot 4 = 1. \]

We are allowed to call this \( y \) the **multiplicative inverse** of the corresponding \( x \); if you take \( x \) and multiply it with its multiplicative inverse, you end up with the multiplicative identity.
10.1.3  The sets \( \mathbb{Z}, \mathbb{Z}_N \) and \( \mathbb{Z}_N^* \)

The set of integers [14], or “whole numbers”, is written using a special symbol:

\[ \mathbb{Z} = \{ \ldots , -3, -2, -1, 0, +1, +2, +3, \ldots \} . \]

Clearly this is an infinite set: the continuation dots at the right- and left-hand ends highlight the fact that we can write down infinitely many positive and negative integers. With modular arithmetic, we deal with a sub-set of the integers. By using a modulus \( N \), we only deal with 0 through to \( N - 1 \) because anything equal to or larger than \( N \) or less than 0 “wraps around” (as we saw with the number line in Chapter 7).

More formally, we deal with the set

\[ \mathbb{Z}_N = \{ 0, 1, 2, 3, \ldots , N - 1 \} , \]

which for \( N = 12 \) would obviously be

\[ \mathbb{Z}_{12} = \{ 0, 1, 2, 3, \ldots , 11 \} . \]

When you see \( \mathbb{Z}_N \), take it to mean the set of integers modulo \( N \); the only numbers we can work with are 0 through to \( N - 1 \).

Given an \( N \) we already know that for some \( y \in \mathbb{Z}_N \), but not all, we can find a multiplicative inverse; this is a bit of a pain unless we know beforehand which values of \( y \) and \( N \) will work. For example, for \( N = 12 \) we can write out a table and fill in the multiplicative inverses for each \( y \):

<table>
<thead>
<tr>
<th>( y )</th>
<th>Multiplicative Inverse (mod 12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>1</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>2</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>3</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>4</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>5</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>6</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>7</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>8</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>9</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>10</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>11</td>
<td>( \equiv 1 )</td>
</tr>
</tbody>
</table>

The question marks in the table highlight entries that we cannot fill in, with the values \( y \in \{ 1, 5, 7, 11 \} \) being those with a multiplicative inverse.

Recall that if some \( p \) is a prime number [20], it can only be divided exactly by 1 and \( p \); we say 1 and \( p \) are the only divisors of \( p \). Strictly speaking 1 is a prime number based on this definition, but 1 is not usually very interesting so we tend to exclude it. Two facts explain why the result above is no accident:

1. 2 and 3 are the two prime divisors of 12, i.e., the divisors of 12 which are prime. For example, 6 is a divisor of 12 but not prime, 5 is not a divisor but is prime, whereas 3 is both a divisor of 12 and prime.

2. \( \{ 1, 5, 7, 11 \} \) is precisely the set of elements in \( \mathbb{Z}_3 \) which are not divisible by 2 or 3. For example, 4 is not in the set because it is divisible by 2, whereas 5 is divisible by neither 2 nor 3 so it is in the set.

The special symbol \( \mathbb{Z}_N^* \) is used to represent the sub-set of \( \mathbb{Z}_N \) for which we can find a multiplicative inverse; for our example this means

\[ \mathbb{Z}_{12}^* = \{ 1, 5, 7, 11 \} . \]

Interestingly, the \( y \in \mathbb{Z}_N^* \) are integers we can safely divide by. Normally if we wrote 2/5 you might say the result is 0.4, but keep in mind that we can only work with 0 through to \( N - 1 \): there are no fractional numbers. So actually, when we write 2/5 (mod 12) this means

\[ 2/5 = 2 \cdot 1/5 = 2 \cdot 5^{-1} \equiv 2 \cdot 5 \equiv 10 \pmod{12} \]

which is “allowed” precisely because, as we saw, 1/5 \( \equiv 5^{-1} \equiv 5 \pmod{12} \), i.e., 5 has a multiplicative inverse modulo 12 which, coincidentally, is also 5.

Now look at a different modulus, namely \( N = 13 \). Something special happens, because if we follow the same reasoning as above we get

\[ \mathbb{Z}_{13} = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 \} , \]

\[ \mathbb{Z}_{13}^* = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 \} . \]

\( \mathbb{Z}_{13} \) and \( \mathbb{Z}_{13}^* \) look fairly similar: all non-zero \( y \in \mathbb{Z}_{13} \) have a multiplicative inverse and appear in \( \mathbb{Z}_{13}^* \) as a result. We can see this by again writing out a table and filling in the multiplicative inverse for each \( y \):

<table>
<thead>
<tr>
<th>( y )</th>
<th>Multiplicative Inverse (mod 13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>1</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>2</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>3</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>4</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>5</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>6</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>7</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>8</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>9</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>10</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>11</td>
<td>( \equiv 1 )</td>
</tr>
<tr>
<td>12</td>
<td>( \equiv 1 )</td>
</tr>
</tbody>
</table>
There are no question marks (except for 0) this time, precisely because 13 is a prime number, and means we can divide by every non-zero \( y \). You can think of this mirroring integer arithmetic, where dividing by zero is not allowed either. It also illustrates the fact that for \( p = 12 \), it was just a coincidence that \( y^{-1} \equiv y \pmod{12} \) for all \( y \) with an inverse: clearly this is not true for \( p = 13 \), where we find \( 5^{-1} \equiv 8 \pmod{13} \) for instance.

### 10.1.4 Some interesting facts about \( \mathbb{Z}_N^* \)

The set \( \mathbb{Z}_N^* \) is interesting because the number of elements it contains can be calculated, using just \( N \), via the so-called Euler \( \Phi \) (or “phi”) function [9]. Specifically,

\[
\Phi(N) = \prod_{i=0}^{l-1} p_i^{e_i-1} \cdot (p_i - 1)
\]

when the prime factorisation [19] of \( N \) is given by

\[
N = \prod_{i=0}^{l-1} p_i^{e_i}
\]

and where each \( p_i \) is a prime number. Basically we express \( N \) as the product of \( l \) prime numbers (or powers thereof). This is quite a nasty looking definition, but some examples should make things clearer:

- If we select \( N = 12 \), then the prime factorisation of \( N \) is

\[
N = 12 = 2 \cdot 2 \cdot 3 = 2^2 \cdot 3^1.
\]

We have expressed \( N \) as the product of \( l = 2 \) primes, namely 2 and 3. Matching this description with what we had above, this means \( p_0 = 2, e_0 = 2, p_1 = 3 \) and \( e_1 = 1 \) so

\[
N = \prod_{i=0}^{l-1} p_i^{e_i} = 2^2 \cdot p_1^{e_1} = 2^2 \cdot 3^1 = 12.
\]

If we select \( N = 13 \) then things are even simpler, i.e.,

\[
N = 13 = 13^1,
\]

due to the fact that 13 is a prime number: the prime factorisation only includes 13 itself! That is, for \( N = 13 \) we just have \( p_0 = 13 \) and \( e_0 = 1 \) so

\[
N = \prod_{i=0}^{l-1} p_i^{e_i} = 13^1 = 13.
\]

- We can now compute

\[
\Phi(12) = 2^{2^1} \cdot (2 - 1) \cdot 3^{3^0} \cdot (3 - 1) = 2 \cdot 1 \cdot 3^0 \cdot 2 = 4
\]

and

\[
\Phi(13) = 13^{1^1} \cdot (13 - 1) = 13^0 \cdot 12 = 12
\]

Both these results make sense because looking back we know that

\[
\mathbb{Z}_{12}^* = \{1, 5, 7, 11\}
\]

\[
\mathbb{Z}_{13}^* = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}
\]

which match up: \( \mathbb{Z}_{12}^* \) contains \( \Phi(12) = 4 \) elements, and \( \mathbb{Z}_{13}^* \) contains \( \Phi(13) = 12 \) elements.
Another way to look at things is that $\Phi(N)$ represents the number of integers less than $N$ which are also coprime [4] to $N$. Two integers $a$ and $b$ are coprime if their Greatest Common Divisor (GCD) is 1; some examples again make the idea clearer:

- If $a = 3$ and $b = 12$, the divisors of $a$ are 1 and 3 while the divisors of $b$ are 1, 2, 3 and 6. The greatest divisor they have in common is 3 so we can write
  
  $$\text{gcd}(a, b) = \text{gcd}(3, 12) = 3$$
  
  and say $a$ and $b$ are not coprime.

- If $a = 3$ and $b = 13$, the divisors of $a$ are 1 and 3 while the divisors of $b$ are 1 and 13. The greatest divisor they have in common is 1 so we can write
  
  $$\text{gcd}(a, b) = \text{gcd}(3, 13) = 1$$
  
  and say $a$ and $b$ are coprime.

So, if $N = 12$ how many integers $y$ exist that are less than $N$ and also coprime to $N$? If we list all the options (except 0)

\[
\begin{align*}
gcd(12, 1) & = 1 & \quad \text{gcd}(12, 6) & = 6 \\
gcd(12, 2) & = 2 & \quad \text{gcd}(12, 7) & = 1 \\
gcd(12, 3) & = 3 & \quad \text{gcd}(12, 8) & = 4 \\
gcd(12, 4) & = 4 & \quad \text{gcd}(12, 9) & = 3 \\
gcd(12, 5) & = 1 & \quad \text{gcd}(12, 10) & = 2 \\
gcd(12, 11) & = 1 & \quad \text{gcd}(12, 11) & = 1
\end{align*}
\]

the answer is four, i.e., 1, 5, 7 and 11, so $\Phi(12) = 4$. Notice that the values of $y$ which produce $\text{gcd}(N, y) = 1$ are precisely those which make up the set $Z_N^*$. We need not go through the same task for $N = 13$: all positive integers less than 13 are coprime to 13 because it is prime, so $\Phi(13) = 12$.

Another interesting fact is that if you take any $x \in Z_N^*$ and raise it to the power $\Phi(N)$ modulo $N$, the result is 1. In short, for every $x \in Z_N^*$ we have

$$x^{\Phi(N)} \equiv 1 \pmod{N}.$$ 

For example, consider $x = 5$ which is an element of both $Z_{12}^*$ and $Z_{13}^*$. In these cases we have

\[
\begin{align*}
x^{\Phi(12)} & = 5^4 = 625 \equiv 1 \pmod{12}, \\
x^{\Phi(13)} & = 5^{12} = 244140625 \equiv 1 \pmod{13}
\end{align*}
\]

Demonstrate this fact is also true for other elements of $Z_{12}^*$ and $Z_{13}^*$ by working out similar results for them.

10.2 Modular arithmetic: the practice

In Chapter 3, one of the goals was to convince ourselves that we could write down an algorithm for multiplication: the premise was that if we could do this, we were closer to writing a program to do the same thing on a computer.

You can easily imagine having a similar motivation with respect to modular arithmetic and cryptographic protocols based on it. The goal is to show that you can write algorithms to compute all the modular arithmetic operations we need using only integer arithmetic as a starting point. Put another way, all the theory that surrounds modular arithmetic is important, but if all you want to do is make some cryptographic protocol work on a computer then there is no magic involved: we can describe such computation entirely in terms of what you already know.
Figure 10.2: Some algorithms for modular arithmetic.
10.2.1 Addition and subtraction

Imagine we start off with $0 \leq x, y < N$, i.e., both $x$ and $y$ are in the set $\mathbb{Z}_N = \{0, 1, \ldots, N - 1\}$. If we add $x$ and $y$ together to get $t = x + y$, we end up with a result that satisfies

$$ 0 \leq t \leq 2\cdot(N - 1). $$

The smallest result we can end up with is $t = 0$ which happens when $x$ and $y$ are as small as they can be, i.e., $x = y = 0$; the largest result we can end up with is $t = 2\cdot(N - 1)$ which happens when $x$ and $y$ are as large as then can be, i.e., $x = y = N - 1$. We would like to apply a modular reduction to $t$, i.e., compute $t' \equiv t$ (mod $N$). This is simple in that to get

$$ 0 \leq t' \leq N - 1, $$

we just subtract $N$ from $t$ if $t \geq N$. This is captured in Algorithm 10.2a, and demonstrated in action by some examples:

- Imagine $x = 8$, $y = 7$ and $N = 12$: we compute $t = 8 + 7 = 15$, and then $t' = 15 - 12 = 3$ since $t \geq 12$. Checking the modular reduction, we can see that
  
  $$ 3 \equiv 15 \pmod{12}. $$

- Imagine $x = 1$, $y = 7$ and $N = 12$: we compute $t = 1 + 7 = 8$, and then $t' = 8$ since $t < 12$, i.e., no modular reduction is needed in this case because we know
  
  $$ 8 \equiv 8 \pmod{12}. $$

So to cut a long story short, computing a modular addition relies only on integer addition and subtraction.

Things work more or less the same way for modular subtraction: if we subtract $y$ from $x$ to get $t = x - y$, we end up with a result that satisfies

$$ -(N - 1) \leq t \leq (N - 1). $$

The smallest result we can end up with is $t = -(N - 1)$ which happens when $x$ is as small as it can be and $y$ is as large as it can be, i.e., $x = 0$ and $y = N - 1$; the largest result we can end up with is $t = N - 1$ which happens when $x$ is as large as it can be and $y$ is as small as it can be, i.e., $x = N - 1$ and $y = 0$. Again, we would like to apply a modular reduction to get

$$ 0 \leq t' \leq N - 1 $$

which is again simple: we just add $N$ to $t$ if $t < 0$. This is captured in Algorithm 10.2b, and demonstrated in action by some examples:

- Imagine $x = 8$, $y = 7$ and $N = 12$: we compute $t = 8 - 7 = 1$, and then $t' = 1$ since $t \geq 0$. No modular reduction is required in this case because we know
  
  $$ 1 \equiv 1 \pmod{12}. $$

- Imagine $x = 1$, $y = 7$ and $N = 12$: we compute $t = 1 - 7 = -6$, and then $t' = -6 + 12 = 6$ since $t < 0$. Checking the modular reduction, we can see that
  
  $$ 6 \equiv -6 \pmod{12}. $$

Once more it is clear that to compute a modular subtraction, integer addition and subtraction are all we need.

10.2.2 Multiplication

Imagine we start off with $0 \leq x, y < N$, i.e., both $x$ and $y$ are in the set $\mathbb{Z}_N = \{0, 1, \ldots, N - 1\}$. Following the cases for addition and subtraction above, if we now multiply $x$ and $y$ together to get $t = x \cdot y$, we end up with a result that satisfies

$$ 0 \leq t \leq (N - 1)^2. $$

The smallest result we can end up with is $t = 0$ which happens when $x$ and $y$ are as small as they can be, i.e., $x = y = 0$; the largest result we can end up with is $t = (N - 1)^2$ which happens when $x$ and $y$ are as large as then can be, i.e., $x = y = N - 1$. 

186
We would like to apply a modular reduction to \( t \), but this time things are more tricky. For addition and subtraction, the \( t \) we compute is only ever wrong by at most \( N \), i.e., we only need to subtract or add \( N \) once to get the right result; here we might need to subtract \( N \) many times. Consider an example: imagine we set \( N = 12 \) and select \( x = 10 \) and \( y = 11 \). If we compute \( t = x + y = 10 + 11 = 21 \), then we only need to subtract \( N \) once from \( t \) to get the result \( t' = 9 \equiv 21 \pmod{12} \). If however we compute \( t = x \cdot y = 10 \cdot 11 = 110 \), then we need to subtract \( N \) nine times to get \( t' = 2 \equiv 110 \pmod{12} \). So basically, instead of

\[
1 \text{ if } t \geq N \text{ then} \\
2 \quad t' \leftarrow t - N \\
3 \text{ end} \\
4 \text{ else} \\
5 \quad t' \leftarrow t \\
6 \text{ end}
\]

we need something that \textit{repeatedly} subtracts \( N \) until we get the right result, i.e., something more like

\[
1 \text{ while } t' \geq N \text{ do} \\
2 \quad t' \leftarrow t' - N \\
3 \text{ end}
\]

Potentially this will be \textit{very} inefficient however; the reason is more or less the same as when we talked about repeated addition in Chapter 3. Instead, we can calculate how many multiples of \( N \) we \textit{would} need to subtract using this approach, and then simply do them all in one go. The idea is basically to compute

\[
t' = t - (N \cdot \left\lfloor \frac{t}{N} \right\rfloor).
\]

The term \( \left\lfloor \frac{t}{N} \right\rfloor \) is the number of times \( N \) goes into \( t \); this is basically the number of times we would go around the loop above. As a result \( N \cdot \left\lfloor \frac{t}{N} \right\rfloor \) tells us what to subtract from \( t \), which we can now do in one step. This is captured in Algorithm 10.2c, and demonstrated in action by some examples:

- Imagine \( x = 8, y = 7 \) and \( N = 12 \): we compute \( t = 8 \cdot 7 = 56 \), and then
  \[
t' = 56 - (12 \cdot 4) = 8
\]
  since \( \left\lfloor \frac{56}{12} \right\rfloor = 4 \). Checking the modular reduction, we can see that
  \[
  8 \equiv 56 \pmod{12}.
  \]
- Imagine \( x = 1, y = 7 \) and \( N = 12 \): we compute \( t = 1 \cdot 7 = 7 \), and then
  \[
t' = 7 - (12 \cdot 0) = 7
\]
  since \( \left\lfloor \frac{7}{12} \right\rfloor = 0 \), i.e., no modular reduction is needed in this case because we know that
  \[
  7 \equiv 7 \pmod{12}.
  \]

Things are more involved here than in the case of modular addition and subtraction, but to compute a modular multiplication \textit{we still} only rely on integer multiplication, division and subtraction.

### 10.2.3 Exponentiation

The case of modular exponentiation is (believe it or not) one of the easiest to resolve. What we end up with is Algorithm 10.2d, which should look at least a bit familiar: basically we have just taken the old algorithm based on Horner’s Rule from Chapter 3 and replaced the normal, integer multiplications with modular multiplications instead. The idea is that if each multiplication applies a modular reduction, then of course the exponentiation algorithm as a whole will be sane.

Consider an example where \( x = 3 \) and \( y = 6_{(10)} = 110_{(2)} \). We know that \( 3^6 = 729 \) and that \( 729 \equiv 9 \pmod{12} \), so the question is whether the new algorithm will give us the same result. The steps it performs demonstrate that it does:
Step #1 Assign \( t \leftarrow 1 \).

Step #2 Assign \( t \leftarrow t^2 \pmod{N} \), i.e., \( t \leftarrow 1^2 \pmod{12} = 1 \).

Step #3 Since \( y_2 = 1 \), assign \( t \leftarrow t \cdot x \pmod{N} \), i.e., \( t \leftarrow 1 \cdot 3 \pmod{12} = 3 \).

Step #4 Assign \( t \leftarrow t^2 \pmod{N} \), i.e., \( t \leftarrow 3^2 \pmod{12} = 9 \).

Step #5 Since \( y_1 = 1 \), assign \( t \leftarrow t \cdot x \pmod{N} \), i.e., \( t \leftarrow 9 \cdot 3 \pmod{12} = 3 \).

Step #6 Assign \( t \leftarrow t^2 \pmod{N} \), i.e., \( t \leftarrow 3^2 \pmod{12} = 9 \).

Step #7 Since \( y_0 = 0 \), skip the assignment \( t \leftarrow t \cdot x \pmod{N} \).

Step #8 Return \( t = 9 \).

All we are relying on to do this is modular multiplication, and we already know that modular multiplication can be described using integer operations only.

### 10.2.4 Division (via inversion)

We have already seen that in modular arithmetic, dividing by some \( y \) only makes sense if \( y \) has a multiplicative inverse. This is only true if \( y \in \mathbb{Z}^*_N \), which, in turn, is only true if \( N \) and \( y \) are coprime, i.e., \( \gcd(N, y) = 1 \). When this all works out, we can write

\[
x/y = x \cdot y^{-1} \pmod{N}
\]

which implies that modular division of \( x \) by \( y \) boils down to finding the multiplicative inverse of \( y \) modulo \( N \), and multiplying this by \( x \) modulo \( N \).

#### 10.2.4.1 Problem #1: computing \( \gcd(N, y) \)

The first challenge is to test whether \( y \) has a multiplicative inverse or not: for large values of \( N \) we cannot simply write out the whole set \( \mathbb{Z}_N^* \), we need to compute and test whether \( \gcd(N, y) = 1 \) or not.

One way would be to write out the prime factorisation of \( N \) and \( y \) and simply pick out the greatest divisor they have in common; if this turns out to be 1, we know \( N \) and \( y \) are coprime. We performed this exact task when we originally discussed coprimality, but imagine we now select the larger examples

\[
N = 1426668559730 = 2^1 \cdot 5^1 \cdot 157^1 \cdot 271^1 \cdot 743^1 \cdot 4513^1, \\
y = 810653094756 = 2^2 \cdot 3^2 \cdot 61^1 \cdot 157^1 \cdot 521^1 \cdot 4513^1.
\]

Using their prime factorisations (the right-hand part of each line), we can deduce that

\[
\gcd(N, y) = \gcd(1426668559730, 810653094756) = 2 \cdot 157 \cdot 4513 = 1417082
\]

because both \( N \) and \( y \) have 2, 157 and 4513 as common divisors. However factoring is relatively hard work, particularly if \( N \) and \( y \) are large; if this was the best way to compute \( \gcd(N, y) \), we would need a very fast computer or plenty of time on our hands! Luckily the Euclidean algorithm [8], an ancient algorithm due to Greek mathematician Euclid, can compute the same result rather more quickly.

Imagine we want to compute

\[
\gcd(a, b)
\]

assuming \( a \geq b \) (swapping \( a \) and \( b \) if not). Understanding how the Euclidean algorithm works hinges on writing

\[
a = b \cdot q + r
\]

where \( q \) is called the quotient and \( r \) is called the remainder. This is the same as saying that if we divide \( a \) by \( b \), we get a quotient (how many times \( b \) divides into \( a \)) and a remainder (what is left after such a division). An important fact is that \( 0 \leq r < b \leq a \) because if \( r \) were larger, \( b \) would divide into \( a \) one more time than \( q \) says it should. Taking this as our starting point, we can formulate two rules:

1. If you divide \( a \) by \( b \) and the remainder is 0, then \( a \) is a multiple of \( b \) (because \( b \) divides \( a \) exactly): this implies that \( b \) is a common divisor of \( a \) and \( b \).
2. If you divide \( a \) by \( b \) and the remainder is not 0, then we can say that
\[
gcd(a, b) = gcd(b, r).
\]

Why is this? Imagine we write out the prime factorisation of \( a \) as follows:
\[
a = p_0^{e_0} \cdot p_1^{e_1} \cdots p_{l-1}^{e_{l-1}}.
\]
One of those terms will be \( d = gcd(a, b) \). Therefore you could imagine \( a \) written instead as
\[
a = d \cdot a'.
\]
where \( a' \) represents all the \"other\" terms. We can do the same thing with \( b \), because we know \( d \) must divide it as well, to get
\[
b = d \cdot b'.
\]
As a result
\[
a - b = (d \cdot a') - (d \cdot b') = d \cdot (a' - b')
\]
and hence \( d \) clearly divides \( a - b \). Now going back to what we know from above, i.e.,
\[
r = a - b \cdot q,
\]
it follows that
\[
a - b \cdot q = (d \cdot a') - (d \cdot b' \cdot q) = d \cdot (a' - b' \cdot q).
\]
So basically
\[
r = d \cdot (a' - b' \cdot q)
\]
which means that \( d \) also divides \( r \). To cut a long story short, \( gcd(b, r) \) will therefore give us the same result as \( gcd(a, b) \); because \( r \) is smaller than \( a \) and \( b \), computing \( gcd(b, r) \) retains the restrictions we had to start with that said \( a \geq b \).

Based on these two rules, the Euclidean algorithm computes the following sequence:

\[
\begin{align*}
  r_0 &= a \\
  r_1 &= b \\
  r_2 &= r_0 - r_1 \cdot q_2 \\
  r_3 &= r_2 - r_3 \cdot q_3 \\
  &\vdots
\end{align*}
\]

Basically the sequence is describing repeated application of the second rule above: the \( i \)-th remainder \( r_i \), which we call \( r_i \), is computed via
\[
r_i = r_{i-2} - r_{i-1} \cdot q_i
\]
where the \( i \)-th quotient is \( q_i = r_{i-2}/r_{i-1} \). The sequence will finish in some \( m \)-th step when \( r_{m-1} \) divides \( r_{m-2} \) exactly (meaning we compute \( r_m = 0 \)), at which point the answer we want is \( r_{m-1} \); this corresponds to the first rule above. There are plenty of ways to write this down as an algorithm; one simple approach is shown in Algorithm 10.2e.

If we invoke \textsc{Euclidean}(21, 12), i.e., try to compute \( gcd(a, b) \) with \( a = 21 \) and \( b = 12 \) as in the example above, the algorithm computes the sequence

\[
\begin{align*}
  r_0 &= 21 \\
  r_1 &= 12 \\
  r_2 &= 21 - 12 \cdot 1 = 9 \\
  r_3 &= 12 - 9 \cdot 1 = 3 \\
  r_4 &= 9 - 3 \cdot 3 = 0
\end{align*}
\]
meaning that \( m = 4 \) and hence \( r_{m-1} = r_3 = 3 \). The algorithm itself does this via the following steps:

\textbf{Step #1} Assign \( r_0 \leftarrow 21 \).

\textbf{Step #2} Assign \( r_1 \leftarrow 12 \).
Step #3 Assign $i \leftarrow 2$.

Step #4 Since $r_0 \mod r_1 = 21 \mod 12 \neq 0$, skip the return.

Step #5 Assign $r_2 \leftarrow r_0 \mod r_1 \cdot \lfloor \frac{r_0}{r_1} \rfloor = 21 - 12 \cdot \lfloor \frac{21}{12} \rfloor = 21 - 12 \cdot 1 = 9$.

Step #6 Assign $i \leftarrow i + 1 = 3$.

Step #7 Since $r_1 \mod r_2 = 12 \mod 9 \neq 0$, skip the return.

Step #8 Assign $r_3 \leftarrow r_1 \mod r_2 \cdot \lfloor \frac{r_1}{r_2} \rfloor = 12 - 9 \cdot \lfloor \frac{12}{9} \rfloor = 12 - 9 \cdot 1 = 3$.

Step #9 Assign $i \leftarrow i + 1 = 4$.

Step #10 Since $r_2 \mod r_3 = 9 \mod 3 = 0$, return $r_3 = 3$.

What about the example we started off with originally? The $a$ and $b$ are larger so we have a longer sequence, but by invoking Euclidean($1426668559730, 810653094756$)

we get

$$
egin{align*}
    r_0 &= 1426668559730 \\
    r_1 &= 810653094756 \\
    r_2 &= 1426668559730 - 810653094756 \cdot 1 = 616015464974 \\
    r_3 &= 810653094756 - 616015464974 \cdot 1 = 194637629782 \\
    r_4 &= 616015464974 - 194637629782 \cdot 3 = 32102575628 \\
    r_5 &= 194637629782 - 32102575628 \cdot 6 = 2022176014 \\
    r_6 &= 32102575628 - 2022176014 \cdot 15 = 1769935418 \\
    r_7 &= 2022176014 - 1769935418 \cdot 1 = 252240596 \\
    r_8 &= 1769935418 - 252240596 \cdot 7 = 4251246 \\
    r_9 &= 252240596 - 4251246 \cdot 59 = 1417082 \\
    r_{10} &= 4251246 - 1417082 \cdot 3 = 0
\end{align*}
$$

via similar steps, meaning that $m = 10$ and hence $r_{m-1} = r_9 = 1417082$ which matches the original result.

10.2.4.2 Problem #2: computing $y^{-1}$ (mod $N$)

Now we can tell whether or not $y$ has a multiplicative inverse modulo $N$, the next challenge is to compute the inverse itself, i.e., $y^{-1}$ (mod $N$). One option is to use the fact that we know

$$
    y^{\Phi(N)} \equiv 1 \pmod{N}.
$$

Based on this, it also makes sense that

$$
\begin{align*}
    y^{\Phi(N)-2} &\equiv y^{-2} \equiv 1/y^2 \pmod{N} \\
    y^{\Phi(N)-1} &\equiv y^{-1} \equiv 1/y \pmod{N} \\
    y^{\Phi(N)+0} &\equiv y^0 \equiv 1 \pmod{N} \\
    y^{\Phi(N)+1} &\equiv y^1 \equiv y \pmod{N} \\
    y^{\Phi(N)+2} &\equiv y^2 \equiv y^2 \pmod{N}
\end{align*}
$$

Or, put another way, we can easily compute $y^{-1}$ (mod $N$) if we know $\Phi(N)$ because we already know how to do modular exponentiation. On the other hand, we already said that computing $\Phi(N)$ is relatively hard because we need to factor $N$.

An alternative option is to use use a variant of the Euclidean algorithm, called the Extended Euclidean Algorithm (EEA) [11]; sometimes this is called “XGCD” as in “eXtended”. Recall that the Euclidean algorithm used

$$
    r_i = r_{i-2} - r_{i-1} \cdot q_i.
$$
The idea of the extended Euclidean algorithm is to unwind the steps and write each \( r_i \) in terms of \( a \) and \( b \). This produces

\[
\begin{align*}
  r_0 &= a \\
  r_1 &= b \\
  r_2 &= r_0 - r_1 \cdot q_2 \\
        &= a - b \cdot q_2 \\
  r_3 &= r_1 - r_2 \cdot q_3 \\
        &= b - (a - b \cdot q_2) \cdot q_3 \\
        &= -a \cdot q_3 + b \cdot (1 + q_2 \cdot q_1) \\
\vdots
\end{align*}
\]

This quickly starts to look nasty, so we simplify it by saying that in the \( i \)-th step we have

\[
r_i = a \cdot s_i + b \cdot t_i
\]

where \( s_i \) and \( t_i \) collect together all the nasty looking parts in one place. The sequence again finishes in the \( m \)-th step. But if we keep track of \( s_i \) and \( t_i \), as well as getting \( r_{m-1} \) we also get \( s_{m-1} \) and \( t_{m-1} \). Algorithm 10.2f shows the updated algorithm that keeps track of the extra parts.

If we invoke the new algorithm via \texttt{Extended-Euclidean}(21, 12), we get

\[
\begin{align*}
  r_0 &= 21 \\
  s_0 &= 1 \\
  t_0 &= 0 \\
  r_1 &= 12 \\
  s_1 &= 0 \\
  t_1 &= 1 \\
  r_2 &= 21 - 12 \cdot 1 = 9 \\
  s_2 &= 1 - 0 \cdot 1 = 1 \\
  t_2 &= 0 - 1 \cdot 1 = -1 \\
  r_3 &= 12 - 9 \cdot 1 = 3 \\
  s_3 &= 0 - 1 \cdot 1 = -1 \\
  t_3 &= 1 - (-1) \cdot 1 = 2 \\
  r_4 &= 9 - 3 \cdot 3 = 0 \\
  s_4 &= 1 - (-1) \cdot 3 = 4 \\
  t_4 &= -1 - 2 \cdot 3 = -7
\end{align*}
\]

meaning that \( m = 4 \). We can verify that this is sane because

\[
\begin{align*}
  r_{m-1} &= 3 = a \cdot s_{m-1} + b \cdot t_{m-1} \\
         &= 21 \cdot (-1) + 12 \cdot 2 \\
         &= -21 + 24 \\
         &= 3
\end{align*}
\]

as expected. The point is that the extra parts the algorithm computes turn out to be very useful because given

\[
r_{m-1} = \gcd(a, b) = 1
\]

if \( a \) and \( b \) are coprime, we therefore know

\[
1 = a \cdot s_{m-1} + b \cdot t_{m-1}.
\]

As a result, we can compute the multiplicative inverse of \( x \) modulo \( N \) by setting \( a = x \) and \( b = N \). This means we get

\[
1 = x \cdot s_{m-1} + N \cdot t_{m-1}
\]

and, if we look at this modulo \( N \),

\[
1 \equiv x \cdot s_{m-1} \pmod{N}
\]

since \( N \cdot t_{m-1} \equiv 0 \pmod{N} \) (in fact any multiple of \( N \) is equivalent to 0 when reduced modulo \( N \)) or rather

\[
x^{-1} \equiv s_{m-1} \pmod{N}
\]

meaning that \( s_{m-1} \) is the multiplicative inverse we want.
10.3 From modular arithmetic to cryptographic protocols

As we saw in the introduction, a cryptographic protocol is a description of how the parties involved accomplish some task (e.g., sending messages between each other securely). The two tasks we focus on relate to the key distribution problem: can Alice and Bob communicate in a secure way without having to agree on a shared key before they start? We can describe two potential solutions using just the modular arithmetic studied so far:

1. key agreement protocols [15], and
2. public-key encryption schemes [21].

One of the motivations for solving this problem is that we, as Alice say, often want to communicate with a non-human Bob. Imagine for example if Bob were a web-site we want to buy something from; it is not feasible for us to agree a shared key with every possible web-site, so solutions like those above are critically important.

10.3.1 Diffie-Hellman key exchange

Suppose Alice wants to send a message to Bob, but they do not have a shared key; they could use the protocol we saw in the introduction, but first they need a secure way to agree on a shared key. The Diffie-Hellman key exchange [5] protocol, invented in 1976 by cryptographers Whitfield Diffie and Martin Hellman, offers one solution to this problem. A historically interesting fact is that the same protocol was invented independently, and around the same time, by Malcolm Williamson, a cryptographer at GCHQ [12] in the UK; this only became apparent when the intelligence agency declassified the work, making it public in 1997.

Before they start, Alice and Bob agree on two public values: a large prime number \( p \), and some \( g \in \mathbb{Z}_p \).

The values are made available to everyone, e.g., by publishing them on a web-site. You can think of them as fixed settings which allow the protocol to work, a bit like the settings that tell a web-browser how to work.

As a result, Alice and Bob can agree on a key as follows:

<table>
<thead>
<tr>
<th>Alice</th>
<th>Eve</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \leftarrow Z_p )</td>
<td>( y \leftarrow Z_p )</td>
<td></td>
</tr>
<tr>
<td>( a \leftarrow g^x \mod p )</td>
<td>( b \leftarrow g^y \mod p )</td>
<td></td>
</tr>
<tr>
<td>( k \leftarrow b^x \mod p )</td>
<td>( k \leftarrow a^y \mod p )</td>
<td></td>
</tr>
</tbody>
</table>

The protocol Alice and Bob engage in has five simple steps:

1. Alice selects \( x \), a random element in \( Z_p \); at the same time Bob selects \( y \), a random element in \( Z_p \).
2. Alice computes \( a = g^x \mod p \); at the same time Bob computes \( b = g^y \mod p \).
3. Alice communicates \( a \) to Bob.
4. Bob communicates \( b \) to Alice.
5. Alice computes \( k = b^x \mod p \); at the same time Bob computes \( k = a^y \mod p \).

Notice that at the end of the protocol Alice and Bob end up with the same shared key \( k \); this is because for Alice

\[
\begin{align*}
    k &= b^x \mod p \\
    &= (g^y)^x \mod p \\
    &= g^{yx} \mod p
\end{align*}
\]

while for Bob we find

\[
\begin{align*}
    k &= a^y \mod p \\
    &= (g^x)^y \mod p \\
    &= g^{xy} \mod p
\end{align*}
\]

A concrete example demonstrates that crucially, this happens even though Alice does not know \( y \), Bob does not know \( x \) and Eve knows neither \( y \) nor \( x \). Imagine we make the settings \( p = 13 \) and \( g = 7 \) available to everyone. The protocol can then proceed as follows:

\[
\begin{align*}
    k_1 &= 7^3 \mod 13 = 1 \\
    k_2 &= (7^2)^3 \mod 13 = 1
\end{align*}
\]
1. Alice selects \( x = 9 \); at the same time Bob selects \( y = 3 \).

2. Alice computes \( a = g^x \pmod{p} = 7^9 \pmod{13} = 8 \); at the same time Bob computes \( b = g^y \pmod{p} = 7^3 \pmod{13} = 5 \).

3. Alice communicates \( a = 8 \) to Bob.

4. Bob communicates \( b = 5 \) to Alice.

5. Alice computes \( k = b^x \pmod{p} = 5^9 \pmod{13} = 5 \); at the same time Bob computes \( k = a^y \pmod{p} = 8^3 \pmod{13} = 5 \).

What if all you have is a block cipher? Clearly Alice and Bob cannot rely on the fact they know \( k \) before they start, but what if they each share a different key with some trusted third-party Trent (call them \( k_{\text{Alice}} \) and \( k_{\text{Bob}} \) say): can you write down a protocol whereby they end up with a shared \( k \) using Trent to help out?

10.3.2 RSA encryption

Now suppose Alice wants to send a message to Bob, without the need to agree on a shared key at all. The RSA public-key encryption [22], invented in 1978 by Ron Rivest, Adi Shamir and Leonard Adleman (who lend their initials to the name), represents an efficient and widely-used solution. In short, it allows us to avoid the need for shared keys entirely, and hence side-step the key distribution problem. Again, cryptographers at GCHQ [12] produced similar results behind closed doors: Clifford Cocks had developed a similar scheme independently in 1973, but this was not declassified until 1997.

Before Alice can encrypt a message, Bob has to do some work. He picks two large prime numbers \( p \) and \( q \) and uses them to first compute \( N = p \cdot q \), then

\[
\Phi(N) = (p-1) \cdot (q-1).
\]

Notice that only Bob can do this because only he knows \( p \) and \( q \), the prime factors of \( N \). He then selects a small number \( e \) which is coprime to \( \Phi(N) \), and solves

\[
e \cdot d \equiv 1 \pmod{\Phi(N)},
\]

that is, he computes \( d \), the multiplicative inverse of \( e \) modulo \( \Phi(N) \). Remember, this means there exists a \( \kappa \) (known only by Bob in this case) such that

\[
e \cdot d = 1 + \kappa \cdot \Phi(N).
\]

The pair \((N, e)\) is the public-key for Bob, while \((N, d)\) is the private-key. The idea is that Bob can make his public-key available to everyone (e.g., by publishing it on his web-site), and that by downloading it doing so anyone can encrypt messages for him. Only Bob knows the corresponding private-key, so only he will be able to decrypt said messages.

As a result, Alice can send \( M \in \mathbb{Z}_N^* \) to Bob as follows:

\[
\begin{align*}
\text{Alice} & \quad \text{Eve} & \quad \text{Bob} \\
C & \leftarrow M^e \pmod{N} & C & \rightarrow & M & \leftarrow C^d \pmod{N}
\end{align*}
\]

As in the introduction, the protocol that Alice and Bob engage in has three simple steps:

1. Alice encrypts a plaintext \( M \) using the public-key \((N, e)\) to produce the ciphertext message

\[
C = M^e \pmod{N}.
\]

2. Alice communicates \( C \) to Bob.
3. Bob decrypts the ciphertext $C$ using the private-key $(N, d)$ to recover the plaintext message

$$M = C^d \quad (\text{mod } N).$$

But why does this work? Well, it follows from a fact we saw earlier: for any $x \in \mathbb{Z}_N^*$ we have that

$$x^{\Phi(N)} \equiv 1 \quad (\text{mod } N).$$

This means

$$C^d \equiv (M^e)^d \quad (\text{mod } N)$$
$$\equiv M^{ed} \quad (\text{mod } N)$$
$$\equiv M \cdot M^{e \cdot \Phi(N)} \quad (\text{mod } N)$$
$$\equiv M \cdot (M^{e \cdot \Phi(N)})^x \quad (\text{mod } N)$$
$$\equiv M \cdot 1^x \quad (\text{mod } N)$$
$$\equiv M \cdot 1 \quad (\text{mod } N)$$
$$\equiv M \quad (\text{mod } N)$$

and so we get the message $M$ as a result! The crucial difference between this and what we saw in the introduction is that Alice and Bob no longer use a shared key: everyone knows the public-key owned by Bob that was used for encryption, but only Bob knows the corresponding private-key used for decryption. We can demonstrate this using another concrete example. Imagine Bob selects

$$p = 5$$
$$q = 11$$

meaning $N = 55$ and $\Phi(N) = (p - 1) \cdot (q - 1) = 40$. He also selects $e = 7$, noting that gcd($e, \Phi(N)$) = 1, and computes $d = 23$, again noting that $e \cdot d = 161 \equiv 1 \pmod{\Phi(N)}$. The protocol can then proceed as follows:

1. Alice encrypts a plaintext $M = 46$ using the public-key $(N, e) = (55, 7)$ to produce the ciphertext message

$$C = M^e \quad (\text{mod } N) = 46^7 \quad (\text{mod } 55) = 51.$$

2. Alice communicates $C$ to Bob.

3. Bob decrypts the ciphertext $C = 51$ using the private-key $(N, d) = (55, 23)$ to recover the plaintext message

$$M = C^d \quad (\text{mod } N) = 51^{23} \quad (\text{mod } 55) = 46.$$

The ElGamal [7] public-key encryption scheme is an alternative to RSA; it also makes use of modular arithmetic. Three public values, namely $g, p$ and $q$, are known to everyone:

- Alice chooses a random private-key $x$ from the set $\{1, 2, \ldots, q - 1\}$ and computes

$$h = g^x \quad (\text{mod } p)$$

which is her public-key made available to everyone.

- Bob wants to send a message $m$ to Alice: he first chooses a random $k$ from the set $\{1, 2, \ldots, q - 1\}$ and computes

$$c_1 = g^k \quad (\text{mod } p)$$
$$c_2 = h^k \cdot m \quad (\text{mod } p)$$

The pair $(c_1, c_2)$ is the ciphertext sent to Alice.

How can Alice decrypt $(c_1, c_2)$ to recover the plaintext $m$? Work out the steps required, and show that the overall scheme works using an example (e.g., using the public values $g = 105, p = 107$ and $q = 53$).
10.3.3 Functional versus secure

In describing the protocols above, we showed they do what they should: they satisfy the functional requirements. But we forgot about (or more like ignored) Eve: how can we be sure the protocols are secure? We need to be able to reason about what Eve can do as well: it is not good enough to say in the case of RSA “Eve does not see M communicated between Alice and Bob so the protocol is secure”. A standard approach is to model all the things that Eve must do in order that a protocol is insecure, and relate them to a “hard” Mathematical problem: if we can prove things about the problem, those proofs mean something in terms of Eve. For example, if we can reason about how hard the problem is (e.g., how many steps an algorithm will take to solve it), we can reason about how easily Eve will be able to attack the protocol.

Both of the protocols we introduced rely on a common building block, namely the computation of

\[ z = x^y \pmod N \]

for various x, y and N. This allows us to consider (at least) two problems:

- If N is prime, say \( N = p \), then given N, x and z computing y is believed to be hard (for large values of N). This is called the Discrete Logarithm Problem (DLP) [6]; we write down an instance of this problem as DLP\((z, x, N)\).

- If N is a product of two primes, say \( N = p \cdot q \), then given N, y and z computing x is believed to be hard without also knowing p and q. This is called the RSA problem [23]; we write down an instance of this problem as RSA\((z, y, N)\).

In both cases, a problem instance is simply a challenge to Eve. When you see DLP\((z, x, N)\) this should be read as “find y such that \( z \equiv x^y \pmod N \)”; when you see RSA\((z, y, N)\) this should be read as “find an x such that \( z \equiv x^y \pmod N \)’’.

In general, a function \( f \) which is easy to compute but which is hard to invert or “reverse” (meaning \( f^{-1} \) is hard to compute) is called a one-way function [18]. Our examples fall exactly into this category: it is easy to compute \( z = x^y \pmod N \), but (depending on the N we choose) hard to solve either DLP\((z, x, N)\) or RSA\((z, y, N)\). The question is, what do we mean by “hard”? Usually we try to answer this by using the same ideas as in Chapter 3. By saying that computing \( f \) should be easy, we mean that for a problem of size \( n \) (for example the number of bits in \( N \)), we might have an algorithm which does not take too many steps to compute \( f \). So maybe \( f \) is \( O(n) \) or \( O(n^2) \) or even \( O(n^3) \). In contrast, computing \( f^{-1} \) should be much harder; for example we might select \( f \) so that the best known algorithm to compute \( f^{-1} \) takes \( O(2^n) \) steps.

So how does this relate to what Eve might do? Consider a model where Eve observes the communication between Alice and Bob (in addition to getting any publicly available values). How might she attack the protocol?

1. In the Diffie-Hellman example, Eve (like everyone else) gets access to the settings \( p \) and \( g \) and sees \( a \) and \( b \) communicated from Alice to Bob and vice versa.

Assuming Eve wants to recover the shared key \( k \), one approach would be to solve the problem instance DLP\((a, g, p)\); this yields \( x \) and means Eve can compute

\[ b^x = k \]

just like Bob did. Alternatively, Eve could solve the problem instance DLP\((b, g, p)\) to get \( y \) and compute

\[ a^y = k \]

just like Alice did. But both problem instances are as hard as each other, so either way we might argue that if solving them is hard then Diffie-Hellman key exchange is secure.

2. In the RSA example, Eve (like everyone else) gets access to the public-key for Bob, i.e., \( (N, e) \) and sees \( C \) communicated by Alice to Bob.

Assuming Eve wants to recover the message \( M \), one can imagine two approaches:

(a) Factor \( N \) to recover \( p \) and \( q \), then compute \( \Phi(N) \). By doing this, \( d \) can be computed and Eve can decrypt \( C \) just like Bob did.

(b) Solve the problem instance RSA\((C, e, N)\) which gives the result \( M \).

As a result we might argue that RSA is secure if factoring is hard, and solving instances of the RSA problem is hard.
This seems great: if Eve wants to do anything we might regard as “bad”, she has to work out how to solve what we are assuming are hard mathematical problems. If the problems really are hard, then we can say the protocols are secure.

But what if Eve does something other than what we expect? What if she acts outside our current model, which assumes she can only observe the communication between Alice and Bob? For example, imagine Eve has cut the network cable that links Alice and Bob and placed her own computer in between them; this is known as a man-in-the-middle attack [16]. Now Eve becomes the active adversary Mallory, and can do all sorts of other things. The question is, are the two protocols still secure?

1. For the Diffie-Hellman example, imagine the protocol changes because of how Mallory behaves; Alice and Bob behave in exactly the same way. The result is as follows:

<table>
<thead>
<tr>
<th>Alice</th>
<th>Mallory</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leftarrow Z_p$</td>
<td>$x' \leftarrow Z_p$</td>
<td>$y \leftarrow Z_p$</td>
</tr>
<tr>
<td>$a \leftarrow g^x \pmod{p}$</td>
<td>$a' = g^{x'} \pmod{p}$</td>
<td>$b \leftarrow g^y \pmod{p}$</td>
</tr>
<tr>
<td>$k_1 = a^{y'} \pmod{p}$</td>
<td>$k_2 = b^{x'} \pmod{p}$</td>
<td>$k_2 \leftarrow a'^y \pmod{p}$</td>
</tr>
</tbody>
</table>

Since

$$k_1 = b^{x'} \pmod{p}$$
$$= (g^y)^{y'} \pmod{p}$$
$$= a^{y'} \pmod{p}$$

and

$$k_2 = a'^y \pmod{p}$$
$$= (g^y)^{x'} \pmod{p}$$
$$= b^{x'} \pmod{p}$$

Mallory agrees one key with Alice and one with Bob: Alice thinks she is communicating with Bob and vice versa, but actually they are communicating through Mallory. When Alice sends a message to Bob encrypted using $k_1$, Mallory can simply decrypt the message and read it, then re-encrypt it using $k_2$ and send it on: neither Alice nor Bob are any the wiser because from their point of view, nothing has gone wrong.

