Principles of Algorithmic Techniques
CS-E3190

Pekka Orponen

Department of Computer Science
Aalto University

Autumn 2017
Lecture 13: Algorithms with numbers I

- Basics of modular arithmetic (computing with integers modulo $N$)
- Greatest common divisors, Euclid’s algorithm, extended Euclid’s algorithm
- Multiplicative inverses and modular division
- Computations on primes, primality testing and generating random primes
13.1 Modular Arithmetic

- Computing with integers modulo $N$, or “computing with remainders”.
- Let $x$ and $N \geq 2$ be integers.
- The remainder when $x$ is divided by $N$ is denoted $x \mod N$.
- This is the least nonnegative integer $r$ such that $x = qN + r$, for some $q$.
- Integers $x$ and $y$ are **congruent modulo $N$** if $x \mod N = y \mod N$. This is denoted also:
  
  \[ x \equiv y \pmod{N}. \]

- Note that $x$ and $y$ are congruent modulo $N$ if and only if $N$ divides $x - y$, denoted $N | x - y$. 
Congruence modulo $N$ is an equivalence relation, and partitions integers in *equivalence classes* of the form

$$\{ r + qN \mid q \in \mathbb{Z} \}, \quad 0 \leq r \leq N - 1.$$  

The number of congruence classes mod $N$ is $N$.

The congruence class to which integer $x$ belongs is sometimes denoted as

$$[x] = \{ y \mid y = x + qN, q \in \mathbb{Z} \}.$$  

A congruence class is often represented simply by its smallest nonnegative member, i.e. the common remainder $r$ of all the numbers $x$ that belong to the class.
Modular arithmetic operations

**Substitution rule** If \( x \equiv x' \pmod{N} \) and \( y \equiv y' \pmod{N} \), then
\[
x + y \equiv x' + y' \pmod{N} \quad \text{and} \quad x \times y \equiv x' \times y' \pmod{N}
\]

For example, suppose you watch an entire season of your favourite television show in one sitting, starting at midnight. There are 25 episodes, each lasting 3 hours. At what time of day are you done? *Answer:* The hour of completion is \((25 \times 3) \mod 24\), which (since \(25 \equiv 1 \pmod{24}\)) is \(1 \times 3 = 3 \mod 24\), or three o’clock in the morning.

\[
x + (y + z) \equiv (x + y) + z \pmod{N} \quad \text{Associativity}
\]
\[
x \times y \equiv y \times x \pmod{N} \quad \text{Commutativity}
\]
\[
x \times (y + z) \equiv x \times y + x \times z \pmod{N} \quad \text{Distributivity}
\]
Modular reductions

When performing a sequence of operations in modular arithmetic, with respect to the same modulus $N$, it is legal to reduce intermediate results to their remainders modulo $N$ at any stage.

Such simplifications can be a dramatic help in big calculations. Witness, for instance:

$$2^{345} \equiv (2^5)^{69} \equiv 32^{69} \equiv 1^{69} \equiv 1 \pmod{31}$$

*Note.* It is not legal to reduce the exponent modulo 31. (Below we will see that it would be legal to reduce the exponent modulo 30.)
Modular addition

To add two integers $x$ and $y$ modulo $N$, we start with regular addition. Since $x$ and $y$ are each in the range 0 to $N - 1$, their sum is between 0 and $2(N - 1)$. If the sum exceeds $N - 1$, we merely need to subtract off $N$ to bring it back into the required range.

The overall computation therefore consists of an addition, and possibly a subtraction, of integers that never exceed $2N$. Its running time is linear in the sizes of these numbers, in other words $O(n)$, where $n = \lceil \log N \rceil$ is the size of $N$. 
Modular multiplication

To multiply two integers $x$ and $y$ modulo $N$, we start with regular multiplication and then reduce the answer modulo $N$. The product can be as large as $(N - 1)^2$, but this is still at most $2n$ bits long since $\log((N - 1)^2) = 2\log(N - 1) \leq 2n$. To reduce the answer modulo $N$, we compute the remainder upon dividing it by $N$, using our quadratic-time division algorithm. Multiplication thus remains a quadratic operation.

