

Algebraic Foundations of Computer Science.

Computational Introduction to Number theory (III)

Ferucio Laurențiu Tiplea

Department of Computer Science
"AL.I.Cuza" University of Iași
Iași, Romania
E-mail: ftiplea@mail.dntis.ro

Spring 2015

1 *Asymptotic notation*

2 *Complexity of the basic arithmetic operations*

3 *Course readings*

Given $g : \mathbb{N} \rightarrow \mathbb{R}_+$ a function, define the following sets:

$$\mathcal{O}(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_+ \mid (\exists c \in \mathbb{R}_+^*)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(f(n) \leq cg(n))\}$$

$$\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_+ \mid (\exists c \in \mathbb{R}_+^*)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(cg(n) \leq f(n))\}$$

$$\Theta(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_+ \mid (\exists c_1, c_2 \in \mathbb{R}_+^*)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)$$

$$(c_1g(n) \leq f(n) \leq c_2g(n))\}$$

$$o(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_+ \mid (\forall c \in \mathbb{R}_+^*)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(f(n) \leq cg(n))\}$$

Definition 1

Let $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$ and $X \in \{\mathcal{O}, \Omega, \Theta, o\}$. f is said to be X of g , denoted $f(n) = X(g(n))$, if $f \in X(g)$.

\mathcal{O} (“big O”), Ω (“big Ω ”), Θ (“big Θ ”), and o (“little o”) are **order of magnitude symbols**.

Asymptotic notation illustrated

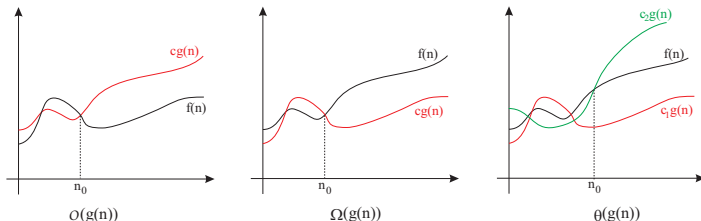
Algebraic
Foundations of
Computer Science
(AFCS)

Prof.Dr. F.L.
Tiplea

Asymptotic
notation

Complexity of
arith. operations

Course readings



- $f(n) = \mathcal{O}(g(n))$
 - $g(n)$ is an asymptotic upper bound for $f(n)$
 - $f(n)$ is **no more than** $g(n)$
 - used to state the **complexity of a worst case** analysis;
- $f(n) = \Omega(g(n))$ – similar interpretation;
- $f(n) = o(g(n))$ – $f(n)$ is **less than** $g(n)$ (the difference between \mathcal{O} and o is analogous to the difference between \leq and $<$).

Proposition 1

Let $f, g, h, k : \mathbb{N} \rightarrow \mathbb{R}_+$. Then:

- 1 $f(n) = \mathcal{O}(f(n))$;
- 2 if $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$, then $f(n) = \mathcal{O}(h(n))$;
- 3 $f(n) = \mathcal{O}(g(n))$ iff $g(n) = \Omega(f(n))$;
- 4 $f(n) = \Theta(g(n))$ iff $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$;
- 5 if $f(n) = \mathcal{O}(h(n))$ and $g(n) = \mathcal{O}(k(n))$, then
 $(f \cdot g)(n) = \mathcal{O}(h(n)k(n))$ and
 $(f + g)(n) = \mathcal{O}(\max\{h(n), k(n)\})$;
- 6 if there exists $n_0 \in \mathbb{N}$ such that $g(n) \neq 0$ for any $n \geq n_0$, then
 $f(n) = o(g(n))$ iff $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Some useful inequalities are in order:

- (Stirling's formula)

$$\sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n+1}} \leq n! \leq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n}},$$

for any $n \geq 1$;

- for any real constants ϵ and c such that $0 < \epsilon < 1 < c$,

$$1 < \ln \ln n < \ln n < e^{\sqrt{(\ln n)(\ln \ln n)}} < n^\epsilon < n^c < n^{\ln n} < c^n < n^n < c^{c^n}$$

(each inequality holds for all $n \geq n_0$, where n_0 is suitable chosen).

Example 2

- ❶ If $f(x) = a_0 + a_1x + \dots + a_kx^k$ is a polynomial of degree k with real coefficients and $f(x) \geq 0$ for any $x \in \mathbb{N}$, then $f(n) = \Theta(n^k)$.
- ❷ $\log_c n = \Theta(\log n)$, for any real constant $c > 1$.
- ❸ $\log n = \mathcal{O}(n^\epsilon)$, for any real number ϵ such that $0 < \epsilon < 1$.
- ❹ $\log^k n = \mathcal{O}(n)$, for any natural number $k \geq 1$.
- ❺ $n! = \Omega(2^n)$ and $n! = o(n^n)$.
- ❻ $\log(n!) = \Theta(n \log n)$.
- ❼ If $f : \mathbb{N} \rightarrow \mathbb{R}_+$ satisfies $f(n) \geq 1$ for any $n \geq n_0$ and some $n_0 \in \mathbb{N}$, then

$$f(n) = \Theta(2^{\lceil \log_2 f(n) \rceil}).$$
- ❽ $4^n \neq \mathcal{O}(2^n)$.