2. For the RSA example, imagine the protocol changes to the following:

<table>
<thead>
<tr>
<th>Alice</th>
<th>Mallory</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C \leftarrow M^e \pmod{N}$</td>
<td>$C \leftarrow 2^e \cdot C \pmod{N}$</td>
<td>$M' \leftarrow C'^d \pmod{N}$</td>
</tr>
</tbody>
</table>

Since

$$C' \equiv 2^e \cdot C \pmod{N}$$
$$2^e \cdot M^e \pmod{N}$$
$$2 \cdot M)^e \pmod{N},$$
when Bob decrypts $C'$, he gets an $M'$ equal to $2 \cdot M$. That seems quite bad: if $M$ was a message saying “Alice owes Bob £100” then Alice might actually end up owing twice as much!

Both cases are meant to illustrate that Eve (now Mallory) has avoided the hard mathematical problems: in order to do “something bad”, there was no need to solve an instance of the RSA problem for example. What does this mean: is the protocol insecure? Or is the security model wrong? The combined difficulty of

1. understanding what a protocol should do,
2. understanding and modelling what any adversaries can do, and
3. designing a protocol that is efficient while also matching all functional requirements

make good cryptography very challenging. The fact that people are constantly developing new capabilities for Eve (e.g., new ways to factor numbers or solve the RSA problem) adds even more of a challenge; what is secure today may not be secure in ten or twenty years time, and equally the types of protocol parties want to engage in might change. But these challenges also make cryptography a fascinating subject: it must, by definition, constantly evolve to keep pace with both functionality and attack landscapes.


Part III

Security
CHAPTER

11

HIDING A NEEDLE IN A HAYSTACK: CONCEALED MESSAGES

Imagine you are employed as a spy or, to add a modicum of glamour to the story, the spy, James "007" Bond himself [5]. You have two suspects under surveillance and are tasked with identifying which one is plotting to end the world. The usual laser pen and exploding pants were loaned to 006, but you do have a watch that can intercept the emails people send; you use the watch to capture some evidence:

Email from suspect #1:

Dear Mum,

I am having a strange holiday. The weather here is nice enough, but there is a weird man in the next room who does nothing but drink vodka martinis and smile at women.

Love, David.

Email from suspect #2:

DE259236 4D581352 8D9ABE72 818B4040 856ECEC7 90DB9F2 7E85AF98 9BBD0F34 D14EC5DC 683D69C5 49C7AA7D A65DB65A 77DDE93B 815275EC 6F66DD23 22A0333A B1641A64 FC5333AF 8E21BB7 81115EF2 7DB41239 44942DCF E2C9AE6E D0BDB547 3B9F1BC2 23D1E844 7FD9474A 9B8C9D3B FAD65181 3EED4F8B 5D71AE1E 28F878B2

Love, Mr. X.

Which suspect should you throw in the shark infested swimming pool? Clearly suspect #2: by the look of it he is sending encrypted emails so if he does not want people to know what they say, he must be up to no good. What went wrong for suspect #2? Cryptography was meant to solve all his secrecy problems! In short, the problem here is not really one of secrecy because the message suspect #2 sent is still secure in the sense that it cannot be read. Rather, the problem is that the message he is sending stands out and therefore attracts attention. It would be nice if the message was also secret, but in this case hiding the message is perhaps more important to suspect #2. The topic of steganography [16], a Greek word meaning "concealed writing", gives us a solution. As we have described previously, cryptography is all about preserving the secrecy of data; steganography on the other hand relates to hiding secret data, typically within non-secret data. For example, imagine we intercept a million emails with the watch. Suspect #2 could hide a secret message inside an innocuous email similar to the one suspect #1 sent: the secret message in this case is one needle in a haystack of a million emails, and we may totally overlook it as a result.

Although steganography is more generally interesting, one reason it makes a good topic is the connection to use within digital media: we get to look at some pictures for a change, rather than long lists of numbers! In this context, steganography can be used as a way to watermark [4] digital media. Imagine we have taken a brilliant photograph, and upload it to our web-site. What is to stop a competitor downloading the image, making a copy of it on their web-site, and claiming they took the photograph not us? The answer is not much because the beauty (and curse) of digital media is that it is so easy to copy. In Chapter 2 we already saw that CDs and DVDs are just well organised sequences of numbers; digital images and MP3 files are the same thing. So, unlike an oil painting or an analogue cassette tape, it is trivial to make a perfect copy...
of most digital media. One way to combat this problem is to hide a message within whatever we want to protect: imagine we hide the message “this photograph was taken by X on the date Y” inside the image. Even if someone else copies it, they would have a hard time convincing a court of law that they had taken the image and just happened to put our name in it!

Of course, real digital watermarking techniques usually have additional and complicated requirements that we will not worry about; obviously we require the watermark to be robust, in the sense that it cannot easily be removed, for instance. Our aim here is to use the scenario above as a motivation to look at two types of digital image, and two forms of steganography that follow quite naturally from how such images are represented.

11.1 Digital images

It pains me to say it, but I am old enough to remember a time before digital photography existed. The question is, how do modern digital cameras do the same thing as older ones that were based mainly on chemical processes? That is, how do they represent a physical image using numbers so that we can store and manipulate them using a computer? The answer is not too involved, but it turns out that there are (at least) two quite different varieties of digital image.

11.1.1 Rasterised images

A rasterised image (or “bitmap” image) measures or samples the colour of the physical image at regular intervals; the samples are the digital representation we keep. You can think of a rasterised image as being a matrix where each element represents one sample called a pixel (or picture element) [11].

The number of rows and columns for a particular image is usually fixed; if we have a \((m \times n)\)-pixel image then the resolution is \(n\) by \(m\), meaning it has \(n\) columns and \(m\) rows. You might have seen images described as “1024 by 768” for example, which would mean \(n = 1024\) and \(m = 786\). When talking about digital cameras, it is also common to refer to the total number of pixels rather than the number of rows and columns. So a camera described as “having 10 megapixels” produces images where \(n \cdot m \sim 10 \cdot 10^6\), i.e., we might have a matrix with just over \(n = 3000\) columns and just over \(m = 3000\) rows.

In a sense, the total number of pixels tells us something about the image detail. Basically, if there are more pixels, more samples have been taken from the physical image and therefore there is more detail:

- A 1 megapixel digital camera produces a \((1000 \times 1000)\)-pixel image, i.e., one million pixels in total. Imagine we use the camera to take a picture of an object that is 5 m long: we sample about one pixel every 5 mm or so. What if there is an ant on the object which is only 2 mm long? Chances are we might miss him out if he happens to fall between two samples.

- A 10 megapixel digital camera produces an image with ten times as many pixels, i.e., ten million pixels in total. Now, taking a picture of the same 5 m object means we sample about one pixel every 1.6 mm. There is more detail in this image: we can capture smaller features within the physical image.

The next question is how we describe the pixels themselves; remember that they are meant to represent samples, or colours, from the physical image. To do this, we use a colour model [3] which tells us which numbers represent which colours.

11.1.1.1 The RGB colour model

The RGB colour model [14] represents colours by mixing together red, green and blue (i.e., “R”, “G” and “B”). The idea is to write a sequence of three numbers, i.e., a triple

\[ (r, g, b) \]

where each of \(r\), \(g\) and \(b\) is called a channel and represents the proportion of red, green or blue in the pixel. There are several ways we could specify \(r\), \(g\) and \(b\); for example we could use a percentage, e.g., 100% red, which is both easy to read and understand. However, it is more common to see them specified as an \(n\)-bit integer between zero and some maximum \(2^n - 1\). Imagine we set \(n = 8\): this means each channel is some number \(x\) in the range 0 . . . 255 and the proportion specified is \(x/(2^n - 1)\), in this case \(x/255\). Here are some examples to make things clearer:

1. The triple \((255, 255, 255)\) is white: it specifies 255/255 (or 100% red), 255/255 (or 100%) green and 255/255 (or 100%) blue.
2. The triple \(\langle 255, 0, 0 \rangle\) is a pure red colour: it specifies 255/255 (or 100% red), 0/255 (or 0%) green and 0/255 (or 0%) blue.

3. The triple \(\langle 255, 255, 0 \rangle\) is a pure yellow colour: it specifies 255/255 (or 100% red), 255/255 (or 100%) green and 0/255 (or 0%) blue.

4. The triple \(\langle 51, 255, 102 \rangle\) is a pea green colour: it specifies 51/255 (or 20% red), 255/255 (or 100%) green and 102/255 (or 40%) blue.

Since we need an \(n\)-bit integer for each channel, and we need three channels to specify the colour of each pixel, we need \(3n\) bits for each pixel. This is sometimes called the colour depth because it tells us how many colours we can represent, i.e., the number of bits-per-pixel (sometimes abbreviated as “bpp”). With \(n = 8\), each pixel needs 24 bits and we can represent nearly 17 million colours. This is enough colours to approximate physical images, and is used by many image formats that are produced by digital cameras.

The obvious next step is to put the two concepts together and arrange the RGB triples in a matrix to represent the pixels of an image. Figure 11.1 demonstrates what this actually looks like, both as a matrix of numbers and the pixels they represent. We can write the numbers that represent each colour channel in any base, so the matrix

\[
\begin{pmatrix}
\langle 255, 255, 255 \rangle & \langle 0, 0, 0 \rangle \\
\langle 255, 0, 0 \rangle & \langle 0, 255, 0 \rangle \\
\langle 0, 0, 255 \rangle & \langle 255, 255, 0 \rangle \\
\langle 255, 0, 255 \rangle & \langle 0, 255, 255 \rangle \\
\end{pmatrix}
\]

is equivalent: it is just written down in a different way.
You might be used to creating images using software that allows you to “paint” interactively (using a mouse). When we need careful control over pixel values, however, it can be easier to describe images using a text format such as PPM [9]. For example,

```
P3       # this is an ASCII PPM file
2        # there are 2 columns
4        # there are 4 rows
255      # maximum R, G or B is 255

255 255 255 # P(0,0)
0 0 0       # P(0,1)
255 0 0     # P(1,0)
0 255 0     # P(1,1)
0 0 255     # P(2,0)
255 255 0   # P(2,1)
255 0 255   # P(3,0)
0 255 255   # P(3,1)
```

is a PPM description of Figure 11.1. The first four lines specify information about the image, namely

- the PPM file identifier (which ensures any software reading the file can interpret the associated content correctly),
- the image dimensions, first the number of columns then the number of rows, and
- the maximum value any colour channel can have.

The subsequent lines specify image content: each line specifies a pixel, from top-left to bottom-right in the image. So the first pixel in the top-left corner is (255, 255, 255) (meaning white).

Try to reproduce this example using image manipulation software of your choice to view the result.

As well as the red, green and blue channels, it is common to include an *alpha* channel. This forms the RGBA colour space [15], and typically means that each pixel is represented by 32 rather than 24 bits. Find out about the purpose of this addition, then try to show the effect it has using some experimental images.

### 11.1.1.2 The CMYK colour model

RGB is not the *only* colour model; you might reasonably argue that it is not even the most sensible since it does not correspond to what happens when paints are mixed together. Consider some examples:

1. To get green coloured paint you need to mix yellow and blue paint together. With RGB on the other hand, you can get a green colour “for free”; adding red gives a yellow colour. Try mixing red and green paint together: you do not get yellow paint!

2. If you mix together all colours of paint together you get black (or maybe just a mess), but in RGB you end up with a white colour.

So why the difference? Think about what a computer monitor is doing in a physical sense: basically it just emits light. If it emits no light, then we as the viewer see a black colour. If it emits red, green and blue then these “add up” to white. So, with the RGB model, colours are additive. Now think about how we see paint: white light hits the paint, the paint absorbs selected parts of the spectrum and reflects what is left so that we as the viewer can see it. If white light, (255, 255, 255) as an RGB triple, hits yellow paint, it absorbs all the blue component and we get reflected (255, 255, 0) back; if white light hits cyan paint, it absorbs all the red component and we get (0, 255, 255) reflected back. So if we mix yellow and cyan paint the result
will absorb all the blue and red components from the spectrum and hence reflect \( (0, 255, 0) \), i.e. green, back. Hence, paint absorbs (i.e., subtracts) colours of light, whereas a computer monitor emits (i.e., adds) colours of light.

The idea of using cyan, magenta and yellow as primary colours is standard in the printing world since it deals with ink (which is like paint). So whereas we use red, green and blue as primary colours and hence the RGB colour model within computers, when we print rather than display images it is common to use the CMYK colour model instead.

<table>
<thead>
<tr>
<th>Research (task #46)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you are really serious about printing and typesetting, the name Pantone [10] will crop up a lot: the Pantone Matching System (PMS) is a standard colour model used in most commercial settings. Do some research about this colour model, and the legal issues which mean it often cannot be used (e.g., within the open source GIMP software mentioned in Chapter 2).</td>
</tr>
</tbody>
</table>

11.1.2 Vector images

A vector image describes a physical image using a combination of geometric primitives such as points and lines. It is also possible to include more complicated curved surfaces (e.g., circles and ellipses), and even to extend from 2-dimensions into 3-dimensions. An example makes this easy to explain. Imagine you have a large sheet of graph paper: a 2-dimensional point on our graph paper is just a pair of coordinates that we write as \((x, y)\). We can draw such a point on the graph paper; scale is not important, but you can imagine the paper has unit sized or 1 cm squares on it if that helps. Using two such points, we can write

\[(x, y) \rightarrow (p, q)\]

to describe a 2-dimensional line segment between points \((x, y)\) and \((p, q)\). Again, we can draw such a line on the graph paper: just get a ruler and join up the points. Using lots of lines, we can start to draw basic images. Imagine we start with this one

\[
\begin{align*}
(1.0, 1.0) & \rightarrow (2.0, 1.0) \\
(2.0, 1.0) & \rightarrow (2.0, 2.0) \\
(2.0, 2.0) & \rightarrow (1.0, 2.0) \\
(1.0, 2.0) & \rightarrow (1.0, 1.0) \\
(3.0, 1.0) & \rightarrow (4.0, 1.0) \\
(4.0, 1.0) & \rightarrow (3.5, 2.0) \\
(3.5, 2.0) & \rightarrow (3.0, 1.0)
\end{align*}
\]

which describes seven lines in total. The first four lines form a square with sides of length one unit, the last three form an isosceles triangle of base and altitude one unit. It does not look very impressive when written like this, but if we were to draw it on our graph paper, it would look like Figure 11.2a. On one hand, this is still a fairly unimpressive image; we could make it more complicated by drawing more lines, but it still might be difficult to represent a physical image such as a face or a flower. On the other hand, what we might have sacrificed in terms of realism we recoup in terms of precision: since the image is effectively described in terms of Mathematics, we can manipulate it via Mathematics as well. There are applications where this is a real benefit; if you are designing a new car or building, accuracy is vital!

<table>
<thead>
<tr>
<th>Implement (task #47)</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is not as common now, but a system called Logo [7] has been used extensively as a way to teach programming. The idea is that a program controls an on-screen turtle that is guided around, drawing as it moves. In a sense, the Mathematical description of images above is very close to a Logo program: see if you can use an online resource such as <a href="http://turtleacademy.com/">http://turtleacademy.com/</a> to reproduce our simple example. If you have no background in programming, this is a good chance to explore what else is possible.</td>
</tr>
</tbody>
</table>

To demonstrate the significance of this, we can start to think about taking a point \((x, y)\) and translating it into a new point \((x', y')\) using some form of transformation. Three such transformations, which describe
(a) Original.

(b) Translate.

(c) Mirror vertically.

(d) Scale horizontally and vertically.

(e) Scale horizontally only.

(f) Rotate about origin.

Figure 11.2: Some example vector image transformations.
Imagine we select an example point (1.0, 2.0), i.e. we have \( x = 1.0 \) and \( y = 2.0 \), which is the bottom left-hand corner of our square. The first thing we might try is to translate (or “move”) the point to somewhere else in the image. To do this we add a vector \((i, j)\). To move the point one unit horizontally and one unit vertically for example, we compute the new point as

\[
\begin{align*}
\hat{x} &= x + i = 1.0 + 1.0 = 2.0 \\
\hat{y} &= y + j = 2.0 + 1.0 = 3.0
\end{align*}
\]

In isolation this probably does not seem very exciting. But if we repeat the transformation for all the points in the image, we end up with the result in Figure 11.2b. And that is just for starters! We can reflect an image in either axis. For example by negating the \( y \) coordinate we can mirror the image in the \( y \)-axis, as in Figure 11.2c.

If we want to scale a point \((x, y)\) by a factor of \( i \) horizontally and \( j \) vertically we multiply \( x \) and \( y \) by \( i \) and \( j \). Imagine we want to double the size of the image; setting \( i = 2 \) and \( j = 2 \) we compute the new point as

\[
\begin{align*}
\hat{x} &= 1.0 \cdot 2.0 = 2.0 \\
\hat{y} &= 2.0 \cdot 2.0 = 4.0
\end{align*}
\]

Again, doing the same thing with all the points in our image gives the result in Figure 11.2d. This time the result is a bit more impressive: we have doubled the image size, but we have not degraded the quality at all. The Mathematics to describe the lines, and hence our image, is “perfect”. We can play about even more, and stretch the image horizontally; setting \( i = 2 \) and \( j = 1 \) gives Figure 11.2e.

As a final trick we can think about rotating the image, the result of which is shown in Figure 11.2f. If we want to rotate the point by \( \theta \), say \( \theta = 45 \) for instance, we compute the new point (to three decimal places) as follows:

\[
\begin{align*}
\hat{x} &= 1.0 \cdot \cos(45) - 2.0 \cdot \sin(45) = -0.325 \\
\hat{y} &= 1.0 \cdot \sin(45) + 2.0 \cdot \cos(45) = 1.376
\end{align*}
\]

What other types of useful transformation can you think of? As a hint, think about a shearing transform which produces a slanting result. Find out how such transformations can be specified and applied by using matrices [17].

Rotation is the first point where we hit a problem: the “perfectness” of the Mathematics fails us when we want to write down actual values for \( x' \) and \( y' \), because of the limit on precision (put simply, we only have a fixed number of decimal places available). Until then, we can manipulate everything perfectly without ever having to see a number! In a sense, the same problem crops up when we want to print or view a vector image.

Most laser printers [6] or plotters [12] can draw vector images. Many accept PostScript [13], a language for describing vector images, as input: programs written in PostScript are essentially lists of geometry, such as the line segments we started off with. But, eventually, physical constraints will cause problems. Such a printer cannot usually draw infinitely small images, for example, since there is a limit to how accurate mechanical parts can ever be. Some other types of display device [18] can also render vector images; early video games like Asteroids [1] used this sort of technology. These are rare however, and most modern computer monitors work in a different way: they first rasterise the vector image, turning it into a format that can be displayed. And there lies the problem: the process of rasterisation throws away the “perfect” nature of the Mathematics and forces us to do things like round-up the coordinates of a point, which might be represented using many decimal places of precision, so they match the nearest integer pixel location.
11.2 Steganography

Back to James Bond, or rather the problem of stopping his pesky snooping. Remember that the idea is to hide some secret data in non-secret data and, by doing so, throw him off the scent. The secret message will be a string of characters (an email if you like) and the non-secret data will be a digital image. Each type of digital image we have looked at gives quite a neat way to do this.

11.2.1 Rasterised images: “stolen LSBs”

We have already learned that a rasterised image can be represented as a matrix of numbers, and ideally binary numbers. For example, our original (4×2)-pixel example image was

\[
\begin{bmatrix}
\langle 11111111(2), 11111111(2), 11111111(2) \rangle & \langle 00000000(2), 00000000(2), 00000000(2) \rangle \\
\langle 11111111(2), 00000000(2), 00000000(2) \rangle & \langle 00000000(2), 11111111(2), 00000000(2) \rangle \\
\langle 00000000(2), 00000000(2), 11111111(2) \rangle & \langle 11111111(2), 11111111(2), 00000000(2) \rangle \\
\langle 11111111(2), 00000000(2), 11111111(2) \rangle & \langle 00000000(2), 11111111(2), 11111111(2) \rangle
\end{bmatrix}
\]

Some of the bits are given special names:

- The bits at the right-hand end of each number (those coloured red), are termed the **least-significant** bits or LSBs: they contribute weights of $2^1$ and $2^0$ to the overall value, the smallest weights of all.

- The bits at the left-hand end of each number (those coloured blue), are termed the **most-significant** bits or MSBs: they contribute weights of $2^7$ and $2^6$ to the overall value, the largest weights of all.

What happens if we alter either the LSBs or MSBs of the pixels in an image? Imagine we look at each pixel and set the two LSBs of each colour channel to zero. Or, maybe we look at each pixel and randomise the two MSBs of each colour channel. Figure 11.4 demonstrates four images that were created by taking an original and slightly altering each pixel along these lines. Figure 11.4b and Figure 11.4c had the two LSBs in each pixel altered, either set to zero or a random 2-bit value respectively. Although the images are marginally different (perhaps a little darker), without the original you would be hard pressed to pick them out as having been altered at all. This should make sense: we are altering the LSBs and these have the least impact on the value of each colour, so changing them does not have a lot of impact.

In Figure 11.4d and Figure 11.4e, the two MSBs rather than the LSBs were altered and the results are quite striking: if the MSBs are altered, the images are clearly corrupted. In Figure 11.4d, each pixel has become much darker: the image is more or less intact and understandable, but in Figure 11.4e is much less understandable; there is still some structure if you look closely, but it is really quite random. Again, this should make sense, we are altering the MSBs and these have the most impact on the value of each colour. In particular, if we zero the two MSBs we are basically saying that each colour channel can only take a value 0...63 rather than 0...255 so at best, it can only be about a quarter as bright.

The question is, how can we use our findings as a steganographic mechanism? The answer is reasonably simple. We are going to “steal” the LSBs from their original purpose of contributing to an image, and use them to conceal a message. The theory is that altering the LSBs will not corrupt the image too much: we have already seen that even if we randomise them, the end result is very close to the original. Our original (4×2)-pixel example image had eight pixels in total; we are stealing six bits from each pixel (two from each of the colour channels), so 48 bits in total. Chapter 5 already showed us than an ASCII character can be stored using 8 bits. Therefore, we can store a 6-character message in the 48 bits we have at our disposal.

First, we need to choose a message: the 6-character string “hello.” (note the trailing full stop) is not particularly exciting, but will do the job here. Writing the ASCII representation of the string as a sequence we have


If we write this in binary instead our message is

\[\langle 01101000(2), 01100101(2), 01101100(2), 01101100(2), 01101111(2), 00101110(2) \rangle.\]
Bond and Tanaka discuss a surveillance photograph. Tanaka reveals the microdot ... and then enlarges it. Zooming in on the photograph reveals the boat name: “Ning-Po”.

Ernst Blofeld. Bond, Blofeld and the infamous cat.

Figure 11.3: Scenes from You Only Live Twice, 1967 © United Artists; the microdot text translates as “photograph taken by female American tourist from coastal vessel; the woman has been liquidated as a routine precaution”.

Figure 11.4: Four images, each created by altering the pixels from an original image.
and if we split it up into 2-bit chunks then, reading left-to-right and top-to-bottom, we get

\[
\begin{bmatrix}
01_{(2)}, & 10_{(2)}, & 10_{(2)}, & 00_{(2)}, & 01_{(2)}, & 10_{(2)}, \\
01_{(2)}, & 01_{(2)}, & 01_{(2)}, & 10_{(2)}, & 11_{(2)}, & 00_{(2)}, \\
01_{(2)}, & 10_{(2)}, & 11_{(2)}, & 00_{(2)}, & 01_{(2)}, & 10_{(2)}, \\
11_{(2)}, & 11_{(2)}, & 00_{(2)}, & 10_{(2)}, & 11_{(2)}, & 10_{(2)}
\end{bmatrix}
\]

which is now ready to be injected into the image. Again, this is reasonably simple: we just start with the example image and replace the red LSBs with the sequence of 2-bit chunks derived from our message. The result is as follows:

\[
\begin{bmatrix}
\langle 11111101_{(2)}, 11111110_{(2)}, 11111110_{(2)} \rangle, & \langle 00000000_{(2)}, 00000001_{(2)}, 00000010_{(2)} \rangle \\
\langle 11111101_{(2)}, 00000001_{(2)}, 00000001_{(2)} \rangle, & \langle 00000010_{(2)}, 11111110_{(2)}, 00000000_{(2)} \rangle \\
\langle 00000001_{(2)}, 00000010_{(2)}, 11111111_{(2)} \rangle, & \langle 11111100_{(2)}, 11111101_{(2)}, 00000010_{(2)} \rangle \\
\langle 11111111_{(2)}, 00000011_{(2)}, 11111100_{(2)} \rangle, & \langle 00000010_{(2)}, 11111111_{(2)}, 11111110_{(2)} \rangle
\end{bmatrix}
\]

If we turn this back into decimal to make it easier to read, and render the matrix as pixels as well, the end result is Figure 11.5. The left-hand side shows the two matrices: clearly there is a difference as a result of the LSBs having been commandeered to store the message.

Reversing the process to extract the message is just as simple: we take the pixels, extract the LSBs and then recombine the 2-bit chunks into 8-bit bytes which represent the characters. But the point is that we would have to know the message was there in the first place before we even attempted to extract it. Looking at the comparison, it is difficult to see any difference in the images themselves: marginal changes in the colour channels are not easily detected even when we have the original image (which of course we would ordinarily lack).
If you **know** the message (or watermark) has been embedded in an image, it is of course easy to extract it. What about if you just suspect such a watermark is there? Can you think of a way to detect watermarks of this (or some other, similar) type?

### 11.2.2 Vector images: “microdots”

*You Only Live Twice* [19] is the first film to feature cat lover Ernst Blofeld, the head of SPECTRE, as a main character (in previous films you only see a character stroking a white cat, never his face). Part way through the film, Bond recovers a photograph of a cargo ship; on the photograph is something called a **microdot** [8]. A microdot is basically a very tiny image. The idea is to scale one image until it is so small it can be placed, in an inconspicuous way, within another image. Because it is so small, the premise is that it will be overlooked by a casual observer. So we take a secret message, scale it until it looks like a barnacle on the side of the cargo ship and then only someone looking for it (or who is far too interested in barnacles) will be able to recover the message.

Far from being limited to science fiction, there is plenty of evidence to suggest microdots being used as a real steganographic mechanism. Various sources have claimed invention, but it was almost certainly used by German intelligence agents in WW2 to send covert messages. British counter-intelligence nicknamed the microdots “duff” because they were mixed into letters like raisins were mixed into in the steamed pudding “plum duff”. More modern uses include identification of physical resources such as car parts: the car manufacturer prints a unique number on each part in order to trace it in the event it is stolen. Of course a car thief might try to cover their tracks by etching off any obvious markings, but if they cannot even see the microdot then the chances of removing it are slim.

From the terminology we have used so far, it perhaps is not a surprise that we can try to create a real microdot using vector images: remember they these can be scaled without loss of quality, so are an ideal match. Figure 11.6 shows a somewhat basic example where we have scaled a message so that it is small enough to fit inside a full stop. If you are reading an electronic PDF version of this document, you can actually zoom in and enlarge the dot to see for yourself. If you are reading a version of the document printed on paper, this clearly would not work. Why not? We already talked about the problem: a given printer has limits as to how accurate the print mechanism is. So, in printing the electronic version that includes the perfect Mathematical description of our message, we have lost all the detail. Real microdots are therefore created using a traditional photographic process in which (very roughly) a special camera is used which shrinks an image of the message using magnification.
The 1983 film *WarGames* saw lead character David Lightman faced with a problem [9]. David had already hooked his computer up to the telephone system and broken into the school computer to change his grades. He succeeded because someone left the password on a desk in the school office. He then set about finding other computers by performing an automated search (we call this war dialing [8]) of all telephone numbers in the region. After a while he hit the jackpot: an interesting looking computer answered his call. But he could not gain access because this time he did not have the password.

It is easy to model this problem in a more formal way. Imagine the remote computer $C$ has a password $P$ embedded inside it, and that $P$ is a sequence of characters (lower-case alphabetic characters only, to make things easier). Our job as the attacker Eve is to guess $P$. We can make successive attempts, each of which means sending a guess $G$ to the computer: the computer takes $P$ and $G$ and uses an algorithm called Match-Pwd to compare them. If $P$ and $G$ are the same then $\text{Match-Pwd}(P,G)$ returns $\text{true}$, otherwise it returns $\text{false}$. We know whether or not a guess was correct, because if we guessed correctly we obviously then get access to the computer. The following diagram tries to capture this model:

![Diagram of the password guessing model.](image)

So how do we, as the attacker, proceed? One method might be to try all possible passwords; this is called a brute-force attack [1]. Imagine the password has $n$ characters in it, and for the sake of argument say $n = 6$. With our brute-force attack we would perform guesses of the form

$\text{Match-Pwd}(P, "aaaaaa")$
$\text{Match-Pwd}(P, "baaaaa")$
$\text{Match-Pwd}(P, "caaaaa")$

$\vdots$

$\text{Match-Pwd}(P, "zaaaaa")$
$\text{Match-Pwd}(P, "abaaaa")$
$\text{Match-Pwd}(P, "bbaaaa")$
$\text{Match-Pwd}(P, "cbaaaa")$

$\vdots$

$\text{Match-Pwd}(P, "zbaaa")$
$\vdots$

$\text{Match-Pwd}(P, "zzzzz")$
The school password "pencil", left on an unattended desk.

Back at home, David dials the school computer ...

... using an acoustic coupler.

The school computer issues a login prompt.

After entering the password ...

... David changes his Biology grade.

The war dialler trying every telephone number in the area.

AI researcher Prof. Stephen Falken.

David tries the password "joshua" ...

... and gains access to WOPR, the NORAD nuclear war simulator.

The AI within WOPR allows David to play a "game" of Global Thermonuclear War ...

... which looks like a real war to the staff at NORAD.

Figure 12.1: Scenes from WarGames, 1983 © MGM Pictures.
until eventually one of them matched. There are two problems: first, we do not know what \( n \) is (but we can solve this later); second, the number of possible passwords is \( 26^n \) which grows quickly even if \( n \) grows slowly. With \( n = 6 \) for example, \( 26^6 = 308915776 \) passwords are possible: not only will it take some time to make all those guesses, but after the first thousand or so wrong guesses someone in charge of the system should notice there is a problem. On the other hand, there are advantages in the sense that although the brute-force attack might take a long time it at least guarantees success if we wait long enough.

The next thing we could do is try common passwords. This is often called a dictionary attack [2]. The idea is that we have or make a dictionary of words, and use those words (including combinations of them) as our guesses. It turns out that people often select weak (i.e., easy to guess) passwords [4], so this approach can be effective. This is particularly true if we include various common passwords in our dictionary such as system defaults (e.g., “password” or “admin”), names of family and pets, football teams and so on. In fact David Lightman eventually solved his problem in exactly this way: the designer of the computer he had contacted, Prof. Stephen Falken, set the password to the name of his dead son Joshua. Clearly a similar approach could reduce the number of attempts versus a brute-force attack, but on the other hand it does not guarantee success: the actual password might not occur in our dictionary.

As we saw in Chapter 7 and Chapter 10 however, when we design cryptographic schemes we do so in a way that should prevent these two forms of attack. For example, the key (which is of course analogous to the password) should be large enough to prevent a brute-force attack. More often than not, an attacker would need to cryptanalyse the system, usually attempting to find some weakness in the underlying Mathematics. Within this context, the concept of side-channel attacks [6] is relatively new. The idea is that rather than studying just the mathematics of a cryptographic scheme “on paper”, we consider the fact that the scheme must be implemented as a program which executes on a computer. The idea is that as an attacker we might be able to passively monitor, or actively influence, how the computer executes the program. Based on this activity we hope that cryptanalysis could be easier; of course, how feasible this is depends on the exact scenario. However, motivating examples are easy to come by: modern computers (e.g., a chip-and-pin card) are increasingly carried around with us, contain sensitive information (e.g., your banking details) and are used in a setting controlled by other people (e.g., the terminal of a supermarket checkout). So it is not too hard to imagine that side-channel attacks are sometimes a feasible and useful addition to the range of approaches on offer. Our aim in this Chapter is to look at side-channel and fault attacks in a non-technical way: some of the examples do not exactly match what would happen in real life, but act as good metaphors for the concepts involved.

In the description above, there are some hints that our simplified model is not how real access control systems work. Within a system that controls access to a large web-site for instance, it would normally be a bad idea to store \( P \) itself somewhere.

Find out about why this choice is made, e.g., using the LinkedIn web-site breach\(^6\) in 2012 as motivation, and how it is supported: write out a set of requirements, and then steps that describe a better access control system than the one modelled.

\(^6\) http://blog.linkedin.com/2012/06/06/linkedin-member-passwords-compromised/

12.1 Passive physical attacks

12.1.1 Attack

Imagine we represent strings as sequences of characters, for example we might have a sequence

\[ A = \langle 'a', 'b', 'c', 'd' \rangle. \]

Instead of writing \( \langle 'a', 'b', 'c', 'd' \rangle \) for example, we would normally write “abcd” instead. Based on this, how might we write an algorithm to check whether two strings (in our case \( P \) and \( G \)) are the same or not? Easy! We already saw algorithms in Chapter 5 that do this. If we ignore the type of string, since we are not really interested in that, the algorithm is (re)shown in Figure 12.2. Recapping on Chapter 5 a little, have a look at it one step at a time. The first thing that happens is a conditional: if the number of characters in \( P \) or \( G \) are not the same (\( n \) or \( m \)), we return \textbf{false} as the result. If \( P \) and \( G \) are the same length then we need to check each character. To do this, we use a loop which iterates over a block for values of \( i \) in the range \( 0 \ldots n - 1 \). By this point we know \( n = m \) so there is no danger of the \( i \)-th element of \( P \) or \( G \) being invalid. For each value of \( i \), we test if \( P_i = G_i \), i.e., if the \( i \)-th character of \( P \) is equal to the \( i \)-th character of \( G \). If they are not equal, then clearly the two strings are not equal and we can return \textbf{false} as the result; if they are equal, we need to test all the other possible values of...
An aside: other, everyday analogies for information leakage.

The example of guessing passwords might sound a little contrived, especially if you know a little more about how such systems are really implemented. Even so, the problem of information leakage is ubiquitous enough that many other everyday analogies exist. Consider for example

1. a burglar who opens a safe using a stethoscope to collect audible information leaked while operating the lock,
2. a burglar who guesses a door entry code by noting leaked information in the form of worn buttons on the keypad, or
3. a TV license detector van who reasons about your use of a TV set by monitoring the electro-magnetic emission from your house.

In each case, unintentional information leakage of different forms (i.e., different from execution time, which we used for guessing passwords) is the common issue.

```
1 algorithm MATCH-Pwd(P, G) begin
  2     n ← STRING-LENGTH(P)
  3     m ← STRING-LENGTH(G)
  4     if n ≠ m then
  5         return false
  6     end
  7     for i from 0 upto n - 1 do
  8         if P_i ≠ G_i then
  9             return false
 10       end
 11   end
 12   return true
end
```

Figure 12.2: An algorithm to test whether the guess G matches some password P (where both G and P are strings).

i. Finally, after we have completed all our tests we can be confident that the two strings are the same and return true.

Obviously it takes some time for the computer to execute an implementation of the MATCH-Pwd algorithm. Thinking back to Chapter 5, you will remember that the algorithm is O(n), i.e., the number of steps it takes is tied to the number of times the loop iterates. But a subtle issue is at the crux of what we are interested in: the loop does not always make n iterations because if there is a case where P_i ≠ G_i, then the algorithm terminates early. Some examples make this clear:

- MATCH-Pwd(“joshua”, “bob”) returns false and takes 1 step because n ≠ m.
- MATCH-Pwd(“joshua”, “daniel”) returns false and takes 2 steps because n = m but at i = 0, P_0 = ‘j’ ≠ ‘d’ = G_0.
- MATCH-Pwd(“joshua”, “joanne”) returns false and takes 4 steps because n = m but at i = 2, P_2 = ‘s’ ≠ ‘a’ = G_2; obviously for i = 0 and i = 1 we have P_i ≠ G_i.
- MATCH-Pwd(“joshua”, “joshua”) returns true and takes 7 steps because n = m and for all i we have P_i = G_i.

The idea is now to imagine that we can actually time how long it takes the computer to execute MATCH-Pwd and hence recover the information above. This is not difficult: since we are communicating with the remote computer, we just measure the time that elapses between sending it a guess and getting a response back. We can redraw the original diagram to look more like this:
The question is, how can we exploit this side-channel to help us solve the original problem of guessing the password? Notice that the more characters at the start of our guess that match those in the password, the longer the computer will take to give us this result. We say that information has leaked through a side-channel related to execution time; we still do not know $P$, but we do know a bit more than we thought we did. Also notice that we have solved the problem of how long $P$ is: if the algorithm takes more than one step then we know our guess is the right length.

So let us return to the brute-force attack, but each time we make a guess we time how long it takes to get the result back and use this information to help us. We start by cycling through guesses of $G_0$:

$\text{Match-Pwd}(P, "aaaaaa") \Rightarrow 2$ steps

$\text{Match-Pwd}(P, "baaaaa") \Rightarrow 2$ steps

$\text{Match-Pwd}(P, "caaaaa") \Rightarrow 2$ steps

$\vdots$

$\text{Match-Pwd}(P, "jaaaaa") \Rightarrow 3$ steps

$\vdots$

$\text{Match-Pwd}(P, "zaaaaa") \Rightarrow 2$ steps

One of the guesses will take slightly longer to return a result than the rest. This is because one of the guesses will make $P_0 = G_0$ and so will take 3 steps rather than 2 steps as in the cases where $P_i \neq G_i$. So we know this guess must be the real value of $P_0$. We carry on, but now we keep $G_0$ set to $P_0$, which we now know, and cycle through guesses at $G_1$:

$\text{Match-Pwd}(P, "jaaaaa") \Rightarrow 3$ steps

$\text{Match-Pwd}(P, "jbaaaa") \Rightarrow 3$ steps

$\text{Match-Pwd}(P, "jcaaa") \Rightarrow 3$ steps

$\vdots$

$\text{Match-Pwd}(P, "jaaaaa") \Rightarrow 4$ steps

$\vdots$

$\text{Match-Pwd}(P, "jzaaaa") \Rightarrow 3$ steps

Again, one of the guesses will take slightly longer to return a result than the rest. This time one of the guesses will make $P_1 = G_1$ and so will take 4 steps rather than 3. Again we know this guess must be the real value of $P_1$; we carry on by keeping $G_1$ set to $P_1$, which we now know, and cycle through guesses at $G_2$:

$\text{Match-Pwd}(P, "jaaaaa") \Rightarrow 4$ steps

$\text{Match-Pwd}(P, "jobaaa") \Rightarrow 4$ steps

$\text{Match-Pwd}(P, "jocaa") \Rightarrow 4$ steps

$\vdots$

$\text{Match-Pwd}(P, "josaaa") \Rightarrow 5$ steps

$\vdots$

$\text{Match-Pwd}(P, "jozaa") \Rightarrow 4$ steps

If we follow this approach, we eventually guess correctly. The crucial thing to realise is that by using the side-channel information we have dramatically decreased the number of guesses we need to make. In the brute-force attack we needed $26^n$ guesses in the worst case, whereas now we need $26 \cdot n$; more formally, the brute-force attack can be described as $O(26^n)$ whereas the side-channel attack is $O(n)$. The latter is clearly better since it is at least somewhat feasible even with quite large $n$: with $n = 6$ that is only 156 guesses rather than 308915776.
Cast your mind back to Chapter 10, and even more specifically to the Exponentiate-Mod algorithm. Imagine this is used by a target computer that computes RSA decryptions for users; using a similar model, we would have something like

$$\text{Eve} \xrightarrow{\text{Exponentiate-Mod}(C, d, N)} C$$

Think about this setting: given Eve can still measure the execution time, what information leaks from $C$? Based on what you know about RSA, do you think this alone is a problem or not? Whatever your answer is, explain why.

### 12.1.2 Countermeasures

If we wanted to prevent this sort of side-channel leakage, what could we do? Rather than come up with a totally different algorithm for checking passwords, one idea is to add some countermeasures to our existing Match-Pwd algorithm: they are like a bandage which should patch up the algorithm and prevent the attack working. There are lots of potential ideas; the plan here is to explore four of them, plus any advantages or disadvantages they might have. To highlight the differences, we have collected the altered algorithms together in Figure 12.3.

#### 12.1.2.1 Detect the attack

We have already mentioned one idea that could be quite effective: if we notice that numerous incorrect guesses have been made, we assume they are being made by an attacker and shut down the computer to prevent access. A basic version is captured by Figure 12.3a. Of course it is a little simplistic, but the central idea is that we maintain a counter somewhere called $c$ and initially set it to zero when the computer is turned on. Notice that each time the algorithm detects an incorrect guess it adds one to $c$, so if lots and lots of incorrect guesses are made in a row then $c$ will grow quite large. When a correct guess is registered, we set $c$ back to zero again and we forget about any incorrect guesses up to that point. The algorithm capitalises on this behaviour by checking $c$ before it does anything else: if $c$ is larger than some threshold value $t$ it instantly returns false and denies access to the computer, in effect denying access forever or at least until we manually reset $c$ back to zero somehow. So basically what we are saying is that if $t$ incorrect guesses in a row are made, then the computer will shut down. Choosing a suitable value for $t$ will result in the attack becoming much less feasible.

#### 12.1.2.2 Slow down the attack

If you were the administrator of the computer, you might be having a heart attack after reading the previous idea: even legitimate users make mistakes, so you would probably be called out at all hours of the day to reset $c$ after someone forgot the password! Another less problematic idea is not to shut down the computer, but to slow it down: in short, the more incorrect guesses the attacker makes the longer the computer takes to use Match-Pwd. This is a simple alteration from our previous algorithm, and is shown in Figure 12.3b.

The alteration is simply in the first step: instead of testing $c$ against a threshold, we wait for $c$ steps. This means as more and more incorrect guesses are made, $c$ grows larger and larger and the computer is slower and slower to respond: making many incorrect guesses, as necessitated by the attack, thus becomes laborious at best! This approach is a variant of something called exponential back-off [3]; you would see similar approaches to controlling network congestion for example.

#### 12.1.2.3 Take a fixed number of steps

We could force Match-Pwd to take the same, fixed number of steps no matter what values of $P$ and $G$ are given to it. If this is possible, the idea is that information previously leaked is hidden: if the attacker cannot determine the number of steps taken, they cannot use the same strategy.
\begin{figure}
\centering
\begin{minipage}{0.45\textwidth}
\begin{algorithm}
\textbf{Match-Pwd}(P, G) \begin{algorithmic}
\State \text{if } c > t \text{ then} \\
\State \quad \text{return false} \\
\State \end{algorithmic}
\end{algorithm}
\end{minipage}\hfill
\begin{minipage}{0.45\textwidth}
\begin{algorithm}
\textbf{Match-Pwd}(P, G) \begin{algorithmic}
\State \text{Pause for } c \text{ steps} \\
\State \text{return false} \\
\State \end{algorithmic}
\end{algorithm}
\end{minipage}
\caption{(a) Detect the attack.}
\end{figure}

\begin{figure}
\centering
\begin{minipage}{0.45\textwidth}
\begin{algorithm}
\textbf{Match-Pwd}(P, G) \begin{algorithmic}
\State \text{if } c \neq m \text{ then} \\
\State \quad \text{return false} \\
\State \end{algorithmic}
\end{algorithm}
\end{minipage}\hfill
\begin{minipage}{0.45\textwidth}
\begin{algorithm}
\textbf{Match-Pwd}(P, G) \begin{algorithmic}
\State \text{Pause for } c \text{ steps} \\
\State \text{return false} \\
\State \end{algorithmic}
\end{algorithm}
\end{minipage}
\caption{(b) Slow down the attack.}
\end{figure}

\begin{figure}
\centering
\begin{minipage}{0.45\textwidth}
\begin{algorithm}
\textbf{Match-Pwd}(P, G) \begin{algorithmic}
\State \text{for } i \text{ from } 0 \text{ upto } n - 1 \text{ do} \\
\State \quad \text{if } P_i \neq G_i \text{ then} \\
\State \quad \quad \text{return false} \\
\State \end{algorithmic}
\end{algorithm}
\end{minipage}\hfill
\begin{minipage}{0.45\textwidth}
\begin{algorithm}
\textbf{Match-Pwd}(P, G) \begin{algorithmic}
\State \text{for } i \text{ from } 0 \text{ upto } n - 1 \text{ do} \\
\State \quad \text{if } P_i \neq G_i \text{ then} \\
\State \quad \quad \text{return false} \\
\State \end{algorithmic}
\end{algorithm}
\end{minipage}
\caption{(c) Take a fixed number of steps.}
\end{figure}

\begin{figure}
\centering
\begin{minipage}{0.45\textwidth}
\begin{algorithm}
\textbf{Match-Pwd}(P, G) \begin{algorithmic}
\State \text{for } i \text{ from } 0 \text{ upto } \min(n, m) - 1 \text{ do} \\
\State \quad \text{if } P_i \neq G_i \text{ then} \\
\State \quad \quad \text{return false} \\
\State \end{algorithmic}
\end{algorithm}
\end{minipage}\hfill
\begin{minipage}{0.45\textwidth}
\begin{algorithm}
\textbf{Match-Pwd}(P, G) \begin{algorithmic}
\State \text{for } i \text{ from } 0 \text{ upto } n - 1 \text{ do} \\
\State \quad \text{if } P_i \neq G_i \text{ then} \\
\State \quad \quad \text{return false} \\
\State \end{algorithmic}
\end{algorithm}
\end{minipage}
\caption{(d) Take a random number of steps.}
\end{figure}

\textbf{Figure 12.3:} Four example countermeasures to harden \textbf{Match-Pwd} against side-channel attack.
Figure 12.3c implements this idea. To keep track of when a difference is found, it maintains a flag called \( f \): it starts with \( f = \text{true} \), then when it finds a difference sets \( f = \text{false} \) to note this fact. The flag is only returned once the algorithm finishes checking characters in \( P \) and \( G \): it \textit{always} checks all characters to determine \( f \), meaning it \textit{always} takes a fixed number of steps. Or at least it does if \( P \) and \( G \) are the same length. Imagine we have \( P = \text{"joshua"} \) for example, and make two guesses \( G = \text{"daniel"} \) and \( G' = \text{"joanne"} \). The algorithm will take 8 steps in both cases, so the original attack strategy fails.

Of course, we have paid a price for this improved security: previously, if \( P \) and \( G \) were long strings then our algorithm was efficient in the sense that as soon as we definitely knew the result, we returned it straight away. With our new algorithm, we are always looking at the worst case; the algorithm always takes the longest possible time it could.

![Implement (task #52)](implement.png)

With reference to Task 51, how could you apply this same idea as a countermeasure for \textsc{Exponentiate-Mod}? Do you think this is a good approach? For instance, does using this countermeasure imply any disadvantages (versus an alternative say)?

