Modular Arithmetics and Cryptography

Lecture 1

Complexity. Modular Arithmetics

Index

Contents of the Course

1. Modular Arithmetics
2. Private Key Cryptography
3. Public Key Cryptography
4. Applications

Selective bibliography

- M. Cozzens, S.J. Miller, *The Mathematics of Encryption: An Elementary Introduction*, American Mathematical Society, 2013.
- S. Crivei, A. Marcus, C. Săcărea, C. Szántó, *Computational algebra with applications to coding theory and cryptography*, Ed. EFES, Cluj-Napoca, 2006.
- J. Daemen, V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*, Springer-Verlag, 2002.
- N. Ferguson, B. Schneier, *Practical Cryptography*, Wiley, 2003.
- C. Gherghe, D. Popescu, *Criptografie. Coduri. Algoritmi*, Univ. București, 2005.
- D. Kahn, *The Codebreakers*, Macmillan, 1967.
- N. Koblitz, *A Course in Number Theory and Cryptography*, Springer, 1994.

Selective bibliography (cont.)

- R. Lidl, G. Pilz, *Applied Abstract Algebra*, Springer-Verlag, 1998.
- A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997. [http://www.cacr.math.uwaterloo.ca/hac]
- C. Paar, J. Pelzl, *Understanding cryptography*, Springer, 2009.
- B. Schneier, *Applied Cryptography*, John Wiley and Sons, 1996.
- S. Singh, *The Code Book on CD-ROM* [http://www.simonsingh.net]
- S. Singh, *Cartea codurilor*, Humanitas, 2005.
- W. Stallings, *Cryptography and Network Security*, Prentice-Hall, 1999.
- D.R. Stinson, *Cryptography: Theory and Practice*, CRC Press, 1995.

Running Times

Complexity Theory:

- Goal: to offer methods to classify computational problems according to the resources needed to solve them.
- The classification should not depend on the computational model, but on the intrinsic difficulty of the problem.
- By "needed resources" one should understand the storage space, number of processors etc., and especially, time.
- According to Information Theory, almost any cryptographic algorithm can be broken. Complexity Theory tells us the time needed to do that.

Definition

Algorithm: a well-defined computational method, that takes a variable input and offers an output after a finite number of steps (computer program).

Running Times (cont.)

Definition

Size of an input: the total number of bits needed to represent it in the usual binary notation.

Examples. (a) Let $n \in \mathbb{N}$ and suppose that it has k binary digits. Then we have

$$2^{k-1} \leq n < 2^k,$$

$$k - 1 \leq \log_2 n < k,$$

$$[\log_2 n] = k - 1,$$

$$k = 1 + [\log_2 n] \sim \log n.$$

This will mean the base 2 logarithm if it is not stated otherwise.

(b) If $f = a_0 + a_1 X + \cdots + a_k X^k \in \mathbb{Z}[X]$, *degree*$(f) = k$ and $0 \leq a_i \leq n$, $\forall i = 0, \ldots, k$, then the size of f is of $(k+1) \log n$ bits.

Running Times (cont.)

Definition

- *Bit operation*: the operation of addition (or subtraction) of two bits together with everything that this involves.
- *Running time* of an algorithm with a given input: the number of primitive operations or steps performed.
  In general, by *step* one means a bit operation and we will do this throughout the course.
- *Estimation* of the time needed to perform an operation: the number of bit operations needed, neglecting memorization, logical tests etc., that in general are less time-consuming than the bit operations.

In general it is difficult to determine the exact running time of an algorithm. We need to find approximations for it, such as the *asymptotic running time*, that means the way of how the running time increases when the size of the input increases unbounded.

Big O Notation

We consider functions of the following type: $f : \mathbb{N} \to \mathbb{R}$ with $f(n) \geq 0,\ \forall n \geq n_0$ (we are interested in big values of n).

