Number Theoretic Algorithms

- **Number theory** is basically a branch of pure mathematics.
  - The topic is typically described in the context of cryptography.
  - Mainly, it is simply concerned with the properties of integer numbers.
- Number theory is interesting in this context for a couple of important reasons:
  - There are some ancient and classic algorithms that you should know about.
  - We are dealing with large integers, so algorithm performance is crucial.
  - It has a tight connection to complexity and the theory of computation.

GCD and XGCD

- The greatest common divisor or GCD of two numbers is an easy concept to describe:
  - Given \( a \) and \( b \) we want to find \( c \), the largest number that exactly divides both \( a \) and \( b \).
  - If \( c = 1 \), then \( a \) and \( b \) are termed co-prime or relatively prime.
- Consider some simple examples:
  - If we set \( a = 6 \) and \( b = 9 \) then \( c = \text{gcd}(a, b) = 3 \) since 3 is the largest number to exactly divide 6 and 9.
  - If we set \( a = 5 \) and \( b = 7 \) then \( c = \text{gcd}(a, b) = 1 \) and \( a \) and \( b \) are co-prime.
- A basic real-world use of GCD is to reduce fractions into the simple form:
  - The GCD of numerator and denominator cancel:
    \[
    \frac{42}{56} = \frac{3 \cdot 14}{4 \cdot 14} = \frac{3}{4}.
    \]

GCD and XGCD

- The most naive way to solve this problem would be to write a brute force check:
  \[
  \text{GCD}(a, b) \begin{align*}
  \text{begin} & \quad \text{for } \min(a, b) \text{ downto } 1 \text{ do} \\
  & \quad \text{if } a \equiv 0 \pmod{i} \text{ and } b \equiv 0 \pmod{i} \text{ then} \\
  & \quad \text{return } i \\
  \text{end}
  \end{align*}
  \]
  - We loop through all possible divisors from the largest to the smallest and return the first that divides both \( a \) and \( b \).
  - Clearly, this isn’t very clever since we need to do a lot of loop iterations for large values of \( a \) or \( b \).
- A better way is Euclid’s algorithm which can be expressed recursively or (more inefficiently) iteratively:
  \[
  \text{GCD}(a, b) \begin{align*}
  \text{begin} & \quad \text{while } a \neq b \text{ do} \\
  & \quad \text{if } a > b \text{ then} \\
  & \quad \quad a \leftarrow a - b \\
  & \quad \text{else} \\
  & \quad \quad b \leftarrow b - a \\
  & \quad \text{end} \\
  & \quad \text{return } a \\
  \text{end}
  \end{align*}
  \]
- They key fact that shows correctness is that
  \[
  \text{gcd}(a, b) = \text{gcd}(b, a \mod b)
  \]
- We now do fewer loop iterations than the naive implementation but need to do some more costly arithmetic:
GCD and XGCD

▶ Say for example the remainder of the division of \(a\) by \(b\) is \(t\).
   ▶ In this case we have \(a = q \cdot b + t\) where \(q\) is the quotient of the division of \(a\) by \(b\).
▶ Any divisor of both \(a\) and \(b\) also divides \(t\).
   ▶ Since \(t\) can be written as \(t = a - q \cdot b\).
▶ In the same way, any common divisor of \(b\) and \(t\) will also divide \(a\).
   ▶ So the GCD of \(a\) and \(b\) is the same as the GCD of \(b\) and \(t\).
▶ Therefore it is enough if we continue the process with the numbers \(b\) and \(t\).

GCD - Running Time

Before we start we must convince ourselves that Euclid's algorithm actually terminates.
▶ Observe that the second term strictly decreases at every recursive call
   ▶ The second term is also integer valued and bounded below by 0
   ▶ Therefore Euclid's algorithm must terminate

The overall running time of \textsc{Euclid} is proportional to the number of recursive calls it makes. The running time turns out to have its worst case when \(a\) and \(b\) are consecutive Fibonacci numbers.
▶ Recall that \(F_0 = 0, F_1 = 1\) and \(F_i = F_{i-1} + F_{i-2}\) for \(i \geq 2\)

GCD - Running Time

Lemma

\[\text{If } a > b \geq 0, \text{ then } a \mod b \leq a/2\]

Proof.

Either \(b \leq a/2\), in which case \(a \mod b \leq b \leq a/2\). Or \(b > a/2\), in which case \(a \mod b = a - b \leq a/2\).