Division is not quite so easy. In ordinary arithmetic there is just one tricky case: division by zero. It turns out that in modular arithmetic there are potentially other such cases as well, which we will characterise toward the end of this section. Whenever division is legal, however, it can be managed in cubic time, $O(n^3)$. 
Two’s complement representation

A common format of storing signed integers in the range $-2^{n-1}$ to $2^{n-1} - 1$ using $n$ bits.

- Positive integers, in the range $0$ to $2^{n-1} - 1$, are stored in regular binary and have a leading bit of 0.
- Negative integers $-x$, with $x$ in the range $1$ to $2^{n-1}$, are stored by first constructing $x$ in binary, then flipping all the bits, and finally adding 1. The leading bit in this case is 1.
Modular exponentiation

How to compute $x^e \mod N$ quickly for values of $x$, $e$, and $N$ that are several hundred bits long?

The result is some number modulo $N$ and is therefore itself a few hundred bits long.

However, the raw value of $x^e$ could be much, much longer than this. Even when $x$ and $e$ are just 20-bit numbers, $x^e$ is at least $(2^{19})^{2^{19}} = 2^{19 \times 524288}$, about 10 million bits long! Imagine what happens if $e$ is a 500-bit number!

Solution:

- perform all intermediate computations modulo $N$, and
- do repeated squaring modulo $N$.

$x \mod N \rightarrow (x \mod N)^2 \mod N = x^2 \mod N \rightarrow \cdots \rightarrow x^{2^{\lfloor \log e \rfloor}} \mod N$. 
The square and multiply algorithm (1/2)

To determine $x^e \mod N$, we multiply together an appropriate subset of these squared powers, those corresponding to 1’s in the binary representation of $e$. For instance,

$$25 = 2^4 + 2^3 + 1 \rightarrow x^{25} = x^{16} \times x^8 \times x^1.$$ 

In general:

$$x^e = \begin{cases} 
(x^\lceil e/2 \rceil)^2 & \text{if } e \text{ is even} \\
 x \times (x^\lceil e/2 \rceil)^2 & \text{if } e \text{ is odd.}
\end{cases}$$

This closely parallels recursive multiplication à la Française.

Let $n$ be the size in bits of $x$, $e$, and $N$ (actually we can always reduce to a situation where $N$ is the largest of the three). As with multiplication, the algorithm will halt after at most $n$ recursive calls, and during each call it multiplies $n$-bit numbers (doing computation modulo $N$ saves us here), for a total running time of $O(n^3)$. 
### The square and multiply algorithm (2/2)

#### Algorithm 1: Modular exponentiation

```python
1 function modexp (x, e, N);
    Input: Two n-bit integers x and N, an integer exponent e
    Output: \(x^e \mod N\)
2 if e = 0 then
3     return 1;
4 else
5     set z = modexp(x, \([e/2]\), N);
6     if e is even then
7         return \(z^2 \mod N\);
8     else
9         return \(x \times z^2 \mod N\);
10 end
11 end
```
13.2 GCDs and Euclid’s Algorithm

Given two integers $a$ and $b$, Euclid’s algorithm finds the largest integer that divides both $a$ and $b$, known as their greatest common divisor $\gcd(a, b)$.

**Euclid’s rule** If $x$ and $y$ are positive integers, then
$\gcd(x, y) = \gcd(x \mod y, y) = \gcd(y, x \mod y)$.

---

**Algorithm 2**: Euclid’s algorithm for finding the greatest common divisor of two nonnegative integers

1. function Euclid $(a, b)$;
2. Input: Two nonnegative integers $a$ and $b$
3. Output: $\gcd(a, b)$
4. if $b = 0$ then
5. return $a$;
6. else
7. return Euclid($b, a \mod b$);
8. end
Complexity of Euclid’s algorithm

**Lemma** If $a \geq b$ then $a \mod b < \frac{a}{2}$.

*Proof.* Two cases: $b \leq \frac{a}{2}$ or $b > \frac{a}{2}$. The first case is clear. In the second case, $a \mod b = a - b < \frac{a}{2}$.

This means that after any two consecutive rounds, both arguments, $a$ and $b$, are at the very least halved in value, and the length of each decreases by at least one bit. If they are initially $n$-bit integers, then the base case ($b = 0$) will be reached within $2n$ recursive calls. And since each call involves a quadratic-time division, the total time is $O(n^3)$. 
Lemma Let $d, a, b$ be nonnegative integers. If $d$ divides both $a$ and $b$, and $d = xa + yb$ for some integers $x$ and $y$, then $d = \gcd(a, b)$.