Definition

We write $f(n) = O(g(n))$ if $\exists c \in \mathbb{R}_+$ and $\exists n_0 \in \mathbb{N}$ such that $0 \leq f(n) \leq cg(n)\ \forall n \geq n_0$.

In fact, the "big O notation" tells us that f *does not increase faster* than g, up to a constant.
If $\lim\limits_{n \to \infty} \frac{f(n)}{g(n)}$ is finite, then $f(n) = O(g(n))$. Moreover, if the limit is non-zero, then we have both $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Theorem

(i) $f(n) = O(f(n))$.
(ii) $f(n) = O(g(n)),\ g(n) = O(h(n)) \Longrightarrow f(n) = O(h(n))$.
(iii) $f(n) = O(h(n)),\ g(n) = O(h(n)) \Longrightarrow (f+g)(n) = O(h(n))$.
(iv) $f(n) = O(h(n)),\ g(n) = O(l(n)) \Longrightarrow$
$(f \cdot g)(n) = O(h(n)l(n))$.

Examples. (a) If $f(n) = a_0 + a_1 n + \cdots + a_k n^k$ is a polynomial in n of degree k with all $a_k > 0$, then $f(n) = O(n^k)$.

(b) Let $0 < \delta < 1 < \alpha$. Then:

$$1 < \log\log n < \log n < e^{\sqrt{\log n \log\log n}} < n^{\delta}$$

$$n^{\delta} < n^{\alpha} < n^{\log n} < \alpha^{n} < n^{n} < \alpha^{(\alpha^{n})}.$$

Note that the function $f(n) = \log n$ is less than any exponential function $g(n) = n^{\delta}$.

(c) Let $a, b \in \mathbb{N}$ with $a, b \leq n$.

- The algorithm that adds (subtracts) a and b is of complexity $O(\log n)$. More precisely, $O(\log a + \log b)$.
- The algorithm that multiplies (divides) a and b is of complexity $O(\log^2 n)$. More precisely, $O((\log a) \cdot (\log b))$.

Complexity Classes

Definition

- *Polynomial-time algorithm*: an algorithm of complexity $O(N^k)$, where N is the size of the input (that is, $log\ n$ if n is the input) and k is a constant.
- *Exponential-time algorithm*: any non-polynomial-time algorithm.

Algorithm	Complexity	No. operations for $N = 10^6$	Time needed at 10^6 operations / sec.
constant	$O(1)$	1	1 μ sec.
linear	$O(N)$	10^6	1 sec.
quadratic	$O(N^2)$	10^{12}	1,6 days
cubic	$O(N^3)$	10^{18}	32000 years
exponential	$O(2^N)$	10^{301030}	10^{301006}· age of universe

- Polynomial-time algorithms $\mapsto$ efficient.
- Exponential-time algorithms $\mapsto$ inefficient.

Complexity Classes (cont.)

For simplicity, we may consider that Computational Complexity Theory reduces to decision (YES or NO) problems.

Definition

We define the following complexity classes:

- **P**: all decision problems that can be solved in polynomial time.
- **NP**: all decision problems for which the answer YES can be checked in polynomial time using an extra information, called *certificate*.
- **NPC**: the most "difficult" problems in **NP** (*NP-complete problems*).

If a problem belongs to **NP**, this does not necessarily mean that a certificate for the answer YES can be easily obtained, but just that it does exist and, if known, can be used to check the answer YES.

Examples. (a) Consider the following problem: given $n \in \mathbb{N}$, decide if n is composite, that is, $\exists a, b \in \mathbb{N}$, $a, b \geq 2$, such that $n = ab$.
This problem belongs to **NP**, because if n is composite, then this fact can be checked in polynomial time if one knows a divisor $1 < a < n$ of n. Here the certificate is the divisor a of n.

(b) *Subset Sum Problem*
Given a set $A = \{a_1, \ldots, a_n\}$ of natural numbers and $s \in \mathbb{N}$, determine whether or not there are some elements of A whose sum is s. This is an NP-complete problem.