▶ The total number of recursive calls to \textsc{Euclid} must be bounded above by \(2 \log a\) since in two recursive invocations of the algorithm, the larger number, \(a\), is replaced by at most \(a/2\).
▶ Since each call requires a division (to reduce \(a \mod b\)), each division requires \(O(\log^2 a)\) steps, for a grand total of \(O(\log^3 a)\) operations for the whole of \textsc{Euclid}.
▶ If we assume that number of bits \(b\) needed to represent \(a\) is \(\Theta(\log a)\), then the total running time is \(O(b^3)\)
▶ Problem 33-2 in CLRS asks you to show a tighter \(O(b^2)\) bound on the number of bit operations for \textsc{Euclid}.

GCD and XGCD

▶ For example, the GCD of 1071 and 1029 is computed by the algorithm to be 21 as follows:

\[
\begin{array}{c|c|c}
  a & b & a \mod b \\
  \hline
  1071 & 1029 & 42 \\
  1029 & 42 & 21 \\
  42 & 21 & 0 \\
  21 & 0 & \\
\end{array}
\]

▶ Note that this is far fewer loop iterations than the naive algorithm would perform.
GCD and XGCD

- In practise, this is still bad news for large numbers because modular reduction of large a and b values is expensive.
- To improve on practical performance (without changing the time complexity), we use the binary GCD algorithm:
  - The basic idea is to utilise the fact that we are working in binary, or base-2.
  - Recall that division by two is just a right shift by one bit.
  - Checking if a value is odd or even is just a test of the least significant bit.
- You can think of this as an optimisation of the GCD algorithm rather than a new algorithm:
  - We are reducing the constants in the complexity by matching the implementation against the capabilities of our processor.

GCD and XGCD

- So instead of a general modular reduction, we apply a few basic rules to reduce a and b:
  - If a is even and b is even, then \( \gcd(a, b) = 2 \cdot \gcd(a/2, b/2) \).
  - Since we know that 2 is a common divisor.
  - If a is even and b is odd, then \( \gcd(a, b) = \gcd(a/2, b) \).
  - Since we know that 2 is not a common divisor.
  - If a is odd and b is even, then \( \gcd(a, b) = \gcd(a, b/2) \).
  - Since we know that 2 is not a common divisor.
  - If a is odd and b is odd, then \( \gcd(a, b) = \gcd(|a - b|/2, b) \) which is also the same as \( \gcd(a, |a - b|/2) \).
  - Which we get from the original Euclidean algorithm.

GCD and XGCD

- The result is a longer but far more efficient algorithm:

\[
\begin{align*}
\text{GCD}(a, b) & \quad \text{begin} \\
& \quad k \leftarrow 0 \\
& \quad \text{while } a \equiv 0 \pmod{2} \text{ and } b \equiv 0 \pmod{2} \text{ do} \\
& \quad \quad a \leftarrow a/2 \\
& \quad \quad b \leftarrow b/2 \\
& \quad \quad k \leftarrow k + 1 \\
& \quad \text{repeat} \\
& \quad \quad \text{if } a \equiv 0 \pmod{2} \text{ then} \\
& \quad \quad \quad a \leftarrow a/2 \\
& \quad \quad \text{else if } b \equiv 0 \pmod{2} \text{ then} \\
& \quad \quad \quad b \leftarrow b/2 \\
& \quad \quad \text{else if } a \geq b \text{ then} \\
& \quad \quad \quad a \leftarrow (a - b)/2 \\
& \quad \quad \text{else} \\
& \quad \quad \quad b \leftarrow (b - a)/2 \\
& \quad \quad \text{until } a \leq 0 \\
& \quad \text{return } b/2^k \\
\end{align*}
\]

- Further minor improvements to the Euclidean algorithm can give us even more useful information.
  - As well as \( g = \gcd(a, b) \), we can compute \( x \) and \( y \) so \( a \cdot x + b \cdot y = g \).
  - This is usually called the extended Euclidean algorithm or XGCD.
- The XGCD algorithm has some more interesting uses than the more basic GCD.
  - Consider working with integers modulo some prime number \( p \).
  - Given \( a \), we often want to find a number \( b \) so that \( a \cdot b \equiv 1 \pmod{p} \).
  - If \( (g, x, y) = \text{xgcd}(p, a) \) then \( g = 1 \) given that \( p \) is prime, and since \( x \cdot a + y \cdot M = 1 \) and \( y \cdot M = 0 \pmod{M} \), then \( b = x \).

- First we make sure both \( a \) and \( b \) are odd and then iteratively apply our set of rules.
GCD and XGCD

- The algorithm takes the same structure as the original Euclidean algorithm:
  - The basic idea is to keep track of the factors of \(a\) and \(b\) we are eliminating.

\[
\text{XGCD}(a,b) \begin{align*}
\text{if } b = 0 & \text{ then return } (a, 1, 0) \\
\text{else } & (g', x', y') \leftarrow \text{XGCD}(b, a \mod b) \\
& \text{return } (g', y', x' - \lfloor \frac{a}{b} \rfloor \cdot y')
\end{align*}
\]

- Clearly we can do the same sorts of optimisation to produce a binary version ...