Proof. Since $d$ divides $a$ and $b$, it also divides $\gcd(a, b)$. Conversely, since $\gcd(a, b)$ divides $a$ and $b$, and $d = xa + yb$, it follows that $\gcd(a, b)$ divides $d$. Thus, $d = \gcd(a, b)$.

Also the converse holds: if $d = \gcd(a, b)$ then there exist integers $x$ and $y$ such that $d = xa + yb$. This presentation is not unique. One such pair of “Bézout coefficients” $x$ and $y$ for $d$ can be found using the following extension of Euclid’s algorithm.
Extended Euclid’s algorithm (2/2)

Algorithm 3: A simple extension of Euclid’s algorithm

1 function extended-Euclid (a, b);

Input: Two nonnegative integers $a$ and $b$

Output: Integers $x$, $y$, $d$ such that $d = \gcd(a, b)$ and $xa + yb = d$

2 if $b = 0$ then
3     return $(1, 0, a)$
4 else
5     set $(x', y', d) = \text{extended-Euclid}(b, a \mod b)$;
6     return $(y', x' - \lfloor a/b \rfloor y', d)$
7 end
Correctness of extended Euclid’s algorithm

**Lemma** For any positive integers $a$ and $b$, the extended Euclid’s algorithm returns integers $x$, $y$ and $d$ such that $\gcd(a, b) = d = xa + yb$.

**Proof.** By the basic Euclid’s algorithm $d = \gcd(a, b)$. If $b = 0$, the algorithm returns $(x, y, d) = (1, 0, a)$ which satisfies $xa + yb = d$.

Assume now that $b > 0$ and we have used the algorithm to produce $(x', y', d)$ such that $x'b + y'(a \mod b) = d$. Then writing $a \mod b$ as $a - \lfloor a/b \rfloor b$ gives

$$d = x'b + y'(a - \lfloor a/b \rfloor b) = y'a + (x' - \lfloor a/b \rfloor y')b,$$

thus validating the algorithm’s design.
**Example:** \( \gcd(25, 11) \)

\[
\begin{align*}
25 &= 2 \times 11 + 3 & 3 &= 25 - 2 \times 11 \\
\downarrow & \quad \uparrow & \uparrow & \uparrow \\
11 &= 3 \times 3 + 2 & 2 &= 11 - 3 \times 3 \\
\downarrow & \quad \uparrow & \uparrow & \uparrow \\
3 &= 1 \times 2 + 1 & 1 &= 3 - 1 \times 2 \\
\downarrow & \quad \uparrow & \uparrow & \uparrow \\
2 &= 2 \times 1 + 0 & 0 &= 2 - 2 \times 1 \\
\end{align*}
\]

\[
\therefore 1 = 3 - 1 \times 2 \\
= 3 - 1 \times (11 - 3 \times 3) \\
= -11 + 4 \times 3 \\
= -11 + 4 \times (25 - 2 \times 11) \\
= 4 \times 25 - 9 \times 11
\]
13.3 Multiplicative Inverses and Modular Division

We say that \( x \) is a *multiplicative inverse of \( a \) mod \( N \) if \( a \times x \equiv 1 \pmod{N} \), and denote this condition as \( x \equiv a^{-1} \pmod{N} \).

If \( a \times x \equiv 1 \pmod{N} \), then there exist integers \( x \) and \( y \) such that \( a \times x + N \times y = 1 \). It follows that if \( a \) has a multiplicative inverse mod \( N \), then \( \gcd(a, N) = 1 \). If on the other hand \( \gcd(a, N) = 1 \), then a suitable \( x \) can be found using the extended Euclid’s algorithm.

We say that \( a \) is *relatively prime to \( n \) if \( \gcd(a, N) = 1 \). Thus we have:

**Modular division theorem** For any \( a, a \) has a multiplicative inverse mod \( N \) if and only if it is relatively prime to \( N \). When this inverse exists, it can be found in time \( O(n^3) \), where as usual \( n \) denotes the number of bits of \( N \), by running the extended Euclid algorithm.