### 12.1.2.4 Take a random number of steps

Another approach would be to force \textsc{Match-Pwd} to take a random number of steps no matter what values of \( P \) and \( G \) are given to it. The end result is similar: the information previously leaked is now masked, meaning the attacker cannot use the same strategy.

Figure 12.3d shows that only a simple alteration is required (although, in reality, we also need a way to generate suitable random numbers). The main algorithm is the same, so the matching process is the same. However, there is an additional line before we start: by waiting a random amount of time, the actual number of steps observed by the attacker is randomised. For example, imagine we have \( P = \text{"joshua"} \) and make the two guesses \( G = \text{"daniel"} \) and \( G' = \text{"joanne"} \); if the algorithm takes 8 steps, which guess did this come from? It could be \( G \) if the algorithm waits for \( r = 6 \) steps initially, then takes 2 steps in the matching phase; on the other hand, it could be \( G' \) if the algorithm waits for \( r = 4 \) steps initially, then takes 4 steps in the matching phase. The point is that we cannot relate the number of steps \textit{observed} to the value of \( P \) being used; again, we have prevented the use of execution time as a useful side-channel.

Is this a \textit{robust} countermeasure though? We could try to filter out the randomness using elementary statistics. Imagine we take our two guesses and use them each 100 times, capturing the results. If we find the \textit{average} time taken by attempts with \( G \) and \( G' \), this might tell us something that comparing the time taken by a single attempt could not. For example, if \textit{on average} the number of steps for guess \( G \) is significantly longer than the number of steps for \( G' \) then we might reason there is a good chance that more characters at the start of \( G \) are correct than \( G' \). We cannot say for definite, but by capturing more and more results we can get more and more confident. So we have partially removed the effect of the randomness by simply using enough attempts that a general trend emerges.

### 12.2 Active physical attacks

#### 12.2.1 Attack

Imagine a real scenario; things get a little more technical but we will try to keep the discussion simple. A mobile telephone uses a different type of password to a computer login screen: normally the telephone can be configured to ask for and test a four digit \textbf{Personal Identification Number (PIN)} when turned on [5]. There are not that many different four digit PIN numbers, only 10000 in fact, so one might immediately start to think a brute-force attack could work here. Usually the telephone would be equipped with a countermeasure to prevent such an attack. Do not try this at home, but after a few wrong attempts the telephone usually locks itself and refuses further attempts before you take it to the shop to be unlocked. This prevents a brute-force attack unless the shopkeeper is willing to reactivate a telephone for you thousands of times!

In the previous example, we had a computer with a password \( P \) embedded inside it. Here we have a telephone with a PIN number \( P \) and a counter \( c \) embedded inside it; this is more or less the same as our example countermeasure where we wanted to shut down the computer if there were lots of incorrect password guesses. In this new context, we can be a bit more accurate about the details. Typically the \( P \) and \( c \) values are stored on the SIM card [7] which means their values are retained even if the telephone is turned off. The PIN number plays the same role as the password did: the telephone is supposed to take a guess \( G \),
compare \( P \) and \( G \) and allow access if they are equal. This time we will use an algorithm called \texttt{Check-PIN} detailed in Figure 12.4a.

Again, look at the algorithm one step at a time. The first thing that happens is that the counter \( c \) is read from where it is stored and tested: if \( c = 0 \) we have run out of attempts and the telephone is locked. In this case we return \texttt{false} to indicate that the attempt at accessing the telephone failed. However, if \( c > 0 \) then the telephone is unlocked and we are allowed at least one more attempt. In this case we call the algorithm \texttt{Request-PIN} to read a guess \( G \) from the user and then proceed to check it. Next we check if the guess \( G \) is the same as the real PIN number \( P \) stored in the telephone. If the two are equal then we return \texttt{true} to indicate that the attempt succeeded. If they are not equal then we first subtract one from the counter \( c \) to indicate there is one less incorrect attempt allowed, and then return \texttt{false}.

As the attacker, our goal is to guess \( P \) without the telephone locking itself. Maybe we have "acquired" a telephone from somewhere, or maybe we have forgotten our own PIN number. Unlike the \texttt{Match-Pwd} function, there is not really anything we can infer by timing \texttt{Check-PIN} since the actual test (i.e., \( P \neq G \)) takes the same time no matter what \( P \) and \( G \) are (\( P \) and \( G \) here are numbers, not strings). But imagine that rather than just passively monitoring \texttt{Check-PIN}, we could actively influence how it is executed. This should not sound too amazing, for example we have the telephone in our hand so we could do all sorts of things to it: we might pull out the battery, detach the keypad, or put the telephone in the oven. Of course, the goal is not to destroy the telephone but to aid cryptanalysis, so imagine we are restricted to turning the telephone off mid-execution.

The trick is to turn off the power at just the right point. A good point would be just before line \#9 (which updates \( c \)) gets executed. If we turn off the power before \( c \) is updated then basically it never changes: the new value of \( c \) is not stored on the SIM card. So if we had a way to sound an alarm when line \#9 is going to be executed and then turn off the power whenever the alarm sounds, then \( c \) will never be updated and we have unlimited attempts at guessing \( P \) without the telephone being locked. This turns out to be very feasible. A suitable alarm can be rigged up by monitoring the telephone communicating with the SIM card, both of which we have easy access to. One can imagine an attacker that can now perform a brute-force attack on the telephone:

1. Set the current guess \( G = 0 \).
2. Take \( G \) and enter it into the telephone as a guess at \( P \).
3. Monitor communication between the telephone and the SIM card:
   - If the telephone sends a command to update \( c \), turn off the power then wait for a few seconds and skip forward to line \#4.
(a) Vodafone branded SIM cards.

(b) A Vodafone branded Sony-Ericsson V600i mobile telephone ... 

(c) ... which when opened, exposes the power and SIM card interfaces.

Figure 12.5: Demonstrating viability of the SIM card physical attack.
• If the telephone does not send a command to decrement \(c\) then \(G\) was the correct guess and we have access to the telephone.

4. Add one to \(G\) and go back to line #2.

### 12.2.2 Countermeasures

This is not good news for whoever made and/or sold the telephone: they would claim the PIN number password and associated locking mechanism is offering some level of security, but with only reasonable assumptions we can circumvent that security with some ease. So how might we go about solving the problem? Have a look at the slightly modified version of CHECK-PIN in Figure 12.4b.

At face value the two algorithms are quite similar, but now we alter \(c\) in two places: on line #8 where one is taken away from it, and then in line #10 when one is added to it again if it turns out that the guess was correct. What happens if we set the original attacker to work on the new algorithm? If the alarm sounds just before line #8 is executed we can prevent one being subtracted from \(c\). However \(c\) is always updated at this point regardless of whether \(P = G\) or not, so if we turn off the power when the telephone communicates with the SIM then we can never get access to the telephone: we would turn off the power even if the guess was correct. On the other hand, if the alarm sounds just before line #10 is executed then it is already too late to turn off the power since \(c\) will already have been updated in line #8.

<table>
<thead>
<tr>
<th>Research</th>
<th>Think about the types of electronic device you carry around with you: can you identify any other environmental factors (other than the power supply) that could be manipulated by Eve?</th>
</tr>
</thead>
</table>

227
BIBLIOGRAPHY

Part IV

Advanced
Here is a simple (or at least simple looking) question: if we take two sequences
\[ T = \langle 0, 1 \rangle \]
and
\[ S = \langle 2, 3, 4 \rangle, \]
what happens if we copy all the elements from \( S \) into \( T \)? At face value, this seems like a reasonable thing to do: afterwards we expect the sequences to be equal, i.e., that \( T_0 = S_0 = 2 \), \( T_1 = S_1 = 3 \) and \( T_2 = S_2 = 4 \). However, a problem is lurking. Namely, \( S_0 \), \( S_1 \) and \( S_2 \) are valid elements since \( |S| = 3 \), but \( |T| = 2 \) so \( T_2 \) is not valid. In other words, we would put too much content \[10\] into \( T \). When defining how sequences can be accessed we include an implicit bounds check \[2\] so before each step, we implicitly do something like “if \( 0 \leq i < |T| \) and \( 0 \leq i < |S| \) then copy \( S_i \) into \( T_i \), otherwise cause an error” behind the scenes. So to cut a long story short, an error occurs because trying to assign \( T_2 \) the value 4 is problematic.

Next question: does this match what a computer does when executing a program to copy \( S \) into \( T \)? The answer is no, not really. If you look at the instruction set from Chapter 4, specifically at the few instructions which access memory, the bounds check is missing; the computer just accesses memory without checking the address we give it. Suppose \( T \) and \( S \) are stored in memory somewhere, e.g., something like
\[ \begin{align*}
  i &= \ldots, 3, 4, 5, 6, 7, \ldots \\
  MEM &= \langle \ldots, 0, 1, 2, 3, 4, \ldots \rangle \\
  &= T_0 \ T_1 \ S_0 \ S_1 \ S_2 
\end{align*} \]

If we write a program that tries to store a value in \( MEM_5 \), i.e., what it thinks should be \( T_2 \), something else is overwritten: in this case \( S_0 \). Clearly this is not ideal; the program has a bug \[14\] or mistake in it which will probably causes it to malfunction. That is, it will overwrite some data or an instruction that will be wrong if subsequently loaded. In this case, if we load \( S_0 \) after the copying has finished we end up with 4 rather than 2 because it was overwritten.

One might argue that this is a simple problem to solve: just avoid writing programs with bugs in! This is more tricky than you might think. Large programs are complex and human programmers will make mistakes, so bugs that cause functional problems in a program, i.e., mean it does not do what it should, are irritatingly common. However, the issue is magnified further if the bug implies a security problem, which of course is our focus. Specifically, what if an attacker intentionally tries to find and trigger bugs in our program so they can cause something “bad” to happen, e.g., the program allows access to a file which could not access otherwise? Given the overwriting bug might occur in a program, the real question is can it produce a security problem, and if so how?

Of course the answer to the first part is yes; the rest of this Chapter tries to explain the answer to the second part. Within the context of security, the overwriting bug is more generically called a buffer overflow \[3\], the idea being we overflow (i.e., put too much content into) a buffer (i.e., the memory allocated
for something). The issue of buffer overflows has been described as the “vulnerability of the decade” since they act as an underlying technique that supports all sorts of other security issues. For example, the Morris Worm [9] mentioned in Chapter 4 used a buffer overflow in the fingerd server on UNIX-based computers, among other techniques, to propagate itself. Clearly there are ways to prevent or limit such attacks, but they have remained a clear and present danger for more than twenty years; the cost resulting from the Morris Worm was estimated at up to $100 million for example.

There are a huge range of attacks and techniques that fall under this more general umbrella, but the basic idea is not hard to describe. Suppose we are an attacker, and we want to attack a web-server. A web-server could be privileged [12] in the sense that it can do things a normal program cannot; for example, it might have access to files that a normal program does not. Normally, we would interact with the web-server via a web-browser. Behind the scenes, the web-server and web-browser interact by simply sending commands and responses between each other in the form of strings. Something like “please send me web-page X”, just less polite. Both programs use a data structure to store these strings: if they are written in C, they would use the C-string method from Chapter 5. The basic idea of a buffer overflow attack is for us, i.e., the attacker, to send a large string \( S \) to the web-server: the hope is that it copies the string into some buffer \( T \) which is too small to hold the content, thereby overwriting something else. The tricky part is to make the bug do something useful for us rather than simply crash the web-server. Typically, the idea is that we, i.e., the attacker, fix things up so that the bug causes some other instructions to be executed rather than those representing the web-server program. That is, the bug causes the program to jump somewhere unexpected. Ideally, we end up executing instructions of our choosing so that in short, we force execution of our program rather than the web-server. Of course this is difficult, but if we get everything just right we can execute a program of our choice using the privileges of the web-server. This might mean we can access a web-page we could not access otherwise: all manner of bad things are possible.

### 13.1 From algorithms to sub-routines

Imagine we write two simple algorithms, one of which invokes the other; the algorithms are shown in Figure 13.1. We can easily work out what should happen if we invoke \texttt{MAIN}:

**Step #1** Assign \( a \leftarrow 10 \).

**Step #2** Assign \( b \leftarrow 20 \).

**Step #3** Invoke \texttt{Add}(a, b), i.e., invoke \texttt{Add}(10, 20).

**Step #3.1** Assign \( z \leftarrow x + y \), i.e., assign \( z \leftarrow 10 + 20 = 30 \).

**Step #3.2** Return \( z \), i.e., return 30.

then assign \( c \leftarrow 30 \).

**Step #4** Return.

Basically, \texttt{MAIN} assigns values to \( a \) and \( b \), and then passes them as input to an invocation of \texttt{Add}. The arguments are added together and their sum \( z \) is produced as output returned to \texttt{MAIN}; this is assigned to \( c \). The point is, there are some crucial details missing if we think about similar steps within a corresponding program: exactly how are \( a \) and \( b \) are passed from \texttt{MAIN} to \texttt{Add}, and how is \( z \) returned back to \texttt{MAIN}?
13.1.1 A simple approach to sub-routine calls

13.1.1.1 Attempt #1: a simple starting point

Imagine we implement Main and Add in the program described by Figure 13.2. There are two parts:

1. Main is represented by the instructions held in addresses #0...#11, and uses a, b and c held in addresses #12...#24.

2. Add is represented by the instructions held in addresses #15...#18, and uses x, y and z held in addresses #19...#21.

Each of the parts would be termed a sub-routine [7]. Well, actually we would more usually term them functions but this name makes them sound like Mathematical functions: in Chapter 3 we already described why there might be important differences between the two. So for the sake of avoiding confusion, we stick with sub-routine as a name because either way the idea is the same: the sub-routines capture steps required to complete a specific task somewhat independently from the rest of the program. While algorithms are invoked, sub-routines are called: the implementation of Main would be the caller (i.e., the part that does the calling), Add the callee (i.e., the part that is called).

The program itself is a lot longer than those we have looked at before, but if you read down the right-hand column, the instructions loosely replicate lines of the original algorithms. A step-by-step description of execution is too verbose to include, but here is an abridged version including just the most important aspects:

1. The instructions at addresses #0...#3 act to assign a and b the initial values of 10 and 20.

2. The instructions at addresses #4...#7 copy a and b into x and y, ready for use as arguments by Add; control is then passed to Add when the instruction at address #8 sets PC = 15.

3. The instructions at addresses #15...#17 compute z = x + y = 30; the instruction at address #18 then passes control back to Main by setting PC = 9.

4. The instructions at addresses #9...#10 copy the return value z into c and the program is then halted by the instruction at address #11; the result is that c = 30.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A ← 10</td>
<td>compute a = 10</td>
</tr>
<tr>
<td>1</td>
<td>MEM12 ← A</td>
<td>store a</td>
</tr>
<tr>
<td>2</td>
<td>A ← 20</td>
<td>compute b = 20</td>
</tr>
<tr>
<td>3</td>
<td>MEM13 ← A</td>
<td>store b</td>
</tr>
<tr>
<td>4</td>
<td>A ← MEM12</td>
<td>load a</td>
</tr>
<tr>
<td>5</td>
<td>MEM19 ← A</td>
<td>store x = a</td>
</tr>
<tr>
<td>6</td>
<td>A ← MEM13</td>
<td>load b</td>
</tr>
<tr>
<td>7</td>
<td>MEM20 ← A</td>
<td>store y = b</td>
</tr>
<tr>
<td>8</td>
<td>PC ← 15</td>
<td>call</td>
</tr>
<tr>
<td>9</td>
<td>A ← MEM21</td>
<td>load z</td>
</tr>
<tr>
<td>10</td>
<td>MEM14 ← A</td>
<td>store c = z</td>
</tr>
<tr>
<td>11</td>
<td>HALT</td>
<td>halt</td>
</tr>
<tr>
<td>12</td>
<td>NOP a</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>NOP b</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>NOP c</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>A ← MEM19</td>
<td>load x</td>
</tr>
<tr>
<td>16</td>
<td>A ← A + MEM20</td>
<td>compute z = x + y</td>
</tr>
<tr>
<td>17</td>
<td>MEM21 ← A</td>
<td>store z</td>
</tr>
<tr>
<td>18</td>
<td>PC ← 9</td>
<td>return</td>
</tr>
<tr>
<td>19</td>
<td>NOP x</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>NOP y</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>NOP z</td>
<td></td>
</tr>
</tbody>
</table>

Figure 13.2: Attempt #1 at implementing Main and Add.
1 algorithm Main begin
2    \( a \leftarrow 10 \)
3    \( b \leftarrow 20 \)
4    \( c \leftarrow \text{Add}(a, b) \)
5    \( d \leftarrow 30 \)
6    \( e \leftarrow 40 \)
7    \( f \leftarrow \text{Add}(d, e) \)
8    return
9 end

Figure 13.3: An altered Main that now invokes Add twice rather than once.

1 algorithm Fibonacci-Iterative\((n)\) begin
2    \( i \leftarrow 1 \)
3    \( p \leftarrow 0 \)
4    \( q \leftarrow 1 \)
5    while \( i < n \) do
6        \( i \leftarrow i + 1 \)
7        \( t \leftarrow p + q \)
8        \( p \leftarrow q \)
9        \( q \leftarrow t \)
10 end
11 return \( q \)
12 end

1 algorithm Fibonacci-Recursive\((n)\) begin
1    if \( n = 0 \) then
2        return 0
3    end
4    if \( n = 1 \) then
5        return 1
6    end
7    \( p \leftarrow \text{Fibonacci-Recursive}(n - 1) \)
8    \( q \leftarrow \text{Fibonacci-Recursive}(n - 2) \)
9    return \( p + q \)
10 end

(a) An iterative approach.

(b) A recursive approach.

Figure 13.4: Two algorithms to compute the \( n \)-th Fibonacci number.

13.1.1.2 Problem #1: multiple callers

Algorithms can be invoked from more than one place. In a sense, this is also one of the main attractions of using sub-routines in a program: they allow us to describe how to perform some task once, and then use that description as many times as we like. Clearly this is more attractive than writing out the same lines every time we need to use them.

Imagine we take our original Main algorithm and alter it slightly to produce Figure 13.3; the crucial difference is that it now invokes Add twice to compute \( a + b \) and then \( d + e \). This highlights a problem: if we follow the same implementation strategy as with the original Main and Add, we will get confused when returning from Add. How do we know where to return to? In our previous implementation, we fixed the address of where the next instruction in Main should be after we finish Add; now there are two “next instructions”. Clearly this is a problem that needs to be solved, preferably without having to write two different implementations of Add since that defeats the purpose of reusing one description.

13.1.1.3 Problem #2: local variables

The concept of recursion [13] is commonplace in nature, and a great analogy is provided by the Droste effect [5]: basically this is where an image contains a smaller instance of itself, with the name taken from a Dutch brand of hot chocolate mixture whose packaging is shown in Figure 13.5.

By replacing “image” with “algorithm” the concept still makes sense in that a recursive algorithm is one that contains invocations of itself. At a high level, the idea is to divide a problem into smaller sub-problems of the same type, then solve the sub-problems: the smaller solutions can then be combined to solve the original problem. Since the sub-problems will be solved using the same algorithm as the original, recursive algorithms tend to be very concise and admired for their elegance. However, despite their advantages, the concept can be hard to grasp: an example will make this a lot easier. Consider computation of the Fibonacci
Figure 13.5: The Droste effect whereby an image contains a smaller instance of itself (public domain image, source: http://en.wikipedia.org/wiki/File:Droste.jpg)
numbers [6] which form a sequence defined by

\[
F_0 = 0 \\
F_1 = 1 \\
F_i = F_{i-1} + F_{i-2}
\]

That is, the first two numbers in the sequence are fixed and the rest we compute out: each \(i\)-th element in the sequence is just the sum of the previous two. So for example

\[
F_2 = F_1 + F_0 = 1 + 0 = 1 \\
F_3 = F_2 + F_1 = 1 + 1 = 2 \\
F_4 = F_3 + F_2 = 2 + 1 = 3
\]

... ... ...

Put another way, to compute \(F_i\) we invoke the same algorithm to solve the smaller sub-computations \(F_{i-1}\) and \(F_{i-2}\) then combine the solutions by adding them together. That should sound sensible, but what does an algorithm to compute the \(n\)-th Fibonacci number look like?

1. We could adopt an iterative approach: this basically means that we write a loop that computes successive Fibonacci numbers until it finds the one we want. The algorithm FIBONACCI-ITERATIVE in Figure 13.4a uses this approach; imagine we invoke it with \(n = 3\):

   - **Step #1** Assign \(i \leftarrow 1, p \leftarrow 0, q \leftarrow 1\).
   - **Step #2** Since \(i < n\), perform the next loop iteration.
   - **Step #3** Assign \(i \leftarrow i + 1 = 2, t \leftarrow p + q = 0 + 1 = 1, p \leftarrow q = 1, q \leftarrow t = 1\).
   - **Step #4** Since \(i < n\), perform the next loop iteration.
   - **Step #5** Assign \(i \leftarrow i + 1 = 3, t \leftarrow p + q = 1 + 1 = 2, p \leftarrow q = 1, q \leftarrow t = 2\).
   - **Step #6** Since \(i = n\), stop the loop.
   - **Step #7** Return \(q\), i.e., return 2.

2. We could adopt an recursive approach: this basically means that we write an algorithm that invokes itself to solve small sub-problems (a recursive case) until it finds one where the solution can be returned trivially (a base case). FIBONACCI-RECURSIVE in Figure 13.4b uses this approach; imagine we invoke it with \(n = 3\):

   - **Step #1** Since \(n \neq 0\) and \(n \neq 1\), continue.
   - **Step #2** Invoke FIBONACCI-RECURSIVE\((n - 1)\), i.e., invoke FIBONACCI-RECURSIVE(2).
     - **Step #2.1** Since \(n \neq 0\) and \(n \neq 1\), continue.
     - **Step #2.2** Invoke FIBONACCI-RECURSIVE\((n - 1)\), i.e., invoke FIBONACCI-RECURSIVE(1).
       - **Step #2.2.1** Since \(n = 1\), return 1.
         and assign \(p \leftarrow 1\)
     - **Step #2.3** Invoke FIBONACCI-RECURSIVE\((n - 2)\), i.e., invoke FIBONACCI-RECURSIVE(0).
       - **Step #2.3.1** Since \(n = 0\), return 0.
         and assign \(q \leftarrow 0\)
     - **Step #2.4** Return \(p + q\), i.e., return \(1 + 0 = 1\).
         and assign \(p \leftarrow 1\).
   - **Step #3** Invoke FIBONACCI-RECURSIVE\((n - 2)\), i.e., invoke FIBONACCI-RECURSIVE(1).
     - **Step #3.1** Since \(n = 1\), return 1.
       and assign \(q \leftarrow 1\).
   - **Step #4** Return \(p + q\), i.e., return \(1 + 1 = 2\).
It makes sense to explore the concept of recursion by writing your own recursive algorithms. We have already seen two iterative candidates that could be translated into a recursive alternative:

1. an algorithm for finding the length of a string, e.g., C-STRING-LENGTH from Chapter 5, or
2. an algorithm for performing exponentiation, e.g., EXPONENTIATE-HORNER from Chapter 3.

Try to write the recursive alternatives, and show how they work using the same example inputs as originally.

Since both approaches compute the same result, selecting between them could be viewed as a matter of taste. Ignoring this however, imagine we write a program that uses recursion; in particular, imagine the program corresponds to FIBONACCI-RECURSIVE. In the algorithm we can only perform one step at a time; the same is true for the program. But, in a sense, more than one instance of FIBONACCI-RECURSIVE is active at a given step: looking at the example, step #2.1 is a step in an invocation of FIBONACCI-RECURSIVE nested inside another one, and another one again! Each active instance has some state we need to keep track of. We need to remember that the value of \( p \) assigned to by step #2 does not conflict with that resulting from step #2: each \( p \) is “owned” by the associated instance of FIBONACCI-RECURSIVE.

This discussion of how the algorithms behave highlights a problem: if we follow the same implementation strategy as we did previously with MAIN and ADD, we will confuse the values of \( p \). In Figure 13.2 we only have one place for \( z \); the analogy of \( z \) here is \( p \), so basically one instance of the sub-routine would overwrite the \( p \) owned by another. Clearly this is another problem that needs a solution.

13.2 Constructing an improved sub-routine call mechanism

13.2.1 A stack data structure

A stack [15] is an important data structure in Computer Science. The easiest way to think about a stack is as a “container” that organises data in a certain way: we can add items to the stack and remove items from it, and the stack basically keeps track of where all the items are. Sometimes a stack is called a First-In Last-Out (FILO), but another way to say the same thing would be to call it a Last-In First-Out (LIFO): the idea is that we can “push” (or add) items onto the stack, but when we “pop” (or remove) items off the stack we get the data on the Top of Stack (ToS), i.e., the last item pushed. To get the required behaviour we need three components, namely

1. an address, called the Stack Pointer (SP), that keeps track of where the ToS is,
2. an algorithm called Push that takes some item \( x \) and adds it to the stack, and
3. an algorithm called Pop that removes an item and returns it as a result.

Based on the fact that \( SP \) is just a number we keep somewhere, the two algorithms we need are actually quite simple:

```
algorithm Push(x) begin
  MEM_{SP} ← x
  SP ← SP − 1
return
end
```

```
algorithm Pop begin
  SP ← SP + 1
return MEM_{SP}
end
```

To reinforce what is going on, consider a simple example where we have the following initial memory content and set \( SP = 7 \):

\[
\begin{align*}
SP &= \ldots, 3, 4, 5, 6, 7, \ldots \\
i &= \ldots, 0, 0, 0, 0, \ldots \\
MEM &= \langle \ldots, 0, 0, 0, 0, \ldots \rangle
\end{align*}
\]

Notice that we have marked where the current value of \( SP \) says the ToS is to make things easier to read, and that stacks traditionally grow downwards as items are added (i.e., from higher address to lower addresses)
so we follow this. By invoking Push(104), our intention is to add 104 to the stack; the behaviour of the algorithm is simple

**Step #1** Assign $MEM_7 \leftarrow 104$.

**Step #2** Assign $SP \leftarrow SP - 1 = 6$.

After it terminates, we end up with $SP = 6$ and the memory content

$$SP \quad i = \quad \ldots, \ 3, \ 4, \ 5, \ 6, \ 7, \ \ldots $$

$$MEM = \langle \ldots, \ 0, \ 0, \ 0, \ 0, \ 104, \ \ldots \rangle$$

We can carry on pushing more items if we want; for example, after two more invocations Push(101) and Push(108) we end up with $SP = 4$ and the memory content

$$SP \quad i = \quad \ldots, \ 3, \ 4, \ 5, \ 6, \ 7, \ \ldots $$

$$MEM = \langle \ldots, \ 0, \ 0, \ 108, \ 101, \ 104, \ \ldots \rangle$$

What happens if we now try to remove an item? If we invoke Pop then our intention is to remove (i.e., retrieve) the last item we pushed. The behaviour we get is again simple

**Step #1** Assign $SP \leftarrow SP + 1 = 5$.

**Step #2** Return $MEM_5$, i.e., return 108.

and we get what we expected: 108 was the last item previously pushed. After the algorithm terminates we end up with $SP = 5$ and the memory content

$$SP \quad i = \quad \ldots, \ 3, \ 4, \ 5, \ 6, \ 7, \ \ldots $$

$$MEM = \langle \ldots, \ 0, \ 0, \ 108, \ 101, \ 104, \ \ldots \rangle$$

Notice that we have not deleted 108: it is still there in memory. All that has happened is that by updating $SP$, the organisation now excludes whatever is in $MEM_5$ from being part of the stack data structure.

Going back to the topic of sub-routines, our approach to solving the problems we identified will be to use a stack just like this one. To illustrate the concept, imagine trying to pass an argument from $MAIN$ to $ADD$: we just make $MAIN$ push the argument onto the stack, and then make $ADD$ pop it off again. Employing a more diagrammatic description, invoking $MAIN$ would produce something like the following behaviour:

<table>
<thead>
<tr>
<th>MAIN</th>
<th>ADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step #1</td>
<td>Assign $a \leftarrow 10$</td>
</tr>
<tr>
<td>Step #2</td>
<td>Assign $b \leftarrow 20$</td>
</tr>
<tr>
<td>Step #3</td>
<td>Invoke Push($b$)</td>
</tr>
<tr>
<td>Step #4</td>
<td>Invoke Push($a$)</td>
</tr>
<tr>
<td>Step #5</td>
<td>Pass control to $ADD$ →</td>
</tr>
<tr>
<td>Step #6</td>
<td>Assign $x \leftarrow$ Pop()</td>
</tr>
<tr>
<td>Step #7</td>
<td>Assign $y \leftarrow$ Pop()</td>
</tr>
<tr>
<td>Step #8</td>
<td>Invoke Push($x + y$)</td>
</tr>
<tr>
<td>Step #9</td>
<td>← Pass control to $MAIN$</td>
</tr>
<tr>
<td>Step #10</td>
<td>Assign $c \leftarrow$ Pop()</td>
</tr>
</tbody>
</table>

Notice that because of the way the stack works, we push $b$ then $a$ (i.e., sort of in reverse) so that when we pop $x$ then $y$ in $MAIN$, they get assigned the right values. To prove this really works, we need to look at the gruesome detail of how the algorithms progress. Imagine we reset $SP = 7$ and the memory content to

$$SP \quad i = \quad \ldots, \ 3, \ 4, \ 5, \ 6, \ 7, \ \ldots $$

$$MEM = \langle \ldots, \ 0, \ 0, \ 0, \ 0, \ 0, \ \ldots \rangle$$

If we invoke $MAIN$ then the resulting behaviour modelled by the diagram above is given by the following:
Step #1 Assign $a \leftarrow 10$.

Step #2 Assign $b \leftarrow 20$.

Step #3 Invoke $\text{Push}(b)$, i.e., invoke $\text{Push}(20)$.
  
  Step #3.1 Assign $\text{MEM}_7 \leftarrow 20$.
  
  Step #3.2 Assign $SP \leftarrow SP - 1 = 6$.

Step #4 Invoke $\text{Push}(a)$, i.e., invoke $\text{Push}(10)$.
  
  Step #4.1 Assign $\text{MEM}_6 \leftarrow 10$.
  
  Step #4.2 Assign $SP \leftarrow SP - 1 = 5$.

Step #5 Pass control to $\text{Add}$.
  
  Step #5.1 Invoke $\text{Pop}()$.
    
    Step #5.1.1 Assign $SP \leftarrow SP + 1 = 6$.
    
    Step #5.1.2 Return $\text{MEM}_6$, i.e., return 10.
    
    then assign $x \leftarrow 10$.
  
  Step #5.2 Invoke $\text{Pop}()$.
    
    Step #5.2.1 Assign $SP \leftarrow SP + 1 = 7$.
    
    Step #5.2.2 Return $\text{MEM}_7$, i.e., return 20.
    
    then assign $y \leftarrow 20$.
  
  Step #5.3 Assign $z \leftarrow x + y$, i.e., assign $z \leftarrow 10 + 20 = 30$.
  
  Step #5.4 Invoke $\text{Push}(z)$, i.e., invoke $\text{Push}(30)$.
    
    Step #5.4.1 Assign $\text{MEM}_7 \leftarrow 30$.
    
    Step #5.4.2 Assign $SP \leftarrow SP - 1 = 6$.
  
  Step #5.5 Pass control to $\text{Main}$.

Step #6 Invoke $\text{Pop}()$.
  
  Step #6.1 Assign $SP \leftarrow SP + 1 = 7$.
  
  Step #6.2 Return $\text{MEM}_7$, i.e., return 30.
  
  then assign $c \leftarrow 30$.

On one hand this description is quite long and therefore not so nice to read through, but on the other hand there is no longer any missing detail: the advantage we now have is that provided the stack exists, there is no hidden mystery behind the process of $\text{Main}$ invoking $\text{Add}$.

### 13.2.2 Adding a stack to the example computer

To support the approach described above, we need a minor upgrade to the computer. None of the changes are particularly radical, with the main goal being addition of instructions to manage and access stack content:

1. We add some new instructions that allow us to move $PC$ into $A$ and vice versa:
   
   - $43nnnn$ means $PC \leftarrow A$.
   - $44nnnn$ means $A \leftarrow PC$.

   Before this upgrade, we could only set $PC$ to fixed addresses via an instruction of the form

   $$PC \leftarrow n.$$ 

   Now we can compute an address in $A$ during execution of the program, and set $PC$ to this new value.

2. We introduce another accumulator called $SP$ which represents the stack pointer; when the computer is reset, we assume $SP$ is set to the highest address in memory (in the same way $PC$ is set to 0, the lowest address).
3. We introduce two new instructions:

- **50nnnn** means \( \text{MEM}\_\text{SP} \leftarrow A, \text{SP} \leftarrow \text{SP} - 1 \).
- **51nnnn** means \( \text{SP} \leftarrow \text{SP} + 1, A \leftarrow \text{MEM}\_\text{SP} \).

which essentially to the same thing as the Push and Pop algorithms: they put the current value of \( A \) onto the ToS and set the value of \( A \) to that from the ToS, updating \( SP \) accordingly.

4. We add some new instructions that allow us to use \( SP \) in a wider variety of ways, mainly in line with how we already use \( A \):

- **52nnnn** means \( \text{MEM}\_\text{SP} + n \leftarrow A \).
- **53nnnn** means \( A \leftarrow \text{MEM}\_\text{SP} + n \).
- **54nnnn** means \( A \leftarrow A + \text{MEM}\_\text{SP} + n \).
- **55nnnn** means \( A \leftarrow A - \text{MEM}\_\text{SP} + n \).
- **56nnnn** means \( A \leftarrow A \oplus \text{MEM}\_\text{SP} + n \).
- **57nnnn** means \( \text{SP} \leftarrow \text{SP} + n \).
- **58nnnn** means \( \text{SP} \leftarrow \text{SP} - n \).

For example, now we can perform loads and stores relative to the current value of \( SP \), i.e., directly access the stack content. You can think of this as bending the rules a little: with these new instructions, we are not strictly limited to accessing the ToS only.

### 13.2.3 A stack-based approach to sub-routine calls

Armed with a more capable computer, improving on our original strategy for sub-routine calls is fairly straightforward. The basic idea is that each time we call a sub-routine, we create some space for it on the stack called a stack frame [4] (or sometimes an activation record); each time a sub-routine returns, we remove the associated stack frame. The stack frame itself is home to various items of data, for example

1. a Return Address (RA), i.e., where the sub-routine should return to once it finishes,
2. any incoming arguments and outgoing Return Value (RV), and
3. any local variables, i.e., variables it “owns”.

We takes things step-by-step, starting with the original program (i.e., attempt #1) and making a series of small changes to end up with a better program; each step essentially adds one of the items above to the stack frame, each addition makes things a little more complicated but eventually solve the problems we started o ff with.

#### 13.2.3.1 Attempt #2: using return addresses

The first change, detailed in Figure 13.6, is fairly modest: the idea is simply to adopt the idea of computing and using a return address. The program is still in two parts, with a third part representing the stack:

1. **Main** is represented by the instructions held in addresses #0 ... #14, and uses \( a, b \) and \( c \) held in addresses #15 ... #17. A constant called \( o f f \) is held in address #18: this allows computation of the return address, representing the offset from where we copy the value of PC to where we should return.
2. **Add** is represented by the instructions held in addresses #19 ... #23, and uses \( x, y \) and \( z \) held in addresses #24 ... #26.
3. The stack occupies addresses #27 ... #34; remember that it grows downward in memory, and that \( SP = 34 \) initially.

Notice the difference from the previous attempt: there we had a fixed return address (Add set \( PC = 9 \)), but here we use the return address passed on the stack (Add sets \( PC \) equal to a value popped off the stack). This essentially solves the first problem we identified; Add can be called from wherever we want because as long as we push the right return address onto the stack, it will always know where to return. Adopting a similar abridged description as before, execution of the program can be described as follows:
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 200010</td>
<td>A ← 10</td>
<td>compute $a = 10$</td>
</tr>
<tr>
<td>1 210015</td>
<td>MEM$_{15}$ ← A</td>
<td>store $a$</td>
</tr>
<tr>
<td>2 200020</td>
<td>A ← 20</td>
<td>compute $b = 20$</td>
</tr>
<tr>
<td>3 210016</td>
<td>MEM$_{16}$ ← A</td>
<td>store $b$</td>
</tr>
<tr>
<td>4 440000</td>
<td>A ← PC</td>
<td>copy PC</td>
</tr>
<tr>
<td>5 300018</td>
<td>A ← A + MEM$_{18}$</td>
<td>compute RA = PC + off</td>
</tr>
<tr>
<td>6 500000</td>
<td>MEM$_{SP}$ ← A, SP ← SP − 1</td>
<td>push RA</td>
</tr>
<tr>
<td>7 220015</td>
<td>A ← MEM$_{15}$</td>
<td>load $a$</td>
</tr>
<tr>
<td>8 210024</td>
<td>MEM$_{24}$ ← A</td>
<td>store $x = a$</td>
</tr>
<tr>
<td>9 220016</td>
<td>A ← MEM$_{16}$</td>
<td>load $b$</td>
</tr>
<tr>
<td>10 210025</td>
<td>MEM$_{25}$ ← A</td>
<td>store $y = b$</td>
</tr>
<tr>
<td>11 400019</td>
<td>PC ← 19</td>
<td>call</td>
</tr>
<tr>
<td>12 220026</td>
<td>A ← MEM$_{26}$</td>
<td>load $z$</td>
</tr>
<tr>
<td>13 210017</td>
<td>MEM$_{17}$ ← A</td>
<td>store $c = z$</td>
</tr>
<tr>
<td>14 100000</td>
<td>HALT</td>
<td>halt</td>
</tr>
<tr>
<td>15 000000</td>
<td>NOP</td>
<td>a</td>
</tr>
<tr>
<td>16 000000</td>
<td>NOP</td>
<td>b</td>
</tr>
<tr>
<td>17 000000</td>
<td>NOP</td>
<td>c</td>
</tr>
<tr>
<td>18 000007</td>
<td>NOP</td>
<td>off</td>
</tr>
<tr>
<td>19 220024</td>
<td>A ← MEM$_{24}$</td>
<td>load $x$</td>
</tr>
<tr>
<td>20 300025</td>
<td>A ← A + MEM$_{25}$</td>
<td>compute $z = x + y$</td>
</tr>
<tr>
<td>21 210026</td>
<td>MEM$_{26}$ ← A</td>
<td>store $z$</td>
</tr>
<tr>
<td>22 510000</td>
<td>SP ← SP + 1, A ← MEM$_{SP}$</td>
<td>pop RA</td>
</tr>
<tr>
<td>23 430000</td>
<td>PC ← A</td>
<td>return</td>
</tr>
<tr>
<td>24 000000</td>
<td>NOP</td>
<td>x</td>
</tr>
<tr>
<td>25 000000</td>
<td>NOP</td>
<td>y</td>
</tr>
<tr>
<td>26 000000</td>
<td>NOP</td>
<td>z</td>
</tr>
<tr>
<td>27 000000</td>
<td>NOP</td>
<td>stack</td>
</tr>
<tr>
<td>28 000000</td>
<td>NOP</td>
<td>stack</td>
</tr>
<tr>
<td>29 000000</td>
<td>NOP</td>
<td>stack</td>
</tr>
<tr>
<td>30 000000</td>
<td>NOP</td>
<td>stack</td>
</tr>
<tr>
<td>31 000000</td>
<td>NOP</td>
<td>stack</td>
</tr>
<tr>
<td>32 000000</td>
<td>NOP</td>
<td>stack</td>
</tr>
<tr>
<td>33 000000</td>
<td>NOP</td>
<td>stack</td>
</tr>
<tr>
<td>34 000000</td>
<td>NOP</td>
<td>stack (initial ToS)</td>
</tr>
</tbody>
</table>

**Figure 13.6:** Attempt #2 at implementing `main` and `add`. 
• Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:

\[
\begin{align*}
SP_i &= \ldots, 27, 28, 29, 30, 31, 32, 33, 34, \\
MEM &= \langle \ldots, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ldots \rangle
\end{align*}
\]

• Execution starts at address #0: **Main** first initialises \( a \) and \( b \) (addresses #0…#3). It then computes and pushes the return address onto the stack (addresses #4…#6): the instructions copy the value of \( PC \), the add an offset to it giving the address #12, i.e., just after the instruction that performs the call to **Add**. Next it copies \( a \) and \( b \) into \( x \) and \( y \) (addresses #7…#10).

**Main** finally passes control to **Add** by setting \( PC = 19 \) (address #11). At this point, the memory content is as follows:

\[
\begin{align*}
SP_i &= \ldots, 27, 28, 29, 30, 31, 32, 33, 34, \\
MEM &= \langle \ldots, 0, 0, 0, 0, 0, 0, 0, 0, 12, \ldots \rangle
\end{align*}
\]

• **Add** computes \( z = x + y \) (addresses #19…#21) and then pops the return address from the stack. Finally, it passes control to **Main** by copying the return address into \( PC \). At this point, the memory content is as follows:

\[
\begin{align*}
SP_i &= \ldots, 27, 28, 29, 30, 31, 32, 33, 34, \\
MEM &= \langle \ldots, 0, 0, 0, 0, 0, 0, 0, 0, 12, \ldots \rangle
\end{align*}
\]

• **Main** concludes execution by copying \( z \) into \( c \) (addresses #12…#13), then halting (address #14).

### 13.2.3.2 Attempt #3: passing arguments, returning values

The next change, detailed in Figure 13.7, is again fairly modest: since we already use the stack to house a return address, it makes sense to use it for the arguments and return value associated with the call to **Add**.

The program is still in two parts, with a third part again representing the stack:

1. **Main** is represented by the instructions held in addresses #0…#14, and uses \( a, b \) and \( c \) held in addresses #15…#17; the constant \( off \) is held in address #18.

2. **Add** is represented by the instructions held in addresses #19…#29, and uses \( x \) and \( y \) held in addresses #30…#31. Notice that we no longer need any space for \( z \) (since we use the stack to hold it) but do need some space (address #32) to temporarily hold the return address.

3. The stack occupies addresses #33…#40; remember that it grows downward in memory, and that \( SP = 40 \) initially.

The program execution can be described as follows:

• Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:

\[
\begin{align*}
SP_i &= \ldots, 33, 34, 35, 36, 37, 38, 39, 40, \\
MEM &= \langle \ldots, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ldots \rangle
\end{align*}
\]

• Execution starts at address #0: **Main** first initialises \( a \) and \( b \) (addresses #0…#3). It then computes and pushes the return address, \( a \) and \( b \) onto the stack (addresses #4…#10).

**Main** finally passes control to **Add** by setting \( PC = 19 \) (address #11). At this point, the memory content is as follows:

\[
\begin{align*}
SP_i &= \ldots, 33, 34, 35, 36, 37, 38, 39, 40, \\
MEM &= \langle \ldots, 0, 0, 0, 0, 0, 0, 0, 0, 12, \ldots \rangle
\end{align*}
\]

\[
= \langle y, x, RA \rangle
\]
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$A \leftarrow 10$</td>
<td>compute $a = 10$</td>
</tr>
<tr>
<td>1</td>
<td>$\text{MEM}_{15} \leftarrow A$</td>
<td>store $a$</td>
</tr>
<tr>
<td>2</td>
<td>$A \leftarrow 20$</td>
<td>compute $b = 20$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{MEM}_{16} \leftarrow A$</td>
<td>store $b$</td>
</tr>
<tr>
<td>4</td>
<td>$A \leftarrow \text{PC}$</td>
<td>copy PC</td>
</tr>
<tr>
<td>5</td>
<td>$\text{MEM}<em>{18} \leftarrow A + \text{MEM}</em>{18}$</td>
<td>compute $RA = \text{PC} + \text{off}$</td>
</tr>
<tr>
<td>6</td>
<td>$\text{MEM}_{19} \leftarrow A, \text{SP} \leftarrow \text{SP} - 1$</td>
<td>push $RA$</td>
</tr>
<tr>
<td>7</td>
<td>$A \leftarrow \text{MEM}_{15}$</td>
<td>load $a$</td>
</tr>
<tr>
<td>8</td>
<td>$\text{MEM}_{16} \leftarrow A, \text{SP} \leftarrow \text{SP} - 1$</td>
<td>push $a$</td>
</tr>
<tr>
<td>9</td>
<td>$A \leftarrow \text{MEM}_{16}$</td>
<td>load $b$</td>
</tr>
<tr>
<td>10</td>
<td>$\text{MEM}_{17} \leftarrow A, \text{SP} \leftarrow \text{SP} - 1$</td>
<td>push $b$</td>
</tr>
<tr>
<td>11</td>
<td>$\text{PC} \leftarrow 19$</td>
<td>call</td>
</tr>
<tr>
<td>12</td>
<td>$\text{SP} \leftarrow \text{SP} + 1, A \leftarrow \text{MEM}_{18}$</td>
<td>pop $c$</td>
</tr>
<tr>
<td>13</td>
<td>$\text{MEM}_{19} \leftarrow A$</td>
<td>store $c$</td>
</tr>
<tr>
<td>14</td>
<td>$\text{HALT}$</td>
<td>halt</td>
</tr>
<tr>
<td>15</td>
<td>$\text{NOP}$</td>
<td>$a$</td>
</tr>
<tr>
<td>16</td>
<td>$\text{NOP}$</td>
<td>$b$</td>
</tr>
<tr>
<td>17</td>
<td>$\text{NOP}$</td>
<td>$c$</td>
</tr>
<tr>
<td>18</td>
<td>$\text{NOP}$</td>
<td>$\text{off}$</td>
</tr>
<tr>
<td>19</td>
<td>$\text{SP} \leftarrow \text{SP} + 1, A \leftarrow \text{MEM}_{19}$</td>
<td>pop $b$</td>
</tr>
<tr>
<td>20</td>
<td>$\text{MEM}_{31} \leftarrow A$</td>
<td>store $y$</td>
</tr>
<tr>
<td>21</td>
<td>$\text{SP} \leftarrow \text{SP} + 1, A \leftarrow \text{MEM}_{18}$</td>
<td>pop $a$</td>
</tr>
<tr>
<td>22</td>
<td>$\text{MEM}_{30} \leftarrow A$</td>
<td>store $x$</td>
</tr>
<tr>
<td>23</td>
<td>$\text{SP} \leftarrow \text{SP} + 1, A \leftarrow \text{MEM}_{17}$</td>
<td>pop $RA$</td>
</tr>
<tr>
<td>24</td>
<td>$\text{MEM}_{32} \leftarrow A$</td>
<td>store $RA$</td>
</tr>
<tr>
<td>25</td>
<td>$A \leftarrow \text{MEM}_{30}$</td>
<td>load $x$</td>
</tr>
<tr>
<td>26</td>
<td>$\text{MEM}<em>{31} \leftarrow A + \text{MEM}</em>{33}$</td>
<td>compute $z = x + y$</td>
</tr>
<tr>
<td>27</td>
<td>$\text{MEM}_{19} \leftarrow A, \text{SP} \leftarrow \text{SP} - 1$</td>
<td>push $z$</td>
</tr>
<tr>
<td>28</td>
<td>$A \leftarrow \text{MEM}_{32}$</td>
<td>load $RA$</td>
</tr>
<tr>
<td>29</td>
<td>$\text{PC} \leftarrow A$</td>
<td>return</td>
</tr>
<tr>
<td>30</td>
<td>$\text{NOP}$</td>
<td>$x$</td>
</tr>
<tr>
<td>31</td>
<td>$\text{NOP}$</td>
<td>$y$</td>
</tr>
<tr>
<td>32</td>
<td>$\text{NOP}$</td>
<td>$RA$</td>
</tr>
<tr>
<td>33</td>
<td>$\text{NOP}$</td>
<td>stack</td>
</tr>
<tr>
<td>34</td>
<td>$\text{NOP}$</td>
<td>stack</td>
</tr>
<tr>
<td>35</td>
<td>$\text{NOP}$</td>
<td>stack</td>
</tr>
<tr>
<td>36</td>
<td>$\text{NOP}$</td>
<td>stack</td>
</tr>
<tr>
<td>37</td>
<td>$\text{NOP}$</td>
<td>stack</td>
</tr>
<tr>
<td>38</td>
<td>$\text{NOP}$</td>
<td>stack</td>
</tr>
<tr>
<td>39</td>
<td>$\text{NOP}$</td>
<td>stack</td>
</tr>
<tr>
<td>40</td>
<td>$\text{NOP}$</td>
<td>stack (initial ToS)</td>
</tr>
</tbody>
</table>

Figure 13.7: Attempt #3 at implementing $\text{MAIN}$ and $\text{ADD}$. 
• **Add** first pops \(b, a\) and the return address off the stack and stores them (addresses \#19...\#24). It then computes \(x + y\), pushing the result onto the stack rather than storing it in \(z\) as previously (addresses \#25...\#27). Finally, it passes control to **Main** by loading the return address (previously popped off the stack and stored) and copying it into PC. At this point, the memory content is as follows:

\[
\begin{align*}
SP &= \ldots, 33, 34, 35, 36, 37, 38, 39, 40, \ldots \\
\downarrow & \\
MEM &= \langle \ldots, 0, 0, 0, 0, 0, 20, 10, 30, \ldots \rangle \\
&= z
\end{align*}
\]

• **Main** concludes execution by popping \(z\) off the stack and storing it into \(c\) (addresses \#12...\#13). At this point, the memory content is as follows:

\[
\begin{align*}
SP &= \ldots, 33, 34, 35, 36, 37, 38, 39, 40, \ldots \\
\downarrow & \\
MEM &= \langle \ldots, 0, 0, 0, 0, 0, 20, 10, 30, \ldots \rangle \\
&= z
\end{align*}
\]

It then halts (address \#14).

