In this lecture we look at string matching algorithms.

In simple terms, we want to find all the occurrences of some string $P$ in a larger string $T$. 
Introduction

- This seems like a very simple problem, and it is:
  - The real problem is one of efficiency in time and space.
  - Doing many matching operations demands a better than naive approach.

- High performance string matching is vital in many applications:
  - In web searches or databases:
    - We might search stored text for a keyword supplied by the user.
  - In Unix tools like grep and sed:
    - We need to match regular expressions against input text streams.
  - In DNA matching:
    - We match a small DNA strand against a large corpus
    - Here, as in many situations, inexact matching is also required
Introduction

- We look at four different exact string matching methods:
  - The naive, obvious method.
  - The Knuth-Morris-Pratt (KMP) algorithm.
  - The Boyer-Moore-Horspool (BMH) algorithm.
  - The Finite State Machine (FSM) algorithm.

- To compare each different approach, we count the number of comparisons they do:
  - Performance is mostly determined by how many comparisons are performed.
  - It is therefore a good candidate as our computational step.
  - To make absolutely sure, we should check that the run-time is linearly related to the number of comparisons in each case (it is).
Naive Method – Algorithm

- The **basic idea** is this:
  - Match pattern string against input string character by character.
  - When there is a **mismatch**, shift the whole input string down by one character in relation to the pattern string, and start again at the beginning.

Input: Strings $P$ and $T$

```plaintext
n ← |$T$|;
m ← |$P$|;
for $i$ ← 0 to $n - m$ do
  if $P[1, \ldots, m] = T[i + 1, \ldots, i + m]$ then
    OUTPUT($i + 1$);
  end
end
```
Consider matching some example arrays where we set \( T = \text{ababaabbbababb} \) and \( P = \text{ababb} \)

\[
\begin{array}{c}
\text{ababaabbbababb} \\
\text{ababb} \\
\text{ababaabbbababb} \\
\quad \text{ababb} \\
\text{ababaabbbababb} \\
\quad \text{ababb} \\
\text{ababaabbbababb} \\
\end{array}
\quad
\begin{array}{c}
\text{ababaabbbababb} \\
\text{ababb} \\
\text{ababaabbbababb} \\
\quad \text{ababb} \\
\text{ababaabbbababb} \\
\quad \text{ababb} \\
\text{ababaabbbababb} \\
\end{array}
\]

The underlined characters are where the match fails.

This performs a total of 23 comparisons to find a match.
However, consider a worst case example:

- The input text is $aaa\ldots b$ and of total length $n$.
- That is, $n - 1$ a characters followed by one b.
- The pattern is $aaa\ldots b$ and of total length $m$.
- That is, $m - 1$ a characters followed by one b.

Using the naive method, we match up to the $m$-th character and after each mismatch, restart at the first character:

- The mismatch occurs $n - m$ times.
- The match succeeds at position $n - m + 1$.
- The total number of comparisons is therefore $(n - m + 1) \cdot m \in \Theta(n^2)$ if $n > m$. 
When a mismatch occurs at index \( j \) in the naive method, we have found \( j - 1 \) characters that do match.

We can take advantage of this when deciding where to restart the match:

- Imagine a case where the string \( ababb \) mismatches at the 5th character.
- The matched text consists of \( abab? \) where \( ? \) is unknown.
- We restart the match by comparing the 3rd character, an \( a \), against \( ? \).

In short, since we know the pattern beforehand we can work out where to restart the match.
Knuth-Morris-Pratt – Example

Consider matching the same example arrays where we set $T = \text{ababaabbbababb}$ and $P = \text{ababb}$:

Note that we are shifting the pattern either by one or a distance from a precomputed prefix table.

Now we only perform a total of 17 comparisons.
The basic idea is this:

- First compute the **prefix** table of the pattern which tells us where to restart.
- Then when we run the KMP matcher, use the table when a mismatch occurs.

What is a prefix table? The prefix table tells us how far we can shift the pattern along at each turn without missing any matches. Let \( P = ababb \).