(c) *Travelling Salesman Problem*
A travelling salesman has to visit n different cities using only one tank of gas (there is a maximum distance he can travel). Is there a route that allows him to visit each city exactly once?
This is an NP-complete problem.

The Euclidean Algorithm

Division Algorithm

$\forall a, b \in \mathbb{N},\ b \neq 0,\ \exists! q, r \in \mathbb{N}$ such that $a = bq + r$, where $r < b$.

One of the most efficient ways to compute $gcd(a, b)$, also denoted (a, b), is the Euclidean Algorithm.

Example. We have $(1547, 560) = 7$, because:

$$\begin{aligned} 1547 &= 2 \cdot 560 + 427 \\ 560 &= 1 \cdot 427 + 133 \\ 427 &= 3 \cdot 133 + 28 \\ 133 &= 4 \cdot 28 + 21 \\ 28 &= 1 \cdot 21 + 7 \\ 21 &= 3 \cdot 7 \end{aligned}$$

The Euclidean Algorithm (cont.)

The Euclidean Algorithm

- Input: $a, b \in \mathbb{N}$, $a, b \leq n$, $a \geq b$.
- Output: (a, b).
- Algorithm:
  while $b > 0$ do
  $\quad r := a \bmod b$; $a := b$; $b := r$;
  return(a).
- Time: $O(\log^2 n)$.

Theorem

Let $a, b \in \mathbb{N}$ and $d = (a, b)$. Then $\exists u, v \in \mathbb{Z}$: $d = au + bv$.

Corollary

Let $a, b \in \mathbb{N}$. Then $(a, b) = 1 \iff \exists u, v \in \mathbb{Z} : 1 = au + bv$.

Example. We already know that $(1547, 560) = 7$. Using the Euclidean Algorithm we succesively deduce, starting from the last equality:

$$
\begin{aligned}
7 &= 28 - 1 \cdot 21 \\
&= 28 - 1 \cdot (133 - 4 \cdot 28) \\
&= 5 \cdot 28 - 1 \cdot 133 \\
&= 5 \cdot (427 - 3 \cdot 133) - 1 \cdot 133 \\
&= 5 \cdot 427 - 16 \cdot 133 \\
&= 5 \cdot 427 - 16 \cdot (560 - 1 \cdot 427) \\
&= 21 \cdot 427 - 16 \cdot 560 \\
&= 21 \cdot (1547 - 2 \cdot 560) - 16 \cdot 560 \\
&= 21 \cdot 1547 - 58 \cdot 560.
\end{aligned}
$$

The Extended Euclidean Algorithm (cont.)

The Extended Euclidean Algorithm

- Input: $a, b \in \mathbb{N}$, $a, b \leq n$, $a \geq b$.
- Output: $d = (a, b)$ and $u, v \in \mathbb{Z}$ such that $au + bv = d$.
- Algorithm:

  $u_2 := 1$; $u_1 := 0$; $v_2 := 0$; $v_1 := 1$;
  while $b > 0$ do
      $q := \lfloor a/b \rfloor$; $r := a - qb$; $u := u_2 - qu_1$; $v := v_2 - qv_1$;
      $a := b$; $b := r$; $u_2 := u_1$; $u_1 := u$; $v_2 := v_1$; $v_1 := v$;
  $d := a$; $u := u_2$; $v := v_2$;
  write(d, u, v).
- Time: $O(\log^2 n)$.

Congruences Modulo n

Definition

Let $n \in \mathbb{N}$, $a, b \in \mathbb{Z}$. We have:

$$a \equiv b \pmod{n} \iff n | a - b.$$

If $n \neq 0$, then $a \equiv b \pmod{n} \iff$ a and b give the same remainder when divided by n.