In a sense, the end result segregates **Main** and **Add** more than previously. No longer does **Main** access the local variables “owned” by **Add**, instead the two communicate with each other using the stack.

### 13.2.3.3 Attempt #4: avoiding individual pushes and pops

The length of (i.e., number of instructions in) the previous attempt is annoying. One of the causes is the large number of pushes and pops. The first thing **Add** does for example is pop \(x\) and \(y\) since it has nowhere to keep them, it then has to store them back into memory which seems like a waste of time. The idea of our next attempt is to combine together individual pushes and pops, and to totally avoid them wherever possible.

For example, to push multiple items onto the stack the idea is to first update \(SP\) to make space for all the items in one go, then store the items relative to the new value of \(SP\); this contrasts with updating \(SP\) for each item. Likewise instead of popping items off the stack and then having to store them somewhere, we access the items on the stack using the appropriate load and store instructions. Just as before, the program is in two parts with a third part representing the stack:

1. **Main** is represented by the instructions held in addresses \#0...\#16, and uses \(a, b\) and \(c\) held in addresses \#17...\#19; the constant \(f\) is held in address \#20.
2. **Add** is represented by the instructions held in addresses \#21...\#25. Notice that we no longer need any space for \(x, y\) nor \(z\) since they are all held on the stack.
3. The stack occupies addresses \#26...\#33; remember that it grows downward in memory, and that \(SP = 33\) initially.

The program execution can be described as follows:

• Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:

\[
\begin{align*}
SP &= \ldots, 26, 27, 28, 29, 30, 31, 32, 33, \ldots \\
\downarrow & \\
MEM &= \langle \ldots, 0, 0, 0, 0, 0, 0, 0, 0, \ldots \rangle
\end{align*}
\]

• Execution starts at address \#0: **Main** first initialises \(a\) and \(b\) (addresses \#0...\#3). It then creates a stack frame (addresses \#4) ready for the call to **Add**: there is space for four items in the frame, namely the return address, \(x\), \(y\) and the return value \(z\). **Main** proceeds by computing and storing the return address, then the values of \(a\) and \(b\) into the stack frame (addresses \#5...\#11) to form the arguments \(x\) and \(y\).

Finally it passes control to **Add** by setting \(PC = 21\) (address \#12). At this point, the memory content is as follows:

\[
\begin{align*}
SP &= \ldots, 26, 27, 28, 29, 30, 31, 32, 33, \ldots \\
\downarrow & \\
MEM &= \langle \ldots, 0, 0, 0, 0, 0, 0, 20, 10, 13, \ldots \rangle \\
&= z \quad y \quad x \quad RA
\end{align*}
\]
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$A \leftarrow 10$</td>
<td>compute $a = 10$</td>
</tr>
<tr>
<td>1</td>
<td>$MEM_{17} \leftarrow A$</td>
<td>store $a$</td>
</tr>
<tr>
<td>2</td>
<td>$A \leftarrow 20$</td>
<td>compute $b = 20$</td>
</tr>
<tr>
<td>3</td>
<td>$MEM_{18} \leftarrow A$</td>
<td>store $b$</td>
</tr>
<tr>
<td>4</td>
<td>$SP \leftarrow SP - 4$</td>
<td>create stack frame</td>
</tr>
<tr>
<td>5</td>
<td>$A \leftarrow PC$</td>
<td>copy $PC$</td>
</tr>
<tr>
<td>6</td>
<td>$A \leftarrow A + MEM_{20}$</td>
<td>compute $RA = PC + \text{off}$</td>
</tr>
<tr>
<td>7</td>
<td>$MEM_{SP+4} \leftarrow A$</td>
<td>store $RA$</td>
</tr>
<tr>
<td>8</td>
<td>$A \leftarrow MEM_{17}$</td>
<td>load $a$</td>
</tr>
<tr>
<td>9</td>
<td>$MEM_{SP+3} \leftarrow A$</td>
<td>store $a$</td>
</tr>
<tr>
<td>10</td>
<td>$A \leftarrow MEM_{18}$</td>
<td>load $b$</td>
</tr>
<tr>
<td>11</td>
<td>$MEM_{SP+2} \leftarrow A$</td>
<td>store $b$</td>
</tr>
<tr>
<td>12</td>
<td>$PC \leftarrow 21$</td>
<td>call</td>
</tr>
<tr>
<td>13</td>
<td>$A \leftarrow MEM_{SP+1}$</td>
<td>load $c$</td>
</tr>
<tr>
<td>14</td>
<td>$MEM_{19} \leftarrow A$</td>
<td>store $c$</td>
</tr>
<tr>
<td>15</td>
<td>$SP \leftarrow SP + 4$</td>
<td>remove stack frame</td>
</tr>
<tr>
<td>16</td>
<td>$HALT$</td>
<td>halt</td>
</tr>
<tr>
<td>17</td>
<td>$NOP$</td>
<td>$a$</td>
</tr>
<tr>
<td>18</td>
<td>$NOP$</td>
<td>$b$</td>
</tr>
<tr>
<td>19</td>
<td>$NOP$</td>
<td>$c$</td>
</tr>
<tr>
<td>20</td>
<td>$NOP$</td>
<td>off</td>
</tr>
<tr>
<td>21</td>
<td>$A \leftarrow MEM_{SP+3}$</td>
<td>load $x$</td>
</tr>
<tr>
<td>22</td>
<td>$A \leftarrow A + MEM_{SP+2}$</td>
<td>compute $z = x + y$</td>
</tr>
<tr>
<td>23</td>
<td>$MEM_{SP+1} \leftarrow A$</td>
<td>store $z$</td>
</tr>
<tr>
<td>24</td>
<td>$A \leftarrow MEM_{SP+4}$</td>
<td>load $RA$</td>
</tr>
<tr>
<td>25</td>
<td>$PC \leftarrow A$</td>
<td>return</td>
</tr>
<tr>
<td>26</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>27</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>28</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>29</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>30</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>31</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>32</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>33</td>
<td>$NOP$</td>
<td>stack (initial ToS)</td>
</tr>
</tbody>
</table>

**Figure 13.8:** Attempt #4 at implementing `Main` and `Add`.
• *Add* no longer needs to pop *x* and *y*; it simply loads the values from the stack frame, computes *z = x + y* and then stores the result back into the stack frame (addresses #21...#23). At this point, the memory content is as follows:

\[
\begin{align*}
SP & = \ldots, 26, 27, 28, 29, 30, 31, 32, 33, \ldots \\
MEM & = \langle \ldots, 0, 0, 0, 0, 30, 20, 10, 13, \ldots \rangle
\end{align*}
\]

Finally, it passes control to *Main* by loading the return address from the stack frame and copying it into *PC* (addresses #24...#25).

• *Main* concludes execution by loading *z* from the stack frame and storing it into *c* (addresses #13...#14); it then removes the stack frame (addresses #15). At this point, the memory content is as follows:

\[
\begin{align*}
SP & = \ldots, 26, 27, 28, 29, 30, 31, 32, 33, \ldots \\
MEM & = \langle \ldots, 0, 0, 0, 0, 30, 20, 10, 13, \ldots \rangle
\end{align*}
\]

It then halts (address #14).

Notice that now *Add* in particular is more efficient: it no longer has to pop then store *x* and *y* since it can simply access them using the appropriate offset from *SP*. The act of *Main* creating and then removing the stack frame for *Add* is also more explicit since it happens in one step rather than within a series of pushes and pops.

13.2.3.4 Attempt #5: using the stack frame to house local variables

Now we are fairly close to what we want; the only lingering issue relates to the second problem we originally identified. Look at the previous attempt: *Main* has a single place in memory it can store *a*, *b* and *c*. In this case *Main* is not recursive so there is no major problem, but we need a solution for more general cases.

So the question is, where can we store local variables? The answer is to pull the same trick again, and simply store them on the stack: if each instance of a sub-routine has a dedicated stack frame, then storing local variables in the stack frame will mean they cannot be overwritten by some other instance. In the last attempt, the caller took the responsibility of mangling the stack frame. In this attempt the callee will also play a part:

1. *Main* is represented by the instructions held in addresses #0...#18. Notice that we no longer need any space for *a*, *b* nor *c* since they are all held on the stack; *off* is a constant (we never change it) so that does not need to be held on the stack.

2. *Add* is represented by the instructions held in addresses #20...#24.

3. The stack occupies addresses #25...#32; remember that it grows downward in memory, and that *SP* = 32 initially.

The program execution can be described as follows:

• Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:

\[
\begin{align*}
SP & = \ldots, 25, 26, 27, 28, 29, 30, 31, \ldots \\
MEM & = \langle \ldots, 0, 0, 0, 0, 0, 0, 0, \ldots \rangle
\end{align*}
\]

• Execution starts at address #0: the first thing *Main* now does is create space in the stack frame for local variables (address #0). There is space for three items in the frame, namely *a*, *b* and *c*. Next it initialises *a* and *b* (addresses #1...#4) and creates a stack frame (addresses #5) ready for the call to *Add*. *Main* proceeds by computing and storing the return address, then *a* and *b* into the stack frame (addresses #6...#12).

Finally it passes control to *Add* by setting *PC* = 20 (address #13). At this point, the memory content is as follows:

\[
\begin{align*}
SP & = \ldots, 25, 26, 27, 28, 29, 30, 31, 32, \ldots \\
MEM & = \langle \ldots, 0, 0, 20, 10, 14, 0, 20, 10, \ldots \rangle \\
& = z \ y \ x \ RA \ c \ b \ a
\end{align*}
\]
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$SP \leftarrow SP - 4$</td>
<td>create local space</td>
</tr>
<tr>
<td>1</td>
<td>$A \leftarrow 10$</td>
<td>compute $a = 10$</td>
</tr>
<tr>
<td>2</td>
<td>$MEM_{SP+4} \leftarrow A$</td>
<td>store $a$</td>
</tr>
<tr>
<td>3</td>
<td>$A \leftarrow 20$</td>
<td>compute $b = 20$</td>
</tr>
<tr>
<td>4</td>
<td>$MEM_{SP+3} \leftarrow A$</td>
<td>store $b$</td>
</tr>
<tr>
<td>5</td>
<td>$SP \leftarrow SP - 4$</td>
<td>create stack frame</td>
</tr>
<tr>
<td>6</td>
<td>$A \leftarrow PC$</td>
<td>copy $PC$</td>
</tr>
<tr>
<td>7</td>
<td>$A \leftarrow A + MEM_{19}$</td>
<td>compute $RA = PC + off$</td>
</tr>
<tr>
<td>8</td>
<td>$MEM_{SP+4} \leftarrow A$</td>
<td>store $RA$</td>
</tr>
<tr>
<td>9</td>
<td>$A \leftarrow MEM_{SP+8}$</td>
<td>load $a$</td>
</tr>
<tr>
<td>10</td>
<td>$MEM_{SP+3} \leftarrow A$</td>
<td>store $a$</td>
</tr>
<tr>
<td>11</td>
<td>$A \leftarrow MEM_{SP+7}$</td>
<td>load $b$</td>
</tr>
<tr>
<td>12</td>
<td>$MEM_{SP+2} \leftarrow A$</td>
<td>store $b$</td>
</tr>
<tr>
<td>13</td>
<td>$PC \leftarrow 20$</td>
<td>call</td>
</tr>
<tr>
<td>14</td>
<td>$A \leftarrow MEM_{SP+1}$</td>
<td>load $c$</td>
</tr>
<tr>
<td>15</td>
<td>$MEM_{SP+6} \leftarrow A$</td>
<td>store $c$</td>
</tr>
<tr>
<td>16</td>
<td>$SP \leftarrow SP + 4$</td>
<td>remove stack frame</td>
</tr>
<tr>
<td>17</td>
<td>$SP \leftarrow SP + 4$</td>
<td>remove local space</td>
</tr>
<tr>
<td>18</td>
<td>$HALT$</td>
<td>halt</td>
</tr>
<tr>
<td>19</td>
<td>$NOP$</td>
<td>off</td>
</tr>
<tr>
<td>20</td>
<td>$A \leftarrow MEM_{SP+3}$</td>
<td>load $x$</td>
</tr>
<tr>
<td>21</td>
<td>$A \leftarrow A + MEM_{SP+2}$</td>
<td>compute $z = x + y$</td>
</tr>
<tr>
<td>22</td>
<td>$MEM_{SP+1} \leftarrow A$</td>
<td>store $z$</td>
</tr>
<tr>
<td>23</td>
<td>$A \leftarrow MEM_{SP+4}$</td>
<td>load $RA$</td>
</tr>
<tr>
<td>24</td>
<td>$PC \leftarrow A$</td>
<td>return</td>
</tr>
<tr>
<td>25</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>26</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>27</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>28</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>29</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>30</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>31</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
<tr>
<td>32</td>
<td>$NOP$</td>
<td>stack</td>
</tr>
</tbody>
</table>

Figure 13.9: Attempt #5 at implementing `Main` and `Add`. 

249
• Add is basically the same as the last attempt: it loads \( x \) and \( y \) from the stack frame, computes \( z = x + y \) and then stores the result back into the stack frame (addresses #20...#22). At this point, the memory content is as follows:

\[
\begin{align*}
SP \quad i & = \ldots, 25, 26, 27, 28, 29, 30, 31, 32, \ldots \\
MEM & = \langle \ldots, 0, 30, 20, 10, 14, 0, 20, 10, \ldots \rangle \\
& = z \ y \ x \ RA \ c \ b \ a
\end{align*}
\]

Finally, it passes control to Main by loading the return address from the stack frame and copying it into PC (addresses #23...#24).

• Main concludes execution by loading \( z \) from the stack frame and storing it into \( c \) (addresses #14...#15); it then removes the stack frame (addresses #16) to give

\[
\begin{align*}
SP \quad i & = \ldots, 25, 26, 27, 28, 29, 30, 31, 32, \ldots \\
MEM & = \langle \ldots, 0, 30, 20, 10, 14, 30, 20, 10, \ldots \rangle \\
& = c \ b \ a
\end{align*}
\]

and the space for local variables (addresses #17) to give

\[
\begin{align*}
SP \quad i & = \ldots, 25, 26, 27, 28, 29, 30, 31, 32, \ldots \\
MEM & = \langle \ldots, 0, 30, 20, 10, 14, 30, 20, 10, \ldots \rangle \\
& = c \ b \ a
\end{align*}
\]

It then halts (address #18).

At this point we are done: the end result is more or less a real sub-routine calling mechanism. Of course there are aspects that could be improved (some depend on the instruction set we are using), but this really is how things work.

### 13.3 A buffer overflow attack

Now we finally have enough of a background to look at the original problem: buffer overflow attacks. The attack is based on the fact that the stack holds a mix of return addresses (which determine control-flow) and local variables (which are used in computation). The basic idea is that if we put too much content into the local variables, the overwriting bug is triggered; what we overwrite is essentially other stack content, including return addresses. That is only ever going to be bad, basically because it breaks our sub-routine calling mechanism.

#### 13.3.1 Modelling and implementing a “web-server”

First we need something to attack, so we return to our original motivating example of a web-server. The scenario is that a remote computer \( C \) acts as a target web-server, and our task as the attacker Eve is to send it a malicious request \( S \); the request is specifically constructed to make \( C \) do whatever we want, rather than whatever it was designed to. Figure 13.10 describes Web-Server and Process-Req, two algorithms that act as a model for \( C \):

• The Web-Server algorithm simply loops continuously; this models the fact that \( C \) endlessly listens for a connection by some web-browser, then services the request it makes.

• The Process-Req algorithm is where the action happens; it models \( C \) reading a request and then processing it somehow. The request itself is a sequence of numbers terminated by a zero; this is a bit like the C-string method from Chapter 5.

Imagine that we send the request

\[
S = \langle 1, 2, 0 \rangle.
\]

The web-server \( C \) processes the request by following the Process-Req algorithm as follows:

**Step #1** Assign \( T \leftarrow \langle \bot, \bot \rangle, i \leftarrow 0. \)

**Step #2** Assign \( t \leftarrow \text{Network-Rd}, \) i.e., read the value 1 from the network.
1 algorithm PROCESS-REQ begin
2 \[ T \leftarrow \langle \bot, \bot \rangle \]
3 \[ i \leftarrow 0 \]
4 \[ \text{repeat} \]
5 \[ t \leftarrow \text{Network-Read()} \]
6 \[ \text{if } t \neq 0 \text{ then} \]
7 \[ T_i \leftarrow t \]
8 \[ i \leftarrow i + 1 \]
9 \[ \text{end} \]
10 \[ \text{until } t = 0 \]
11 \[ \text{Process } T \text{ somehow return} \]
12 end

(a) An algorithm that models a web-server.

(b) An algorithm that models the reading and processing of some request.

Figure 13.10: Two algorithms used to represent the web-server subjected to a buffer overflow attack.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#3</td>
<td>Since ( t \neq 0 ), assign ( T_0 \leftarrow t = 1 ) and ( i \leftarrow i + 1 = 1 ),</td>
</tr>
<tr>
<td>#4</td>
<td>Since ( t \neq 0 ), perform the next loop iteration.</td>
</tr>
<tr>
<td>#5</td>
<td>Assign ( t \leftarrow \text{Network-Rd} ), i.e., read the value 2 from the network.</td>
</tr>
<tr>
<td>#6</td>
<td>Since ( t \neq 0 ), assign ( T_1 \leftarrow t = 2 ) and ( i \leftarrow i + 1 = 2 ),</td>
</tr>
<tr>
<td>#7</td>
<td>Since ( t \neq 0 ), perform the next loop iteration.</td>
</tr>
<tr>
<td>#8</td>
<td>Assign ( t \leftarrow \text{Network-Rd} ), i.e., read the value 0 from the network.</td>
</tr>
<tr>
<td>#9</td>
<td>Since ( t = 0 ), skip the conditional block.</td>
</tr>
<tr>
<td>#10</td>
<td>Since ( t = 0 ), stop the loop.</td>
</tr>
<tr>
<td>#11</td>
<td>Process ( T ) somehow.</td>
</tr>
<tr>
<td>#12</td>
<td>Return.</td>
</tr>
</tbody>
</table>

That is, it reads each element of the request (apart from the zero which terminates it) into the sequence \( T \), then processes \( T \) in some way. The type of processing performed is not important; you can think of it being interpreted as as some request for a particular web-page if you like.

The very first step highlights a problem however: how do we know the size of \( T \)? Of course we cannot know the size until we have read it, so we simply allocate a 2-element buffer and hope this is enough (i.e., that the request is not longer).

Remember this is just a model: the claim is not that this is a real web-server, just that we have an outline of the parts which are important to our discussion. Put another way, this is a compromise between reality and the ability to explain the main points clearly.

Now we need to implement these algorithms in a program; this means another upgrade for our example computer that adds network access. This sounds like (and would be) a complicated task if it were not for the fact that we can avoid almost all the detail involved: all we want is some instructions that can read and write values, acting as a model for what a real network connection would do. In short, we add the following:

- \( 12nnnn \) means read \( A \) from network port \( n \).
- \( 13nnnn \) means write \( A \) onto network port \( n \).

This means that we, as the attacker, can supply numbers which are read using the first instruction and view numbers written using the second instruction. Armed with the new instructions, Figure 13.11 describes the implementation. Following the same style as throughout this Chapter, the program is in two parts, with a third part representing the stack:

1. **WEB-SERVER** is represented by the instructions held in addresses \#0 \ldots \#6. A constant called \( off \) is held in address \#7: this allows computation of the return address.
2. Process-Req is represented by the instructions held in addresses #8...#18. A constant called inc is held in address #19: this allows addition of a fixed value (in this case one) to a value of our choice. Unlike most of the previous programs we have looked at before, this one uses one fairly opaque technique to get the job done. The problem is, how can we store \( t \) into \( T_i \)? Since \( T \) is a local variable, it is held on the stack; ideally we would do something like

\[
\text{MEM}_{SP+1} \leftarrow A
\]

This would allow us to read \( t \) into \( A \), then store it at the right offset from \( SP \). But we cannot do this; there is no appropriate instruction. So instead, we “make” one ourselves. The idea is to make constructive use of the self-modifying code from Chapter 4. Notice that the instruction held in address #11 initially reads

\[
510001 \mapsto \text{MEM}_{SP+1} \leftarrow A
\]

How can we make it access the next element of the stack (i.e., the next element of \( T \))? Easy! Just add one to it; we get

\[
510002 \mapsto \text{MEM}_{SP+2}.
\]

So in each iteration of the loop, we load the instruction and update it so that in the next iteration it will store into the next element of the stack; this is achieved by the instructions held in addresses #12...#14. Put another way, we do not need \( i \): we just have an instruction and make sure the operand is updated to “point” at where \( i \) would.

3. The stack occupies addresses #19...#27; remember that it grows downward in memory, and that \( SP = 27 \) initially.
13.3.2 A normal, valid request

When we feed the web-server a normal request, things proceed as we would expect. Imagine we go back to the previous example and feed our program the input

\[ S = (1, 2, 0). \]

The behaviour of the program is as follows:

- Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:
  \[ \text{SP} \quad i = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \ldots \]
  \[ \text{MEM} = \langle \ldots, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ldots \rangle \]

- Execution starts at address #0: \texttt{Web-Server} creates a stack frame (addresses #0) then computes and stores a return address (addresses #1\ldots#3) in it. Then it passes control to \texttt{Process-Req} by setting \texttt{PC} = 8 (address #4). At this point, the memory content is as follows:
  \[ \text{SP} \quad i = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \ldots \]
  \[ \text{MEM} = \langle \ldots, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, \ldots \rangle \]

- \texttt{Process-Req} first creates space in the stack frame for local variables (address #8); there is space for the 2-element buffer \( T \). At this point, the memory content is as follows:
  \[ \text{SP} \quad i = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \ldots \]
  \[ \text{MEM} = \langle \ldots, 0, 0, 0, 0, 0, 0, 1, 0, 5, \ldots \rangle \]

  It then loads \( l \) from the network (address #9), checks to see if this means the end of the sequence (address #10) and if not, stores \( l \) in \( T \) (address #11). In this case \( l = 1 \) since this is the first element of input, so at this point the memory content is:
  \[ \text{SP} \quad i = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \ldots \]
  \[ \text{MEM} = \langle \ldots, 0, 0, 0, 0, 0, 1, 0, 5, \ldots \rangle \]

  One more element is read from the network, which means the memory content once \( T \) is ready for processing is
  \[ \text{SP} \quad i = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \ldots \]
  \[ \text{MEM} = \langle \ldots, 0, 0, 0, 0, 1, 2, 5, \ldots \rangle \]

  After processing \( T \) (which is not included in the program), the space for local variables is removed (addresses #16) to give
  \[ \text{SP} \quad i = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \ldots \]
  \[ \text{MEM} = \langle \ldots, 0, 0, 0, 0, 0, 0, 0, 1, 2, 5, \ldots \rangle \]

  and control is passed back to \texttt{Web-Server} by loading the return address from the stack frame and copying it into \texttt{PC} (addresses #17\ldots#18).

- \texttt{Main} removes the stack frame (addresses #16) to give
  \[ \text{SP} \quad i = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \ldots \]
  \[ \text{MEM} = \langle \ldots, 0, 0, 0, 0, 0, 0, 1, 2, 5, \ldots \rangle \]

  then sets \( PC = 0 \) (address #6) to repeat the whole process all over again.
13.3.3 A not-so-normal, attack request

As you might have guessed, we can feed the web-server a not-so-normal request and cause it problems. This time imagine we feed the program the input

\[ S = (111111, 111111, 25, 0). \]

Already this looks quite bad because the sequence has three elements in it (excluding the zero), and we only have enough space for two elements in the buffer for \( T \). So what happens? This does:

- Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:
  \[
  \begin{align*}
  SP & = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \\
  MEM & = \langle \ldots, 0, 0, 0, 0, 0, 0, 0, 5, \ldots \rangle
  \end{align*}
  \]

- Execution starts at address #0: **Web-Server** creates a stack frame (addresses #0) then computes and stores a return address (addresses #1...#3) in it. Then it passes control to **Process-Req** by setting \( PC = 8 \) (address #4). At this point, the memory content is as follows:
  \[
  \begin{align*}
  SP & = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \\
  MEM & = \langle \ldots, 0, 0, 0, 0, 0, 0, 5, \ldots \rangle
  \end{align*}
  \]

- **Process-Req** first creates space in the stack frame for local variables (address #8); there is space for the 2-element buffer \( T \). At this point, the memory content is as follows:
  \[
  \begin{align*}
  SP & = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \\
  MEM & = \langle \ldots, 0, 0, 0, 0, 0, 0, 5, \ldots \rangle
  \end{align*}
  \]

It then loads \( t \) from the network (address #9), checks to see if this means the end of the sequence (address #10) and if not, stores \( t \) in \( T_i \) (address #11). In this case \( t = 111111 \) since this is the first element of input, so at this point the memory content is:

\[
\begin{align*}
SP & = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \\
MEM & = \langle \ldots, 0, 0, 0, 0, 0, 111111, 0, 5, \ldots \rangle
\end{align*}
\]

Two more elements are read from the network, which means the memory content once \( T \) is ready for processing is

\[
\begin{align*}
SP & = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \\
MEM & = \langle \ldots, 0, 0, 0, 0, 0, 0, 111111, 111111, 25, \ldots \rangle
\end{align*}
\]

Notice that the overwriting bug has been triggered. After processing \( T \) (which is not included in the program), the space for local variables is removed (addresses #16) to give

\[
\begin{align*}
SP & = \ldots, 20, 21, 22, 23, 24, 25, 26, 27, \\
MEM & = \langle \ldots, 0, 0, 0, 0, 0, 111111, 111111, 25, \ldots \rangle
\end{align*}
\]

and an *attempt* to pass control to **Web-Server** is made. The return address is loaded from the stack frame and copied into \( PC \) (addresses #17...#18).

*But*, since we read more data into \( T \) than it could hold, we overwrote the return address: this means that \( PC \) is set to 25 which we (as the attacker) provided. Instead of jumping back to address #5, i.e., returning to **Web-Server**, we instead end up executing the instruction at address #25. What is this instruction? None other than

\[ 111111 \mapsto \text{payload} \]

which we (as the attacker) provided, and does something “bad”.

\[ \]
13.3.4 The end result

We started with an abstract example of the overwriting bug, and now we have a more concrete example of what it implies: the buffer overflow attack allows us to send the web-server some input, and have it execute a program (i.e., one or more instructions) of our choice. As we mentioned previously, this is bad. The things to note are that we

- needed no access to the web-server in the sense of having to log in and execute our program (meaning we side-step any password),
- needed no physical access to the web-server in the sense that the attack can be carried out across a network (from another country if we want), and
- tricked the web-server to execute our program as itself, meaning our program can enjoy access to any resource the web-server has.

Just like at the end of Chapter 4 where we tried to come up with better approaches to stopping viruses, we can try to think of ways to stop the buffer overflow attack. There are a lot of options, so as an exercise try to think of some: imagine you have free reign to alter the computer or the web-server program, and try to work out countermeasures that prevent our attack from working.

This is an ideal moment to point out that the impact of buffer overflow attacks (and related techniques) mean this is an active field of research; black hat[1] and white hat[16] researchers both push the field forward. In addition to our normal set of references to Wikipedia, a great online resource is Phrack[11] magazine. Some people might argue some of the content is morally dubious, but you can (mostly) ignore them: Phrack was, and still is, both an outstanding technical resource and documentary of information security and hacker [8] culture. You can find back-issues of Phrack here

http://www.phrack.com/

and one of the first articles on buffer overflow attacks here


The article has a lot more technical detail than what we have covered, and applies to more realistic scenarios, but basically the concept is exactly the same: go and read it!
BIBLIOGRAPHY


IS THE KEY TO YOUR CASH BEING LEAKED BY YOUR CACHE?

If you talk to people who design computers, they might curse what is often called the memory wall. The problem is that memory is quite slow in comparison to everything else; accesses to memory are often many orders of magnitude slower than other operations (e.g., performing arithmetic). This acts as a barrier, or wall, that prevents computers executing programs quicker that they do. In a rough sense, the reason the memory wall exists is to do with how the electronic components that memory is built from behave. Ideally we would like very large, very fast memories but because of how the electronics work, this is usually impossible: either we can have small, fast memories or large, slow memories. Although we made no reference to this until now, our example computer from Chapter 4 has one of the latter: MEM is large, but slow.

Of course we know computers do execute programs quickly, so how has the memory wall being knocked down? The answer is basically a compromise called the memory hierarchy. The idea is to have multiple memories each representing a different compromise between size and speed, and to use the right one for the right job. This is already evident in our example computer: when we need to store data and access it quickly we use the accumulator A, but when we need to store larger quantities of data or for longer periods of time we use MEM. More complicated computer designs have more complicated memory hierarchies; a vital component within such computers is a level of the hierarchy called the cache, literally a “treasure trove” in which useful data can be hoarded. Cache memories improve performance, but as with everything in life, we seldom get something for nothing. The idea of this Chapter is to explore a disadvantage of caches. Specifically, while they improve performance they also introduce a security issue when the programs a computer executes need to prevent leakage of information: we want to demonstrate a side-channel attack based on execution time, putting it within a similar category to the material from Chapter 12.

14.1 Cache memories

A cache is a small, fast memory placed between the computer and MEM: each time the computer tries to access MEM, the access goes through the cache first. Unlike MEM which is “dumb”, the cache is “smart” in that some rules govern what it holds. This is important since the cache is smaller than MEM, so it cannot simply hold the same content: instead it just holds some of the content in MEM, replicating it with the aim of reducing the total number of times that MEM is accessed.

Imagine the computer performs a memory access using address \( x \), e.g., tries to load from MEM\(_x\). The access goes through the cache, so the cache first checks whether it is currently holding MEM\(_x\) or not. One of two cases will clearly occur:

- If the cache does hold MEM\(_x\), a cache-hit occurs and the value MEM\(_x\) can be returned to the computer quickly and without accessing MEM.
• If the cache does not hold $MEM_x$, a cache-miss occurs and the value $MEM_x$ can be returned to the computer only after a subsequent, slower access to $MEM$:

Note that because of the size difference, the cache may have to evict some of the content it currently holds in order to make space for the new content relating to $MEM_x$.

As a short-hand, we use $H$ and $M$ to mean a cache-hit and cache-miss where need be. Clearly the hope is that there are lots of cache-hits and few cache-misses. This relies on a property of programs called the principle of locality. Imagine you execute a program and pause it at some point in time. Our claim is that whatever the program is doing at that point requires a small working set of data relative to the total size of $MEM$: only a small part of $MEM$ is used at once. Of course this is not always true, but for average programs there are two good reasons it makes sense; both can be illustrated by an algorithm we saw in Chapter 5.

Consider the following:

```
algorithm C-STRING-LENGTH(x) begin
  i ← 0
  while $MEM_{x+i} \neq 0$ do
    i ← i + 1
  end
  return i
end
```

in which two forms of locality can be identified:

**Spacial locality** is where having made one access to memory, there is a good chance the next one will be close to the first. That is, the two accesses use addresses that are close together.

Look at the example: the string is stored in memory, with the algorithm loading characters from $MEM_{x+i}$. If the $i$-th character is loaded in some iteration of the loop, we know that the next iteration will load the $(i+1)$-th character which is close by in memory. So accesses to the string are spatially local in this case.

**Temporal locality** is where having made one access to memory, there is a good chance that if you look at another access within a small window of time, it will be the same. That is, the two accesses use the same address.

Look at the example: there is just one assignment in the loop (it adds one to the current value of $i$). One can easily imagine this is achieved using a single instruction; during each iteration of the loop, the computer fetches this instruction from memory in order to then execute it. If the string is $n$ characters long, having fetched the instruction once it will be fetched again $n - 1$ times: only once (i.e., when the loop finishes) will it not be fetched again. So accesses to the instruction are temporally local in this case.

The fact that these features are true for average programs basically means that at a given point in time the working set is small: if there is a memory access, there is a good chance it accesses data that is the same as or close to data accessed recently.

There is a more human-centric way to look at the same thing. Psychologists often cite Miller’s Law [8]: it says that an average human can keep $7 \pm 2$, or between five and nine, items of information (e.g., concepts or facts) in their short-term memory at once. That might seem surprisingly low, so how do we get anything useful done? One factor is the ability for humans to concentrate: they work most effectively when they concentrate on one task. In a way, this is the same thing as locality. What we are saying is that humans can hold a working set of $7 \pm 2$ items of information; their access to information is localised within this working set due to the fact that they concentrate on the related task.
14.1.1 The design of a cache mechanism

Now we have a rough idea about what a cache should do, it makes sense to introduce a real design so as to explain what it does do. To make sure this explanation is not too overwhelming, we opt to avoid a lot of detail:

1. We consider a simple (perhaps the most simple) cache design.
2. Since both instructions and data are stored in MEM, the cache can hold both as well. However, this makes things more complicated. As a result we will imagine that instructions are fetched directly from MEM (i.e., they bypass the cache) so as to focus on data only.
3. Clearly accesses to MEM can be either loads or stores, but coping with stores is a little more complicated than loads; as a result, we ignore them and focus on loads only.

Numerous resources exist which can flesh out the missing detail if you are interested. The thing to keep in mind is that although we have strayed a little away from reality, our description is close enough to discuss a related security issue later; of course this, rather than the cache itself, is our overall aim so it seems like a good compromise.

So how can we add a cache to our example computer? At the moment, every time the computer needs to load from MEM, we can imagine it following a (very) simple algorithm:

\begin{verbatim}
1 algorithm Load(x) begin
2 return MEM_x
3 end
\end{verbatim}

To add a cache, all we really need to think about is how \texttt{Load} should be altered so that it applies the appropriate rules using some extra components that we also add.

14.1.1.1 An outline of the cache mechanism

The cache itself is typically organised as a table where each row is called a cache line. It is important that we can refer to the different components within a particular cache line, so assume \( C \) is the \( i \)-th cache line:

- \( C_{\text{valid}} \) indicates whether the \( i \)-th cache line is valid or not, i.e., whether if there is any content in it or not.
- \( C_{\text{tag}} \) is the tag for the cache line; this acts as identifier for exactly what the cache line contains.
- \( C_{\text{data}} \) is the actual content, i.e., data, for the \( i \)-th cache line. This is basically a just sequence whose elements are typically called sub-words.

If we want to describe a real cache, it is important we are precise about some of the quantities involved: we assume throughout that there are \( l \) cache lines with \( w \) sub-words in each. Now we can draw a picture which hopefully makes things a bit clearer:

<table>
<thead>
<tr>
<th>Line</th>
<th>Valid</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>true</td>
<td>1</td>
<td>(4, 1, 7, 5)</td>
</tr>
<tr>
<td>1</td>
<td>false</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
<td>1</td>
<td>(0, 2, 6, 3)</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>5</td>
<td>false</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>6</td>
<td>false</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>7</td>
<td>false</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>

In this case, \( l = 8 \) and \( w = 4 \) so the cache has eight cache lines, each of which can accommodate four sub-words; only the 0-th and 2-nd cache lines have valid content. For example if \( C \) refers to the 2-nd cache line, then \( C_{\text{valid}} = \text{true} \) and \( C_{\text{tag}} = 1 \); if \( D = C_{\text{data}} \) then \( D_0 = 0, D_1 = 2, D_2 = 6 \) and \( D_3 = 3 \). Based on this table we can dive in and rewrite the \texttt{Load} algorithm:
algorithm \texttt{Load}(x) \begin{algorithmic}[1]
  \STATE \texttt{x}_{\text{sub-word}} \leftarrow x \mod w 
  \STATE \texttt{x}_{\text{line}} \leftarrow \lfloor x/w \rfloor \mod l 
  \STATE \texttt{x}_{\text{tag}} \leftarrow \lfloor x/w \rfloor /l \n  \STATE \texttt{x}_{\text{sub-wordless}} \leftarrow \lfloor x/w \rfloor \cdot w 
  \STATE \text{Let } C \text{ refer to cache line number } \texttt{x}_{\text{line}} 
  \STATE \text{Let } D \text{ refer to } \text{C}_{\text{data}}, \text{i.e., the data held in } C 
  \IF {\text{C}_{\text{valid}} = \text{false} \text{ or } \text{C}_{\text{tag}} \neq \text{\texttt{x}_{\text{tag}}}} 
    \FOR {i \text{ from } 0 \text{ upto } w-1} 
      \STATE \text{D}_i \leftarrow \text{MEM}_{\text{sub-wordless}+i} 
    \END 
    \STATE \text{C}_{\text{valid}} = \text{true} 
    \STATE \text{\texttt{x}_{\text{tag}}} = \text{\texttt{x}_{\text{tag}}} 
  \ENDIF 
  \STATE \RETURN \text{D}_{\text{sub-wordless}} 
\end{algorithmic}

Wow! This is quite a bit more complicated than usual, but we can explain what is going on step-by-step:

- Given the address \( x \), lines \#2 through \#5 perform what is typically called \textbf{address translation}: they apply the rules which decide where in the cache the associated data should be stored.

  The idea is fairly simple: \texttt{MEM} is large, but the cache is smaller. Or put another way, there are \(|\text{MEM}| \) elements in \texttt{MEM} but only a total of \( l \cdot w \) sub-words in the cache. We know
  \[ l \cdot w < |\text{MEM}| \]
  because that was the point in the first place, i.e., we wanted a smaller, faster version of \texttt{MEM}. So, if we just map the \( i \)-th element of \texttt{MEM} onto the \( i \)-th sub-word that would work for the first \( l \cdot w \) elements, i.e., elements \( 0 \ldots l \cdot w - 1 \). But what about element \( l \cdot w \) which has no corresponding sub-word? To cope, we “wrap around” and map element \( l \cdot w \) onto the sub-word 0, element \( l \cdot w + 1 \) onto sub-word 1 and so on. The assignments to \texttt{x}_{\text{sub-word}} and \texttt{x}_{\text{line}} decodes \( x \) to apply exactly this mapping process.

- The condition construct on line \#8 checks if there is valid content in the cache line that \( x \) should be in, and whether it is actually the content related to \( x \); it does this by checking \text{C}_{\text{valid}} and then comparing the tag computed from \( x \) with the one stored in \text{C}_{\text{tag}}. If both checks pass, we have a cache-hit: the data is there, we just need to access it. If either check fails, we have a cache-miss: before accessing the data we need to fetch it into the cache. Lines \#9 through \#13 handle this by including a loop construct that loads each sub-word from \texttt{MEM} (we discuss this in more detail below).

  Why would \( \text{C}_{\text{tag}} \neq \text{\texttt{x}_{\text{tag}}} \)? Remember above that we said elements from \texttt{MEM} wrap around: following our reasoning, the 0-th sub-word of cache line 0 could contain element 0 from \texttt{MEM} or element \( l \cdot w \). The reason we do not get confused between the two is that the tag for element 0 will be different from the tag for element \( l \cdot w \). Thus if the comparison fails, we know that although the cache line has valid content in it, that content is not want we are looking for.

- Finally, line \#15 returns the data we want to access. Note that by this point we know it must be resident in the cache: either it was there in the first place and there was a cache-hit, or we fetched it due to a cache-miss.

Since we will use the concept later, line \#5 is particularly important: the idea is that it calculates a “sub-wordless” version of the address \( x \). That is, \texttt{x}_{\text{sub-wordless}} is \( x \) but without the part that determines the sub-word \( x \) should map to. We have \( w = 4 \) sub-words in each cache line; have a look at what the calculation does if we select \( x = 1 \) or \( x = 34 \):

\[
\begin{array}{llll}
  x & = & 1_{(10)} & = & 000001_{(2)} \\
  \texttt{x}_{\text{sub-wordless}} & = & \lfloor x/w \rfloor \cdot w & = & 0_{(10)} & = & 000000_{(2)} \\
  x & = & 34_{(10)} & = & 100010_{(2)} \\
  \texttt{x}_{\text{sub-wordless}} & = & \lfloor x/w \rfloor \cdot w & = & 32_{(10)} & = & 1000000_{(2)} 
\end{array}
\]

Since we have written the result in decimal \textit{and} binary, the process is clearer: to get \texttt{x}_{\text{sub-wordless}}, we basically set the least-significant (i.e., those on the right-hand side) two bits of \( x \) to zero.
This is made use of in line #10 when the cache fetches sub-words from MEM due to a cache-miss. Remember that $x_{\subwordless}$ is $x$ without the part that determines which sub-word $x$ should map to. By looping through all $i \in \{0, 1, \ldots, w - 1\}$, the expression

$$x_{\subwordless} + i$$

therefore cycles through all the addresses in the cache line $x$ has been mapped into. Consider $x = 34$ again: we know that

$$x_{\subword} = x \mod w = 2$$
$$x_{\line} = \lfloor x/w \rfloor \mod l = 0$$
$$x_{\subwordless} = \lfloor x/w \rfloor \cdot w = 32$$

so address $x = 34$ maps to sub-word 2 of cache line 0. When the loop fetches the corresponding sub-words from MEM, the steps it does are something like

$$D_0 \leftarrow MEM_{x_{\subwordless} + 0} = MEM_{32}$$
$$D_1 \leftarrow MEM_{x_{\subwordless} + 1} = MEM_{33}$$
$$D_2 \leftarrow MEM_{x_{\subwordless} + 2} = MEM_{34}$$
$$D_3 \leftarrow MEM_{x_{\subwordless} + 3} = MEM_{35}$$

which means we end up with the data we wanted in the right place.

14.1.1.2 An example of cache behaviour

The mechanism we described above is (part of) a direct-mapped cache. The name basically reflects how the address translation works: an address maps directly to just one one cache line and sub-word. Confused? Imagine we set $l = 8$ and $w = 4$ again as above, and write a program that performs a sequence of nine loads using the addresses

$$1, 34, 35, 36, 37, 1, 38, 39, 40.$$ 

Such a sequence is often called an address stream. Basically we invoke Load nine times, i.e., the program execution can be modelled by

$$
\begin{align*}
220001 & \mapsto A \leftarrow MEM_1 \mapsto \text{Load(1)} \\
220034 & \mapsto A \leftarrow MEM_{34} \mapsto \text{Load(34)} \\
220035 & \mapsto A \leftarrow MEM_{35} \mapsto \text{Load(35)} \\
220036 & \mapsto A \leftarrow MEM_{36} \mapsto \text{Load(36)} \\
220037 & \mapsto A \leftarrow MEM_{37} \mapsto \text{Load(37)} \\
220001 & \mapsto A \leftarrow MEM_1 \mapsto \text{Load(1)} \\
220038 & \mapsto A \leftarrow MEM_{38} \mapsto \text{Load(38)} \\
220039 & \mapsto A \leftarrow MEM_{39} \mapsto \text{Load(39)} \\
220040 & \mapsto A \leftarrow MEM_{40} \mapsto \text{Load(40)}
\end{align*}
$$

Of course there are probably other instructions in there as well, but we can ignore them: from the point of view of the cache, we only care about the loads. One thing worth noting is the address stream already roughly follows the principle of locality: we load from $MEM_1$ twice (which is temporal locality), and also perform consecutive loads from $MEM_{34}$ through $MEM_{37}$ and $MEM_{38}$ through $MEM_{40}$ (which is spacial locality).

In addition to the direct-mapped cache design outlined, other examples also exist; often, one design is preferred over another based on how or where it will be used. Find out about at least one other design: explain the motivation for using it, and try to write an algorithm (of a similar style to Load) that describes how accesses are satisfied.

The question is, for this program how does the cache behave? In a sense, the content of MEM does not matter: we just want to know when cache-hits and cache-misses occur, not necessarily the values actually loaded. On the other hand, some concrete values might make the example easier to grasp, so imagine that
before we execute the program the memory content is as follows:

\[
\begin{align*}
  i &= 0, 1, 2, 3, \ldots \\
  MEM &= \langle 220001, 220034, 220035, 220036, \ldots \rangle \\
  i &= \ldots 34, 35, 36, 37, \ldots \\
  MEM &= \langle \ldots 3, 1, 4, 1, \ldots \rangle \\
  i &= 38, 39, 40, 41, \ldots \\
  MEM &= \langle 5, 9, 2, 6, \ldots \rangle
\end{align*}
\]

Now we are ready: from the address stream

\[
1, 34, 35, 36, 37, 1, 38, 39, 40
\]

we get the cache behaviour $M, M, H, M, H, M, H, H, M,$

i.e., a cache-miss followed by another cache-miss, then a cache-hit and so on. Of course this is not much of an explanation; tracing through each step in the algorithm invocations is a little tedious, but here is at least a little more detailed account of what the cache does:

**Step #1**: Reset the cache to an initial, empty state.

**Step #2**: Address 1 decoded into sub-word 1, line 0 and tag 0; this implies a cache-miss (the content of line #0 is not valid). Line #0 is filled by fetching $MEM_0$ through $MEM_3$. Finally, the value 220034 is loaded from line #0, sub-word #1 which acts as an alias for $MEM_1$.