This resolves the issue of modular division: when working mod \( N \), we can divide by integers relatively prime to \( N \) and only by these. And to actually carry out the division, we multiply by the inverse.
A positive integer $p$ is *prime* if it has no nontrivial factors, or equivalently if all integers $a$ in the range 1 to $p - 1$ are relatively prime to $p$, i.e. $\gcd(a, p) = 1$. A positive integer $N$ that is not prime is *composite*.

For composite $N$, it is in general difficult to find its nontrivial factors, or equivalently an integer less than $N$ which is not relatively prime to $N$. This is called the *factoring problem*. No efficient algorithms are known for factoring.

On the other hand, it is quite feasible to prove that an integer is prime using *primality testing algorithms*.

Factoring is hard and primality is easy.
Fermat’s little theorem

Theorem If $p$ is prime, then for every $1 \leq a < p$ we have

$$a^{p-1} \equiv 1 \pmod{p}.$$ 

Proof. Denote $S = \{1, 2, \ldots, p - 1\}$. For all $i \neq j \in S$, the numbers $a \times i \mod p$ and $a \times j \mod p$ are different.$^1$ It follows that the number of elements in the set

$$\{a \times 1 \mod p, a \times 2 \mod p, \ldots, a \times (p - 1) \mod p\}$$

is equal to $p - 1$. Hence the set is equal to $S$. Multiplying $\mod p$ all elements in this set gives $a^{p-1} \times (p - 1)!$, and hence we obtain

$$a^{p-1} \times (p - 1)! \equiv (p - 1)! \pmod{p}.$$ 

Since $p$ is prime, $(p - 1)!$ is relatively prime to $p$, so we can divide by $(p - 1)!$ to get the desired result

$$a^{p-1} \equiv 1 \pmod{p}.$$ 

---

$^1$The numbers being equal implies $i - j \equiv a^{-1} \times 0 \equiv 0 \pmod{p}$.
Application: reducing exponents

**Corollary** If $p$ is prime, then for every integer $a$ and exponent $e$ we have

$$a^e \equiv a^{e \mod (p-1)} \pmod{p}.$$

**Corollary** If $p$ and $q$ are distinct primes, then for every integer $a$ and exponent $e$ with $e \not\equiv 0 \pmod{(p-1)(q-1)}$, we have

$$a^e \equiv a^{e \mod (p-1)(q-1)} \pmod{pq}.$$

*Proof.* Exercise.
Algorithm 4: A randomised algorithm for testing primality

1 function primality (N);

Input: Positive integer \( N \)
Output: yes/no

2 Pick a positive integer \( a < N \) uniformly at random;

3 if \( a^{N-1} \not\equiv 1 \pmod{N} \) then

4 \hspace{1em} return no;

5 else

6 \hspace{1em} return yes;

7 end

The answer "no" is always correct by Fermat’s little theorem. The answer "yes" may sometimes be incorrect.
Error probability of the Fermat test

**Lemma** If $a^{N-1} \not\equiv 1 \pmod{N}$ holds for some $2 \leq a \leq N - 2$ relatively prime to $N$, then it holds for at least half the choices of $a$.

**Proof.** Let $a$ be such that $a^{N-1} \not\equiv 1 \pmod{N}$. If there is no $b$ such that $b^{N-1} \equiv 1 \pmod{N}$, then we are done. Suppose now that for some number $b$ it is the case that $b^{N-1} \equiv 1 \pmod{N}$. Then for $a \times b$ it holds that

$$(a \times b)^{N-1} \equiv a^{N-1} b^{N-1} \not\equiv 1 \pmod{N},$$

from which the claim follows. (Note that if $b \not\equiv b' \pmod{N}$, then $a \times b \not\equiv a \times b' \pmod{N}$.)