Theorem

"$\equiv$" is an equivalence relation on $\mathbb{Z}$ and its corresponding partition is

$$\mathbb{Z}_n = \{x + n\mathbb{Z} \mid x \in \mathbb{Z}\} = \{\widehat{0}, \widehat{1}, \ldots, \widehat{n-1}\} \quad (\textit{for } n \geq 2).$$

Sometimes we simply write the classes without hats.

Congruences Modulo n (cont.)

Theorem

(i) $(\mathbb{Z}_n, +, \cdot)$ *is a ring, where*

$$\begin{aligned} \widehat{a} + \widehat{b} &= \widehat{a+b} \\ \widehat{a} \cdot \widehat{b} &= \widehat{a \cdot b}. \end{aligned}$$

(ii) $\widehat{0} \neq \widehat{a}$ *is invertible in* $\mathbb{Z}_n \iff (a, n) = 1 \iff \exists u, v \in \mathbb{Z}$ *such that* $au + nv = 1$. *In this case* $\widehat{a}^{-1} = \widehat{u}$.
(iii) $(\mathbb{Z}_n, +, \cdot)$ *is a field* $\iff$ n *is prime.*

Proposition

The complexity of the algorithm to compute the inverse modulo n *is* $O(\log^2 n)$.

Theorem

Consider

$$ax \equiv b \pmod{n} \tag{1}$$

where $0 \leq a, b < n$.
(i) If $(a, n) = 1$, *then* (1) *has solution, namely*

$$x_g \equiv a^{-1} b \pmod{n}.$$

(ii) If $(a, n) = d > 1$, *then (1) has solution* $\iff d | b$. *In this case, (1) has the same solutions as*

$$\frac{a}{d} x \equiv \frac{b}{d} \left(\mathrm{mod}\ \frac{n}{d}\right).$$

(iii) The complexity is $O(\log^2 n)$.

Example. Let us solve the congruences:
(a) $3x \equiv 4 \pmod{12}$.
(b) $8x \equiv 48 \pmod{18}$.

Since $(3, 12) = 3$ and $3 \nmid 4$, (a) does not have solution.

(b) is equivalent to

$$8x \equiv 12 \pmod{18}.$$

Since $(8, 18) = 2$ and $2|12$, (b) has the same solutions as

$$4x \equiv 6 \pmod{9},$$

that is,

$$x_g \equiv 4^{-1}6 \pmod{9},$$

where $4^{-1} = 7$ is the inverse modulo 9 of 4. Then the general solution of (b) is

$$x_g \equiv 6 \pmod{9}.$$

Chinese Remainder Theorem

Consider the system

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ \quad \cdots\cdots\cdots \\ x \equiv a_r \pmod{n_r} \end{cases}$$

where $a_i < n_i$, $n_i \in \mathbb{N}^*$, *and* $(n_i, n_j) = 1$, $\forall i, j \in \{1, \ldots, r\}$, $i \neq j$.
(i) The system has a unique solution modulo $N = n_1 n_2 \ldots n_r$, *namely*

$$x = \sum_{i=1}^{r} a_i N_i K_i,$$

where $N_i = N/n_i$, $K_i = N_i^{-1} \bmod n_i$.
(ii) The complexity is $O(\log^2 N)$.

Chinese Remainder Theorem (cont.)

Example. Let us solve the system:

$$\begin{cases} x \equiv 2 \pmod 5 \\ x \equiv 4 \pmod 7 \\ x \equiv 5 \pmod{11} \end{cases}$$

We have:

$$N = n_1 n_2 n_3 = 5 \cdot 7 \cdot 11 = 385,$$
$$N_1 = 77, \quad N_2 = 55, \quad N_3 = 35.$$

Compute $K_1 = N_1^{-1} \bmod n_1 = 77^{-1} \bmod 5$. So we have to solve $77y \equiv 1 \pmod 5$, that is, $2y \equiv 1 \pmod 5$. Then $K_1 = y = 3$.

Similarly, $K_2 = 55^{-1} \bmod 7 = 6$ and $K_3 = 35^{-1} \bmod 11 = 6$.