**Step #3**: Address 34 decoded into sub-word 2, line 0 and tag 1; this implies a cache-miss (it did not match the tag in line #0). Resident content of line #0 is evicted, and refilled by fetching $MEM_{32}$ through $MEM_{35}$. Finally, the value 3 is loaded from line #0, sub-word #2 which acts as an alias for $MEM_{34}$.

**Step #4**: Address 35 decoded into sub-word 3, line 0 and tag 1; this implies a cache-hit (it matched the tag in line #0). Finally, the value 1 is loaded from line #0, sub-word #3 which acts as an alias for $MEM_{35}$.

**Step #5**: Address 36 decoded into sub-word 0, line 1 and tag 1; this implies a cache-miss (the content of line #1 is not valid). Line #1 is filled by fetching $MEM_{36}$ through $MEM_{39}$. Finally, the value 4 is loaded from line #1, sub-word #0 which acts as an alias for $MEM_{36}$.

**Step #6**: Address 37 decoded into sub-word 1, line 1 and tag 1; this implies a cache-hit (it matched the tag in line #1). Finally, the value 1 is loaded from line #1, sub-word #1 which acts as an alias for $MEM_{37}$.

**Step #7**: Address 1 decoded into sub-word 1, line 0 and tag 0; this implies a cache-miss (it did not match the tag in line #0). Resident content of line #0 is evicted, and refilled by fetching $MEM_0$ through $MEM_3$. Finally, the value 220034 is loaded from line #0, sub-word #1 which acts as an alias for $MEM_1$.

**Step #8**: Address 38 decoded into sub-word 2, line 1 and tag 1; this implies a cache-hit (it matched the tag in line #1). Finally, the value 5 is loaded from line #1, sub-word #2 which acts as an alias for $MEM_{38}$.

**Step #9**: Address 39 decoded into sub-word 3, line 1 and tag 1; this implies a cache-hit (it matched the tag in line #1). Finally, the value 9 is loaded from line #1, sub-word #3 which acts as an alias for $MEM_{39}$.

**Step #10**: Address 40 decoded into sub-word 0, line 2 and tag 1; this implies a cache-miss (the content of line #2 is not valid). Line #2 is filled by fetching $MEM_{40}$ through $MEM_{43}$. Finally, the value 2 is loaded from line #2, sub-word #0 which acts as an alias for $MEM_{40}$.

The same thing can be visualised as a series of diagrams, such as those in Figure 14.1 that show the state of the cache after each of the loads has been performed.

Interestingly, although we said that the address stream looked good in terms of locality the result is not that good: we get the same number of cache-hits as cache-misses in this case. In part this is because the access stream creates a lot of interference. This is where accesses evict each other from the cache, by virtue of the rules it applies, even though keeping the corresponding content there would be of more benefit.

264
Figure 14.1: The behaviour of a small direct-mapped cache when a sequence of loads from addresses 1, 34, 35, 36, 37, 1, 38, 39 and 40 is performed.
14.1.2 Reasoning about cache effectiveness

Imagine we write a program and execute it on our newly upgraded computer; say there are \( m \) memory accesses in total, of which \( m_H \) result in cache-hits and \( m_M \) result in cache-misses. This means

\[
m_H + m_M = m.
\]

Using these quantities we can measure the proportion of accesses which were cache-hits or cache-misses; we call these the hit ratio and miss ratio, and write them as

\[
R_H = \frac{m_H}{m} \quad \text{and} \quad R_M = \frac{m_M}{m}
\]

where clearly now

\[
R_H + R_M = 1.
\]

Another way to think of these quantities is as probabilities: if \( m \) is large enough, then our reasoning is that \( R_H \) and \( R_M \) are good general estimates as well as specific results for our program. Remember that our program is average in some sense, so basically we can say that on average an access is a cache-hit with probability \( R_H \) or conversely a cache-miss with probability \( R_M \).

From the hit and miss ratios, we can start to reason about mean access time, i.e., the average time it takes to perform a memory access. When there was no cache, this was simple: all we needed to do was load from or store into \( MEM \). We can think the time taken to do this as being described by a constant \( T_{MEM} \). Now we have a cache, the time taken depends on whether the access was a cache-hit or a cache-miss. But this is not too hard to accommodate: remember we want the mean (or average) access time so our probabilities from above become useful.

Imagine the time taken to access the cache is \( T_{CACHE} \). Obviously an access can either cause a cache-hit or a cache-miss, so the mean access time (the average time taken for such an access) is given in two parts:

- For a cache-hit, the element of \( MEM \) we are interested in is already resident in the cache; we just need to load from or store into it. Since an access is a cache-hit with probability \( R_H \), the time taken on average will be

\[
R_H \cdot T_{CACHE}.
\]

- For a cache-miss, the element of \( MEM \) we are interested in is not resident in the cache; first we just need to fetch it from \( MEM \) into the cache and then load from or store into it once it becomes resident. Since the access is a cache-miss with probability \( R_M \), the time taken on average will be

\[
R_M \cdot (T_{MEM} + T_{CACHE}).
\]

The mean access time is therefore the sum of these, i.e.,

\[
T = R_H \cdot T_{CACHE} + R_M \cdot (T_{MEM} + T_{CACHE}),
\]

or in words: the probability of the access being a cache-hit multiplied by the time taken to deal with a cache-hit, plus the probability of the access being a cache-miss multiplied by the time taken to deal with a cache-miss.

We can do some analysis and see if this makes sense. If the hit ratio tends to 1 (meaning the miss ratio tends to 0), the first term above dominates and the mean access time tends to \( T_{CACHE} \). This is good: if there are lots of cache-hits, \( T \) will be small so the time taken to perform the associated accesses will be small. However, if the hit ratio tends to 0 (meaning the miss ratio tends to 1), the second term above dominates and the mean access time now tends to \( T_{MEM} + T_{CACHE} \). This is bad: if there are lots of cache-misses, \( T \) will be large so the time taken to perform the associated accesses will be large. In fact this is very bad because if the mean access time is \( T_{MEM} + T_{CACHE} \) then clearly this is larger than \( T_{MEM} \) and so we are actually slower than if the cache was removed entirely!

14.2 On straight-line execution

14.2.1 Applying specialisation and loop unrolling

Imagine we write an algorithm with a loop in it; the idea of the algorithm is to set the first \( n \) elements in a sequence called \( A \) to zero:
The steps performed when \texttt{Zero} is invoked are easy to work out. Writing them out in a more verbose way than before, we would do something like the following:

\textbf{Step} #1 Assign \( i \leftarrow 0 \).

\textbf{Step} #2 Since \( i \leq n - 1 \), perform the next loop iteration.

\textbf{Step} #3 Assign \( A_i \leftarrow 0 \), i.e., \( A_0 \leftarrow 0 \).

\textbf{Step} #4 Assign \( i \leftarrow i + 1 \), i.e., \( i \leftarrow 1 \).

\textbf{Step} #5 Since \( i \leq n - 1 \), perform the next loop iteration.

\textbf{Step} #6 Assign \( A_i \leftarrow 0 \), i.e., \( A_1 \leftarrow 0 \).

\textbf{Step} #7 Assign \( i \leftarrow i + 1 \), i.e., \( i \leftarrow 2 \).

\textbf{Step} #8 Since \( i \leq n - 1 \), perform the next loop iteration.

\textbf{Step} #9 Assign \( A_i \leftarrow 0 \), i.e., \( A_2 \leftarrow 0 \).

\textbf{Step} #10 Assign \( i \leftarrow i + 1 \), i.e., \( i \leftarrow 3 \).

\textbf{Step} #11 Since \( i \leq n - 1 \), perform the next loop iteration.

\textbf{Step} #12 Assign \( A_i \leftarrow 0 \), i.e., \( A_3 \leftarrow 0 \).

\textbf{Step} #13 Assign \( i \leftarrow i + 1 \), i.e., \( i \leftarrow 4 \).

\textbf{Step} #14 ... 

Eventually, \( i = n \) and the loop stops. It might seem odd that we do so many steps yet accomplish so little: where have the extra steps come from? The answer is obvious in the sense that the extra steps relate to control of the loop. Although the block we iterate over (i.e., line #3 of the algorithm) needs one step per-iteration, we need an extra two steps to test and increment \( i \). Since we do \( n \) iterations, the number of steps is therefore roughly \( 3n \) rather than just \( n \).

Clearly this is unattractive: the more steps the algorithm takes, the longer it takes to give us a result. This is especially annoying if we know the number of iterations which should be performed before we start. In this situation the loop just acts like a short description for a potentially long sequence of steps. We can resolve this problem by in two steps. First imagine \( n = 4 \), though any value will do. Using this choice, we can specialise the original algorithm:

\begin{verbatim}
algorithm Zero(A) begin
    for i from 0 upto 3 do
        Ai ← 0
    end
    return
end
\end{verbatim}

Notice that now the algorithm is a special-purpose version, the loop now always does \( n = 4 \) iterations. We cannot ask it to do more, or less, by changing \( n \) like we could with the original one: \( n \) is now fixed. On one hand this does not give us any advantage because this loop still needs about \( 3n = 12 \) steps, just like the original one. On the other hand, now it is clear how many iterations the loop should perform, we can unroll it. The idea is to take the block representing the loop body and copy it \( n \) times, each time replacing \( i \) with the right value. The end result is:
14.2.2 Estimating execution time

Looking at the discussion above, the aim of unrolling a loop seems to be an improvement in performance, i.e., a reduction in steps. This is true, but it also gives a second advantage: the fact that the unrolling process yields a straight-line version (i.e., just a sequence of assignments) means we can reason about it more easily.

Imagine we have a straight-line version of some algorithm, and implement it as a program with $m$ instructions in it. We can estimate the execution time of the program using

$$T = m \cdot T_{EXE}$$

where $T_{EXE}$ is a constant used to represent how long a single instruction takes to execute. If we really wanted to be accurate, we might have a different value of $T_{EXE}$ for each type of instruction. This would allow us to capture the idea that some instructions might take longer to execute than others for example.

This is not the end of the story however: remember that some instructions will access memory and when they do so, the cache plays a part. Put more specifically, some memory accesses will provoke cache-hits and others will provoke cache-misses. Imagine there are $n \leq m$ instructions that access memory. Based on what we already know about mean access time for such instructions, we can rewrite our estimate for the time the program takes to execute as

$$T = m \cdot T_{EXE} + n \cdot (R_H \cdot T_{CACHE} + R_M \cdot (T_{MEM} + T_{CACHE})).$$

Now, the total is now the time taken to execute all the instructions plus the overhead of accessing memory: the latter term is simply the mean access time for a single access (which we already worked out) multiplied by $n$, the number of accesses.

A concrete example will probably make this much clearer. Think back to the straight-line version of the Zero algorithm which sets four elements of a sequence to zero, and imagine we implement the corresponding program. Given we are going to execute the program on a real computer, we need to given concrete values to the constants in our estimations. Imagine that $T_{EXE} = 1$, $T_{CACHE} = 5$, $T_{MEM} = 10$ so that accessing the cache takes exactly half the time of memory, and both are quite a bit larger than the time taken to execute instructions. There are five instructions in the program; the first four set elements of the sequence to zero. So using our initial estimate we get

$$T = 5 \cdot T_{EXE} = 5 \cdot 1 = 5$$

But the first four instructions all access memory, so to be more accurate we should use the better estimation method. To do that we need to fix the hit and miss ratio of our cache; imagine $R_H = 0.5$ and therefore $R_M = 0.5$, so a healthy 50% of accesses will cause cache-hits on average. This means

$$T = 5 \cdot T_{EXE} + 4 \cdot (R_H \cdot T_{CACHE} + R_M \cdot (T_{MEM} + T_{CACHE}))$$

$$= 5 \cdot 1 + 4 \cdot (0.5 \cdot 5 + 0.5 \cdot (10 + 5))$$

$$= 45$$

What happens if the cache is designed to be more effective: now $R_H = 0.8$ and therefore $R_M = 0.2$, so now 80% of accesses will cause cache-hits on average. Clearly this alters our estimate

$$T = 5 \cdot T_{EXE} + 4 \cdot (R_H \cdot T_{CACHE} + R_M \cdot (T_{MEM} + T_{CACHE}))$$

$$= 5 \cdot 1 + 4 \cdot (0.8 \cdot 5 + 0.2 \cdot (10 + 5))$$

$$= 33$$

which now only takes four steps to do the same thing!
in that now the program executes more quickly. In fact, the more cache-hits we get, the faster the program will execute. This is good news because this was the whole point of putting it there in the first place! The key thing to notice is that the memory accesses are what cause variation in our straight-line program: the number of instructions in the program is fixed, as are the constants such as $T_{EXE}$, so if the execution time varies we know that cache behaviour is the cause.

14.3 An example block cipher

Discussing an attack without discussing the target of said attack is a bit pointless: we need some cryptography to apply the attack to! Now would be a good time to go back and recap on Chapter 7 and Chapter 10. We need two functions which constitute a block cipher [2]:

- **Enc** encrypts a plaintext message $P$ using a key $K$ to produce a ciphertext $C$, and
- **Dec** decrypts a ciphertext message $C$ using a key $K$ to produce a plaintext $P$.

To make the cipher work, we need to ensure

$$\text{Dec}(K, \text{Enc}(K, P)) = P$$

i.e., Dec is the inverse function [4] of Enc for a given key, but clearly only someone who knows the key $K$ can use either of them.

To specify an example cipher, we need only specify how the Enc and Dec functions work; Figure 14.2 includes two algorithms which do just that. We will focus on Enc: it encrypts a plaintext message $P$ using a key $K$, both of which are just $n$-element sequences of numbers. It certainly has flaws, but one major advantage of this example is that the structure resembles a real block cipher. In particular, it looks similar to the Advanced Encryption Standard (AES) [1] which is now the de facto standard choice made in most real applications. AES is based on a Substitution-Permutation network [7] or “SP-network”: it performs substitutions (i.e., replaces elements with other elements) and permutations (i.e., rearranges elements), and also mix in key material, within each of several repeated rounds. Looking at Figure 14.2a for example, we can identify similar features:

- Lines #3 to #5 mix each element from $P$ with one from $K$, and then replaces each element in $P$ with another element.
- Lines #6 to #10 rearrange the elements in $P$.
- Lines #3 to #10 represent a single round which is applied $r$ times by the outer-loop.

Of course there are massive differences as well, but hopefully this adds at least some authenticity: we have an example which is not too far away from reality.

AES very roughly relates to the use of $n = 16$ and $r = 10$: for a 16-element $P$, it applies 10 rounds of processing using a $K$ of the same size. We will make life significantly easier by using $n = 2$ and $r = 2$ for our example. One advantage of this is that we can draw Figure 14.6 to show how the example cipher works visually. Starting at the top left-hand corner and reading downwards, the diagram applies the steps of Enc; reading from the bottom right-hand corner upwards applies the steps of Dec. Notice how each step of decryption “undoes” the corresponding step of encryption; to explain why this works, we need to explain what the mysterious ET and DT functions are.

14.3.1 Applying the substitution step using an S-box

The substitution steps are particularly important. The idea is that during encryption (resp. decryption) we replace the $j$-th element of $P$ (resp. $C$) by applying a function ET (resp. DT). The two functions are each others inverse so

$$\text{DT}(\text{ET}(x)) = x$$

and

$$\text{ET}(\text{DT}(x)) = x$$

for any $x$. These functions are given a special name; they are called an S-box [6] or “substitution box”. For our purposes it does not matter what what result ET(255) produces for example, the only thing that matters is the functions act as each others inverse. So to continue with our aim of staying as close to AES as we can,
Figure 14.2: An example cipher, somewhat similar to AES, which acts as the target of an example cache-based side-channel attack.

Figure 14.3: The original example cipher specialised and unrolled for $n = 2$ and $r = 2$ to yield simpler descriptions, and now using the ET-TAB and DT-TAB tables rather than the ET and DT functions.
\[\text{ET-TAB} = \begin{array}{cccccccc}
63_{(16)}, & 7C_{(16)}, & 77_{(16)}, & 7B_{(16)}, & F2_{(16)}, & 6B_{(16)}, & 6F_{(16)}, & C5_{(16)} \\
30_{(16)}, & 01_{(16)}, & 67_{(16)}, & 2B_{(16)}, & FF_{(16)}, & D7_{(16)}, & AB_{(16)}, & 76_{(16)} \\
CA_{(16)}, & 82_{(16)}, & C9_{(16)}, & 7D_{(16)}, & FA_{(16)}, & 59_{(16)}, & 47_{(16)}, & F0_{(16)} \\
AD_{(16)}, & D4_{(16)}, & A2_{(16)}, & AF_{(16)}, & 9C_{(16)}, & A4_{(16)}, & 72_{(16)}, & C0_{(16)} \\
B7_{(16)}, & FD_{(16)}, & 93_{(16)}, & 26_{(16)}, & 36_{(16)}, & 3F_{(16)}, & F7_{(16)}, & CC_{(16)} \\
34_{(16)}, & A5_{(16)}, & E5_{(16)}, & F1_{(16)}, & 71_{(16)}, & DB_{(16)}, & 31_{(16)}, & 15_{(16)} \\
04_{(16)}, & C7_{(16)}, & 23_{(16)}, & C3_{(16)}, & 18_{(16)}, & 96_{(16)}, & 05_{(16)}, & 9A_{(16)} \\
07_{(16)}, & 12_{(16)}, & 80_{(16)}, & E2_{(16)}, & EB_{(16)}, & 27_{(16)}, & B2_{(16)}, & 75_{(16)} \\
09_{(16)}, & 83_{(16)}, & 2C_{(16)}, & 1A_{(16)}, & 1B_{(16)}, & 6E_{(16)}, & 5A_{(16)}, & A0_{(16)} \\
52_{(16)}, & 3B_{(16)}, & D6_{(16)}, & B3_{(16)}, & 29_{(16)}, & E3_{(16)}, & 2F_{(16)}, & 84_{(16)} \\
53_{(16)}, & D1_{(16)}, & 00_{(16)}, & ED_{(16)}, & 20_{(16)}, & FC_{(16)}, & B1_{(16)}, & 5B_{(16)} \\
6A_{(16)}, & CB_{(16)}, & BE_{(16)}, & 39_{(16)}, & 4A_{(16)}, & 4C_{(16)}, & 58_{(16)}, & CF_{(16)} \\
DO_{(16)}, & EF_{(16)}, & AA_{(16)}, & FB_{(16)}, & 43_{(16)}, & 4D_{(16)}, & 33_{(16)}, & 85_{(16)} \\
45_{(16)}, & F9_{(16)}, & 02_{(16)}, & 7F_{(16)}, & 50_{(16)}, & 3C_{(16)}, & 9F_{(16)}, & A8_{(16)} \\
51_{(16)}, & A3_{(16)}, & 40_{(16)}, & 8F_{(16)}, & 92_{(16)}, & 9D_{(16)}, & 38_{(16)}, & F5_{(16)} \\
BC_{(16)}, & B6_{(16)}, & DA_{(16)}, & 21_{(16)}, & 10_{(16)}, & FF_{(16)}, & F3_{(16)}, & D2_{(16)} \\
CD_{(16)}, & 0C_{(16)}, & 13_{(16)}, & EC_{(16)}, & 5F_{(16)}, & 97_{(16)}, & 44_{(16)}, & 17_{(16)} \\
C4_{(16)}, & A7_{(16)}, & 7E_{(16)}, & 3D_{(16)}, & 64_{(16)}, & 5D_{(16)}, & 19_{(16)}, & 73_{(16)} \\
60_{(16)}, & 81_{(16)}, & 4F_{(16)}, & DC_{(16)}, & 22_{(16)}, & 2A_{(16)}, & 90_{(16)}, & 88_{(16)} \\
46_{(16)}, & EE_{(16)}, & B8_{(16)}, & 14_{(16)}, & DE_{(16)}, & 5E_{(16)}, & 0B_{(16)}, & DB_{(16)} \\
EO_{(16)}, & 32_{(16)}, & 3A_{(16)}, & 0A_{(16)}, & 49_{(16)}, & 06_{(16)}, & 24_{(16)}, & 5C_{(16)} \\
C2_{(16)}, & D3_{(16)}, & AC_{(16)}, & 62_{(16)}, & 91_{(16)}, & 95_{(16)}, & E4_{(16)}, & 79_{(16)} \\
E7_{(16)}, & C8_{(16)}, & 37_{(16)}, & 6D_{(16)}, & 8D_{(16)}, & D5_{(16)}, & 4E_{(16)}, & A9_{(16)} \\
6C_{(16)}, & 56_{(16)}, & F4_{(16)}, & EA_{(16)}, & 65_{(16)}, & 7A_{(16)}, & AE_{(16)}, & 08_{(16)} \\
BA_{(16)}, & 78_{(16)}, & 25_{(16)}, & 2E_{(16)}, & 1C_{(16)}, & 9A_{(16)}, & 8A_{(16)}, & 5C_{(16)} \\
EB_{(16)}, & DD_{(16)}, & 74_{(16)}, & 1F_{(16)}, & 4B_{(16)}, & BD_{(16)}, & 8R_{(16)}, & 8A_{(16)} \\
70_{(16)}, & 3E_{(16)}, & B7_{(16)}, & 66_{(16)}, & 48_{(16)}, & 03_{(16)}, & F6_{(16)}, & 0E_{(16)} \\
61_{(16)}, & 35_{(16)}, & 57_{(16)}, & B9_{(16)}, & 86_{(16)}, & C1_{(16)}, & 1D_{(16)}, & 9E_{(16)} \\
El_{(16)}, & F8_{(16)}, & 99_{(16)}, & 11_{(16)}, & 69_{(16)}, & D9_{(16)}, & 8E_{(16)}, & 94_{(16)} \\
9B_{(16)}, & 1E_{(16)}, & 87_{(16)}, & E9_{(16)}, & CE_{(16)}, & 55_{(16)}, & 28_{(16)}, & DF_{(16)} \\
8C_{(16)}, & A1_{(16)}, & 89_{(16)}, & 0D_{(16)}, & BF_{(16)}, & E6_{(16)}, & 42_{(16)}, & 68_{(16)} \\
41_{(16)}, & 99_{(16)}, & 2D_{(16)}, & 0F_{(16)}, & B0_{(16)}, & 54_{(16)}, & BB_{(16)}, & 10_{(16)} \}
\]

Figure 14.4: ET-TAB, a tabular description of ET.
DT-Tab = \[
\begin{array}{cccccccccc}
52_{(16)} & 09_{(16)} & 6A_{(16)} & D5_{(16)} & 30_{(16)} & 36_{(16)} & A5_{(16)} & 38_{(16)} \\
BF_{(16)} & 40_{(16)} & A3_{(16)} & 9E_{(16)} & 81_{(16)} & F3_{(16)} & D7_{(16)} & FB_{(16)} \\
7C_{(16)} & E3_{(16)} & 39_{(16)} & 82_{(16)} & 9B_{(16)} & 2F_{(16)} & FF_{(16)} & 87_{(16)} \\
34_{(16)} & 8E_{(16)} & 43_{(16)} & 44_{(16)} & C4_{(16)} & DE_{(16)} & E9_{(16)} & CB_{(16)} \\
54_{(16)} & 7B_{(16)} & 94_{(16)} & 32_{(16)} & A6_{(16)} & C2_{(16)} & 23_{(16)} & 3D_{(16)} \\
EE_{(16)} & 4C_{(16)} & 95_{(16)} & 0B_{(16)} & 42_{(16)} & FA_{(16)} & C3_{(16)} & 4E_{(16)} \\
08_{(16)} & 2E_{(16)} & A1_{(16)} & 66_{(16)} & 28_{(16)} & D9_{(16)} & 24_{(16)} & B2_{(16)} \\
76_{(16)} & 5B_{(16)} & A2_{(16)} & 49_{(16)} & 6D_{(16)} & 8B_{(16)} & D1_{(16)} & 25_{(16)} \\
72_{(16)} & F8_{(16)} & F6_{(16)} & 64_{(16)} & 66_{(16)} & 98_{(16)} & 16_{(16)} & \\
D4_{(16)} & A4_{(16)} & 5C_{(16)} & CC_{(16)} & 5D_{(16)} & 65_{(16)} & B6_{(16)} & 92_{(16)} \\
6C_{(16)} & 70_{(16)} & 48_{(16)} & 50_{(16)} & FD_{(16)} & ED_{(16)} & B9_{(16)} & DA_{(16)} \\
5E_{(16)} & 15_{(16)} & 46_{(16)} & 57_{(16)} & A7_{(16)} & 8D_{(16)} & 9D_{(16)} & 84_{(16)} \\
90_{(16)} & D8_{(16)} & AB_{(16)} & 00_{(16)} & 8C_{(16)} & BC_{(16)} & D3_{(16)} & 0A_{(16)} \\
F7_{(16)} & E4_{(16)} & 58_{(16)} & 05_{(16)} & 88_{(16)} & B3_{(16)} & 45_{(16)} & 06_{(16)} \\
D0_{(16)} & 2C_{(16)} & 1F_{(16)} & 8F_{(16)} & CA_{(16)} & 3F_{(16)} & 0F_{(16)} & 02_{(16)} \\
Cl_{(16)} & AF_{(16)} & BD_{(16)} & 03_{(16)} & 01_{(16)} & 13_{(16)} & 8A_{(16)} & 6B_{(16)} \\
3A_{(16)} & 91_{(16)} & 11_{(16)} & 41_{(16)} & 4F_{(16)} & 67_{(16)} & DC_{(16)} & EA_{(16)} \\
97_{(16)} & F2_{(16)} & CF_{(16)} & CE_{(16)} & F0_{(16)} & B4_{(16)} & E6_{(16)} & 73_{(16)} \\
96_{(16)} & 0C_{(16)} & 74_{(16)} & 22_{(16)} & E7_{(16)} & AD_{(16)} & 35_{(16)} & 85_{(16)} \\
E2_{(16)} & F9_{(16)} & 37_{(16)} & E8_{(16)} & 1C_{(16)} & 75_{(16)} & DF_{(16)} & 6E_{(16)} \\
47_{(16)} & F1_{(16)} & 1A_{(16)} & 71_{(16)} & 1D_{(16)} & 29_{(16)} & C5_{(16)} & 89_{(16)} \\
6F_{(16)} & B7_{(16)} & 62_{(16)} & 0E_{(16)} & AA_{(16)} & 18_{(16)} & BE_{(16)} & 1B_{(16)} \\
FC_{(16)} & 56_{(16)} & 3E_{(16)} & 4B_{(16)} & C6_{(16)} & D2_{(16)} & 79_{(16)} & 20_{(16)} \\
9A_{(16)} & DB_{(16)} & C0_{(16)} & FE_{(16)} & 78_{(16)} & CD_{(16)} & 5A_{(16)} & F4_{(16)} \\
1F_{(16)} & DD_{(16)} & AB_{(16)} & 33_{(16)} & 88_{(16)} & 07_{(16)} & C7_{(16)} & 31_{(16)} \\
B1_{(16)} & 12_{(16)} & 10_{(16)} & 59_{(16)} & 27_{(16)} & 80_{(16)} & EC_{(16)} & 5F_{(16)} \\
60_{(16)} & 51_{(16)} & 7F_{(16)} & A9_{(16)} & 19_{(16)} & B5_{(16)} & 4A_{(16)} & 0B_{(16)} \\
2D_{(16)} & E5_{(16)} & 7A_{(16)} & 9F_{(16)} & 93_{(16)} & C9_{(16)} & 9C_{(16)} & EF_{(16)} \\
A0_{(16)} & E0_{(16)} & 38_{(16)} & 4D_{(16)} & AE_{(16)} & 2A_{(16)} & F5_{(16)} & B0_{(16)} \\
Cb_{(16)} & EB_{(16)} & BB_{(16)} & 3C_{(16)} & 83_{(16)} & 53_{(16)} & 99_{(16)} & 61_{(16)} \\
17_{(16)} & 2B_{(16)} & 04_{(16)} & 7E_{(16)} & BA_{(16)} & 77_{(16)} & D6_{(16)} & 26_{(16)} \\
E1_{(16)} & 69_{(16)} & 14_{(16)} & 63_{(16)} & 55_{(16)} & 21_{(16)} & 0C_{(16)} & 7D_{(16)} \\
\end{array}
\]
Figure 14.6: A diagrammatic description of \( \text{Enc} \) and \( \text{Dec} \) invoked using \( n = 2 \) and \( r = 2 \); the left-hand portion represents the encryption of the plaintext message \( P \) using the key \( K \), while the right-hand portion represents decryption of the resulting ciphertext \( C \).
we borrow the AES S-box. There are quite complicated ways to describe the AES S-box, but a much easier way is to simply write out all the entries. That is, we write out two sequences (or rather tables) where

\[
\begin{align*}
ET(x) &= ET_{\text{TAB}_x} \\
DT(x) &= DT_{\text{TAB}_x}
\end{align*}
\]

meaning each table captures the result of the corresponding function for all possible inputs. Figure 14.4 and Figure 14.5 show the table content. Looking at the entries, notice for example that

\[
\begin{align*}
ET(255_{(10)}) &= ET_{255_{(30)}} = 16_{(16)} = 22_{(10)} \\
DT(22_{(10)}) &= DT_{22_{(30)}} = FF_{(16)} = 255_{(10)}
\end{align*}
\]

i.e., the tables demonstrate the inverse nature of ET and DT. Beyond this, their descriptions give us two significant advantages:

1. Using the ET and DT functions implies extra computation; by using the ET-TAB and DT-TAB tables, we instead store all possible results for ET and DT then just look them up when we need to. There is no magic going on: there is just no point repeatedly invoking ET and DT given their result is fixed for a given input, and there is a fairly small number of inputs. This is a standard approach called \textit{pre-computation}, i.e., “computation performed before we start”: one one hand it make things more efficient in terms of time, but the trade-off is the space required to store the tables.

2. We can actually perform an example encryption and decryption. If we select

\[
K = \langle 11_{(16)}, 22_{(16)} \rangle
\]

and

\[
P = \langle 33_{(16)}, 44_{(16)} \rangle
\]

say, we can invoke \texttt{Enc} to perform an encryption

\textbf{Step} #1 Assign \(P_0 \leftarrow ET_{\text{TAB}_{P_0 \oplus K_0}} = ET_{\text{TAB}_{33_{(16)} \oplus 11_{(16)}}} = 93_{(16)}\).

\textbf{Step} #2 Assign \(P_1 \leftarrow ET_{\text{TAB}_{P_1 \oplus K_1}} = ET_{\text{TAB}_{44_{(16)} \oplus 22_{(16)}}} = 33_{(16)}\).

\textbf{Step} #3 Assign \(t \leftarrow P_0 = 93_{(16)}, P_0 \leftarrow P_1 = 33_{(16)}, P_1 \leftarrow t = 93_{(16)}\).

\textbf{Step} #4 Assign \(P_0 \leftarrow ET_{\text{TAB}_{P_0 \oplus K_0}} = ET_{\text{TAB}_{33_{(16)} \oplus 11_{(16)}}} = 93_{(16)}\).

\textbf{Step} #5 Assign \(P_1 \leftarrow ET_{\text{TAB}_{P_1 \oplus K_1}} = ET_{\text{TAB}_{44_{(16)} \oplus 22_{(16)}}} = C8_{(16)}\).

\textbf{Step} #6 Assign \(t \leftarrow P_0 = 93_{(16)}, P_0 \leftarrow P_1 = C8_{(16)}, P_1 \leftarrow t = 93_{(16)}\).

\textbf{Step} #7 Return \(P = \langle C8_{(16)}, 93_{(16)} \rangle\).

that produces the ciphertext

\[
\langle C8_{(16)}, 93_{(16)} \rangle.
\]

Then, we can invoke \texttt{Dec} to perform the corresponding decryption

\textbf{Step} #1 Assign \(t \leftarrow C_1 = 93_{(16)}, C_1 \leftarrow C_0 = C8_{(16)}, C_0 \leftarrow t = 93_{(16)}\).

\textbf{Step} #2 Assign \(C_0 \leftarrow DT_{\text{TAB}_{C_0 \oplus K_0}} = DT_{\text{TAB}_{93_{(36)} \oplus 11_{(16)}}} = 33_{(16)}\).

\textbf{Step} #3 Assign \(C_1 \leftarrow DT_{\text{TAB}_{C_1 \oplus K_1}} = DT_{\text{TAB}_{C8_{(36)} \oplus 22_{(16)}}} = 93_{(16)}\).

\textbf{Step} #4 Assign \(t \leftarrow C_1 = 93_{(16)}, C_1 \leftarrow C_0 = 33_{(16)}, C_0 \leftarrow t = 93_{(16)}\).

\textbf{Step} #5 Assign \(C_0 \leftarrow DT_{\text{TAB}_{C_0 \oplus K_0}} = DT_{\text{TAB}_{93_{(36)} \oplus 11_{(16)}}} = 11_{(16)}\).

\textbf{Step} #6 Assign \(C_1 \leftarrow DT_{\text{TAB}_{C_1 \oplus K_1}} = DT_{\text{TAB}_{33_{(36)} \oplus 22_{(16)}}} = 22_{(16)}\).

\textbf{Step} #7 Return \(C = \langle 11_{(16)}, 22_{(16)} \rangle\).

and recover the same plaintext as we started with, i.e.,

\[
\langle 11_{(16)}, 22_{(16)} \rangle.
\]
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$A \leftarrow \text{MEM}_{40}$</td>
<td>load $P_0$</td>
</tr>
<tr>
<td>1</td>
<td>$A \leftarrow A \oplus \text{MEM}_{37}$</td>
<td>XOR $P_0$ and $K_0$</td>
</tr>
<tr>
<td>2</td>
<td>$A \leftarrow A + \text{MEM}_{42}$</td>
<td>compute “ET-Tab access” instruction</td>
</tr>
<tr>
<td>3</td>
<td>$\text{MEM}_{4} \leftarrow A$</td>
<td>store “ET-Tab access” instruction</td>
</tr>
<tr>
<td>4</td>
<td>NOP</td>
<td>no operation</td>
</tr>
<tr>
<td>5</td>
<td>$\text{MEM}_{40} \leftarrow A$</td>
<td>store $P_0$</td>
</tr>
<tr>
<td>6</td>
<td>$A \leftarrow \text{MEM}_{41}$</td>
<td>load $P_1$</td>
</tr>
<tr>
<td>7</td>
<td>$A \leftarrow A \oplus \text{MEM}_{38}$</td>
<td>XOR $P_1$ and $K_1$</td>
</tr>
<tr>
<td>8</td>
<td>$A \leftarrow A + \text{MEM}_{42}$</td>
<td>compute “ET-Tab access” instruction</td>
</tr>
<tr>
<td>9</td>
<td>$\text{MEM}_{10} \leftarrow A$</td>
<td>store “ET-Tab access” instruction</td>
</tr>
<tr>
<td>10</td>
<td>NOP</td>
<td>no operation</td>
</tr>
<tr>
<td>11</td>
<td>$\text{MEM}_{41} \leftarrow A$</td>
<td>store $P_1$</td>
</tr>
<tr>
<td>12</td>
<td>$A \leftarrow \text{MEM}_{40}$</td>
<td>load $P_0$</td>
</tr>
<tr>
<td>13</td>
<td>$\text{MEM}_{39} \leftarrow A$</td>
<td>store $t$</td>
</tr>
<tr>
<td>14</td>
<td>$A \leftarrow \text{MEM}_{41}$</td>
<td>load $P_1$</td>
</tr>
<tr>
<td>15</td>
<td>$\text{MEM}_{40} \leftarrow A$</td>
<td>store $P_0$</td>
</tr>
<tr>
<td>16</td>
<td>$A \leftarrow \text{MEM}_{39}$</td>
<td>load $t$</td>
</tr>
<tr>
<td>17</td>
<td>$\text{MEM}_{41} \leftarrow A$</td>
<td>store $P_1$</td>
</tr>
<tr>
<td>18</td>
<td>$A \leftarrow \text{MEM}_{40}$</td>
<td>load $P_0$</td>
</tr>
<tr>
<td>19</td>
<td>$A \leftarrow A \oplus \text{MEM}_{37}$</td>
<td>XOR $P_0$ and $K_0$</td>
</tr>
<tr>
<td>20</td>
<td>$A \leftarrow A + \text{MEM}_{42}$</td>
<td>compute “ET-Tab access” instruction</td>
</tr>
<tr>
<td>21</td>
<td>$\text{MEM}_{22} \leftarrow A$</td>
<td>store “ET-Tab access” instruction</td>
</tr>
<tr>
<td>22</td>
<td>NOP</td>
<td>no operation</td>
</tr>
<tr>
<td>23</td>
<td>$\text{MEM}_{40} \leftarrow A$</td>
<td>store $P_0$</td>
</tr>
<tr>
<td>24</td>
<td>$A \leftarrow \text{MEM}_{41}$</td>
<td>load $P_1$</td>
</tr>
<tr>
<td>25</td>
<td>$A \leftarrow A \oplus \text{MEM}_{38}$</td>
<td>XOR $P_1$ and $K_1$</td>
</tr>
<tr>
<td>26</td>
<td>$A \leftarrow A + \text{MEM}_{42}$</td>
<td>compute “ET-Tab access” instruction</td>
</tr>
<tr>
<td>27</td>
<td>$\text{MEM}_{28} \leftarrow A$</td>
<td>store “ET-Tab access” instruction</td>
</tr>
<tr>
<td>28</td>
<td>NOP</td>
<td>no operation</td>
</tr>
<tr>
<td>29</td>
<td>$\text{MEM}_{41} \leftarrow A$</td>
<td>store $P_1$</td>
</tr>
<tr>
<td>30</td>
<td>$A \leftarrow \text{MEM}_{40}$</td>
<td>load $P_0$</td>
</tr>
<tr>
<td>31</td>
<td>$\text{MEM}_{39} \leftarrow A$</td>
<td>store $t$</td>
</tr>
<tr>
<td>32</td>
<td>$A \leftarrow \text{MEM}_{41}$</td>
<td>load $P_1$</td>
</tr>
<tr>
<td>33</td>
<td>$\text{MEM}_{40} \leftarrow A$</td>
<td>store $P_0$</td>
</tr>
<tr>
<td>34</td>
<td>$A \leftarrow \text{MEM}_{39}$</td>
<td>load $t$</td>
</tr>
<tr>
<td>35</td>
<td>$\text{MEM}_{41} \leftarrow A$</td>
<td>store $P_1$</td>
</tr>
<tr>
<td>36</td>
<td>$\text{HALT}$</td>
<td>halt</td>
</tr>
<tr>
<td>37</td>
<td>NOP</td>
<td>$K_0$</td>
</tr>
<tr>
<td>38</td>
<td>NOP</td>
<td>$K_1$</td>
</tr>
<tr>
<td>39</td>
<td>NOP</td>
<td>$t$</td>
</tr>
<tr>
<td>40</td>
<td>NOP</td>
<td>$P_0$</td>
</tr>
<tr>
<td>41</td>
<td>NOP</td>
<td>$P_1$</td>
</tr>
<tr>
<td>42</td>
<td>$A \leftarrow \text{MEM}_{43}$</td>
<td>“ET-Tab access” instruction</td>
</tr>
<tr>
<td>43</td>
<td>NOP</td>
<td>ET-Tab0, i.e., ET(0)</td>
</tr>
</tbody>
</table>

Figure 14.7: A partial (i.e., omitting most of the ET-Tab table) implementation of the Enc algorithm using the example computer.
14.3.2 Implementing a simpler, straight-line version of the cipher

The example encryption and decryption above illustrate an important (and hopefully somewhat familiar) fact: once we select values for \(n\) and \(r\), the loops basically disappear and the example cipher just performs a sequence of assignments. Now we have settled on \(n = 2\) and \(r = 2\), we can apply the same approach as previously and write out straight-line versions of the \(E\) and \(D\) algorithms; the end result is shown in Figure 14.3.

Given that the algorithms are now so simple, we can actually go one better by translating the new straight-line version of \(E\) into a program for our example computer from Chapter 4. The implementation is shown in Figure 14.7 and is quite long in comparison to some others we have looked at. On the other hand, it is easy to explain step-by-step:

- Addresses #37 and #38 hold \(K_0\) and \(K_1\), (i.e., the key); addresses #40 and #41 hold \(P_0\) and \(P_1\), (i.e., the plaintext message). Addresses #43 onwards hold the entries of the S-box (i.e., the ET-Tb table) but to save space, these are not repeated: just imagine the content replicates Figure 14.4.
- The instructions in addresses #0 to #5 of mix together \(K_0\) and \(P_0\) the apply the substitution step using the S-box. This is actually quite tricky because we have no clear way to load values from the ET-Tb table. One neat solution is to make constructive use of self-modifying code from Chapter 4. Since the ET-Tb table starts at address #43, we ideally want to do something like
  \[ A \leftarrow \text{MEM}_{A+43}. \]
  This would allow us to compute \(P_0 \oplus K_0\) into \(A\), then use it as an offset from the start of the table to perform the look-up. But we cannot do this; there is no appropriate instruction. So instead, we “make” one ourselves: the idea is to take the value \(P_0 \oplus K_0\) and add it to another value so as to produce the instruction we want as a result.
  This is better explained using an example. Imagine that \(P_0 \oplus K_0 = 51 \oplus 17 = 34\) and that we store this in \(A\) via the instruction at address #1. Next we add the value at address #42, i.e., compute
  \[ 34 + 220043 = 220077 \implies A \leftarrow \text{MEM}_{77}. \]
  So what we have is an instruction that loads from address #77 which just so happens to be the 34-th element of the ET-Tb table! We store this instruction at address #4, which is executed in the next step: we load \(\text{MEM}_{77} = 147\) and use the result to replace \(P_0\).
- The instructions in addresses #6 to #11 do the same thing as addresses #0 to #5, but for \(K_1\) and \(P_1\) rather than \(K_0\) and \(P_0\).
- The instructions in addresses #12 to #17 implement the permutation step, swapping \(P_0\) and \(P_1\) using the temporary value \(t\) which is held in address #39.
- The instructions in addresses #18 to #35 apply the second round, and basically just repeat addresses #0 to #17 again; address #36 is where execution halts.

The crucial thing to realise is that the program has 37 instructions it in; it takes 112 steps to execute on our example computer (a fetch, decode and execute step for each instruction plus one extra to reset it at the beginning) regardless of \(K\) and \(P\). However, some of those 37 instructions access memory. Remember that we are focusing on loads only: there are 22 in total including the ones we put there via self-modification. Based on what we have previously looked at, any variation in how long the program takes to execute is clearly down to how long these loads take.

Showing this in detail is a little laborious, but imagine we select
\[
K = \langle 11_{(16)}, 22_{(16)} \rangle
\]
and
\[
P = \langle 33_{(16)}, 44_{(16)} \rangle
\]
then execute the program on the example computer: it is equipped with the same cache as discussed previously, i.e., with \(l = 8\) and \(w = 4\) so the cache has eight lines, each of which can accommodate four
sub-words. The 22 loads cause 16 cache-hits and 6 cache-misses in this case. If we use the same constants as previously, i.e.,

\[
T_{\text{EXE}} = 1 \\
T_{\text{CACHE}} = 5 \\
T_{\text{MEM}} = 10
\]

and apply the same sort of estimation method, we get

\[
T = 37 \cdot T_{\text{EXE}} + 16 \cdot T_{\text{CACHE}} + 6 \cdot (T_{\text{MEM}} + T_{\text{CACHE}}) \\
= 37 \cdot 1 + 16 \cdot 5 + 6 \cdot (10 + 5) \\
= 207
\]

as an estimate for the execution time. The point is that if we select

\[K = \langle00_{(16)}, 00_{(16)}\rangle\]

and

\[P = \langle00_{(16)}, 00_{(16)}\rangle\]

instead, then the 22 loads cause 17 cache-hits and 5 cache-misses. This means the execution time is less:

\[
T = 37 \cdot T_{\text{EXE}} + 17 \cdot T_{\text{CACHE}} + 5 \cdot (T_{\text{MEM}} + T_{\text{CACHE}}) \\
= 37 \cdot 1 + 17 \cdot 5 + 5 \cdot (10 + 5) \\
= 197
\]

14.4 **A side-channel attack based on cache behaviour**

Consider a similar problem to Chapter 12: some remote computer \(C\) has a key \(K\) embedded inside it, and our task as the attacker Eve is to guess \(K\). We can model what is going on using a similar diagram as before:

![Diagram](http://www.cs.bris.ac.uk)

So basically we send a plaintext message \(P\) (which is public since we know it) to \(C\), and get the ciphertext \(C\) sent back; as well as \(C\), we can measure how long the encryption takes. Of course the ET-Tab table is stored in the memory of \(C\) so that it can apply the substitution step during encryption. When the cipher loads elements from the table, it does so using addresses derived from \(K\) and \(P\); as a result, the cache behaviour that results is partly dictated by \(K\). Sometimes there will be lots of cache-hits, other times lots of cache-misses, but in either case the behaviour is in part due to \(K\): if we time how long the cipher takes to execute, we learn something about \(K\) because the only variation in execution time will be because of the cache behaviour. Neat huh? Or maybe not so neat if it is your \(K\) we are talking about.

14.4.1 **A brute-force attack**

\(K\) is an \(n\)-element sequence: if each element, i.e., each \(K_i\), is an 8-bit value then we know there must be \(2^{8 \cdot n}\) possible values for \(K\). As such, a brute-force attack would proceed as follows:

1. Pick a single random plaintext message \(P\) (any one will do), and send it to \(C\) to get the corresponding ciphertext

\[C = \text{Enc}(K, P)\]

2. For each possible key \(K'\), compute

\[P' = \text{Dec}(K', C)\]

and then check whether \(P' = P\). If the two match, we know that \(K' = K\) and we have recovered the key.
The first step performs 1 encryption, while the second step performs $2^8 = 2^{16} = 65536$ encryptions in the worst case. Our goal therefore is to beat the brute-force attack by recovering $K$ using less than $1 + 65536 = 65537$ encryptions.

As an aside, you could argue that the second step is less important than the first: we can do this offline at our leisure rather than online, having to interact with $C$.