If we run the algorithm on the same example as before, we get:

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(a \mod b)</th>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1071</td>
<td>1029</td>
<td>42</td>
<td>-24</td>
<td>25</td>
</tr>
<tr>
<td>1029</td>
<td>42</td>
<td>21</td>
<td>1</td>
<td>-24</td>
</tr>
<tr>
<td>42</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table is a little confusing but when the recursion unwinds we find that \(g = 21\), \(x = -24\) and \(y = 25\).

- We can verify this using the formula \(a \cdot x + b \cdot y = g\) and substituting our results to get \(1071 \cdot (-24) + 1029 \cdot 25 = 21\).

Primality Testing (1)

- A prime number \(n\) can only be exactly divided by 1 and \(n\).
  - Otherwise the number is termed composite and is the product of some factors.

- In cryptography, we need to generate and use very large prime numbers.
  - Your system parameters used for RSA encryption within SSL are such numbers.

- There are two ways we might try and go about finding large primes:
  - Somehow construct the number so that we know that it will definitely be prime.
  - Generate and test random numbers until we come across one that is prime.

Primality Testing (2)

- The generate and test method turns out to be much more viable.

- Given we can generate random numbers, a naive way to test for primality is:

\[
\text{TRIAL-DIVISION}(n) \begin{align*}
\text{begin} \\
\text{for } i = 2 \text{ upto } \lfloor \sqrt{n} \rfloor \text{ do} \\
\text{if } n \equiv 0 \text{ (mod } i \text{) then return } n \text{ is composite} \\
\text{return } n \text{ is prime}
\end{align*}
\]  

- This is called trial division and is slow for large values of \(n\).
A more advanced approach is due to Fermat and uses the following theorem he discovered in about 1636. For some prime \( n \) and \( 1 < b \leq n \):

\[ b^{n-1} \equiv 1 \pmod{n} \]

and is called Fermat’s little theorem to differentiate it from the more famous Fermat’s last theorem.

The value \( b \) is called the base of the Fermat test.

If we want to test the primality of some number \( n \) then we perform lots of iterations of the following steps:

1. Select a random value \( 1 < b \leq n \) and test if the theorem holds for \( b \) and \( n \).
2. If the theorem doesn’t hold, \( n \) is definitely composite.
3. If the theorem does hold, \( n \) might be prime.

By doing this, we have constructed a Monte Carlo algorithm.

For each \( b \) we test, we decrease the probability that \( n \) is composite.

Values of \( n \) that passes a given number of tests are called pseudo-prime or probably prime.

Actually, they are base-\( b \) pseudo-primes since their passing the test relates to the choice of \( b \).

So the overall algorithm is very simple to write down:

```plaintext
FERMAT-TEST(n, k)
begin
for i = 1 upto k do
b ← random
if \( b^{n-1} \not\equiv 1 \pmod{n} \) then
return \( n \) is composite
return \( n \) is probably prime
end
end
```

Note that we need efficient exponentiation, a topic covered elsewhere...

Actually, in reality we can use a combined algorithm:

1. For large values of \( n \), use the Fermat method or something even better called Miller-Rabin.
2. For small values of \( n \), use trial division or even a look-up table.

You can think of this as a similar sort of approach as quick-sort:

1. For large lists use the text-book quick-sort algorithm.
2. When the recursion in quick-sort needs to sort small lists, bail out and use insertion-sort.

Combining algorithms in this way allows us to utilise the best properties of both ...
A natural question is how many iterations of the Fermat test do we need?
- It obviously depends on how sure we need to be that \( n \) is prime.
- However, after as few as 10 to 20 iterations the probability of error starts to get quite small.

However, you should beware: there are some problems with the Fermat method:
- There are some composite values of \( n \) for which all values of \( b \) say \( n \) is pseudoprime.
- These are called Carmichael numbers and are quite rare, \( n = 561 \) is one example.

We have looked at just two number theoretic algorithms:
- You can think of this as an application of the work we did previously.
- From the point of view of software engineering, there are a few key points to note about what we did and how we did it:
  - Reusing algorithmic techniques like randomised algorithms helped us improve performance.
  - Reusing optimisation techniques like \( w \)-ary exponentiation helped us improve performance.
  - Reusing the idea of combined algorithms like quick-sort helped us improve performance.

Further Reading
- **Introduction to Algorithms**
  - Chapter 31 – Number Theoretic Algorithms
- **A Computational Introduction to Number Theory and Algebra**
  V. Shoup.
  - Chapter 4 – Euclid’s Algorithm
  - Chapter 10 – Probabilistic Primality Testing
  - Chapter 22 – Deterministic Primality Testing