- The \( j \)th element of the prefix table for \( P \) is the length of the longest prefix of \( P[1, \ldots, j] \) that is also a proper suffix of \( P[1, \ldots, j] \)
- \( P[1, 1] = a \). There is no proper suffix of \( a \) so the first element of the prefix table is 0
- \( P[1, 2] = ab \). The only proper suffix of \( ab \) is \( b \) which is not a prefix of \( ab \). Therefore the second element of the prefix table is 0
- \( P[1, 3] = aba \). The proper suffixes are \( ba \) and \( a \). \( a \) is a prefix of \( aba \) so the third element of the prefix table is 1.
- The fourth and fifth elements of the prefix table are therefore 2 and 0.
Knuth-Morris-Pratt – Algorithm

The overall algorithm structure is as follows.

KMP-MATCHER \( (T, P) \)
begin
    \( n \leftarrow |T| \)
    \( m \leftarrow |P| \)
    \( \Pi \leftarrow \text{KMP-PREFIX}(P) \)
    \( i \leftarrow 0 \)
    for \( j = 1 \) upto \( n \) step 1 do
        while \( i > 0 \) and \( P[i + 1] \neq T[j] \) do
            \( i \leftarrow \Pi[i] \) \( \triangleright \) Skip using prefix table
        if \( P[i + 1] = T[j] \) then
            \( i \leftarrow i + 1 \) \( \triangleright \) Next character matches
        if \( i = m \) then
            OUTPUT \( (j - m) \) \( \triangleright \) Pattern at shift \( j - m \)
        \( i \leftarrow \Pi[i] \) \( \triangleright \) Look for next match
    end
Knuth-Morris-Pratt – Example

- Let’s look at the example again. \( T = \text{ababaabbababb} \) and \( P = \text{ababb} \):
  - We calculate the prefix table as \( \Pi = \{0, 0, 1, 2, 0\} \).

- We use the prefix table to shift the pattern as required.
What is the largest number of comparison KMP can take for a pattern of length $m$ and a text of length $n$? Recall that $i$ is the current index in the pattern and $j$ is the current index in the text.

- Consider the variable $2j - i$.
- At the beginning of the main `for` loop $2j - i = 2$
- At termination $2j - i \leq 2n$
- At each iteration of the main `for` loop $2j - i$ strictly increases
- At each iteration of the `while` loop $2j - i$ strictly increases
- Therefore the maximum number of comparisons for KMP is no more than $2n$
- KMP takes $O(n)$ time assuming that the prefix table is available
Knuth-Morris-Pratt – Prefix Table

We can compute the prefix table by comparing the pattern against itself.

- For each $j \leq m$ we compute the length of the longest prefix of $P[1, \ldots, j]$ that is also a proper suffix of $P[1, \ldots, j]$.
- This can be done cleverly in $O(m)$ time

```
KMP-PREFIX(P)
begin
    m ← |P|
    Π[1] ← 0
    i ← 0
    for $j = 2$ upto $m$ step 1 do
        while $i > 0$ and $P[i + 1] \neq P[j]$ do
            i ← Π[i]
        if $P[i + 1] = P[j]$ then
            i ← i + 1
            Π[j] ← i
        end
    return Π
end
```
The KMP algorithm never needs to back-track in the input text:
  - This is an advantage if the text is streamed rather than in an array since we don’t have to maintain a buffer for the stream.

The worst case performance is $O(n)$ comparisons.

However, it doesn’t improve much on the average case:
  - Best performance when alphabet is small since this means higher chance of repeated substrings in the input and pattern.
Boyer-Moore-Horspool – Algorithm

- The main goals of BMH are simplicity and improving on the weakness of KMP over large alphabets
  - Use the alphabet that makes up the pattern and input text to skip large distances.
  - Make comparisons starting on the right of the pattern rather than the left, this finds the rightmost mismatch.
- The trick is to consider the alphabet over which the text is defined:
  - The alphabet is the set of characters $C_0, C_1, \ldots, C_{k-1}$.
- We build a table $\Gamma$ which tells us the rightmost occurrence of each letter in the pattern.
  - For each $C_i$ in the pattern string, set $\Gamma[i]$ to be the position of the rightmost occurrence of $C_i$ in the pattern.
Say we want to search for the pattern `lean` in the following string:

```
carpetsneddleaning
  lean
```