### 14.4.2 A side-channel attack

Putting together all the background material allows demonstration of a better attack that uses the side-channel information available to us: we have a cipher (with $n = 2$ and $r = 2$) implemented on a computer, and the computer is equipped with a cache (again with $l = 8$ and $w = 4$ as before). Look at the first statements in our straight-line version of the $Enc$ algorithm in Figure 14.3 which correspond to the instructions in addresses #0...#11 within the program in Figure 14.7. Basically we are interesting in this fragment

\[
\begin{align*}
  \vdots \\
  P_0 &\leftarrow ET-TAB_{P_0 \oplus K_0} \\
  P_1 &\leftarrow ET-TAB_{P_1 \oplus K_1} \\
  \vdots
\end{align*}
\]

which includes two accesses to the ET-TAB table. If the second access causes a cache-hit, we know that the address used (i.e., the index into the ET-TAB table) must match the address used by the first access: this is what a cache-hit means. So essentially we are allowed to write something like

\[P_0 \oplus K_0 = P_1 \oplus K_1.\]

But this is not quite true: while the addresses match according to the rules of the cache, they are not equal. Why not? Remember there is more than one sub-word in each cache line; a match only means the addresses map to the same cache line not that they access the same sub-word. Another way to say the same thing is that the addresses are equal if we compare the “sub-wordless” versions. So instead we write

\[P_0 \oplus K_0 \equiv P_1 \oplus K_1,\]

meaning the left-hand side is equivalent to the right-hand side if we convert both into their sub-wordless form. Even so, this is quite useful: we know $P_0$ and $P_1$, since we decided on their value. In reality this means we can write

\[P_0 \oplus P_1 \equiv K_0 \oplus K_1,\]

where we know the left-hand side but not the right-hand side. Technically speaking, this is a difference between $K_0$ and $K_1$: we know neither $K_0$ nor $K_1$, but do know a relationship between the two. Put another way, if we guess $K_0$ then we can compute

\[K_1 \equiv P_0 \oplus P_1 \oplus K_0,\]

without having to guess this part of the key independently.

Of course the challenge is now to first acquire and then capitalise on this difference. Imagine that

\[K = (11_{(16)}, 22_{(16)})\]

but remember that as the attacker we obviously do not know this. The attack consists of two phases which are described below.

#### 14.4.2.1 Phase #1: acquisition

In the first phase, we try to acquire the difference (or at least a small set of candidates).

1. Set $f = 0$ and send every special plaintext message of the form

\[P = (f, p)\]

for every possible value $p \in \{0, 1, \ldots, 255\}$, to $C$. We throw away the corresponding ciphertexts but capture the execution time and plot it as the graph; this one is shown at the top of Figure 14.8.
Figure 14.8: Graphs showing variation in the execution time caused by different plaintexts of the form $P = \langle f, p \rangle$ for fixed $f$ and variable $p$. 
The plot illustrates an interesting feature: some choices of \( P \) (i.e., choices of \( p \)) mean a lower execution time than others. Most have an execution time of 217, but a few, i.e., those \( p \) in the set
\[
A = \{10, 16, 20, 34, 48, 49, 51, 54, 72, 149, 167, 193, 215, 235, 247\},
\]
have a particularly low execution time of 207. Each \( p \) gives us a candidate difference: remember we can write
\[
P_0 \oplus p_1 = K_0 \oplus K_1
\]
because if the execution time is low, there are many cache-hits. If we fill in what we know, we can calculate each difference as
\[
f \oplus p = K_0 \oplus K_1.
\]
But remember we only really care about the sub-wordless versions of these differences, so actually we can transform \( A \) into
\[
A_{\text{sub-wordless}} = \{8, 16, 20, 24, 32, 48, 52, 72, 148, 164, 192, 212, 232, 244\}.
\]
2. Repeat the first step again, but this time select \( f = 240 \); this yields another set
\[
B = \{10, 16, 34, 53, 85, 149, 167, 192, 195, 198, 215, 235\}
\]
from the middle graph in Figure 14.8, and hence
\[
B_{\text{sub-wordless}} = \{24, 36, 48, 52, 84, 100, 164, 196, 208, 224, 248\}.
\]
3. Repeat the first step again, but this time select \( f = 192 \); this yields another set
\[
\]
from the bottom graph in Figure 14.8, and hence
\[
C_{\text{sub-wordless}} = \{0, 20, 40, 48, 72, 84, 100, 200, 208, 224, 228\}.
\]
Finally the magic happens: notice that
\[
D = A_{\text{sub-wordless}} \cap B_{\text{sub-wordless}} \cap C_{\text{sub-wordless}} = \{48\}
\]
meaning that a single difference, i.e.,
\[
48 \equiv K_0 \oplus K_1
\]
agrees in all three cases. Sometimes this does not work out so well and instead of a single difference we are forced to work with a set of candidates. This does not matter too much: the next phase is the same if we have one candidate or many, having one just means we do less work.

14.4.2.2 Phase #2: analysis

Armed with \( D \) (which for us is just a single candidate difference) we can now perform a “reduced” brute-force search that will recover \( K \) for us:

1. Pick a single random plaintext message \( P \) (any one will do), and send it to \( C \) to get the corresponding ciphertext
\[
C = \text{Enc}(K, P).
\]
2. For every special key of the form
\[
K' = \langle k, (k \oplus d) + i \rangle
\]
for \( k \in \{0, 1, \ldots, 255\}, d \in D \) and \( i \in \{0, 1, \ldots, w - 1\} \), compute
\[
P' = \text{Dec}(K', C)
\]
and then check whether \( P' = P \). If the two match, we know that \( K' = K \) and we have recovered the key. For our example, we succeed when \( k = 17, d = 48 \) and \( i = 1 \) meaning that
\[
K' = \langle 17, (17 \oplus 48) + 1 \rangle = \langle 17, 34 \rangle = K.
\]
Notice that the expression \((k \oplus d) + i\) is a bit like what we did in the LOAD algorithm when fetching a cache line from MEM: basically we have a sub-wordless address computed by fixing up \( k \) using the difference \( d \) being considered, and then cycle through all the addresses in the cache line \( k \oplus d \) maps into.
14.4.2.3 The end result

How many encryptions do we perform? Consider the two phases:

- The acquisition phase timed the execution of $2^8 = 256$ special plaintexts, repeating the process three times and thus performing $3 \cdot 256 = 768$ encryptions.

- The analysis phase is somewhat similar to the brute-force attack: the first step performs 1 encryption, while since we had just one difference in $D$ the second step performs $256 \cdot 1 \cdot w = 1024$ encryptions in the worst case.

In total, the side-channel attack therefore performed $768 + 1 + 1024 = 1793$ encryptions: this is significantly less than the 65537 performed by the brute-force attack. On the other hand, this time we needed $1 + 768$ online encryptions rather than 1, and so need to interact with $C$ more often.

If you look back at all the simplifications we have made (e.g., the cache design, the cipher design etc.) there is a temptation to say “so what”. The point is that although this is a simplification, real attacks on AES exist and really do follow similar reasoning: they are more complicated, but they also start to extract a bigger advantage. Remember that very roughly, AES use $n = 16$. A brute-force attack would therefore perform an unfeasibly large

$$2^8 \cdot 16 = 2^{128} = 340282366920938463463374407431768211456$$

encryptions to recover $K$. Modern cache based side-channel attacks on AES perform somewhat less than

$$2^{32} = 4294967296$$

encryptions: this is very feasible, and hence a much more clear threat.

Research (task #57) In Chapter 12, we explored a range of countermeasures designed to prevent (with varying degrees of success) a side-channel attack based on leakage of execution time from Match-Pwd. Of course we could do a similar thing here: can you think of how similar concepts could be applied to our block cipher ENC (keeping in mind that the combination of program and computer, or cache at least, are now to blame)?
BIBLIOGRAPHY

Part V

Appendices
APPENDIX

A

A SHORT BASH TUTORIAL

Throughout this book, we have tried to use examples that can be typed into a computer and experimented with. The argument for adopting this approach should be obvious: actively doing things, rather than simply reading about them, offers a more interesting way to learn plus allows more in-depth investigation beyond the limited examples presented in print. A number of challenges complicate matters however. First, a vast number of different types (and configurations) of computer and operating system are available; meaning a single set of examples will, inevitably, be less than ideal for one set of people or another. Second, we need to find a careful balance between the advantages above and some potential disadvantages. Specifically, if we assume too much background knowledge or introduce too many new things to remember, there is a danger this confuses you or even discourages you. Much the same could be said of the Mathematics used, where we intentionally limited the notation for example.

To cope with these problems, we make two compromises. First, we focus the examples on use of a UNIX-based [25] operating system via the BASH shell [1]. Such a platform is provided by a variety of freely available options:

1. Use of a Linux (of whatever flavour or origin [7]) Live CD [12] perhaps represents the best choice: this allows use of a working operating system at no cost (bar the download) and without needing to actually install it. Virtually all distributions now offer a Live CD, with Ubuntu

   http://www.ubuntu.com/

   representing a popular and easy to use example

2. The fantastic Raspberry Pi project


   has produced low-cost, complete computer (and associated Linux distribution) which is ideal for the examples in this book: beyond this, it has huge potential for exploring Computer Science more generally.

3. Cygwin [5] is a system that allows use of BASH on other operating systems, specifically versions of Windows:

   http://www.cygwin.com/

The default installation of Cygwin provides the BASH shell itself, plus a basic set of commands; these alone are enough to support most of the examples.
A complete BASH reference guide is beyond our scope; this could easily fill another book entirely! There are plenty of such resources available, for example a free online books relating to the use of BASH and UNIX can be found at

http://en.wikibooks.org/wiki/Bourne_Shell_Scripting/

and


respectively, with the Computing At School (CAS) backed Raspberry Pi Educational Manual also containing highly relevant content. To address the second challenge above however, this Appendix presents an easy to digest (but more limited) introduction to the main concepts and commands used within the examples. We have tried to make it quite short: the goal is simply to demonstrate what each command does and how it can be used, not give a definitive list of every possible option.

To get the most out of what follows, the recommended approach is to first read through Section B.1 which introduces important background concepts and terminology. Then, Section A.2 should be used as a reference when or if you need to, rather than read from start to finish: it probably makes sense simply to refer to this latter section when you encounter an unfamiliar command elsewhere.

A.1 Some basic, background concepts and terminology

The natural place to start is from first principles, assuming little or no prior knowledge. As such, the goal of what follows is to define just enough concepts and terminology to support the rest of this Chapter. Some aspects of this might seem familiar, others overly simple or obvious: keep in mind that the Chapter overall is introductory, and intended to be a tool to explore the examples elsewhere rather than a definitive guide.

- A physical computer is comprised of many hardware components. These will almost certainly include a Central Processing Unit (CPU) [3], or processor, some short-term storage in the form of Random Access Memory (RAM) [16], and some long(er)-term storage such as a hard disk [9].

- The general-purpose hardware executes software programs, represented by a static set of instructions. When executed, the program becomes an separate, active instance of what is termed a process [15]. The concept of a process includes, for example, an environment in which the program executes; very abstractly, you could think of a program as a sort of template, and execution as taking the template and “using” it to form a concrete process.

  This means that several distinct processes could be executed based on the same program, each with their own distinct execution context. If you execute a web-browser three times for example, there is only one web-browser program but three separate processes representing each execution: what you do with one of them does not necessarily affect the others.

- A special, privileged program called the Operating System (OS) manages the underlying hardware and the processes executing on it. The OS is often described as the combination of

  1. a kernel [11], which represents the innards of the OS, and
  2. a shell [19], which is how users interact with the kernel and hence the computer as a whole.

Put another way, the shell is basically a user interface while the kernel is tasked with duties such as (but certainly not limited to)

- responding to commands issued by the user via the shell (e.g., in order to execute a given program),
- ensuring each resulting process has access to resources it needs to execute (e.g., a large enough, distinct allocation of memory),
- management of multi-tasking [13], so that more than one process can be executed concurrently,
- performing operations, or system calls [22] on behalf of processes (e.g., perform low-level operations using the hard disk, or reading and writing data to and from the network).

Depending on the type of shell, some commands might be real programs and some might be provided by the shell itself; these are sometimes called a built-in [20], but we ignore any distinction otherwise.
<table>
<thead>
<tr>
<th>GUI</th>
<th>CLI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability</strong></td>
<td>A GUI is usually carefully designed, with the resulting user interface supporting interaction via mechanisms and metaphors that are easy to understand and learn. As such, a GUI will be easy to use because usability is a specific, central design goal; this is often further enhanced by mechanisms to support disabled users (e.g., visual impairment).</td>
</tr>
<tr>
<td><strong>Resource</strong></td>
<td>Although drawing windows and icons on the screen is not too demanding in comparison to some tasks a CPU has to perform, modern GUIs typically support complex visual effects (often by using dedicated 3D graphics hardware) and need to cope with a range of input and output peripheral. As a result, one could arguably characterise the computational demands as significant (relatively speaking).</td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td>The act of moving a cursor using a mouse, and clicking on icons and menu items, takes time: even proficient users are limited, physically, to how quickly they can operate a GUI-based user interface. Consider a task such as deleting a file for example: a typical GUI might require you to open a window to browse the hard disk, locate and select the file, then navigate a menu to finally select an item to delete the file.</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>The carefully designed user experience underpinning GUI-based interaction is a clear advantage on one hand. However, even minor deviation from the design can have an impact: as a result, control over how a GUI works is often limited. Further, it is more common for a GUI to force abstractions of lower-level concepts on the user: one seldom has direct control over the file system for example, with the GUI providing intermediate services such as a recycle bin. Meanwhile, desktop shortcuts, and similar.</td>
</tr>
<tr>
<td><strong>Automation</strong></td>
<td>There are ways to automate the use of a GUI: a common approach is the use of a macro recorder, which tracks actions performed by a user and is then able to replay them (e.g., to automatically repeat some task a number of times). The Apple Automator application for OS X represents an example. Beyond this, dedicated programming languages also exist, including Rexx and the Visual Basic for Applications (VBA) framework for controlling Microsoft Office applications.</td>
</tr>
</tbody>
</table>

| Usability | Usability is not usually a central design goal for a CLI: more often a compromise is made so that the CLI is harder to use (especially for non-experts). For example, it is usually harder to remember and use text-based commands without error than to perform the same task by clicking on an icon. |
| Resource | In contrast, the computational requirements for a CLI are typically very light: displaying text requires little effort. If a computer is being used remotely (e.g., administration of a server, or use of an embedded platforms via serial interface), CLIs can arguably cope better with any communication constraints (e.g., limited bandwidth) as a result. |
| Speed | Depending on typing speed, use of a CLI is often much quicker. Even short text-based commands, when correctly used, can quickly invoke relatively complex behaviour: the command to delete a file might be just a few characters for example. On the other hand, that command has to be recalled and typed perfectly for this advantage to be capitalised upon. |
| Control | A CLI more often exposes lower-level concepts directly to the user. The commands available are often little more than wrappers around system calls, meaning closer and direct interaction with the OS kernel (and associated resources). This gives a much greater degree of control, but on the other hand implies a need to know how that control can and/or should be used: lacking the safety net a GUI often offers, incorrect use of a lower-level command could arguably have more dire consequences for example! |
| Automation | Since a CLI demands the precise entry of text-based commands, allowing their combination into a sequence of commands (or script [18]) is fairly natural and allows a high degree of automation. Such scripts can reasonably be viewed as programs; the csh (the C shell), an alternative to BASH, makes this connection explicit by allowing users to write and script commands in a form much like the C programming language [2]. |

---

a The first generation of Apple Macintosh computers could already do this proficiently circa 1984, using only a 8 MHz, 16-bit Motorola 68000 CPU and 64 kB of RAM: the comparison with a more recent computer housing a 1 GHz, 64-bit Intel Core2 CPU and 1 GB of RAM is quite stark.  

b GUI-based remote access is of course possible, and increasingly available. The client-server model used by X11 [26], a framework for GUIs on a given UNIX-based OS, means this facility is inherent; likewise, a range of remote desktop applications now exist for Windows.  

c A keyboard short-cut can make this easier of course, but this starts to blur the boundary between what a GUI or CLI actually is: one could cast a keyboard short-cut mechanism within a GUI as a form of CLI for example.  

**Figure A.1:** A rough, and in places subjective comparison (using only a few criteria) between GUI- and CLI-based user interfaces.
The execution of a process is said to terminate (or finish) either

1. normally (e.g., the computation it was executed to perform is complete),
2. abnormally (e.g., due to an error during the computation), or
3. forcibly (e.g., the user decides the computation is not required).

Arguably the most common way to interact with modern computers is via a Graphical User Interface (GUI) [8]. Their exact form and function might differ, but the basic idea is to offer an analogue to how humans interact with the world around them: using a mouse to point and click on icons and therefore issue commands is analogous to seeing a physical button and pressing it with a finger for example. In part, this style of interaction is motivated by the types of computer, user and also program that are most common. For example mobile computers with touch screens lend themselves better to use of a GUI than typed input (due to the lack of physical keyboard), non-expert users can quickly understand metaphors presented by icons (rather than remember text-based commands), and some of the most popular classes of application are naturally suited to graphical presentation of the data they process (web-browsing and word processing are central examples). Such advantages are evident from the first GUI-style shells pioneered [10] at Xerox PARC, through to modern equivalents in Apple and Microsoft products for example.

The taste and general requirements of different users will of course differ however, as will the tasks they undertake and hence programs they execute. Therefore, different non-GUI styles of interaction and hence shell are also useful. The Command Line Interface (CLI) [4] is an important example, where users interact with the computer by typing commands into a text-based terminal [23]. Both GUI and CLI have advantages and disadvantages, but since there is no perfect choice a common compromise is to expose a CLI within a GUI via a terminal window: the idea is that the CLI executes within same framework as other GUI-based programs, giving a compromise between the two styles of interaction.

Our choice of using CLI-based examples is motivated mainly by practicality: they are easier to present in a typeset book, and make it easier to focus on, reproduce and explain concepts we are interested in. The default CLI-based shell within a UNIX-based OS is almost always BASH [1], which we use exclusively.

### A.1.1 Issuing commands to execute processes

Consider an example BASH session, which mirrors the screen capture of a real terminal window in Figure A.2:

```
bash$ echo 'hello'
hello
bash$
```

Here, the `bash$` part of the first and third lines represents a prompt: this shows the user has control, and specifically that the OS is waiting for their input. The user does not type the prompt: it is included simply to make the examples match what you see in the terminal. Rather, the user types the (rest of the) first line (i.e., the text `echo "hello"` then return) as input, and BASH produces the second line (i.e., `hello`) as output. Following the same terminology as above, the first line is a command instructing the OS to execute
a program called echo. A process is created and managed by the kernel, in this case producing output represented by the second line before terminating naturally and returning control to the user.

A.1.2 Controlling execution using options and arguments

In order to control what a given command does and how it does it, we can provide a list of extra information when we invoke it. One way to think of this information is as a form of input: the process might use some items in the list to make decisions about what operation to perform, and other items as data to operate on. All elements in the list are supplied after the command name, separated by spaces, but there are two different types:

1. A command line option is some optional information provided as part of the command. An option is specified using an identifier, and (optionally) some form of parameter associated with it (e.g., a string, or an integer). Where no parameter is necessary, an option is sometimes termed a flag or switch (since it either turns on or turns off some behaviour).

2. A command line argument is some compulsory information provided as part of the command. Unlike an option, the order of arguments is important: since there is no identifier involved, the only way one argument can be distinguished from another is by their position in the ordered list.

Normally the argument list comes after any options, but this rule is not always true. Even so, a set of examples should make things clearer:

```
bash$ echo 'hello'
hello
bash$ echo -e 'hello' 'goodbye'
hello goodbye
bash$ ls -a --format=long
```

Notice for instance that

- in the first command, `echo` is provided no options, but one argument 'hello',
- in the second command, `echo` is provided one option identified by `-e`, which has no parameter, and two arguments 'hello' and 'goodbye',
- in the third command, `ls` is provided one option identified by `-a` but with no parameter, one option identified by `--format` with the parameter `long`, but no arguments, while
- in the fourth command, `ls` is provided one option identified by `--format` with the parameter `long`, one option identified by `-a` but with no parameter, but no arguments.

In the former two commands the differing options and arguments provoke different behaviour (which is clear from the output), whereas the later two commands produce the same output: this illustrates the fact that the order of options (in this case at least) is irrelevant.

A.1.3 Controlling input and output streams

Each process has an associated set of standard streams [21] which it can use to interact with the environment it is executed within. Specifically, each process can access

- standard input (or stdin), which is used to read input into the process,
- standard output (or stdout), which is used to write output from the process, and
- standard error (or stderr), which is used to produce error messages and avoid having them mixed up with actual output.

Consider a process representing an execution of `echo`, as in the example above. We can picture it (in isolation) as follows:
The question is, where do the (currently dangling) streams come from and go to? By default, when we execute a program and a resulting process is created, the associated streams are connected to the terminal. This means any input typed into the terminal by a user via the keyboard can be read by the process on standard input; any output produced by the process on standard output can be viewed by the user via the monitor. We already saw this above for example, where the command `echo “hello”` produced output: `echo` wrote `hello` to standard output, which we were able to see because standard output was connected to the terminal. However, we can, rather handily, choose where they are connected to via two important mechanisms outlined below.

### A.1.3.1 Input and output redirection

Having just one stream for input and one for output might seem limiting: for example if a web-browser is used to open some web-page it must have to access numerous different resources (e.g., the HTML source code, images and so on), right? The solution provided is to allow each process to supplement the standard input and output streams with access to files; the process can open and close, read and write to files at will, vastly extending the range of ways it can interact with the environment.

To make the task easier, the OS manages streams and files used by a given process in the same way. A **file descriptor** [6], basically a number, is assigned to each: the standard input, output and error streams are numbered 0, 1 and 2, with open files numbered 3 and onward. Each time the process wants to do something related to a file descriptor, it asks the OS to do it via an appropriate system call: “open the file `A.txt` and give me a file descriptor for it” or “read data from file descriptor 0” or “write data to file descriptor 3” say. There is (at least) one major advantage to this approach: if they are managed the same way internally, it should seem natural that one of the standard streams can be connected to a file just as easily as it can the terminal. This is termed **redirection** [17]. If we want to connect the standard output stream of a process to a file, for example, the idea is that the OS pulls a slight-of-hand by replacing the entry for file descriptor 0 with one for the file. The process is unaware and still simply produces output using file descriptor 0, asking the OS to “write data to file descriptor 0”. Behind the scenes however, the OS causes the output to end up somewhere else (i.e., in the file, not on the terminal). A simple example should make this clearer:

```bash
bash$ echo 'foo bar baz'
foo bar baz
bash$ echo 'foo bar baz' > A.txt
bash$ cat A.txt
bash$ tr 'a' 'b' < A.txt
foo bbr bbz
bash$ tr 'a' 'b' < A.txt > B.txt
bash$ cat B.txt
foo bbr bbz
bash$
```

Within the various commands, < and > are used to control how redirection should work:

- The first group of three commands illustrate the difference between standard output being connected to the terminal versus a file. The addition of `> A.txt` in the second command should be read as “redirect standard output into the file `A.txt`”.

  As a result, when the command is executed we no longer see the same output: this output is stored into `A.txt`, a fact then illustrated by the third command (which is used to show the content of `A.txt`). Diagrammatically for example, the first command in this group produces
while the second produces

```
  terminal ----> echo ----> A.txt
               \         |
                \      v
               terminal
```

- The second group of three commands illustrate the difference between standard input being connected to the terminal versus a file. The addition of `< A.txt` in the first command should be read as “redirect standard input from the file A.txt”.

Now `tr` is fed input from `A.txt`, meaning we no longer need to type anything but do see the output (since standard output is still connected to the terminal). The second command uses both forms of redirection, so we get standard input redirected from `A.txt` and at the same time standard output redirected into `B.txt`. The third command shows the content of `B.txt`, which as expected matches what we saw directly via the terminal.

Diagrammatically for example, the first command in this group produces

```
  A.txt ----> tr ----> terminal
               \         |
                \      v
               terminal
```

while the second produces

```
  A.txt ----> tr ----> B.txt
               \         |
                \      v
               terminal
```

There are various more advanced ways to extend this ability, although their use is less common in the examples presented. For completeness, just note that the following are possible:

- As shown above, input and output redirection impacts on the behaviour of standard input and output respectively. However, identifying the stream we want to redirect gives some more control. For example,

  ```
  echo 'foo bar baz' 2> B.txt
  ```

  means standard error is redirected into the file `B.txt` with standard output operating as normal, while

  ```
  echo 'foo bar baz' &> C.txt
  ```

  means both standard output and error are redirected into the file `C.txt`.

- By default, redirecting a stream into a given file overwrites the file content: the previous content lost. This behaviour can be changed by using

  ```
  echo 'foo bar baz' >> D.txt
  ```

  or

  ```
  echo 'foo bar baz' 2>> E.txt
  ```

  to mean append what appears from standard output or error to the end of files `D.txt` and `E.txt` respectively. Here, previous content in `D.txt` and `E.txt` is added to rather than overwritten.
A.1.3.2 Command pipelines

Revisiting the example above, the two commands

```
bash$ echo 'foo bar baz' > A.txt
bash$ tr 'a' 'b' < A.txt
foo bbr bbz
```

have a slightly annoying feature: the first command writes output into the file A.txt so it can be read as input by the second command, but then we probably just delete it. So if we already know we are going to delete A.txt, surely there is a way to avoid using it at all? BASH offers a neat solution to this question, which is termed a **command pipeline** [14]. In short, we can rewrite the two commands as one command pipeline

```
bash$ echo 'foo bar baz' | tr 'a' 'b'
foo bbr bbz
```

which produces the same output using two parts:

1. a left-hand part is represented by the command

   ```
   echo 'foo bar baz'
   ```

   and

2. a right-hand part is represented by the command

   ```
   tr 'a' 'b'
   ```

When the single command pipeline is issued, each part creates a separate process that then execute in parallel (i.e., at the same time). Crucially, the standard output stream of the left-hand process is connected to the standard input stream of the right-hand process via a **pipe**, i.e., we have

```
terminal ➔ echo ➔ pipe ➔ tr ➔ terminal
```

meaning any output produced by echo on standard output can be read by tr on standard input.

A.2 A limited BASH command reference

A.2.1 alias

The **alias** command is usually a BASH built-in. The idea is to give a new name to an existing command, or an entire command pipeline: it literally creates an alias that we can later use as if it were a normal command we had typed in. When used with no arguments, alias prints a list of current aliases:

```
bash$ alias
alias ls='ls -d *'
alias ll='ls -l'
alias mc=~/usr/libexec/mc/mc-wrapper.sh
alias vi='vim'
alias which=~/usr/bin/which --tty-only --read-alias --show-dot --show-ti
lde'
bash$
```

Notice for example that 11 is not a real command, but an alias: BASH translates a use of 11 into `ls -l`, so 11 acts as a shorthand for `ls` with a particular option.

To create a new alias, we supply a name (or identifier) and a value that details what the name should mean (i.e., what it should translate into):
In this example, we create a new alias whose name is `oneperline` and whose meaning is `fold -1`: every time BASH sees the command `oneperline`, it translates it into `fold -1`. As the subsequent uses show, `oneperline` works just like a normal command with respect to standard input and output.

A.2.2 cat

The cat command (short for “concatenate”) is one of the most simple listed here, but also one of the most useful. The command has a modest set of features: given a list of file name arguments, `cat` reads each file in turn and copies the content to standard output. Since each file is read in turn, the content on standard output is essentially a concatenation of all the files. Perhaps even more usefully, when no file names are specified `cat` reads from standard input (i.e., takes input from the user). Crucially, these features enable us to use `cat` to

- provide input to other commands in a command pipeline, or
- create files whose content is supplied by user input.

For example, imagine we want to construct a file called `A.txt` whose content is supplied by user input. We can invoke `cat` with no file name arguments so input is read from standard input, and then redirect standard output at the file:

```
bash$ cat > A.txt
a
b
c
d
```

To mark the end of the input, the user types the EOF character (i.e., Ctrl-D) which is not shown above. Now imagine we want to check that `A.txt` has the expected content; `cat` again supplies a solution. We can invoke `cat` with one file name argument, i.e., `A.txt`, and have it copy the file content to standard output for us to see:

```
bash$ cat A.txt
a
b
c
d
```

Now imagine we make another file called `B.txt` in much the same way as above:

```
bash$ cat > B.txt
e
f
g
h
```

Invoking `cat` now with two file name arguments, i.e., both `A.txt` and `B.txt`, shows how their content is concatenated and copied to standard output:
A.2.3 umask and chmod

New file system entries are created with a default set of permissions controlled by a mask. The mask is subtractive: it disables, i.e., takes away, specific permission types from a starting point where every permission type is enabled. The umask command is used to set and inspect this mask:

```
bash$ umask 022
bash$ umask -S
u=rwx,g=rx,o=rx
```

The first command sets the mask using the shorthand 022; the three numbers in this shorthand should be interpreted as follows:

- 0 means no permission types for the user permission set should be disabled,
- 2 means the writable permission type for the group permission set should be disabled, and
- 2 means the writable permission type for the other permission set should be disabled.

With the -S option, umask reports this in an easier to read format by telling us which permissions a new file will have: notice that the writable permission type is missing from the group and other permission sets, whereas no permission types are missing from the user permission set. We can trial this using an example:

```
bash$ umask 022
bash$ umask -S
u=rwx,g=rx,o=rx
bash$ cat > A.txt
a
b
c
d
bash$ ls -l A.txt
-rw-r--r-- 1 page csstaff 8 Jul 2 17:20 A.txt
bash$
```

The final output of ls confirms the discussion above: the new file A.txt is not writable by either members of the group, or by other users.

Of course, we can change this behaviour to whatever suits us. Imagine we are paranoid, and want a default that prevents anyone other than the owner of a new file from reading it:

```
bash$ umask 077
bash$ umask -S
u=rwx,g=,o=
bash$ cat > B.txt
e
f
g
h
bash$ ls -l B.txt
-rw---r-- 1 page csstaff 8 Jul 2 17:23 B.txt
bash$
```

The new mask 077 implies the following meaning

- 0 means no permission types for the user permission set should be disabled,
- 7 means all permission type for the group permission set should be disabled, and
- 7 means all permission type for the other permission set should be disabled
so when B.txt is created, the effect is that the new file cannot be read by anyone other than the owner.

Clearly it can be useful to change the permissions of a file system entry after it is created; this is the job of chmod. The basic idea is that we specify the name of the file system entry as an argument, along with the change to the permissions we want to make. This change can be specified in two ways. To make an update to the permissions, we can use chmod as follows:

```
bash$ cat > C.txt
bash$ ls -l C.txt
-rw-r--r-- 1 page csstaff 8 Jul 2 17:26 C.txt
bash$ chmod g+w C.txt
bash$ ls -l C.txt
-rw-rw-r-- 1 page csstaff 8 Jul 2 17:26 C.txt
bash$ chmod o-r C.txt
bash$ ls -l C.txt
-rw-rw---- 1 page csstaff 8 Jul 2 17:26 C.txt
bash$
```

The two uses of chmod specifies the changes

- **g+w**, read as “add the write permission type to the group permission set”, and
- **o-r**, read as “remove the read permission type from the other permission set”

where checking the result with ls confirms the changes worked as expected in both cases. However, this style only allows us to specify one change per invocation of chmod; for several of changes, as above, this can quickly get tedious. The good news is that we can describe the permission set we want to end up with using a similar shorthand as above:

```
bash$ cat > D.txt
bash$ ls -l D.txt
-rw-r--r-- 1 page csstaff 8 Jul 2 17:30 D.txt
bash$ chmod 777 D.txt
bash$ ls -l D.txt
-rwxrwxrwx 1 page csstaff 8 Jul 2 17:30 D.txt
bash$ chmod 664 D.txt
bash$ ls -l D.txt
-rw-rw-r-- 1 page csstaff 8 Jul 2 17:30 D.txt
bash$
```

This time, the mask is additive: it enables, i.e., adds, specific permission types from a starting point where every permission type is disabled. In the first case, the mask 777 implies the following meaning

- **7** means all permission types for the user permission set should be enabled,
- **7** means all permission type for the group permission set should be enabled, and
- **7** means all permission type for the other permission set should be enabled

That is, we want all of the readable, writable and executable permission types enabled in all of the user, group and other permission sets. In the second case the mask 644 means

- **6** means the readable and writable permission type for the user permission set should be enabled, and
- **4** means the readable permission type for the group permission set should be enabled, and
- **4** means the readable permission type for the other permission set should be enabled

That is, we want the readable and writable permission types enabled in the user permission set, but only the readable permission type enabled in the group and other permission sets.

While chmod can alter permissions, chgrp and chown can be used to alter the ownership of a file system entry. For example, the chgrp command takes a group name and a file system entry as arguments and sets the group accordingly:
bash$ ls -l D.txt  
-rw-rw-r-- 1 page csstaff 8 Jul 2 17:30 D.txt  
bash$ chgrp crypto D.txt  
bash$ ls -l D.txt  
-rw-rw-r-- 1 page crypto 8 Jul 2 17:30 D.txt  
bash$ chgrp csstaff D.txt  
bash$ ls -l D.txt  
-rw-rw-r-- 1 page csstaff 8 Jul 2 17:30 D.txt  
bash$

Of course, the combination of user name and group name must be valid in the sense that the user must  
actually be a member of the specified group.  

Note that chmod, chgrp and chown all recognise the -R option; this instructs the command to recursively  
process the specified file system entry, and can be useful when changing the ownership or permissions of  
an entire directory tree for example.

A.2.4 cut

The cut command is roughly the opposite of paste. There are a number of ways to use cut but the general  
idea is to consider each line of input, by default read from standard input, as a number of columns (or  
fields) separated by a delimiter character; use of cut constructs output by extracting specific columns from  
the input, writing them by default to standard output.

Consider creating an example file called A.txt; the delimiter character in this case is a comma (i.e., ',')  
meaning A.txt represents a so-called Comma Separated Value (CSV) file:

bash$ cat > A.txt  
a,b,c  
d,e,f  
g,h,i  
bash$

There are three lines, and each line has three fields; imagine we want to extract just the first and second  
fields. We can invoke cut as follows:

bash$ cut -d ',' -f 1,2 A.txt  
a,b  
d,e  
g,h  
bash$ cat A.txt | cut -d ',' -f 1,2  
a,b  
d,e  
g,h  
bash$

The -d option sets the delimiter character to match the one used in the file, and the -f option specifies the  
fields we want in the output; note that fields are counted from one rather than zero. Using other delimiter  
characters is obviously possible:

bash$ date  
Tue Jul 2 14:55:36 BST 2013  
bash$ date | cut -d ' ' -f 1  
Tue  
bash$ date | cut -d ' ' -f 2  
Jul  
bash$ date | cut -d ' ' -f 3  
bash$ date | cut -d ' ' -f 4  
2  
bash$ date | cut -d ' ' -f 5  
14:56:51  
bash$ date | cut -d ' ' -f 6  
BST  
bash$

However, in this case specifying the delimiter and character is perhaps overkill since date always uses the  
same format (e.g., the day field is always the first three characters) unless told otherwise. Rather than rely  
on the use of a delimiter character, it can be useful to have cut extract fields based on character position;  
this is achieved by using the the -c option to specify the position to start and finish at:

bash$ date | cut -c 1-3  
Tue  
bash$ date | cut -c 5-7  
Jul  
bash$
Again note that the character number is counted from one rather than zero; if we leave out the finish position, `cut` just produces the rest of the line. Returning to our original example, we could therefore extract the second field with an alternative usage of `cut`:

```
bash$ cat A.txt | cut -c 3-3
  b
  e
  b
bash$
```

or, by leaving out the finish position, get the second and third fields:

```
bash$ cat A.txt | cut -c 3-
  b.c
  e.f
  b.i
bash$
```

### A.2.5 `diff`

Imagine we want to know if the content of two files is identical or not; we might also want to know where the content differs if it does so. Perhaps you are working on a program or document with someone: they make a change, and you want to know where that change is in the new version relative to the old version you have got. For short text files we might just inspect the files visually, but for longer text files or binary files this is problematic. Fortunately two commands, namely `diff` and `cmp`, can help; they operate on textual and binary content respectively.

Both commands take two file name arguments, and compare their content. The `-s` and `-q` options can be used to silence `cmp` and `diff` respectively, meaning they set the exit code to 0 if the files are identical, or 1 if there are differences found:

```
bash$ diff -q A.txt A.txt
bash$ echo "${?}"
0
bash$ diff -q A.txt B.txt
bash$ echo "$?"
1
Files A.txt and B.txt differ
```

However, some information about what and where differences were found is often very useful. This is particularly true when the file content is textual and we use `diff`. Consider an example were the `-q` option is removed:

```
bash$ diff A.txt B.txt
2c2
  < b
  ---
  > e
4c4
  < d
  ---
  > f
bash$
```

The cryptic tokens `2c2` and `4c4` tell us that two differences between `A.txt` and `B.txt` were found. Each token can be broken into three components:

1. a line number in the first file,
2. a character that describes what type of difference was found,
3. a line number in the second file.

Consider the first token, i.e., 2c2, as an example: the difference type is c, meaning “change”; line two of A.txt has been changed and created a difference in line two of B.txt. Looking again at the output, each token is followed the actual text, i.e. the lines from A.tex and B.tex which have been changed.

Here are some more examples:

```
bash$ cat > B.txt
a
b
c
d
bash$ diff A.txt B.txt
2a3
> e
bash$
```

This time the difference type is a, meaning “addition”; relative to line two of A.txt, there is an addition at line three of B.txt. We might also get the opposite:

```
bash$ cat > B.txt
a
b
d
bash$ diff A.txt B.txt
3d2
< c
bash$
```

where now the change type is a, meaning “deletion”.

### A.2.6 du

The du command (short for “disk usage”) allows a direct way to inspect the size of entries in the file system. Invoked with no arguments, du starts in the working directory and then recursively inspects all files and directories from; it prints the cumulative size of each directory to standard output:

```
bash$ cd /etc/X11/
bash$ du | tail -n 5
  4 ./xinit/Xclients.d
  88 ./xinit
  8 ./twm
 12 ./mwm
 180.
bash$ du -k | tail -n 5
  4 ./xinit/Xclients.d
  88 ./xinit
  8 ./twm
 12 ./mwm
 180.
bash$ du -h | tail -n 5
 4.0K ./xinit/Xclients.d
 88K ./xinit
 8.0K ./twm
 12K ./mwm
 180K .
bash$ du -c | tail -n 5
 88 ./xinit
 8 ./twm
 12 ./mwm
 180 .
 180 total
bash$
```

The four uses of du demonstrate different options. Note that the output is restricted to a few lines (using `last`) since there is a lot of it, and that

- the -k option reports sizes in kilobytes,
- the -h option reports sizes in “human readable” units,
- the -c option includes a total size at the end of the output.

Including one or more path names as arguments instructs du to process them, in turn, rather than the working directory. When the path name specifies a file, du simply prints the size of that file; when the path name specifies a directory, du processes it recursively as above. For example:
Notice that by inspecting the output and identifying the largest entries, it is easy to find out which ones are using the most space. This can be useful when trying to clean up in the event that you run out of disk space: removing the largest entries (which are not useful) will give the most benefit.

### A.2.7 echo

The `echo` command has the modest goal of producing output; it takes a string argument and prints the string, potentially after some minor translation, to standard output.

```bash
bash$ echo 'hello world'
hello world
bash$
```

Using the `-e` and `-E` options, one can enable or disable the translation process which converts escaped characters within the string; by default this behaviour is disabled. For example, imagine the string argument contains a backspace character Ctrl-H escaped as `\b`:

```bash
bash$ echo -e 'hello\b world'
hell world
bash$
bash$ echo -E 'hello\b world'
hello\b world
bash$
```

In the first case, where translation is enabled, when the backspace is printed on standard output it deletes the previous character (i.e., the ‘o’ character at the end of “hello”). However, where translation is disabled in the second case, the backspace is printed on standard output as-is (i.e. as the escaped character `\b`).

Finally, note that by default `echo` prints an EOL character at the end of the string. This behaviour can be turned off by using the `-n` option, and can be useful when producing binary output for example:

```bash
bash$ echo -n 'hello world'
hello world
bash$
```

### A.2.8 fold

The `fold` command is an example of something very simple, and perhaps seemingly useless, but which turns out to be very useful in solving all sorts of other problems. The command reads lines of input from standard input (or the content of a file named as an argument if provided) and “wraps” them so they fit a certain length; the output is written to standard output. You can think of this as similar to what happens when you use a word processor: when you write a very long line, it typically moves you onto the next line automatically.

The `-w` option sets the maximum length of each line, i.e., where to wrap each line; the default is 80 characters. As such an example shows how `fold` works:

```bash
bash$ cat > A.txt
ab cd ef gh
ijkl
mnop qr
st
bash$ fold -w 4 A.txt
ab c
d ef
gh
ijkl
mnop qr
st
bash$ cat A.txt | fold -w 4
ab c
d ef
gh
ijkl
mnop qr
st
bash$
```
A specifically important application of `fold` stems from use of `-w 1` as an option: this splits a 1-line input, of say $n$ characters, into an $n$-line output each of which has just one character. Since many other commands are line-oriented, i.e., process each line of input in turn, this can allow them to operate on the characters within a single line.

Finally, when processing text it can be useful to only wrap a line at a point where there is a space rather than at a specific length; this prevents splitting words over two lines for example. This is possible using the `-s` option. Returning to the example above, notice how this alters the output:

```
bash$ cat A.txt | fold -w 4 -s
ab
cd
ef
gh
ijkl
mnop
qr
st
bash$
```

The first line is now split at the space between “ab” and “cd” because, with the same line length of four, we saw above that “cd” was originally split over two lines.

### A.2.9 grep

The `grep` command is possibly the greatest invention ever, bar perhaps the wheel and sliced bread; used as a verb, “to grep” is something you might actually hear in conversations. Perhaps the best way to understand what `grep` does is to start by thinking of it as a stream filter: it reads lines of input, checks whether they match some pattern (described using a regular expression), and only outputs lines that match. Consider a single example: first we construct a file called `A.txt`, then try to filter out all the lines that contain an ‘a’ character:

```
bash$ cat > A.txt
a
aba
cac
add
eea
f
bash$ grep a A.txt
a
aba
cac
add
eea
bash$
```

Notice that by default `grep` reads input from standard input and writes output to standard output; by supplying a file name argument we can have `grep` read the file content instead. Of course, more complicated regular expressions are possible. Imagine we try to filter the same file using regular expressions that match

1. lines containing characters from the set `[af]`, i.e., the characters ‘a’ and ‘f’,
2. lines that start with the character ‘a’,
3. lines containing the sequence ‘a’, then any character then another ‘a’.

The `grep` usage and resulting output is as follows:

```
bash$ cat A.txt | grep [af]
a
aba
cac
add
eea
f
bash$ cat A.txt | grep 'a'
a
aba
add
bash$ cat A.txt | grep a.a
aba
bash$
```
One can also turn the filter the opposite way and have grep only output a line if it does not match the pattern. This is achieved by using the `–v` option:

```
bash$ cat A.txt | grep -v `[af]
cac
eea
f
bash$ cat A.txt | grep -v `a
a
cac
add
eea
f
bash$
```

It is common to execute grep several times within a single command pipeline in order to narrow down (or “thin”) the input until it contains only those lines which are of interest. Imagine we want to end up with just the line containing “cac”. Among many options for this particular file, one way to filter out all other lines would be to use

```
bash$ cat A.txt | grep -v a.a
a
cac
add
eea
f
bash$
```

where the pattern selects all lines that contain any character, then an ‘a’ then any character. Alternatively we could do the same thing with the following command pipeline:

```
bash$ cat A.txt | grep a | grep -v a$ | grep -v `a
```

For this particular file this works by first selecting all lines containing ‘a’, then eliminating all of those lines ending with ‘a’ and finally eliminating any remaining lines that start with ‘a’.

Finally, grep can act as more than just a stream filter. When provided with one or more file name arguments, grep acts to search those files for the given pattern. Imagine we are writing a C program and litter the source code with “todo list” style comments that indicate where work is needed. We might want to see where in a particular file such comments exist, or gather a list of files that include such comments; in both cases grep provides a solution. Imagine we write three source code files:

```
bash$ cat > B.c
/* TODO: finish main function */
int main( int argc , char* argv[] ) {
    return 0;
}
bash$ cat > C.c
/* TODO: fix foo function, it returns the wrong value */
int foo( int x , int y ) {
    return ( x < y ) ? ( x ) : ( y );
}
bash$ cat > D.c
int bar( int x , int y ) {
    return ( x > y ) ? ( x ) : ( y );
}
bash$
```

Providing the same file names to grep as arguments, we can search for the string TODO as follows:

```
bash$ grep TODO B.c C.c D.c
B.c:/* TODO: finish main function */
C.c:/* TODO: fix foo function, it returns the wrong value */
```

Notice that grep has produced the file names and the lines within those files which match the pattern. If you just want to know the file names which match the pattern, use of the `–l` option will suppress output of the actual lines:

```
bash$ grep -l TODO B.c C.c D.c
B.c
C.c
bash$
```

### A.2.10 bzip2, gzip and zip

Most UNIX-based operating systems offer a wide range of commands that perform compression and decompression of data. Their common role is simple: some input data is compressed into a smaller version (e.g., in order to communicate or store it) which can, at a later date, be decompressed to recover the original. Example commands include:
• gzip (and gunzip),
• bzip2 (and bunzip2), and
• zip (and unzip).

The difference between these examples is (very roughly) how effectively they compress data (i.e., how small an output they produce given some input), and how efficiently they perform said compression. However, note that zip represents a combination of archival and compression tasks: it can collect many files together into an archive then compress the result. In contrast, gzip and bzip2 perform compression only; they rely on a separate command such as tar to deal with archives and hence operate with more than one file.

The commands are all used in roughly the same way, and have similar sets of options. Since the exact command does not matter in most cases, we focus only on bzip2 and bunzip2 since they represent a more modern solution. By default bzip2 (resp. bunzip2) reads input from standard input, compresses (resp. decompresses) it and copies the result to standard output. Alternatively, a file name argument can be provided: the file content is compressed (resp. decompressed) and the result is written to a new file. The new file is named by taking the original file name and adding (resp. removing) the .bz2 extension.

Consider an example where we construct a file called A.txt and then compress it using bzip2:

```
bash$ cat > A.txt
a
a
a
a
b
b
b
b
bash$ ls -l A.txt
-rw-r--r-- 1 page csstaff 16 Jul 2 13:31 A.txt
bash$ bzip2 -z A.txt
bash$ ls -l A.txt.bz2
-rw-r--r-- 1 page csstaff 43 Jul 2 13:31 A.txt.bz2
bash$
```

The -z option instructs the command to compress, but this can be left out since bzip2 compresses the input by default. Either way, the result for this small example input is a bit disappointing; in terms of compression, we have actually made A.txt.bz2 larger! However, for larger and less contrived inputs you can hopefully believe that bzip2 is more effective. We can test the integrity of A.txt.bz2 (i.e., whether the file can be decompressed correctly) using the -t option:

```
bash$ bzip2 -t A.txt.bz2
bash$ echo "$?"
0
bash$
```

In this case the lack of output and the exit code indicate success.