Then the solution is:

$$x \equiv 2 \cdot 77 \cdot 3 + 4 \cdot 55 \cdot 6 + 5 \cdot 35 \cdot 6 \equiv 137 \pmod{385}.$$

Euler's Function

Definition

The function $\varphi : \mathbb{N}^* \to \mathbb{N}^*$,
$\varphi(n) = |\{k \in \mathbb{N} \mid k < n \text{ and } (k, n) = 1\}|$ *is called* Euler's function.

Theorem

(i) If $(m, n) = 1$, *then* $\varphi(mn) = \varphi(m)\varphi(n)$.
(ii) If p *is prime, then* $\varphi(p) = p - 1$.
(iii) If $n = p^k$ *for some prime* p, *then* $\varphi(n) = n\left(1 - \frac{1}{p}\right)$.
(iv) If $n = p_1^{k_1} \dots p_j^{k_j}$ *for some primes* $p_1, \dots, p_j$, *then*

$$\varphi(n) = n\left(1 - \frac{1}{p_1}\right) \dots \left(1 - \frac{1}{p_j}\right).$$

One can easily compute $\varphi(n)$ GIVEN the factorization of n.

Example. $\varphi(98) = \varphi(2 \cdot 7^2) = \varphi(2) \cdot \varphi(7^2) = 1 \cdot 7^2 \cdot (1 - \frac{1}{7}) = 42$.

Repeated Squaring Modular Exponentiation

Let us compute $b^k \bmod n$, where $b, k \in \mathbb{N}$ are large.
Write k in binary, say $k = \sum_{i=0}^{t} k_i 2^i$. We have (modulo n):

$$b^k = \prod_{i=0}^{t} b^{k_i 2^i} = (b^{2^0})^{k_0}(b^{2^1})^{k_1}\dots(b^{2^t})^{k_t}.$$

Example. Let us compute $42^{51} \bmod 73$.
We have $51 = 2^0 + 2^1 + 2^4 + 2^5$. Compute modulo 73:

$$\begin{aligned}
42^{(2^0)} &= 42,\\
42^{(2^1)} &= 42^{(2^0)} \cdot 42^{(2^0)} = 42 \cdot 42 = 12,\\
42^{(2^2)} &= 42^{(2^1)} \cdot 42^{(2^1)} = 12 \cdot 12 = 71,\\
42^{(2^3)} &= 42^{(2^2)} \cdot 42^{(2^2)} = 71 \cdot 71 = 4,\\
42^{(2^4)} &= 42^{(2^3)} \cdot 42^{(2^3)} = 4 \cdot 4 = 16,\\
42^{(2^5)} &= 42^{(2^4)} \cdot 42^{(2^4)} = 16 \cdot 16 = 37.
\end{aligned}$$

Then $42^{51} = 42^{2^0+2^1+2^4+2^5} = 42 \cdot 12 \cdot 16 \cdot 37 = 17 \pmod{73}$.

Repeated Squaring Modular Exponentiation (cont.)

Repeated Squaring Modular Exponentiation

- Input: $b, k, n \in \mathbb{N}$ with $b < n$ and $k = \sum_{i=0}^{t} k_i 2^i$.
- Output: $a = b^k \bmod n$.
- Algorithm:

  $a := 1;$
  if $k = 0$ then write(a);
  $c := b;$
  if $k_0 = 1$ then $a := b;$
  for $i = 1$ to t do
      $c := c^2 \bmod n;$
      if $k_i := 1$ then $a := c \cdot a \bmod n;$
  write(a).
- Time: $O((\log k)(\log^2 n))$.

Selective Bibliography

N. Koblitz, *A Course in Number Theory and Cryptography*, Springer, 1994.

A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997. [http://www.cacr.math.uwaterloo.ca/hac]

B. Schneier, *Applied Cryptography*, John Wiley and Sons, 1996.