The characters `n` and `p` mismatch, also `p` doesn’t appear anywhere in the pattern so we can move all the way past `p` and restart:

```
carpetsneddleaning
  leanleanlean
```

The characters `n` and `e` mismatch, but now `e` occurs in the pattern so line them up:

```
carpetsneddleaning
  lean
```

Only 8 comparisons needed in this example. KMP would need 18.
In the worst case no better then naive matching:
  - The Horspool heuristic is very fast in practice
  - The method works best when you have little repetition within the input text.
    - The worst case is $O(nm)$ comparisons. Consider $p = ba^{m-1}$ and $t = a^n$.
    - The best case is now $O(n/m)$ comparisons.
    - Fast in practice.
The previous methods are quite efficient but also quite simplistic in what they can do:
- What happens if we want to consider regular expressions?

This is exactly the problem faced by parsers in a compiler:
- One defines the syntactic tokens as regular expressions:
  
  ```
  value := [0-9]+  
  ident := [a-z][a-z0-9]*
  ```
- The goal is to match these patterns against the input text.

Using the KMP or Boyer-Moore methods is problematic:
- Pre-computing the tables can’t be done since the pattern is not finite.
We can solve this problem by using a fourth type of matching method.

A Finite State Machine (FSM) is formally defined by the following parts. Note that an FSM is in fact a Turing Machine without a tape:

- $Q$, a finite set of states.
- $q_0 \in Q$, a start state.
- $A \subseteq Q$, a set of accepting states.
- $\Sigma$, an input alphabet.
- $\delta$, a function from $Q \times \Sigma$ into $Q$ which we call the transition function.

More simply, it is just a graph where moving between nodes means consuming input characters.
Finite State Machines (3)

- Here are a few examples of basic regular expressions:
  - $a^+$ – one or more repetitions of $a$.
  - $a^*$ – zero or more repetitions of $a$.
  - $.$ – any character.

- The hard bit is constructing the automaton.

- However, once this is done implementing the matching algorithm is quite easy.

```plaintext
FSM-MATCHER(T, P)
begin
   n ← |T|
   δ ← FSM-BUILD(P)
   s ← 0
   for i = 1 upto n step 1 do
      s ← δ(s, T[i])
      if s is an accepting state then
         OUTPUT(i)
   end
end
```
Consider some example FSM constructions:

- Take the input string a character at a time and move along edges which match each character.
- The empty edge denotes the start state, double circled nodes are accepting states which signal a match.

Question: How do we move through the states for the string aababaaba?
Comparison

- To get an idea of which algorithm is best, we can compare their complexities:

<table>
<thead>
<tr>
<th></th>
<th>Pre-computation</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>$\Theta(m)$</td>
<td>$\Theta(nm)$</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>$O(m +</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Boyer-Moore-Horspool</td>
<td>$O(m</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Finite State Machine</td>
<td>$O(m</td>
<td>\Sigma</td>
</tr>
</tbody>
</table>

- Some points to note from this analysis are:
  - For one-off matches on short strings, the naive method isn’t so bad.
  - The methods that require pre-computation may also require extra memory.
Conclusions

- String matching sounded like a trivial problem:
  - Hopefully you can see there is a little more to it than the naive method.
  - You will see in the coursework how to perform string matching without performing any comparisons at all.

- As a general rule, selecting the right algorithm is done as follows:
  - If you need to consider complex matching like regular expressions, use the FSM method.
  - Otherwise, the choice depends on the alphabet size:
    - For large alphabets, like natural language, use the Boyer-Moore-Horspool method.
    - For small alphabets, use the Knuth-Morris-Pratt method.

- However, there are some even more complicated methods than these.
- See [http://tinyurl.com/eolt7](http://tinyurl.com/eolt7) for a long list with example code.
Further Reading

- **Introduction to Algorithms**
  - Chapter 32 – String Matching.