The command bunzip2 is actually just a shorthand for use of bzip2 command with the -d option; this instructs bunzip2 to decompress the input rather than compress it:

```
bash$ ls -l A.txt.bz2
-rw-r--r-- 1 page csstaff 43 Jul 2 13:31 A.txt.bz2
bash$ bunzip2 -d A.txt.bz2
bash$ ls -l A.txt
-rw-r--r-- 1 page csstaff 16 Jul 2 13:31 A.txt
bash$
```

Note that a slew of alternates to commands such as cat exist that will accept compressed input. For example, given a file compressed with gzip, zcat will copy the decompressed file content to standard output:

```
bash$ cat > B.txt
a
b
c
d
bash$ gzip B.txt
bash$ zcat B.txt.gz
a
b
c
d
bash$
```

Usually a similar result can be achieved just by using the right options for the command; for example, bzip2 and gzip will both produce output on standard output (given a file as input) if used with the -c option.
A.2.11 head and tail

The purpose of the `head` (resp. `tail`) command is to output the first (resp. last) part of some input. We can specify the amount of input to retain in various ways: one might want to output the first $n$ lines for example, or the first $n$ bytes. This is a simple remit, but deceptively powerful. Given `A.txt`, a short example file

```
bash$ cat > A.txt
a
b
c
d
e
f
g
h
i
j
k
l
m
n
bash$
```

consider some examples of specifying the number of lines to output using the `-n` option:

```
bash$ head A.txt
a
b
c
d
e
f
g
h
i
j
bash$ cat A.txt | head
a
b
c
d
e
f
g
h
i
j
bash$ head -n 2 A.txt
a
bash$
```

```
bash$ tail A.txt
g
h
i
j
k
l
m
n
o
p
bash$ cat A.txt | tail
g
h
i
j
k
l
m
n
o
p
bash$
```

Notice that the input taken from a file named as an argument, or from standard input by default, and the output is written to standard output. When `-n` is used, `head` and `tail` output the first and last two lines of `A.txt` respectively; when it is not used, the default is ten lines.

In the case of `tail`, when the number of lines is preceded by a plus sign it signals a slightly different behaviour. For example, if used as follows

```
bash$ tail -n 2 A.txt
o
p
bash$
```
tail starts at the second line in A.txt meaning the first line containing ‘a’ is skipped. By using a combination of head and tail, we can therefore split a file into parts at a given line; to split the same A.txt at the second line for example, we could use

```
bash$ cat A.txt | tail -n +2
bc
defg
hi
klmnp
```

Or, to extract a specific number of lines from the middle of the file, we could combine head and tail in a single command pipeline such as:

```
bash$ cat A.txt | head -n 1
a
bash$ cat A.txt | tail -n +2
bc
defg
hi
klmnp
```

To specify a number of bytes instead of a number of lines, we can use the `-c` option instead of `-n`. By first making a new file called B.txt

```
bash$ cat > B.txt
abcd
efgh
ijkl
mnop
```

the following examples demonstrate how this option behaves:

```
bash$ cat B.txt | head -c 10
abcd
efgh
ijkl
mnop
bash$
```

At first the output seems odd. However, remember that EOL characters are included: the output from head, for example, includes eight printable characters plus two EOL characters, making ten in total.

### A.2.12 `ls`

The `ls` command (short for “list”) allows inspection of file system entries: it will list information about the path names given as arguments, or the working directory if none are provided. Imagine we are interested in contents of the root directory of the file system. We might change directory there and issue an `ls` command to list the content:
By default `ls` lists the names of entries in the new working directory, writing to standard output; at the moment it is not clear if these entries are directories or files, we only get their names. The `ls` command permits a huge range of options, but a select few are perhaps the most important. The example

allows us to describe them as follows:

- The `-1` option instructs `ls` to format the list using one entry per-line. This can be useful if the output of `ls` is used within a line-oriented command pipeline.
- The `-a` option instructs `ls` to show all entries including, for example, those hidden by default. In the example, as well as the original entries we now get the special current directory and parent directory entries (i.e., `.` and `..`).
- The `-l` option instructs `ls` to produce a “long” listing which includes a range of details about each entry. In the example, the columns of output detail

```bash
bash$ cd / 
bash$ ls
a cgroup home local mnt proc selinux sys var
bin  lib lost+found opt root space tmp
boot etc lib64 media opt.tar sbin srv usr
bash
```

```bash
bash$ cd / 
bash$ ls -1
a
bin
cgroup
devel
event
home
lib
lib64
local
lost+found
media
mnt
opt
opt.tar
proc
root
sbin
selinux
space
srv
sys
tmp
usr
var
bash$ ls -a
.. .readahead_collect cgroup lib64 opt selinux usr
.. .rnd dev local opt.tar space var
VISION .autofsck a etc lost+found proc srv
VISION .autorelabel bin home media root sys
VISION .kde boot lib mnt sbin tmp
bash$ ls -l
```

```text
```
```
1. the entry type (e.g., file or directory) and permissions,
2. the number of links to the entry,
3. the user who owns the entry,
4. the group which owns the entry,
5. the entry size (in bytes),
6. the modification timestamp for the entry, and
7. the entry name.

The summary line at the beginning of the output gives the total size of all the entries listed.

This is perhaps obvious, but although we have so far used `ls` in the default mode to list entries in the working directory, we can also specify absolute or relative paths to other file system entries:

```bash
bash$ ls -l /etc/passwd
-rw-r--r--. 1 root root 3616 Nov 14 2011 /etc/passwd
bash$ ls -l /etc/ssh/
total 160
-rw-------. 1 root root 125811 Jun 25 2011 moduli
-rw-r--r--. 1 root root 2047 Jun 25 2011 ssh_config
-rw------- 1 root root 668 Nov 14 2011 ssh_host_dsa_key
-rw-r--r-- 1 root root 590 Nov 14 2011 ssh_host_dsa_key.pub
-rw------- 1 root root 963 Nov 14 2011 ssh_host_key
-rw------- 1 root root 627 Nov 2011 ssh_host_key.pub
-rw------- 1 root root 1675 Nov 14 2011 ssh_host_rsa_key
-rw------- 1 root root 963 Nov 14 2011 ssh_host_rsa_key.pub
-rw-------. 1 root root 3313 Nov 14 2011 sshd_config
bash$ ls -ld /etc/ssh/
drwxr-xr-x. 2 root root 4096 Nov 14 2011 /etc/ssh/
bash$
```

Note that in this context, the `-d` option is useful: if the file system entry we specify is a directory, it tells `ls` we are interested in the directory itself rather than the contents. The result above is that with `-d`, we can inspect the permissions and so on for `/etc/ssh/` but without it `ls` lists the contents of `/etc/ssh/`.

Finally, a note on permissions: imagine we create a directory called `A/` using `mkdir` then create a file in it called `B.txt`. Changing the permissions of both file and directory using `chmod` demonstrates some important facts about how we access to them:

```bash
bash$ mkdir -p A/
bash$ cat > A/B.txt
a
b
c
d
bash$ ls -ld A/
drwxr-xr-x 2 root root 4096 Jul 2 16:35 A/
bash$ chmod 300 A/
bash$ ls -ld A/
d-wx------ 2 page csstaff 4896 Jul 2 16:35 A/
bash$ ls -l A/
ls: cannot open directory A/: Permission denied
bash$ chmod 700 A/
bash$ ls -l A/
-rw------- 2 page csstaff 4896 Jul 2 16:35 A/
bash$ ls -l A/
-rw------- 2 page csstaff 4896 Jul 2 16:35 A/
bash$ ls -l A/
total 4
-rw-r--r-- 1 page csstaff 8 Jul 2 16:35 B.txt
bash$
```

First we create `A/` and `A/B.txt`, and then change the permissions of `A` so the user permission set includes only the write and execute permission types; as a result, the first time we try to use `ls` to list the contents of `A/`, the command fails. After altering the permissions of `A/` again to now include the read permission type, the second attempt to list the contents of `A/` succeeds.

### A.2.13 `od`

The `od` command (short for “octal dump”) is a brilliant tool. The main use-case is visualisation of binary data. The idea is that we provide some input (either by supplying an optional file name argument or on standard input by default), and `od` produces an unambiguous representation on standard output. This is a fancy way to say that it inspects the binary input and produces the corresponding “human readable” output that is not complicated by the existence of non-printable characters and so on.

When used with no options, the output from `od` is somewhat difficult to interpret:

```
308
```
bash$ cat > A.txt
a
b
c
d
bash$ od A.txt
0000000 005141 005142 005143 005144
0000010
bash$ cat A.txt | od
0000000 005141 005142 005143 005144
0000010
bash$

The left-hand column is an index (or address); starting at each index is some content, which is listed in
columns to the right. By default however, both the index and content are displayed in octal. Rather than
try to interpret this, a better approach is typically to control od using one or both of two options:

- `-A` allows us to change the index format using a 1-character code to specify the base (or radix):
  - `o` for octal, which is the default,
  - `d` for decimal, or
  - `x` for hexadecimal.
- `-t` allows us to change the content format using a 1- or 2-character code to specify the base (or radix)
  and size, e.g.,
  - `c` for ASCII characters,
  - `a` for “named” ASCII characters,
  - `o` for octal integers,
  - `u` for unsigned decimal integers,
  - `d` for signed decimal integers,
  - `x` for hexadecimal integers,
  - `u1` for 1-byte unsigned decimal integers,
  - `d2` for 2-byte signed decimal integers, or
  - `x4` for 4-byte hexadecimal integers.

Using decimal for the index and the content, `A.txt` now makes more sense for example:

```
bash$ cat A.txt | od -Ad -td1
0000000 97 10 98 10 99 10 100 10
0000008
bash$
```

This shows that starting at index 0 (through to 7, since there is no content on the second line starting at
dindex 8) the content includes the eight 8-bit unsigned decimal values 97, 10, 98, 10, 99, 10, 100 and 10. This
should not be surprising since, for example, 97 is the ASCII value for the character ‘a’ and 10 is the ASCII
value for EOL. Switching to hexadecimal for the content, i.e.,

```
bash$ cat A.txt | od -Ad -tx1
0000000 61 0a 62 0a 63 0a 64 0a
0000008
bash$
```

should still make sense in that 61₁₆ = 91₁₀, and 0A₁₆ = 10₁₀ for example.

Two other options are often useful: `-w` and `-N` respectively specify the number of entries in the content
to display on each line and in total. In a final example using `A.txt`, these are used to inspect only the first
four bytes with two bytes on each line:

```
bash$ cat A.txt | od -Ad -ta -w2 -N4
0000000 a \nl
0000002 b \nl
0000004
bash$
```

This example also uses a “named” ASCII for the content format: each character is translated into a name
(e.g., EOL into `nl`) rather than a character code as it would be in the alternative

```
bash$ cat A.txt | od -Ad -tc -w2 -N4
0000000 a \n
0000002 b \n
0000004
bash$
```
A.2.14 paste

The paste command is roughly the opposite of cut; it takes a list of \( n \) file name arguments as input, and pastes the files content as columns along side each other to form output. That is, the \( i \)-th line of the output is constructed by reading the \( i \)-th lines of each \( n \) inputs and placing them in order along side each other. The end result is perhaps explained more clearly by example:

```
bash$ cat > A.txt
a
b
c
d
bash$ cat > B.txt
e
f
g
h
bash$ paste A.txt B.txt
a e
b f
c g
d h
bash$
```

By default, as above, the output is written to standard output; note that the tab character is used to separate the parts of each line. One can alter this behaviour using the `-d` option to specify an alternative delimiter character; for example to separate the parts using a comma one might use:

```
bash$ paste -d ',' A.txt B.txt
a,e
b,f
c,g
d,h
bash$
```

It is worth pointing out that paste is similar to other commands in that it can accept input from standard input rather than from the content of files. In this case however, there is only one standard input whereas above we specified multiple file name arguments. By using the `-s` flag, paste uses just one input source: the \( i \)-th line of the output is again constructed by reading lines from the input and placing them in order along side each other. You can think of this as serialising the multi-line input into a single-line output:

```
bash$ paste -s A.txt
a b c d
bash$
bash$ cat A.txt | paste -s
a b c d
bash$
```

A.2.15 sort

It should come as no surprise that the sort command reads lines of input and sorts them to produce output. The sorting procedure is controlled by the sorting “key”. The idea is to imagine each line split into a number of fields by a delimiter (e.g., a space or comma); sorting the input essentially means comparing the field in each line indicated by the sorting key.

The input is taken from a file named as an argument, or from standard input by default, and the output is written to standard output. To demonstrate the command we first need to construct A.txt, a short example file:

```
bash$ cat > A.txt
a t 300
A t 300
b s 456
b h 74
d q 0
g n 5999
f o 623
c r 13
bash$
```

By supplying the file to sort without any options, the whole line is used as the sorting key and so the output is:

```
bash$ sort A.txt
A t 300
a t 300
b s 456
b h 74
c r 13
d q 0
g n 5999
f o 623
```
However, imagine we want to sort the file using the second field as the sorting key. This is easy: we simply use the -t option to specify that the delimiter character in this case is a space, and the -k option to specify that the second column should be used as the sorting key:

```
bash$ cat A.txt | sort -t ' ' -k 2
A t 300
a t 300
b s 456
c r 13
d q 0
g n 5999
f o 623
h m 74
```

Trying the same trick with the third field probably does not produce the output one might expect intuitively:

```
bash$ cat A.txt | sort -t ' ' -k 3
d q 0
c r 13
A t 300
a t 300
b s 456
g n 5999
f o 623
h m 74
```

The reason is simple: unless otherwise specified, the sorting procedure is based on alphabetic order. In order to use the third field as the sorting key, we need a numerical order and hence the -n option:

```
bash$ cat A.txt | sort -t ' ' -k 3 -n
d q 0
c r 13
A t 300
a t 300
b s 456
g n 5999
f o 623
h m 74
```

Controlling `sort` via numerous other options is possible: two of the more useful are -r, which reverses the sort order (i.e., backwards rather than forwards, or decreasing rather than increasing), and -c, which checks if the input is sorted (rather than actually sorting it). These options are demonstrated in the following example:

```
bash$ cat A.txt | sort -r
h m 74
g n 5999
f o 623
d q 0
c r 13
b s 456
```

Notice that in the second and third cases, the -c option tells us whether the input is sorted via the exit code; in addition, when the input is not sorted, it tells us where the first “out of order” line of input is.
A.2.16 tr

The tr command (short for “translate”) reads standard input, transforms characters in the input based on rules supplied by the user, and then writes the result to standard output. The command can operate in three modes, meaning it either

1. translates instances of one character into instances of another,
2. delete every instance of a character, or
3. deleting duplicate instances of a character (i.e., “squeeze” them).

The translate mode is the default; -d and -s options instruct tr to operate in delete or squeeze modes respectively. As well as the operating mode, tr needs to know the rules that control how the input is transformed. The rules are specified using one more more character sets (depending on the mode).

An example will make this easier to explain. For translation, two sets need to be specified; imagine tr reads characters of input one at a time. If the i-th character it reads is not found in the first set, it is written to the output unchanged; if it matches the j-th character of the first set then the output is the j-th character in the second set. This is a very formal way to describe something very simple:

```
bash$ cat > A.txt
a
b
c
d
bash$ cat A.txt | tr 'a' 'e'
e
b
c
d
bash$
```

First we construct a file called A.txt, and then use tr to translate every instance of the character ‘a’ into the character ‘e’; described as such, the output is as you might expect.

In this case single characters control the translation process, but the sets are more general. For example, imagine you want to turn all lower-case characters into their upper-case equivalents. We could build a command pipeline of twenty six invocations of tr, one for each character, but this would be awkward to say the least! Instead, one single invocation suffices which we can write in (at least) two ways:

```
bash$ cat A.txt | tr [a-z] [A-Z]
A
B
C
D
bash$ cat A.txt | tr [:lower:] [:upper:]
A
B
C
D
bash$
```

In both cases, tr translates instances of ‘a’ into ‘A’, ‘b’ into ‘B’ and so on:

- The first usage is arguably the most intuitive, and describes the sets concretely in terms of their members: [a-z] and [A-Z] represent the lower- and upper-case characters respectively.
- The second usage arguably requires the least work, and describes the sets abstractly in terms of their properties: [:lower:] and [:upper:] represent the lower- and upper-case characters respectively.

Note that the sets do not need to be the same size:

```
bash$ cat A.txt | tr [a-c] 'e'
e
e
e
d
bash$
```

When this happens, tr pads the second set by repeating the last character as many times as necessary; in this case, the character ‘e’ is repeated three more times. Excess characters in the second set are simply ignored.

The translation mode is the most common use of tr, but remember it also allows one to delete and “squeeze” characters. Deletion is a simple concept: imagine we want to remove the end of line characters from A.txt. As a first attempt we could try
but this has simply translated the end of line characters into spaces, whereas we want to delete them. To do
this, we use the \-d option which instructs tr to delete characters that occur in the first set. As such, we can
accomplish our goal using:

```
bash$ cat A.txt | tr -d \"\n\"
abcdbash$
```

Finally, "squeezing" characters basically means replacing a sequence of some duplicate character that occurs
in the first set with a single copy of that character. Imagine we create a new file, this time called `B.txt`:

```
bash$ cat > B.txt
aaa
bbb
ccc
ddd
bash$
```

Now we have a sequence of 'a' characters for example. To squeeze this sequence into one copy of 'a' we
use:

```
bash$ cat B.txt | tr -s \"a\"
a
bbb
ccc
ddd
bash$
```

Or, to squeeze all the duplicates, in this specific case we could use

```
bash$ cat B.txt | tr -s [a-d]
a
b

c
d
bash$
```

noting the set of characters to squeeze is 'a' through 'd'. An alternative would be to use use a similar
simplification as above, i.e.,

```
bash$ cat B.txt | tr -s [:lower:]
a
b

c
d
bash$
```

A final trick is provided by the \-c option which complements (or inverts) the first set before it is used.
Imagine we want to delete all characters except lower-case characters; this is now a little awkward if we
follow the examples above, because we need to specify a set which has many members. However, the \-c
option makes it much easier:

```
bash$ cat > A.txt
a
b

c
d
bash$ cat A.txt | tr -dc [:lower:]
abcdbash$
```

### A.2.17 uniq

The idea of the `uniq` command (short for "unique") is simple: imagine the input split into lines, the command
compares the i-th line with successive lines (i.e., j-th lines for j > i) and removes all but the first identical
instance. So for example, if line one is the same as line two then only line one appears in the output.

The input taken from a file named as an argument, or from standard input by default, and the output is
written to standard output. To demonstrate the command we first need to construct `A.txt`, a short example
file:

```
bash$ cat > A.txt
a
b
a
```
Using `A.txt`, it is easier to see what is going on:

```bash
bash$ uniq A.txt
a
b
c
d
bash$ cat A.txt | uniq
a
b
a
c
d
bash$
```

Notice that there are two ‘a’ characters on the first and third lines but these duplicates are not removed since the lines are not successive; the three lines at the end of `A.txt` which contain the ‘d’ character are successive and also identical however, so only the first appears in the output. Modern implementations of `uniq` have numerous other options. For example, if we make a new file called `B.txt`:

```bash
bash$ cat > B.txt
a
b
a
c
d
D
D
bash$
```

the `-i` option can be used to ignore case:

```bash
bash$ cat B.txt | uniq -i
a
b
a
c
D
bash$
```

Another nice trick is that `uniq` can be instructed to print a count of how many identical copies of a line were encountered:

```bash
bash$ cat B.txt | uniq -i -c
1 a
1 b
1 a
1 c
3 D
bash$
```

This starts to allow for some useful command pipelines; imagine we want to know how many lines in a given file start with the character ‘a’ (ignoring case). One solution would be to first use `cut` to extract the first character of each line, then use `sort` to sort the characters into order, then finally use `uniq` to count how many identical copies of the characters exits:

```bash
bash$ cat /usr/share/dict/words | cut -c 1-1 | sort | uniq -i -c | grep a
25192 a
bash$
```

By completing the pipeline using `grep` to get the count for ‘a’ alone, the output we want is produced.

### A.2.18 `wc`

The `wc` command counts the number of lines, words and characters (by default all three) in some input. In this context, a “word” is roughly defined as a string of non-space characters surrounded by space characters. The input taken from files named as arguments, or from standard input by default, and the output is written to standard output:

```bash
bash$ wc /usr/share/dict/words
479829 479829 4953699 /usr/share/dict/words
bash$ wc -l /usr/share/dict/words
479829 /usr/share/dict/words
```
Note that the `-l`, `-w` and `-c` options instruct `wc` to output only the count of lines, words and characters; in this case the number of lines and words is the same because `/usr/share/dict/words` is essentially a list of words, one per-line.

The other useful trick `wc` can perform is to output the length of the longest line in the input. Consider an example where `wc` is supplied with input by `cat`, and the `-L` option is used:

```
bash$ cat > A.txt
a
bb
ccc
dddd
bash$ cat A.txt | wc -L
4
bash$
```

The output is 4 which is sane since the longest line in `A.txt`, namely `dddd`, has four characters in it.

In isolation `wc` might not seem useful, but when combined with other commands we can use it to solve real problems. Imagine we want to know how many lines in a given file start with the character `a` (ignoring case). One solution would be to first use `grep` to filter out all lines except those starting with `A` or `a`, and then `wc` to count the number of lines that remain:

```
bash$ cat /usr/share/dict/words | grep `^[Aa]` | wc -l
31788
bash$
```

### A.2.19 `wget`

To call `wget` a command is underselling it. This command has so many different options and uses, it should really be classed as an application: put simply, `wget` is like a web-browser, but one we use to automate tasks rather than use interactively.

Rather than explore the vast range of tasks `wget` can be used for, we focus on a single use-case: imagine we want to download a web-page and, instead of rendering it on the screen, process the content somehow using other commands (e.g., as if it were a file held locally). In order to accomplish this, we use three options

1. the `-q` option tells `wget` to operate quietly (i.e., not print out any progress information, which can be useful within a command pipeline),
2. the `-U` option allows `wget` to masquerade as a real web-browser by mimicking the style of commands and information supplied to the web-server (which can be useful to ensure a download is not blocked), and
3. the `-O` option tells `wget` where to store the output,

plus an argument that specifies the web-page via a URL. Imagine we want to download content from

```
http://www.gutenberg.org/dirs/etext97/1ws1810.txt
```

which is the text for *The Merchant of Venice* by Shakespeare, and save it into a file called `A.txt`. We can accomplish this with the deceptively simple command

```
bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ ls -l A.txt
-rw-r--r-- 1 page csstaff 145500 Sep 6 12:47 A.txt
bash$
```

noting that here the `-U` option means `wget` acts like a Chrome web-browser, and `-O` instructs it to save the content into `A.txt`. We also told `wget` to be quiet, and it certainly was! It offers no output on standard output, but a subsequent `ls` output shows that `A.txt` represents was download successfully:

```
bash$ head -4 A.txt
This Etext file is presented by Project Gutenberg, in cooperation with World Library, Inc., from their Library of the Future and Shakespeare CDROMS. Project Gutenberg often releases Etexts that are NOT placed in the Public Domain!!
bash$
```

315
Imagine we wanted to count the number of lines in the content downloaded in the example above; at least two approaches are possible:

1. download the content, save it into a file and then count the lines in the file content using `wc`, or
2. download the content, copy it to standard output and then feed this via a command pipeline to `wc`.

By replacing the file named after the `-O` option with a dash character (i.e., `'-'`), we can adopt the second approach:

```
bash$ wget -q -U chrome -O B.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ wc B.txt
3436 24017 145500 B.txt
bash$ wget -q -U chrome -O- 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt' | wc
  3436  24017  145500
bash$
```
APPENDIX

B

A TOUR OF COMPUTER NETWORKING AND CRYPTOGRAPHY-BASED NETWORK SECURITY

The term network security is a catch-all for a very broad field. In essence, it describes techniques and technologies relating to secure use of and communication over a network (e.g., with network-attached resources). On one hand, network security implies various specific challenges, and so requires a specific set of related background knowledge. On the other hand, some challenges within this context are fairly generic; the need for confidentiality and integrity of data, plus authentication of parties could also be evident in non-networked information systems for example.

As a result, one can use network security as a specific vehicle to study these more general concepts. At least two benefits, versus an alternative, stem from doing so:

1. it can offer a practical explanation of concepts we all routinely make use of, plus

2. relate this practical exploration to any theoretical introduction of the same concepts you might have encountered.

This is a (very) short tutorial which explores these points, with specific focus on security-related topics. Said topics, and hence the remit of the tutorial as a whole, are limited in two important ways. First, the content tries to follow a similar ethos as elsewhere in the book by using BASH commands to explain each topic, e.g., those from Chapter 7 and Chapter 10. More so than elsewhere however, these examples are dependent on how the Operating System (OS) works. As a result, and though the concepts themselves are of course more general and can be translated, the examples are specific to how networking is dealt with by the Linux kernel [16]. Second, we limit examples to commands you can issue as a normal user: a huge range of interesting topics (e.g., packet filtering and analysis [10, 23]) are possible if you can act as a privileged user (i.e., root or similar [32]), but are outside the remit of this more basic introduction. To mitigate both limitations, note that plenty of resources provide further information with at least one free online book

http://en.wikibooks.org/wiki/Linux_Networking

for example.

To get the most out of what follows, the recommended approach is to first read through Section B.1 which introduces important background concepts and terminology, then each subsequent Section (more or less in any order you want).
Some basic, background concepts and terminology

Probably without realising it, you already have at least an intuitive grip of some standard network concepts: after all, you routinely use them even if unknowingly. Since we need somewhere to start, the following is a brief recap of such concepts and terminology then used throughout:

- **A network stack** allows processes executing on each host (or computer, sometimes termed a node) to access the network. The internal organisation of a concrete network stack often follows the OSI model [22], or similar, in the sense that a series of layers each provide different types of functionality to a process.

  The lower-layers often represent physical hardware devices that form network interfaces (i.e., physical connections to the network infrastructure) via Network Interface Controllers (NICs) [21]. The upper-layers are more typically realised in software as part of the kernel (due, for example, to the need for direct and protected access to the NIC). These layers will manage protocols such as the Internet Protocol (IP) [15] and Transmission Control Protocol (TCP) [34].

  Rather than view it in terms of different layers, we take a more abstract and simplistic approach: we ignore which role the IP or TCP layer has for example, simply assuming a single monolithic network stack within the kernel does everything.

- An IP-based network, such as the Internet, is packet switched. To send a variable-length message $M$ from some source host to a target (or destination), it is first split into one or more fixed-length packets. This can be roughly formalised by saying $M \mapsto P = (P_0, P_1, \ldots, P_{n-1})$, meaning a message $M$ maps to a sequence $P$ of $n$ packets. Each of these packets is routed independently through the network via a series of intermediate hops (rather than in a single, direct hop from source to target), then reassembled back into $M$ once they all reach the target.

- An **IP address** is a unique numerical identifier given to each host on an IP-based network. Depending on the type of network (e.g., IPv4 or IPv6) the IP address format might differ, but here we assume it is a 4-tuple of 8-bit bytes. More formally for example, the IP address 137.222.102.8 could be represented by the tuple

  $A = (A_0, A_1, A_2, A_3) = (8, 102, 222, 137)$

  where each $A_i \in [0,1,\ldots,255]$. Some IP addresses are reserved for special purposes, one example being 127.0.0.1 which identifies the local host, i.e., the computer you are issuing commands on.

- Imagine we have an IP address written as

  $A = (A_0, A_1, A_2, A_3)$.

  If we fix $A_1, A_2$ and $A_3$, this specifies a so-called class C sub-net [31] (or “smaller network”): each of the 256 possible options for $A_0$ can be used to identify a host on said sub-net. Sometimes, a short-hand such as

  $137.222.102.0/24$

  is used in this context. Here, the number after the slash tells us how much of the IP address is fixed: 24 bits of the address are fixed (matching $A_1, A_2$ and $A_3$), and 8 bits can be varied (matching $A_0$).

- A related idea is the application of an **address mask** to some IP address. Imagine we have a mask

  $B = (0, 255, 255, 255)$

  for example. The idea is to combine $M$ and $A$ by using the AND function (outlined in Chapter 2) to get

  $A' = A \land B = (0, A_1, A_2, A_3)$

  because zero AND any $A_0$ will produce zero as a result in $A'_0$. Thus, we can use $M$ to ignore some parts of $A$: if we start with

  $137.222.102.8$
and apply our mask, we get

\[ 137.222.102.0 \]

as a result.

- The **Domain Name System (DNS)** [8] is an infrastructure that supports associations between IP addresses and more easily human-readable string-based identifiers (or names). Translation of a DNS name into an IP address is termed **DNS resolution**; for example, we might resolve the DNS name `www.cs.bris.ac.uk` into the IP address `137.222.102.8`.

- Imagine a process (say \( \mathcal{A} \)) on one host needs to communicate a message to a process (say \( \mathcal{C} \)) on another host. Clearly the host \( \mathcal{C} \) is executing on is identified by an IP address; as such, we can be confident packets sent by \( \mathcal{A} \) will at be routed to the correct target. But how does that target know to deliver the packets to \( \mathcal{C} \) rather, for example, than other processes (say \( \mathcal{B} \) or \( \mathcal{D} \))?
  
  This problem is solved by the network stack (more formally as part of the TCP protocol) maintaining a set of **ports** [26], each of which has a numerical identifier between 1 and 65535. A given process can send and receive data to and from a specific port via the network stack. As long as \( \mathcal{A} \) marks packets it sends with the port number \( \mathcal{C} \) is using, the packets always end up in the right place.

  **Standard services** are often (pre-)assigned standard ports (e.g., HTTP normally uses port 80, SMTP uses port 25 and so on). You might find cases were a port number is appended to an IP address: when we write

  \[ 137.222.102.8:80 \]

  or

  \[ \text{www.cs.bris.ac.uk:80} \]

  the number after the colon refers to port 80 (specified using the IP address 137.222.102.8 or DNS name `www.cs.bris.ac.uk`).

Consider a somewhat simplified example, described using a diagram: we use a similar style to illustrate and explain scenarios throughout the tutorial. The idea here is to capture some (hopefully most) of the concepts above more coherently:

![Diagram](brainiac.cs.bris.ac.uk)

The diagram illustrates two processes, \( \mathcal{A} \) and \( \mathcal{C} \), that represent a web-browser and web-server respectively; \( \mathcal{A} \) executes on a host whose DNS name is `brainiac.cs.bris.ac.uk`, \( \mathcal{C} \) on a host whose DNS name is `www.cs.bris.ac.uk`. Imagine \( \mathcal{A} \) wants to send a 2 kB HTTP request \( M \) to \( \mathcal{C} \). First, \( \mathcal{A} \) resolves the DNS name `www.cs.bris.ac.uk` to the IP address 137.222.102.8; it sends \( M \) via the network stack on `brainiac.cs.bris.ac.uk`. The network stack then splits \( M \) into a 2-packet sequence \( P = \langle P_0, P_1 \rangle \), before sending each of the 1500 B packets via an Ethernet [9] card into the network. In this case, \( P_1 \) is not likely to be full: although two packets are required to communicate the whole of \( M \), they could cope with messages up to 2 · 1500 B = 3000 B in length. Either way, intermediate hosts then forward \( P_0 \) and \( P_1 \) until they reach the target host, where the network stack on `www.cs.bris.ac.uk` receives them: it reassembles the packets into \( M \), and presents them to the process using port 80, which is \( \mathcal{C} \), for it to use somehow.

The challenge now is to explore these concepts in a practical and concrete way. There are lots of related commands we could look at, but four or five alone allow a fairly good coverage and represent the focus in what follows.

### B.1.1 Exploring IP and DNS information for a host

#### B.1.1.1 Using hostname

The hostname command provides access to a limited amount of information about the local host, mainly related to IP and DNS addresses. It does this by invoking system calls such as `gethostname` on our behalf, each of which essentially asks the kernel a question about how the network stack is configured. Consider the following example
which was executed on the host `brainiac.cs.bris.ac.uk`. Notice that

- the `-f` option prompts `hostname` to print the full-qualified DNS name of the local host, which matches the default,
- the `-s` option prompts `hostname` to print the host name of the local host (i.e., the first part of the fully-qualified DNS name),
- the `-d` option prompts `hostname` to print the domain name of the local host (i.e., the last part of the fully-qualified DNS name), and
- the `-i` option prompts `hostname` to print the IP address of the local host.

The last command produces what might seem a surprising result: two IP addresses are printed, suggesting that both `137.222.102.127` and `127.0.0.1` be used to identify this host. Several sensible reasons explain why this is reasonable in general; for example, maybe the host has two network interfaces (e.g., two network cards, perhaps one wired and one wireless). In this specific case however, the IP address `127.0.0.1` always refers to the local host: externally this host is identified by `137.222.102.127`, but internally it can be referred to as either `137.222.102.127` (which might argue change, based on the network configuration) or `127.0.0.1` (which is always fixed).

### B.1.1.2 Using host and dig

`hostname` is all well and good, but has a drawback in that we can only use it to find out information about the local host! This is not a limit however: DNS is a distributed, networked system so we can query it for information about any host from any host. Among several alternatives, two in particular allow some useful examples:

- `host` is simple to use but focuses on the basic remit of translating DNS names to IP addresses and vice versa.
- `dig` is more complex (it has more options for example) but also more powerful: DNS servers house DNS records for each host that include more than simply a mapping between name and IP address, and `dig` enables a more complete exploration of this meta-data.

Consider the following examples, which both retrieve DNS-related information about the host `brainiac.cs.bris.ac.uk`:
The difference is immediately obvious: the output from host is simply the IP address of brainiac.cs.bris.ac.uk, but dig provides much more information (which may or may not be useful). What about the other direction, i.e., looking-up a DNS name given an IP address? Using the -x option, dig can also perform this type of reverse DNS look-up:

```
bash$ dig -x 137.222.102.127
;; Got answer:
;; ANSWER SECTION:
127.102.222.137.in-addr.arpa. 86400 IN PTR brainiac.cs.bris.ac.uk.
```

B.1.2 Checking a host is active using ping

We use the term “ping” as part of everyday life: to ping someone means to check on them, or remind it that something needs to be done, or just get their attention. The ping command applies a similar idea to a target host, testing whether it is operational (i.e., whether it can be connected to, meaning it is reachable via the network). It does this by sending special messages to the target host, formally these are Internet Control Message Protocol (ICMP) echo requests. A corresponding reply (or absence thereof) will allow ping to and interpret as meaning the target is operational (or not).

Consider the following three examples, wherein ping is limited to sending five echo request (by default it will continue until terminated) using the -c option:

```
bash$ ping -c 5 foo
ping: unknown host foo
bash$ ping -c 5 toybox.cs.bris.ac.uk
PING toybox (137.222.162.74) 56(84) bytes of data.
64 bytes from toybox (137.222.162.74): icmp_seq=1 ttl=64 time=1.82 ms
64 bytes from toybox (137.222.162.74): icmp_seq=2 ttl=64 time=0.206 ms
64 bytes from toybox (137.222.162.74): icmp_seq=3 ttl=64 time=0.240 ms
64 bytes from toybox (137.222.162.74): icmp_seq=4 ttl=64 time=0.209 ms
64 bytes from toybox (137.222.162.74): icmp_seq=5 ttl=64 time=0.241 ms
--- toybox ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4001ms
rtt min/avg/max/mdev = 0.206/0.544/1.825/0.640 ms
bash$ ping -c 5 snowy.cs.bris.ac.uk
PING snowy.cs.bris.ac.uk (137.222.162.74) 56(84) bytes of data.
64 bytes from snowy.cs.bris.ac.uk (137.222.162.74): icmp_seq=1 ttl=64 time=1.82 ms
64 bytes from snowy.cs.bris.ac.uk (137.222.162.74): icmp_seq=2 ttl=64 time=0.206 ms
64 bytes from snowy.cs.bris.ac.uk (137.222.162.74): icmp_seq=3 ttl=64 time=0.240 ms
64 bytes from snowy.cs.bris.ac.uk (137.222.162.74): icmp_seq=4 ttl=64 time=0.209 ms
64 bytes from snowy.cs.bris.ac.uk (137.222.162.74): icmp_seq=5 ttl=64 time=0.241 ms
--- toybox ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4001ms
```
In the first case the host foo does not exist, so ping cannot translate the DNS name into a usable target IP address for the ICMP packets. The second case is more interesting: toybox.cs.bris.ac.uk is translated into the IP address 137.222.102.74, but ICMP packets sent cannot be routed to the target for some reason. In the third case, ping demonstrates that snowy.cs.bris.ac.uk (whose IP address is 137.222.103.3) is operational: among other information, notice that

- for each message sent by ping, the round-trip time (i.e., the time taken to send the message and receive a response) is listed, and
- once ping as finished, some statistics are produced that in some sense describe the connection quality: the number of messages which failed to produce a response is recorded for example, as are maximum, minimum and average round-trip times.

### B.1.3 Exploring the path between hosts using traceroute

We already mentioned that messages, in the form of packets, are routed to from source to their target by making hops between intermediate hosts. A earlier network, the Advanced Research Projects Agency Network (ARPANET) [2], pioneered this approach. Among the original design objectives, scalability and reliability (versus a network with direct host-to-host connections) were central: the former is improved by removing for the need for \( n^2 \) potential direct host-to-host connections between \( n \) hosts, the latter is improved by virtue of the potential to change or update the routing strategy (e.g., to avoid some host that has failed).

The traceroute command can be used to give information about the route (i.e., the hops, or path taken between hosts) from the local host (acting as the source) to a target host. Like ping it uses ICMP echo requests, but also the Time-To-Live (TTL) feature. The idea is that if the TTL for some packet is set to \( t \), if the target host is not reached after \( t \) hops then an error is sent back to the source; traceroute harnesses this feature by successively increasing the TTL (sending a so-called probe), so at each step it discovers the next hop made from source to target.

Consider an example where the local host is brainiac.cs.bris.ac.uk, and the target is www.bbc.co.uk:

```
bash$ traceroute www.bbc.co.uk
traceroute to www.bbc.co.uk (212.58.246.91), 30 hops max, 60 byte packets
1 gate102b (137.222.192.252) 1.075 ms 1.207 ms 1.297 ms
2 172.22.0.201 (172.22.0.201) 0.327 ms 0.282 ms 0.256 ms
3 fr2-br4.nwpp.bris.ac.uk (137.222.250.51) 0.495 ms 0.477 ms 0.453 ms
4 xe-0-3-brisub-rbri.ja.net (146.97.144.1) 0.881 ms 0.764 ms 0.748 ms
5 ael.bris-rbri.ja.net (146.97.35.299) 1.066 ms 0.986 ms 0.884 ms
6 ael4.read.rbri.ja.net (146.97.33.118) 2.726 ms 2.755 ms 2.726 ms
7 ae13.lond-rbri.ja.net (146.97.33.146) 5.725 ms 5.575 ms 5.529 ms
8 po1.lond-ban3.ja.net (146.97.35.196) 20.821 ms 19.443 ms 196.399 ms
9 bbc.lond-ban3.ja.net (193.62.157.6) 4.250 ms 4.284 ms 4.250 ms
10 * * *
11 ae0.er01.cwwtf.bbc.co.uk (132.185.254.93) 5.517 ms 54.763 ms 54.744 ms
12 132.185.255.164 (132.185.255.164) 8.311 ms 8.294 ms 8.315 ms
13 bbc-vi012.cwwtf.bbc.co.uk (212.58.246.91) 5.435 ms 5.400 ms 5.358 ms
bash$
```

Of course traceroute accepts numerous other options that each allow more advanced behaviour, but even here we get

- a list of hosts that the packages are routed via, implying the number of hops required, and
- the latency of each hop (one entry per-host, per-probe of that host of which there could be several) between hosts, meaning the delay associated with communication at that point.

In this example, notice that packets start within a network associated with the University of Bristol, then travel over the so-called Joint Academic NETWORK (JANET) which rough acts as an Internet Service Provider (ISP) for Universities in the UK, before reaching a network associated with the BBC.

### B.1.4 Inspecting network configuration using netstat

On most Linux distributions, the file /etc/services holds a list of standard service names and their corresponding port numbers. It does not tell us anything about which services are actually active however. The netstat command can be used to do this, and indeed act as a general way to explore the active network configuration (on the local host). In particular, it provides information and statistics about

```bash

```
An aside: security issues with the ICMP echo (AKA ping) request and reply process.

The word “usually” in the description of how a target host should respond to an ICMP echo request might sound a little vague: the point is that the host can opt out by not producing an ICMP echo reply, meaning ping will assume it is non-operational. Why might it do this? One reason is that the host wants to avoid being (easily) discovered, but others exist as well:

1. It takes some resource (time, bandwidth etc.) in order to process an ICMP echo request and generate a reply; the level of resource might be small, but is certainly non-zero. With this in mind, one class of Denial-of-Service (DoS) attack [6] is the so-called ping flood [24]. The idea is simple: swamp some target host with lots of ICMP echo requests, meaning it has too little time or bandwidth to operate as normal (i.e., execute legitimate processes).

2. Like any software, the network stack on a target host might contain a bug. In fact, several such bugs relating to ICMP echo requests have been found (in more than one OS) and exploited. One example is the so-called Ping-of-Death (PoD) [25] where an oversized ICMP echo request is sent by an attacker to the target host; if the host cannot cope with this gracefully (due to the bug), the resulting error could actually crash the target and force it to reboot!

Both attacks are powerful in the sense they can be mounted remotely over the network: the attack does not need access to log into the target host, for example.

An aside: cryptic hosts and routes.

When using traceroute, there are at least two types of entry in the output which could be confusing:

1. When information about a host cannot be determined, it is replaced by asterisk character. Various reasons exist for such an event, but the most basic is that the host failed to reply within the time limit; this can be extended using the -w option. Another could be a failure to complete a reverse DNS look-up (i.e., the resolution of a DNS name from an IP address): this process can be avoided using the -n option. Finally, it could be the case that the hop implies some non-standard host; examples include a gateway [11] that connects together different types of network.

2. Sometimes, hosts appear which you might not expect. In the example, we get something other than www.bbc.co.uk: why is this? It might be a result of load balancing [18]: if a single host could not cope with the volume of accesses to a web-site for example, a pool of hosts would be tasked with managing the workload between them.
1. network interfaces (i.e., connections from the local host to a given network provided via wired or wireless network cards) via the `--interface` option,

2. network routing tables (i.e., where the local host will send packets via in order to reach a given target host) via the `--route` option,

3. network connections (e.g., incoming and outgoing, active and inactive connections) via options such as `--listening`, and

4. network protocols (e.g., how much data has been sent specifically via TCP connections) via options such as `--tcp`, `--udp` and `--raw`.

It should be noted that a lot of the same information will be exported by the kernel via the `/proc/net/` file system; for example, `/proc/net/route` lists the routing tables. As such, you can think of `netstat` as simple a standard way to collect this and present it in a more human-readable form. Also note that throughout, the option `--numeric` is used to force output in a numeric form (e.g., as an IP address rather than DNS name).

### B.1.4.1 Network interfaces and routing tables

Consider examples of the first two use-cases:

```
bash$ netstat --numeric --interface
Kernel Interface table
Iface MTU Met RX-OK RX-ERR RX-DRP RX-OVR TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0 1500 0 83130698 0 0 0 59854021 0 0 0 BMRU
lo 16436 0 5186068 0 0 0 5186068 0 0 0 LRU
bash$ netstat --numeric --route
Kernel IP routing table
Destination Gateway Genmask Flags MSS Window Irtt Iface
137.222.102.0 0.0.0.0 255.255.255.0 U 0 0 0 eth0
169.254.0.0 0.0.0.0 255.255.0.0 U 0 0 0 eth0
0.0.0.0 137.222.102.0 0.0.0.0 UG 0 0 0 eth0
bash$
```
A packet for the target 169.254.1.1 matches the second rule (since 169.254.1.1 AND’d with the mask 255.255.0.0 is 169.254.0.0) for example, but the gateway entry 0.0.0.0 means it does not need to be forwarded: this is a special link-local [17] range.

All other packets match the third rule (since the destination 0.0.0.0 is a default meaning any packet), and are forwarded to 137.222.102.250 via the eth0 interface (then presumably onward from there).

B.1.4.2 Network connections

The third use-case for netstat is as a means of checking which ports are currently in use, and what for. There are numerous reasons to do this: for example, perhaps you want to disable any services you do not need (as a form of optimisation for both performance and security), or perhaps you want to execute a service but first need to check whether the port is already in use. Either way, netstat can identify ports which are listening for connections:

```
bash$ netstat --numeric --tcp --listening
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 0 0.0.0:5672 0.0.0:* LISTEN
tcp 0 0 0.0.0:621 0.0.0:* LISTEN
tcp 0 0 0.0.0:111 0.0.0:* LISTEN
tcp 0 0 0.0.0:47284 0.0.0:* LISTEN
tcp 0 0 0.0.0:22 0.0.0:* LISTEN
tcp 0 0 0.0.0:55516 0.0.0:* LISTEN
tcp 0 0 0.0.0:57155 0.0.0:* LISTEN
tcp 0 0 :::57901 :::* LISTEN
tcp 0 0 :::59896 :::* LISTEN
tcp 0 0 :::111 0.0.0:* LISTEN
tcp 0 0 :::22 0.0.0:* LISTEN
tcp 0 0 :::25 0.0.0:* LISTEN
tcp 0 0 :::5989 0.0.0:* LISTEN
tcp 0 0 :::58246 0.0.0:* LISTEN
bash$
```

```
cat /etc/services | grep '111/tcp'
sunrpc 111/tcp portmapper rpcbind # RPC 4.0 portmapper TCP
lmsocialserver 1111/tcp # LM Social Server
dsatp 2111/tcp # DSATP
wsynch 3111/tcp # Web Synchronous Services
xgrid 4111/tcp # Xgrid
taep-as-svc 5111/tcp # TAEP AS service
spc 6111/tcp # HP SoftBench Sub-Process Control
vce 11111/tcp # Viral Computing Environment (VCE)
fs-qos 41111/tcp # Foursticks QoS Protocol
bash$ cat /etc/services | grep '1012/tcp'
bash$
```

Each entry details a (potential) connection between the local host and some remote host; the information includes

- the protocol used (in this case, the --tcp option limits this to connections using TCP only),
- a count of the number of bytes in receive and send queues (which essentially represent data which has not yet been processed by the local or remote host),
- the local address (meaning the address on the local host, including the port number),
- the remote (or foreign) address (meaning the address of the remote host),
- the connection state (in this case, the --listening option limits this to connections in the listening state only).

Various obvious cases stand out in our example (the local host is clearly listening for SSH and SMTP connections on ports 22 and 25), but it also includes some less obvious cases; we can try to identify these using /etc/services, but this is not a definitive list. Port 111 seems to relate to the Remote Procedure Call (RPC) [27] system for instance, but port 1012 is not included (this could potentially be in use by a non-standard user program for example).

B.1.4.3 Network protocols

The forth and final use-case for netstat is to dump statistics relating to a particular network protocol. For example, the following