*Caveat:* There are infinitely many composite integers $N$, so called *Carmichael numbers*, for which:

$$a^{N-1} \equiv 1 \pmod{N} \text{ for all } a \text{ relatively prime to } N.$$

For such integers the Fermat test is not guaranteed to be efficient. This issue is avoided by e.g. the *Solovay-Strassen and Rabin-Miller* primality tests. The smallest Carmichael number is 561.
An improved primality test

**Algorithm 5:** A randomised algorithm for testing primality, with in most cases low error probability

1. function primality2 (N);
2. **Input:** Positive integer $N$
3. **Output:** yes/no
4. Pick positive integers $a_1, a_2, \ldots, a_k < N$ unif. at random;
5. if $a_i^{N-1} \equiv 1 \pmod{N}$ for all $i = 1, 2, \ldots, k$ then
6. return yes;
7. else
8. return no;
9. end

The probability that this “iterated Fermat test” returns “yes” when $N$ is not prime and not a Carmichael number is at most $(1/2)^k = 2^{-k}$. 
Error probability

*Textbook:* The probability that the iterated Fermat test returns “yes” when \( N \) is not prime and not a Carmichael number is at most \( 2^{-k} \).

This is correct, but this is not the error probability we want.

Let \( C \) denote the event:

“integer \( N \) is composite”

and let \( Y \) denote the event:

“the algorithm answers “yes” \( k \) times in succession”.

Clearly the conditional probability \( \Pr[Y \mid C] \) is equal to \( 2^{-k} \). But actually the true error probability is \( \Pr[C \mid Y] \), which can be determined using Bayes’ theorem. (More about this later.)
Application: generating random primes

Prime number theorem Let $\pi(x)$ be the number of primes less than or equal to $x$. Then $\pi(x) \approx x/\ln x$, or more precisely,

$$\lim_{x \to \infty} \frac{\pi(x)}{x/\ln x} = 1.$$ 

Algorithm:

- Pick an $n$-bit random number $N$.
- Run a primality test on $N$.
- If it passes the test, output $N$; else repeat the process.

How fast is the algorithm? A randomly chosen odd number $N$ is prime in approximately 2 out of $\ln N$ cases. Therefore the algorithm is expected to halt within $O(\ln N) = O(n)$ rounds.
More on error probability (1/2)

Let us calculate the probability of error for the iterated Fermat primality test.

Let $C$ denote the event:

“a given $n$-bit integer $N$ is composite”

and let $Y$ denote the event:

“the algorithm answers “yes” $k$ times in succession”.

The error probability is $\Pr[C \mid Y]$, which can be determined using Bayes’ theorem:

$$\Pr[C \mid Y] = \frac{\Pr[Y \mid C] \Pr[C]}{\Pr[Y]}$$
More on error probability (2/2)

By the Prime number theorem,

\[ \Pr[C] \approx 1 - \frac{2}{\ln N}. \]

To estimate \( \Pr[Y] \) we write

\[
\Pr[Y] = \Pr[Y | C] \Pr[C] + \Pr[Y | \overline{C}] \Pr[\overline{C}]
\approx \Pr[Y | C](1 - \frac{2}{\ln N}) + \frac{2}{\ln N}
\]

We obtain the following upper bound

\[
\Pr[C | Y] = \frac{\Pr[Y | C](1 - \frac{2}{\ln N})}{\Pr[Y | C](1 - \frac{2}{\ln N}) + \frac{2}{\ln N}}
\]

\[\approx \frac{2^{-k}(1 - \frac{2}{\ln N})}{2^{-k}(1 - \frac{2}{\ln N}) + \frac{2}{\ln N}} \]

\[= \frac{\ln N - 2}{\ln N - 2 + 2^{k+1}}.\]
Example

Suppose $N$ is a 512-bit number. Then $\ln N \approx 355$ and the bound for error probability is

$$\frac{353}{353 + 2^{k+1}}$$

Typical value of $k$ is between 50 and 100. Note that this bound is always somewhat larger than $2^{-k}$. 
The Miller-Rabin primality test

Algorithm 6: The Miller-Rabin primality test

1 function primality3 (N);

Input: Positive integer N
Output: yes/no

2 Write $N - 1 = 2^t u$, where $u$ is odd;
3 Pick a positive integer $a < N$ at random;
4 $b = a^u \mod N$;
5 if $b \equiv 1 \pmod{N}$ then
6     return yes;
7 else
8     for $i = 0$ to $t - 1$ do
9         if $b \equiv -1 \pmod{N}$ then
10            return yes;
11        else
12            $b = b^2 \mod N$
13        end
14     end
15     return no;
16 end
Error probability

The probability that the Miller-Rabin test returns "yes" when $N$ is not prime can be shown to be at most $1/4$.

By repeating the test for a number of randomly selected bases $a$ the error probability can be made arbitrarily small.