```
```

327
B.2 Networked inter-process communication using netcat

netcat (represented by the command nc, and literally a networked version of cat) is often described as a “swiss-army knife” of networking, since it can act as a solution for (or within) such a wide range of tasks. In the following, we use nc within the context of two such tasks in order to illustrate host-to-host communication; this moves the discussion up a level, with less emphasis on the underlying network stack and more on use at an application level.

B.2.1 A simple messaging system

Arguably the simplest use of nc, and certainly a good way to start, is to connect the stdin or stdout streams attached to a given process with a network port instead. More concretely, imagine a task that involves the combination of two processes: we might normally combine the processes locally using a pipe (e.g., via a command pipeline). By using nc this combination can be remote, meaning the pipe is essentially realised by the network. So assume we want to connect nc to port 1234 on some host whose DNS name is foo, replicating Figure B.1a. We can operate the command in two different modes:

1. On foo itself, we can use nc as a server (meaning it waits or listens for a connection) via

   nc -l foo 1234

   noting the -l option which specifies this mode. Used as such, nc will read input from the port and write it as output to stdout.

2. On any host (perhaps foo as well, or some other host), we can use nc as a client (meaning it initiates a connection) via

   nc foo 1234

   In contrast to the above, nc will now read input from stdin and write it as output to the port.
Figure B.1: Three scenarios describing use of $\text{nc}$ to send from a source host to a target host.
Diagrammatically, execution of the two commands can be viewed as implying the scenario in Figure B.1a. Illustrating these commands being executed from the command-line is a little more tricky that some others, due to the level of (concurrent) user interaction. So instead of relying on an example, step through the following tasks to see how this works yourself:

<table>
<thead>
<tr>
<th>Open two terminal windows, referred to as terminal #1 and #2. Strictly in order, first</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. in terminal #1, execute the command</td>
</tr>
<tr>
<td>\texttt{nc -l localhost 1234}</td>
</tr>
<tr>
<td>which acts as the server, then</td>
</tr>
<tr>
<td>2. in terminal #2, execute the command</td>
</tr>
<tr>
<td>\texttt{nc localhost 1234}</td>
</tr>
<tr>
<td>which acts as the client</td>
</tr>
</tbody>
</table>

so both commands refer to the local host via the DNS name \texttt{localhost}. Once connected, this produces a messaging system (akin to the UNIX \texttt{talk} [33] command) between the terminals: whatever you type into terminal #2 will be displayed on terminal #1. You can terminate the connection either via Ctrl-C (forcibly terminating the process) or Ctrl-D (marking the end of input, at which point the process will terminate naturally) on either terminal.

We could replace \texttt{localhost} with any IP address or DNS name, allowing extension from local-only to remote communication. Either

1. log terminal #2 into a second host (using SSH for example), or
2. find a friend already using a second host (e.g., the person next to you in the lab) and collaborate with them.

Remember that you can find the DNS name and IP address of either host using commands outlined in Section B.1.1.

Repeat Task 58 but for terminal #2 acting as the client, replace \texttt{localhost} with a reference to the host you run the server on (i.e., either the IP address or DNS name); note the communication is now genuinely via the network (since the client and server are different computers).

\subsection{B.2.2 Some simple HTTP-based applications}

At a (very) basic level, a web-browser simply reads and writes data to and from a port on some web-server (using the HTTP protocol [13]) and displays the result. On the other side, a web-server is just a process running on some host that receives input (i.e., requests, or commands again using the HTTP protocol) and produces output (i.e., responses, or content). The fact we have already used \texttt{nc} to receive and produce input and output over the network suggests it can be used as a (very) basic web-browser and/or web-server: this is exactly our goal in the following.

\subsubsection{B.2.2.1 A web-browser}

Consider the following command pipeline:

\texttt{echo -n -e 'GET /index.html HTTP/1.0\r\n\r\n' | nc www.cs.bris.ac.uk 80}

The right-hand half should be familiar in that we ask \texttt{nc} to connect to port 80 of \texttt{www.cs.bris.ac.uk}. Rather than read input from the terminal (i.e., have someone type it) however, \texttt{nc} reads input from \texttt{echo} instead: all output the invocation of \texttt{echo} writes to stdout is fed, via the pipe, as input on stdout to the invocation of \texttt{nc}.

The -n and -e options prompt \texttt{echo} to avoid producing a trailing new line automatically, but expand the escaped characters we specify into carriage return and new line respectively. As a result, echo produces
**Figure B.2:** Example output from `nc` when used as a simple web-browser to load the web-page `index.html` from `www.cs.bris.ac.uk`.
GET /index.html HTTP/1.0

followed by a blank line: this is a request for the web-server to supply the file /index.html (i.e., www.cs.bris.ac.uk:80/index.html). The response produced is illustrated in Figure B.2, where use of head produces the first 50 lines only (of what is a large file, preceded by the HTTP header).

<table>
<thead>
<tr>
<th>Implement (task #60)</th>
</tr>
</thead>
</table>
| Reproduce the steps above to download content from http://www.google.com/index.html using nc, then do the same thing with a real web-browser. Depending on where in the world you do this from (e.g., the UK versus the US), the result might be surprising: using output from the former, try to explain (even informally)
| - what each line of the HTTP header and content means, and
| - what the real web-browser is therefore doing automatically on your behalf. |

<table>
<thead>
<tr>
<th>Research (task #61)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The details of HTTP are normally hidden from you by a web-browser, but in the above we use it directly to perform what is termed a GET request (i.e., to get some content). Do some research into other request types (examples include the HEAD, OPTIONS or TRACE request types), then alter the example above to perform at least one different request via port 80 of <a href="http://www.cs.bris.ac.uk">www.cs.bris.ac.uk</a>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Research (task #62)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Although you may be more used to GUI-based web-browsers, e.g. Firefox or Chrome, text-based web-browsers also exist: like nc, these can be extremely useful when automating tasks from the command-line. Standard examples include the venerable links web-browser, and the wget content retrieval command. Focusing on the latter, do some research into how wget works then use it, in a similar way as above, to again retrieve content from <a href="http://www.google.com/index.html">http://www.google.com/index.html</a></td>
</tr>
</tbody>
</table>

B.2.2.2 A (one-shot) web-server

What about the other side of this scenario? Imagine we create a file that represents a web-page we want to serve (via the network) to a web-browser. Given such a file, say A.txt, an nc-based web-server can be executed as follows

```bash
{ echo -n -e 'HTTP/1.0 200 OK\r\n\n' ; cat A.txt ; } | nc -l localhost 1234
```

where within the command pipeline

1. the left-hand side uses echo and cat to print the HTTP header and file content respectively; their combined output (which you can think of as merged together as a result of the curly braces around both commands) is piped into

2. the right-hand side, which uses an invocation of nc as before: it listens for a connection to port 1234 of the local host, and when one is made it produces the content as required.

Try it out yourself:
Open two terminal windows, referred to as terminal #1 and #2. Strictly in order, first

1. create a file called A.txt, containing whatever content you want, but perhaps HTML for example,
2. to make things easier, in terminal #1 first create an alias
   
   alias S="\{ echo -n -e 'HTTP/1.0 200 OK\r\n\r\n' ; cat A.txt ; \}"

   then execute the command

   S | nc -l localhost 1234

   which acts as the web-server, then
3. to make things easier, in terminal #2 first create an alias
   
   alias C="echo -n -e 'GET /index.html HTTP/1.0\r\n\r\n'"

   then execute the command we used previously

   C | nc localhost 1234

   which acts as the web-browser (or client).

This nc-based web-server is very basic of course. For example once it has satisfied the request by sending A.txt to the web-browser it simply terminates; in addition, it makes no difference what request is sent in that the web-browser gets A.txt even if it asked for something else.

We can use a BASH loop to resolve the first problem. Assuming reuse of the alias representing the server, repeat Task 63 but use

   while true ; do S | nc -l localhost 1234 ; done

in terminal #1: now the web-server executes forever (at least until terminated using Ctrl-C), meaning it will now deal with multiple requests.

Rather than use nc as a web-browser, try using a real web-browser such as Chrome or links to access the URL

   http://localhost:1234/A.txt

connected to the web-server from Task 64. In the web-browser, you should see the content of A.txt; in the web-server terminal window you should see the actual HTTP requests sent by the web-browser when trying to access A.txt.

B.3 Using OpenSSL-based cryptographic primitives

You may recognise OpenSSL as an open-source implementation of the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols [35]. Although you probably use OpenSSL (even if unknowingly) as a component within web-browsers for example, the project provides a very general-purpose library and a suite of command-line tools. From a practical perspective, the latter are enormously useful: making use of them to actually do things with cryptography can be both instructive and rewarding. As such, the simple aim of this Section is to explore various common tasks one might undertake using OpenSSL from the command-line.

When invoked from the command-line using the openssl command, note that the first, compulsory option determines the operation performed: this is following by normal options that further control the operation.
B.3.1 Symmetric encryption and decryption operations

The most fundamental cryptographic operation is almost certainly encryption (or decryption) of data, with the easiest approach to doing so being use of a block cipher \[3\]. OpenSSL supports numerous block cipher algorithms, in numerous modes of operation \[4\]: we can check the AES-based \[1\] possibilities as follows:

```
bash$ openssl list -cipher -commands | grep aes
aes -128-cbc
aes -128-ecb
aes -192-cbc
aes -192-ecb
aes -256-cbc
aes -256-ecb
bash$
```

noting that 128-bit, 192-bit and 256-bit key sizes are possible, as are ECB and CBC modes of operation.

B.3.1.1 Using files (and streams)

The first task is to retrieve some data (as elsewhere, some Shakespearean text) to experiment with:

```
bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws4110.txt'
bash$
```

after which we can encrypt it, then decrypt to get the same result:

```
bash$ openssl enc -e -aes -128-ecb -k 'secret' -in A.txt -out B.txt
bash$ openssl enc -d -aes -128-ecb -k 'secret' -in B.txt -out C.txt
```

The first option `enc` dictates use of a block cipher, with the other options providing extra control:

- `-e` (resp. `-d`) specifies that encryption (resp. decryption) of the input should be performed to produce the output,
- `-aes-128-ecb` specifies that the AES block cipher should be used, with a key size of \( k = 128 \) bits and in ECB mode,
- `-k` specifies the password used for the encryption (resp. decryption) operation, and
- `-in` (resp. `-out`) specifies the input (resp. output) file name.

In common with most uses of, the input (resp. output) file can be replaced by the standard stream `stdin` (resp. `stdout`) by simply removing the option. To get the same result, we might therefore

```
bash$ cat A.txt | openssl enc -e -aes -128-ecb -k 'secret' -in -out B.txt
bash$ cat B.txt | openssl enc -d -aes -128-ecb -k 'secret' -in -out C.txt
```

334
Clearly this can, and will be useful later when the command is used in a larger command pipeline. In addition to encryption and decryption, each example includes four extra commands whose purpose is to demonstrate the content of each input and output. Although we know \texttt{A.txt} represents ASCII text, it is important to remember that a block cipher will process it as binary data: it reads (resp. writes) 8-bit bytes of data to form the 128-bit plaintext (resp. ciphertext) message block. The first three commands illustrate this by inspecting the first four blocks (64 bytes in total) of each file. Notice the match between \texttt{A.txt} and \texttt{C.txt}, which is further confirmed by using \texttt{cmp} to perform a complete comparison: the lack of output indicates the files are identical, which we expect as decryption should “undo” encryption (under the same key).

In each of the examples above -k specifies an ASCII password (i.e., the string “secret”). This might seem odd if you know about block ciphers: specifically, they do \textit{not} actually use a password but rather a $k$-bit key $K$. OpenSSL actually forms such a key from the password using a \textit{Key Derivation Function (KDF)}. In more detail, the hash function $\text{MD5}[20]$ is used: without a salt \texttt{MD5} produces $K$ from the password “secret”:

\[
K = \text{MD5}(\text{"secret"}) = 5EBE2294ECD0E0F8EAB7690D2A6EE69_{(16)}
\]

used as the key, but with salt we instead get whatever

\[
K = \text{MD5}(\psi \parallel \text{"secret"})
\]

produces given a random salt value $\psi$. In this case (and others) it can be useful to inspect the resulting key, with the -p option instructing OpenSSL to do so. For example, in the following (where the output is discarded)

\[
\text{bash}\$\text{openssl enc -aes-128-ecb -nosalt -k \"secret\" -p -in A.txt -out /dev/null}
\]

\[
\text{key=5EBE2294ECD0E0F8EAB7690D2A6EE69}
\]

note that we get the same derived key as above from the same password. Alternatively, we could specify the key (and/or IV, where the mode of operation allows one) explicitly: on one hand this is perhaps less user friendly, but on the other hand it allows more direct control over the block cipher. In this example, we set the cipher key to

\[
K = 000102030405060708090A0B0C0D0E0F_{(16)}
\]

using the -K option (plus the -iv option to set a zero‘ed IV) as follows:

\[
\text{bash}\$\text{openssl enc -aes-128-ecb -K 000102030405060708090A0B0C0D0E0F -iv 0 -in A.txt -out B.txt}
\]

\[
\text{bash}\$\text{openssl enc -d -aes-128-ecb -K 000102030405060708090A0B0C0D0E0F -iv 0 -in B.txt -out C.txt}
\]

However, if we specify neither a password or key then OpenSSL prompts us to first enter then verify a password choice by typing it:

\[
\text{bash}\$\text{openssl enc -e -aes-128-ecb -in A.txt -out B.txt}
\]

\[
\text{enter aes-128-ecb encryption password:}
\]

\[
\text{Verifying - enter aes-128-ecb encryption password:}
\]

\[
\text{bash}\$\text{openssl enc -d -aes-128-ecb -in B.txt -out C.txt}
\]

335
B.3.1.2 Across the network

The fact OpenSSL can deal with stream-based as well as file-based input and output suggests an interesting next step: we can already use netcat to perform networked communication, so why not do this in a secure manner by also using OpenSSL to encrypt the communicated data? Following the original example in Figure B.1a, again consider a host whose DNS name is foo. Using nc interactively (per the messaging system example) is a little tricky, but simply communicating the contents of a file is much easier:

1. On foo itself, we can use nc as a server (meaning it waits or listens for a connection) via

   ```bash
   nc -l foo 1234 | openssl enc -d -aes-128-ecb -k 'secret' > B.txt
   ```

   where the output is fed through openssl to decrypt it using AES, forming B.txt.

2. On any host (perhaps foo as well, or some other host), we can use nc as a client (meaning it initiates a connection) via

   ```bash
   cat A.txt | openssl enc -e -aes-128-ecb -k 'secret' | nc foo 1234
   ```

   where the input A.txt is fed through openssl to encrypt it using AES.

Replacing foo with the local host whose DNS name is localhost, as you did before in Task 58, reproduce the steps above to form an encrypted version of the previous messaging system; verify that the file B.txt received by terminal #1 matches the file A.txt sent by terminal #2.

Clearly the client and server must operate symmetrically wrt. encryption. Imagine, for example, they use

- a different key, or
- a different block cipher (or mode of operation).

First try to explain what should happen in theory, and then try the options in practice and see if the result matches what your explanation.

This is in fact so easy, it is difficult to see the value in doing so. Put another way, the point of using encrypted communication is to thwart an attack of some sort: without one, why bother?! In Figure B.1b we find some motivation, where versus the initial scenario in Figure B.1a a man-in-the-middle is now included. In an IP-based network, we have already know packets are routed through intermediate hosts. As a result, it should be no surprise that a host can inspect and/or store any packet routed through it. The following attempts to give a practical illustration of this:
Open three terminal windows, referred to as terminals #1, #2 and #3. Strictly in order, first

1. in terminal #1, execute the command

   nc -l localhost 5678

   which acts as the server, then

2. in terminal #2, execute the command

   nc -l localhost 1234 | tee P.txt | nc localhost 5678

   which acts as the passive man-in-the-middle, then

3. in terminal #3, execute the command

   nc localhost 1234

   which acts as the client.

Once connected, this produces a messaging system as before: a message typed into terminal #3 appears on terminal #1. However, now the passive man-in-the-middle also captures messages into a file called P.txt.

Reproduce Task 67, but incorporate use of OpenSSL into the command pipelines for terminals #3 and #1 so each message they communicate are encrypted (using AES say). Before this change, the man-in-the-middle could simply read P.txt, but now it needs to know (and use) the key to do so.

Verify this: inspect the contents of (the encrypted) P.txt, then decrypt it with the key used by terminals #3 and #1.

### B.3.2 Cryptographic hash and MAC operations

In the same way as block ciphers, OpenSSL allows use of numerous hash functions; the following

```bash
bash$ openssl list -message -digest -commands
md4
md5
mdc2
rmd160
sha
sha1
bash$
```

again checks the possibilities, which include MD4, MD5 and SHA1.

#### B.3.2.1 Using files (and streams)

First fetching the same data to work with as above,

```bash
bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws4110.txt'
bash$
```

applying SHA1 [30], or a Message Authentication Code (MAC) based on it, say HMAC-SHA1 [12], is fairly straightforward:

```bash
bash$ openssl dgst -sha1 A.txt
SHA1(A.txt)= a3983edd17c2c820a19117819b77dd58446491e4f
bash$ openssl dgst -sha1 -hmac 'secret' A.txt
HMAC-SHA1(A.txt)= cef2862a5322880d1f6e5f5f14c487fd9e983219
bash$
```

Of course if we alter A.txt somehow, for instance we change all the ‘a’ characters to ‘b’, forming B.txt, then applying either SHA1 or HMAC-SHA1 produces a totally different output:

```bash
bash$ cat A.txt | tr 'a' 'b' > B.txt
bash$ openssl dgst -sha1 A.txt
SHA1(B.txt)= 2a1312c9d8b92f8fd490736765c59abece3210444
bash$ openssl dgst -sha1 -hmac 'secret' B.txt
HMAC-SHA1(A.txt)= a6126eb95388dc9c89e96672f9a7c15dd6bba9f
bash$
```
OpenSSL uses a slightly annoying output format in the sense we often only want the hash function digest or MAC tag: this can be extracted using \texttt{cut}, for example

\begin{verbatim}
bash$ openssl dgst -sha1 A.txt | cut -d ' ' -f 2
a3983edd17c2cba19137819b7dd5d844c6491e4f
bash$ openssl dgst -sha1 -hmac 'secret' A.txt | cut -d ' ' -f 2
cfe2b6e6f3228b6bf1e5ff5f14c847fd9e983219
bash$
\end{verbatim}

\section*{B.3.2.2 Across the network}

Figure B.1c offers a third scenario where the previously passive man-in-the-middle (in Figure B.1b) now becomes an active man-in-the-middle: instead of just taking a copy of packets communicated through it, packets are now manipulated somehow before they reach the target.

\begin{center}
\begin{tikzpicture}
\node[draw, align=center] (A) {Revisit Task 67 (with a passive man-in-the-middle based on use of the tee command, without any form of encryption), but alter the command used by terminal \#2 from \begin{verbatim}
nc -l localhost 1234 | tee P.txt | nc localhost 5678
\end{verbatim} to \begin{verbatim}
nc -l localhost 1234 | sed -u -e 's/a/e/g' | nc localhost 5678
\end{verbatim} instead: the sed command (using -u to operate in unbuffered mode) means 'a' characters sent by terminal \#3 will now be translated into 'e' before reaching terminal \#1. That is, the active man-in-the-middle manipulates the communication rather than simply observing it.

Verify this works by typing some input into terminal \#3: note that the output in terminal \#1 might differ (where you type an 'a') from that sent by terminal \#3.}
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}
\node[draw, align=center] (A) {sed can also process binary input: replace the command \begin{verbatim}
nc -l localhost 1234 | sed -u -e 's/a/e/g' | nc localhost 5678
\end{verbatim} in Task 69 with \begin{verbatim}
nc -l localhost 1234 | sed -u -e 's/\x00/\x11/g' | nc localhost 5678
\end{verbatim} instead: this means every byte with the value 00\textsubscript{16} is now translated into 11\textsubscript{16}.

Now encrypt communication between terminals \#3 and \#1 by again using OpenSSL. That is, communicate the contents of a file A.txt from terminal \#3 to terminal \#1 (via this new man-in-the-middle) and store the content in B.txt, then verify whether A.txt and B.txt match. Manipulation by the active man-in-the-middle should ensure there are some differences (depending on the original file content, and which bytes the man-in-the-middle manipulates).}
\end{tikzpicture}
\end{center}
Without access to A.txt and B.txt, we cannot verify they match (as per Task 70). So how can we detect whether or not the communication is manipulated? One way is to employ a MAC. This task is somewhat complicated, but the idea is to start by revisiting Task 58 (without a man-in-the-middle, or encryption), and see how to do this:

1. in terminal #1, execute the command

   \texttt{nc -l localhost 1234 > B.txt}

   which acts as the server, then

2. in terminal #2, compute a MAC tag for A.txt

   \texttt{cat A.txt | openssl dgst -sha1 -hmac 'secret' | cut -d ' ' -f 2 > T.txt}

   then send T.txt and A.txt

   \texttt{cat T.txt A.txt | nc localhost 1234}

   rather than just the content in A.txt, then

3. back in terminal #3, in B.txt we now have the MAC tag and file content; first we need to separate them

   \texttt{cat B.txt | head -c 41 > C.txt}

   \texttt{cat B.txt | tail -c +42 > D.txt}

   (where 41 hexadecimal characters represents 160 bits of SHA1 output plus a newline character), before recomputing the MAC on D.txt (i.e., the file content) as follows

   \texttt{cat D.txt | openssl dgst -sha1 -hmac 'secret' | cut -d ' ' -f 2 > E.txt}

   and finally comparing the result in E.txt with the MAC tag sent

   \texttt{cmp C.txt E.txt}

If/when you get this working, the next step is to (re)introduce the active man-in-the-middle: have it manipulate the communication from terminal #3 to terminal #1, and verify that comparing the communicated and recomputed MAC tags can detect this.

\subsection{Asymmetric encryption and decryption operations}

Using an asymmetric primitive to encrypt and decrypt data is almost as easy as with a symmetric primitive (such as the block cipher AES, which we already explored). However, before doing so, and having fetched some data as before

\texttt{bash$ wget -q -U chrome -O A.txt 'http://www.gutenberg.org/dirs/etext97/1ws4110.txt'
bash$}

we first need to generate a pair of public and private keys. When using RSA [28] for example, this is achieved as follows

\texttt{bash$ openssl genrsa -out rsa_sk.pem 1024
Generating RSA private key, 1024 bit long modulus
..............................................++++++
.e is 65537 (0x10001)
bash$ openssl rsa -in rsa_sk.pem -pubout -out rsa_pk.pem
writing RSA key
bash$}

with the second command extracting the public key into a separate file for clarity. We can inspect the resulting key material as follows:

\texttt{bash$ openssl rsa -pubin -in rsa_pk.pem -noout -text
Public-Key: (1024 bit)
Modulus:

339}
You need to know how RSA works to interpret the output, but based on the overview in Chapter 10 one can identify hexadecimal values of $p$ and $q$, $N$, $e$ and $d$ for instance. Now we can try to encrypt A.txt, then decrypt the result again much like we did previously with the symmetric example:

```
bash$ openssl rsautl -encrypt -pubin -inkey rsa_pk.pem -in A.txt -out B.txt
RSA operation error
140448904599208: error :0406D06E:rsa routines:RSA_padding_add_PKCS1_type_2:data to o large for key size:rsa_pk1.c:151:
bash$
```

There is a problem: RSA can only encrypt messages, say $m$, which are smaller than $N$. Put another way, it requires $0 \leq m < N$ but we gave it $m = A.txt$ which is many kilo bytes in size and hence $m > N$. In reality, we would need to split A.txt into suitably sized blocks. Here however, imagine we only care about the first 64 B block of the file. We first extract this block into B.txt using head, then proceed as follows:

```
bash$ cat A.txt | head -c 64 > B.txt
bash$ openssl rsautl -encrypt -pubin -inkey rsa_pk.pem -in B.txt -out C.txt
bash$ openssl rsautl -decrypt -inkey rsa_sk.pem -in C.txt -out D.txt
bash$
```

You need to know how RSA works to interpret the output, but based on the overview in Chapter 10 one can identify hexadecimal values of $p$ and $q$, $N$, $e$ and $d$ for instance. Now we can try to encrypt A.txt, then decrypt the result again much like we did previously with the symmetric example:

```
bash$ openssl rsa -in rsa_sk.pem -noout -text
Private-Key: (1024 bit)
modulus:

c:5b:06:b5:29:92:cc:69:3f
publicExponent: 65537 (0x10001)

```

```
bash$
```

There is a problem: RSA can only encrypt messages, say $m$, which are smaller than $N$. Put another way, it requires $0 \leq m < N$ but we gave it $m = A.txt$ which is many kilo bytes in size and hence $m > N$. In reality, we would need to split A.txt into suitably sized blocks. Here however, imagine we only care about the first 64 B block of the file. We first extract this block into B.txt using head, then proceed as follows:

```
bash$ cat A.txt | head -c 64 > B.txt
bash$ openssl rsautl -encrypt -pubin -inkey rsa_pk.pem -in B.txt -out C.txt
bash$ openssl rsautl -decrypt -inkey rsa_sk.pem -in C.txt -out D.txt
bash$
```

You need to know how RSA works to interpret the output, but based on the overview in Chapter 10 one can identify hexadecimal values of $p$ and $q$, $N$, $e$ and $d$ for instance. Now we can try to encrypt A.txt, then decrypt the result again much like we did previously with the symmetric example:
As an aside, the split command can be useful in circumstances such as this: it could be used to split A.txt into as many equally sized blocks as required (rather than just extracting the first block as above). Either way, notice the encryption step uses the public key rsa_pub.pem, while the decryption step uses the private key rsa_pri.pem; in the former, we tell OpenSSL that we are using a public key via the -pubin option. As in the symmetric case, use of an given input (resp. output) file can of course be replaced by use of stdin (resp. stdout) by omitting the -in (resp. -out) option.

B.3.4 Asymmetric signature and verification operations

As above, using an asymmetric digital signature algorithm instead of a symmetric MAC is fairly similar. Once we have some data:

we again need to start by generating the public and private keys. For DSA [7], a further requirement is that we generate some domain parameters (the first command) which can be viewed as global data that everyone knows:

In this case, the generation process is driven by randomness read from /dev/urandom; once the domain parameters are generated we then generate the public and private keys (second and third commands) much like RSA, again extracting the public key into a separate file for clarity. The end result can again be inspected as follows:
bash$ openssl dsa -in dsa_sk.pem -noout -text
read DSA key
Private-Key: (1024 bit)
bash$ openssl dsa -in dsa_pk.pem -noout -text
read DSA key
Private-Key: (1024 bit)
bash$ openssl dsa -dss1 -sign dsa_sk.pem -out B.txt A.txt
bash$ openssl dss1 -verify dsa_pk.pem -signature B.txt A.txt
Verified OK
bash$

Now we can sign the data using our private key:

bash$ openssl dgst -dss1 -sign dsa_sk.pem -out B.txt A.txt
bash$ openssl dss1 -verify dsa_pk.pem -signature B.txt A.txt
Verified OK
bash$

The first command above produces the signature in B.txt, and the second command verifies this signature on A.txt using our public key: in this case, the two match and so the command succeeds. What happens if we try to verify the signature on some other message, say some C.txt where like the MAC example we change all the ‘a’ characters with ‘b’? As one might expect, this time the verification fails:

bash$ cat A.txt | tr 'a' 'b' > C.txt
bash$ openssl dgst -dss1 -verify dsa_pk.pem -signature B.txt C.txt
Verification Failure
bash$

Of course using an asymmetric primitive to sign a large message is fairly inefficient. To improve the example therefore, we could first apply a hash function the message and then sign the resulting digest, i.e.,

bash$ openssl dgst -sha1 A.txt | openssl dgst -dss1 -sign dsa_sk.pem -out B.txt
bash$ openssl dgst -sha1 A.txt | openssl dgst -dss1 -verify dsa_pk.pem -signature B.txt A.txt
Verified OK
bash$

The same reasoning applies: if we change A.txt then the digest produced by SHA1 will change, so the signature verification will fail. So provided the hash function is secure, signing the (shorter) digest produces the same result from a security perspective but is clearly more efficient.
B.4 Experimenting with SSL and TLS using OpenSSL

As we already mentioned, the high-level goal of OpenSSL is to support use of the SSL and TLS protocols; in a sense, everything else it offers is a side-effect of this. Put another way, SSL and TLS motivates provision of block ciphers etc. by OpenSSL since both protocols combine such primitives to achieve their high-level goal. As a result, experimenting with SSL and TLS can help illuminate more theoretical study. Specifically, we can see how they usefully combine the primitives we have already used by hand from the command-line.

Imagine we want to establish a secure communication channel between two hosts, say $A$ and $B$, which act as a web-browser (or client) and web-server respectively. Assuming the use of RSA-based signatures, we first need to generate appropriate public and private keys for the server:

```bash
genrsa -out server.pem 1024
```

This is fairly complicated, but can be roughly summarised as follows:

- the `-subj` option specifies the identity relating to $B$, representing the subject of the certificate\(^1\),
- the `-key` option specifies the file from which a public-key $PK_B$ and a private-key $SK_B$ are read,
- the `-out` option specifies the file into which a certificate $\sigma = \text{RSA.Sig}(SK_B, (B, PK_B))$ is then written.

We can inspect the result, via

```bash
x509 -in server.cert -noout -text
```

\(^1\) An identity can be formed from various fields; in this case “O” labels the organisation, “OU” labels the organisational unit, “L” labels the locality, and “CN” labels the common name.
noting, for example, that the public key information help within the certificate matches that in server.pem. Using the key material and certificate, we are now ready to establish the secure communication channel using TLS. OpenSSL includes a variety of tools that make this very easy:

- Executing

  ```bash
  openssl s_server -state -accept 1234 -key server.pem -cert server.cert -msg -www
  ```

  on some host, say `foo`, launches a TLS test server, where

  - `-state` instructs the server to produce a dump of all TLS session states,
  - `-accept` specifies the port to listen for connections on, which in this case is 1234,
  - `-key` and `-cert` specify the (previously generated) server key and certificate,
  - `-msg` instructs the server to produce a hexadecimal dump of all communicated messages, and
  - `-www` means the server will act as a (very) simple web-server, which allows connections over HTTP and sends back a simple message (cf. web-page) with information about the server.

- Executing

  ```bash
  openssl s_client -state -connect foo:1234
  ```

  on any host (including `foo`) launches a TLS test client, where

  - `-state` instructs the server to produce a dump of all TLS session states, and
  - `-connect` specifies the port to connect to (i.e., the port on which a server is listening for connections), which in this case is 1234 on `foo`.

Now try the following:

Reproduce the steps above, in both cases replacing `foo` with the local host whose DNS name is `localhost`: open two terminal windows, referred to as terminal #1 and #2, then strictly in order, first

1. in terminal #1, execute the test server as above, then
2. in terminal #2, execute the test client as above.

Once finished, terminate the test client in terminal #2. Then use a real web-browser such as Chrome or links to access the URL

```
https://localhost:1234/
```

and hence interact with the test server.

You already have the tools and have seen how to introduce a passive or active man-in-the-middle between a simple `nc`-based client and server. Can you reproduce such an attacker between a more complex TLS-based OpenSSL test client and server?

You should find TLS detects and/or prevents each attacker: using what you observe for example, identify the feature within TLS that does so in each case.
In Task 72, the information sent from the test server back to the test client or (real) web-browser included the TLS cipher suite [5] agreed: roughly speaking, this represents the set of algorithms used to secure communication. The cipher suite is agreed using a one-sided negotiation: as part of the TLS hand-shake protocol,

- the client sends a list of algorithms it supports (ordered by preference), then
- the server sends back a choice from that list.

We can control the list of algorithms sent by the OpenSSL test client using the -cipher option:

- Use the command
  
  ```bash
  openssl ciphers
  
  ```
  to produce a list of valid algorithms, or more specifically valid combinations of them. Given the output, do some research into the primitives each combination relates to: for example, what does EDH-RSA-DES-CBC-SHA mean?

- Revisit Task 72, but when executing the test client use -cipher to control the cipher suite agreed. By experimenting with the list, can you reason about when and why the server might make one choice over another?
BIBLIOGRAPHY


INDEX

n-gram, 137
Logo, 207
Phrack, 255

1260, 80

sink web-pages, 114

accumulator, 68
activation record, 242
adapt, 30
additive identity, 180
additive inverse, 180
address, 68
address mask, 320
address stream, 263
address translation, 262

AdSense, 103
Advanced Encryption Standard (AES), 269
Advanced Research Projects Agency Network (ARPANET), 65, 324
adversary, 127, 177
AdWords, 103
algorithm, 47
alpha, 206

American Standard Code for Information Interchange (ASCII), 86
anti-virus, 65
argument, 291
arguments, 242
ASCII art, 88
assignment, 49, 52, 53
attacker, 127

Automatic Sequence Controlled Calculator (ASCC), 69

Babington Plot, 127
backward compatibility, 99
base, 26
base case, 238
biased, 161

big-O notation, 58
bigrams, 137
binary, 28
Binary Coded Decimal (BCD), 99
bit, 29
black hat, 255
block, 178
block cipher, 178, 269
bounded, 52
bounds check, 233
Breadth-First Search (BFS), 110
broken, 127
brute-force, 133, 217
buffer overflow, 233
bug, 78, 233
built-in, 288
Bulletin Board Systems (BBSs), 88
byte, 29

cache, 259
cache line, 261
cache-hit, 259
cache-miss, 260
called, 235
callee, 235
caller, 235
Carmichael numbers, 48
ceiling, 57
Central Processing Unit (CPU), 67, 288
channel, 204
characters, 85
cipher, 129
ciphertext, 127
ciphertext only, 134
clock arithmetic, 131
CMYK, 207
co-prime, 48
codeword, 38
colour depth, 205
colour model, 204
column vector, 39  
Comma Separated Value (CSV), 298  
Command Line Interface (CLI), 290  
command pipeline, 294  
commands, 288  
Compact Disk (CD), 25  
compressed, 23  
computational complexity, 56  
computer, 288  
computer virus, 65  
concatenate, 17  
condition, 53  
conditional probability, 117, 162  
construct, 51  
converge, 116  
coprime, 184  
countermeasures, 222  
Creeper, 65, 66  
cryptanalysis, 127  
cryptography, 127  
cycles, 110  
damping factor, 119  
damping ratio, 119  
Data Encryption Standard (DES), 163  
data structure, 85  
debug, 78  
decimal, 25  
degree, 109  
Denial-of-Service (DoS), 325  
Depth-First Search (DFS), 110  
dictionary, 32, 219  
difference, 278  
dimension, 39  
direct-mapped cache, 263  
Discrete Logarithm Problem (DLP), 195  
divisors, 182  
DNS name, 321  
DNS resolution, 321  
Domain Name System (DNS), 321  
dot product, 39  
Droste effect, 236  
edges, 109  
Electronic Codebook (ECB), 178  
elements, 17  
encryption, 127  
entropy, 161  
entropy pool, 164  
equality, 49  
equivalent, 131  
equivalent to, 178  
error correction, 25  
error detection, 25  
escape code, 31, 34  
Euclidean algorithm, 188  
Euler Φ, 183  
even parity code, 35  
evict, 260  
executed, 51  
executes, 67, 288  
exponential back-off, 222  
Fibonacci numbers, 236  
file descriptor, 292  
First-In Last-Out (FILO), 239  
flag, 291  
flow chart, 49  
frequency analysis, 133  
functions, 235  
gateway, 325  
general-purpose, 67  
generating matrix, 41  
graph, 109  
graph exploration, 113  
graph theory, 109, 110  
graph traversal, 110  
Graphical User Interface (GUI), 290  
Greatest Common Divisor (GCD), 184  
Halting problem, 66  
Hamming code, 36  
hard disk, 288  
Harvard architecture, 69  
hexadecimal, 30  
hit ratio, 266  
hops, 320  
host, 65, 320  
HTML, 104  
HTTP, 104  
Hyperlink-Induced Topic Search (HITS), 114  
hyperlinks, 104  
hypertext, 104  
identity function, 180  
identity matrix, 42  
impact factor, 114  
implemented, 51  
index, 17  
infect, 65  
infinite loop, 77  
Instruction Register (IR), 67  
instruction set, 67  
instructions, 67, 288  
integers, 182  
interference, 264  
Internet, 65  
Internet Control Message Protocol (ICMP), 323  
Internet Protocol (IP), 320  
Internet Service Provider (ISP), 324  
inverse function, 269  
inverse permutation, 136  
invoke, 48  
IP address, 320  
iternates, 52  
iterative, 238  
iterative methods, 118  
John von Neumann, 69  
Joint Academic NETwork (JANET), 324
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joint Photographic Experts Group (JPEG), jpeg, jpg</td>
<td>30</td>
</tr>
<tr>
<td>Kerckhoffs Principle</td>
<td>131</td>
</tr>
<tr>
<td>kernel</td>
<td>288</td>
</tr>
<tr>
<td>key agreement protocols</td>
<td>178</td>
</tr>
<tr>
<td>Key Derivation Function (KDF)</td>
<td>335</td>
</tr>
<tr>
<td>key distribution problem</td>
<td>178</td>
</tr>
<tr>
<td>land</td>
<td>25</td>
</tr>
<tr>
<td>Last-In First-Out (LIFO)</td>
<td>239</td>
</tr>
<tr>
<td>least-significant</td>
<td>210</td>
</tr>
<tr>
<td>length</td>
<td>31</td>
</tr>
<tr>
<td>line</td>
<td>47</td>
</tr>
<tr>
<td>Linear Congruence Generator (LCG)</td>
<td>167</td>
</tr>
<tr>
<td>linear equations</td>
<td>118</td>
</tr>
<tr>
<td>load balancing</td>
<td>325</td>
</tr>
<tr>
<td>local</td>
<td>242</td>
</tr>
<tr>
<td>local host</td>
<td>320</td>
</tr>
<tr>
<td>loop</td>
<td>52, 53</td>
</tr>
<tr>
<td>lossless</td>
<td>30</td>
</tr>
<tr>
<td>lossy</td>
<td>30</td>
</tr>
<tr>
<td>malware</td>
<td>65</td>
</tr>
<tr>
<td>man-in-the-middle</td>
<td>196</td>
</tr>
<tr>
<td>mark-up language</td>
<td>113</td>
</tr>
<tr>
<td>matrices</td>
<td>38</td>
</tr>
<tr>
<td>matrix</td>
<td>40</td>
</tr>
<tr>
<td>matrix transpose</td>
<td>40</td>
</tr>
<tr>
<td>matrix-vector multiplication</td>
<td>40</td>
</tr>
<tr>
<td>Maximum Transmission Unit (MTU)</td>
<td>326</td>
</tr>
<tr>
<td>mean access time</td>
<td>266</td>
</tr>
<tr>
<td>member</td>
<td>17</td>
</tr>
<tr>
<td>memory</td>
<td>68</td>
</tr>
<tr>
<td>memory hierarchy</td>
<td>259</td>
</tr>
<tr>
<td>memory wall</td>
<td>259</td>
</tr>
<tr>
<td>meta tags</td>
<td>105</td>
</tr>
<tr>
<td>meta-data</td>
<td>85</td>
</tr>
<tr>
<td>microdot</td>
<td>214</td>
</tr>
<tr>
<td>Miller’s Law</td>
<td>260</td>
</tr>
<tr>
<td>miss ratio</td>
<td>266</td>
</tr>
<tr>
<td>mode of operation</td>
<td>178</td>
</tr>
<tr>
<td>modular arithmetic</td>
<td>41, 131</td>
</tr>
<tr>
<td>modulus</td>
<td>132, 180</td>
</tr>
<tr>
<td>Morris Worm</td>
<td>65, 234</td>
</tr>
<tr>
<td>most-significant</td>
<td>210</td>
</tr>
<tr>
<td>multi-tasking</td>
<td>288</td>
</tr>
<tr>
<td>multiplicative identity</td>
<td>181</td>
</tr>
<tr>
<td>multiplicative inverse</td>
<td>181</td>
</tr>
<tr>
<td>native</td>
<td>85</td>
</tr>
<tr>
<td>Network Interface Controllers (NICs)</td>
<td>320</td>
</tr>
<tr>
<td>network interfaces</td>
<td>320</td>
</tr>
<tr>
<td>network security</td>
<td>319</td>
</tr>
<tr>
<td>network stack</td>
<td>320</td>
</tr>
<tr>
<td>node</td>
<td>320</td>
</tr>
<tr>
<td>nodes</td>
<td>109</td>
</tr>
<tr>
<td>number theory</td>
<td>178</td>
</tr>
<tr>
<td>nybble</td>
<td>29</td>
</tr>
<tr>
<td>odd parity code</td>
<td>36</td>
</tr>
<tr>
<td>offline</td>
<td>105</td>
</tr>
<tr>
<td>one-way function</td>
<td>195</td>
</tr>
<tr>
<td>online</td>
<td>105</td>
</tr>
<tr>
<td>opcode</td>
<td>68, 73</td>
</tr>
<tr>
<td>Open Directory Project (ODP)</td>
<td>104</td>
</tr>
<tr>
<td>operand</td>
<td>73</td>
</tr>
<tr>
<td>operating system</td>
<td>67</td>
</tr>
<tr>
<td>operating system</td>
<td>67</td>
</tr>
<tr>
<td>option</td>
<td>291</td>
</tr>
<tr>
<td>OSI model</td>
<td>320</td>
</tr>
<tr>
<td>packets</td>
<td>320</td>
</tr>
<tr>
<td>PageRank</td>
<td>105</td>
</tr>
<tr>
<td>Pantone</td>
<td>207</td>
</tr>
<tr>
<td>Pantone Matching System (PMS)</td>
<td>207</td>
</tr>
<tr>
<td>parity check matrix</td>
<td>42</td>
</tr>
<tr>
<td>parties</td>
<td>177</td>
</tr>
<tr>
<td>password</td>
<td>217</td>
</tr>
<tr>
<td>path</td>
<td>109</td>
</tr>
<tr>
<td>payload</td>
<td>65</td>
</tr>
<tr>
<td>period</td>
<td>170</td>
</tr>
<tr>
<td>permanent</td>
<td>34</td>
</tr>
<tr>
<td>permutation</td>
<td>135</td>
</tr>
<tr>
<td>Personal Identification Number (PIN)</td>
<td>224</td>
</tr>
<tr>
<td>picture element</td>
<td>204</td>
</tr>
<tr>
<td>ping flood</td>
<td>325</td>
</tr>
<tr>
<td>Ping-of-Death (PoD)</td>
<td>325</td>
</tr>
<tr>
<td>pipe</td>
<td>294</td>
</tr>
<tr>
<td>pit</td>
<td>25</td>
</tr>
<tr>
<td>pixel</td>
<td>204</td>
</tr>
<tr>
<td>plaintext</td>
<td>127</td>
</tr>
<tr>
<td>ports</td>
<td>321</td>
</tr>
<tr>
<td>PostScript</td>
<td>209</td>
</tr>
<tr>
<td>pre-computation</td>
<td>274</td>
</tr>
<tr>
<td>pre-computing</td>
<td>105</td>
</tr>
<tr>
<td>precision</td>
<td>209</td>
</tr>
<tr>
<td>prime</td>
<td>48</td>
</tr>
<tr>
<td>prime divisors</td>
<td>182</td>
</tr>
<tr>
<td>prime factorisation</td>
<td>183</td>
</tr>
<tr>
<td>prime number</td>
<td>182</td>
</tr>
<tr>
<td>principle of locality</td>
<td>260</td>
</tr>
<tr>
<td>private-key</td>
<td>193</td>
</tr>
<tr>
<td>privileged</td>
<td>234</td>
</tr>
<tr>
<td>probability</td>
<td>161</td>
</tr>
<tr>
<td>probability distribution</td>
<td>161</td>
</tr>
<tr>
<td>problem size</td>
<td>57</td>
</tr>
<tr>
<td>process</td>
<td>288</td>
</tr>
<tr>
<td>processed</td>
<td>47</td>
</tr>
<tr>
<td>processor</td>
<td>67, 288</td>
</tr>
<tr>
<td>Program Counter (PC)</td>
<td>73</td>
</tr>
<tr>
<td>programs</td>
<td>67, 288</td>
</tr>
<tr>
<td>prompt</td>
<td>290</td>
</tr>
<tr>
<td>protocol</td>
<td>177</td>
</tr>
<tr>
<td>pseudo-code</td>
<td>48</td>
</tr>
<tr>
<td>pseudo-random</td>
<td>167</td>
</tr>
<tr>
<td>Pseudo-Random Number Generator (PRNG)</td>
<td>168</td>
</tr>
</tbody>
</table>
public-key, 193
public-key encryption schemes, 178
pure, 49

queue, 112

Random Access Memory (RAM), 288
random surfer model, 113
random walk, 113
randomness, 159
range, 16
rasterise, 209
rasterised image, 204
Reaper, 65
receiver, 127, 177
recursion, 236
recursive, 238
recursive case, 238
redirection, 292
redundant, 30
Remote Procedure Call (RPC), 327
resolution, 204
resources, 288
return, 52, 54
Return Address (RA), 242
Return Value (RV), 242
RGB, 204
RGBA colour space, 206
root, 110
roots, 116
rounds, 269
routed, 320
row vector, 39
RSA problem, 195
run, 31

S-box, 269
samples, 204
scanner, 66
script, 169
search engine, 105
Search Engine Optimisation (SEO), 121
search queries, 103
search query, 104
Secure Sockets Layer (SSL), 178, 333
seed, 167
self-replicating, 65
sender, 127, 177
sequence, 17
services, 321
set, 17
shell, 288
shift cipher, 129
Short Message Service (SMS), 23
side effects, 49
side-channel, 219
signature, 66
simple, 109
source web-pages, 114
specialise, 267

square-and-multiply, 56
stack, 112, 239
stack frame, 242
Stack Pointer (SP), 239
state, 49
steganography, 203
step, 47
step counting, 57
stored program architecture, 69
straight-line, 76, 268
streams, 291
strings, 85
Stuxnet, 65
sub-net, 320
sub-routine, 235
sub-words, 261
subject, 31
substitution cipher, 135
Substitution-Permutation network, 269
summation, 26
switch, 291
symmetric-key, 178
syndrome, 42
system calls, 288
tag, 261
tape, 68
terminate, 290
terminates, 48
termination criteria, 116
terminator, 90
The GNU Image Manipulation Program (GIMP), 31
Time-To-Live (TTL), 324
Top of Stack (ToS), 239
transformation, 207
transient, 34
Transmission Control Protocol (TCP), 320
Transport Layer Security (TLS), 333
tree, 110
trigrams, 137
Turing Machines (TMs), 67
turtle, 207
unbounded, 91
uniform, 161
unroll, 267
user interface, 288

vector addition, 39
vector image, 207
vector multiplication, 39
vector transpose, 39
vertices, 109
virus, 65

war dialing, 217
watermark, 203
web, 103
web-browser, 104
web-crawling, 105
web-directory, 104
web-graph, 110
web-index, 105
web-indexing, 105
web-pages, 104
web-search, 103, 104
web-search engine, 105
web-server, 104
weight, 109
white hat, 255
working set, 260
World Wide Web (WWW), 103
worm, 65
WWW Virtual Library, 104
I have a question/comment/complaint for you. Any (positive or negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way. To get in contact, email

page@cs.bris.ac.uk

or

nigel@cs.bris.ac.uk

We are not perfect, so mistakes are of course possible (although hopefully rare). Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Can I use this material for something? We are distributing these PDFs under the Creative Commons Attribution License

http://creativecommons.org/licenses/by/2.0/uk/

This is a fancy way to say you can basically do whatever you want with them (i.e., use, copy and distribute it however you see fit) as long as correct attribution of the original source is maintained.

Is there a printed version of this material I can buy? Yes: Springer have published selected Chapters in

http://link.springer.com/book/10.1007/978-3-319-04042-4

while also allowing us to keep electronic versions online. Any royalties from this published version are donated to the Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it is possible to study programming as a topic in itself, we are more interested in what can be achieved using programs: we treat programming itself as another tool.