Operations with classes of functions

Algebraic
Foundations of
Computer Science
(AFCS)

Prof.Dr. F.L.
Tiplea

Asymptotic
notation

Complexity of
arith. operations

Course readings

Let \mathcal{A} and \mathcal{B} be sets of functions as those defined above (e.g., $\mathcal{O}(g)$ etc.), and let $f : \mathbb{N} \rightarrow \mathbb{R}_+$. Then, we write

$$\bullet f + \mathcal{A} = \{f + g \mid g \in \mathcal{A}\};$$

$$\bullet \mathcal{A} + \mathcal{B} = \{f + g \mid f \in \mathcal{A}, g \in \mathcal{B}\};$$

$$\bullet f\mathcal{A} = \{f \cdot g \mid g \in \mathcal{A}\}. \text{ If } f \text{ is the constant } c \text{ function, then we will write } c\mathcal{A} \text{ instead of } f\mathcal{A};$$

$$\bullet \mathcal{A}\mathcal{B} = \{fg \mid f \in \mathcal{A}, g \in \mathcal{B}\};$$

$$\bullet \mathcal{O}(\mathcal{A}) = \bigcup_{f \in \mathcal{A}} \mathcal{O}(f).$$

Convention: $\mathcal{A} = \mathcal{B}$ stands for $\mathcal{A} \subseteq \mathcal{B}$.

Proposition 2

Let $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$ and $c \in \mathbb{R}_+$. Then:

- 1 $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n));$
- 2 $c\mathcal{O}(f(n)) = \mathcal{O}(f(n));$
- 3 $\mathcal{O}(\mathcal{O}(f(n))) = \mathcal{O}(f(n));$
- 4 $\mathcal{O}(f(n))\mathcal{O}(g(n)) = \mathcal{O}(f(n)g(n));$
- 5 $\mathcal{O}(f(n)g(n)) = f(n)\mathcal{O}(g(n)).$

Programming with large integers (with more than 100 digits) is crucial for cryptography, computer security, computational algebra.

There have been developed several libraries for large integer arithmetic:

- LIDIA (Darmstadt University of Technology)
<http://www.cdc.informatik.tu-darmstadt.de/TI/Lidia>
- PARI/GP (Université Bordeaux I, France)
<http://pari.math.u-bordeaux.fr/>
- NTL (New York University)
<http://www.shoup.net/ntl/>
- GMP (faster than any other multi-precision library)
<http://gmplib.org/>

MpNT is a large integer library developed at “Al.I.Cuza” University of Iasi.

Base representation of integers

Algebraic
Foundations of
Computer Science
(AFCS)

Prof.Dr. F.L.
Tiplea

Asymptotic
notation

Complexity of
arith. operations

Course readings

Let $\beta \geq 2$ be an integer called **base**. Each non-negative integer $a < \beta$ is called a **digit of the base β** or a **base β digit**.

Any non-negative integer a can be represented in a given base β as:

$$a = a_{k-1}\beta^{k-1} + \cdots + a_1\beta + a_0,$$

where $0 \leq a_i < \beta$ for all $0 \leq i < k$, and $a_{k-1} > 0$. The k -ary vector

$$(a_{k-1}, \dots, a_0)_\beta$$

is called the **representation of a in base β** , the numbers a_i are called the **digits of a in base β** , and k is the **length of the representation**. k is also denoted by $|a|_\beta$ and it satisfies

$$|a|_\beta = \lfloor \log_\beta a \rfloor + 1$$

(convention: $\log_\beta 0 = 0$). Therefore, $|a|_\beta = \mathcal{O}(\log a)$.

Time complexity of algorithms will always be measured in **digit operations**, i.e., logical or arithmetic operations on digits. These operations are:

- digit comparison;
- digit addition, subtraction, and multiplication. Any of these operations takes two digits and a carry and produces a digit and a carry;
- division of a 2-digit number by a digit (the result consists of a quotient and a remainder).

Remark 1

The **shifting and copying operations** are not considered digit operations. In practice, these operations are fast in comparison with digit operations, so they can be safely ignored.

Time complexity: addition and subtraction

*Algebraic
Foundations of
Computer Science
(AFCS)*

*Prof.Dr. F.L.
Tiplea*

*Asymptotic
notation*

*Complexity of
arith. operations*

Course readings

Assume that both operands have the same length k of representation in base β :

- ➊ addition with carry: $\mathcal{O}(k)$
- ➋ subtraction with borrow: $\mathcal{O}(k)$

Assume that both operands have the same length k of representation in base β :

- **Schoolbook multiplication:** $\mathcal{O}(k^2)$

It is based on computing partial sums after each row multiplication. In this way, the intermediate numbers obtained by addition do not exceed $\beta^2 - 1$ and, therefore, a basic procedure for multiplication of two base β digits can be used;

- **Karatsuba multiplication:** $\mathcal{O}(k^{\log 3})$

It reduces the multiplication of two k -digit numbers to $\mathcal{O}(k^{\log 3})$ single-digit multiplications by recursively splitting the operands into two smaller parts.

Generalization: **Toom-Cook algorithm**;

- **FFT based multiplication:** $\mathcal{O}(k \log k)$

Uses the Fast Fourier Transform and it is efficient for very large inputs (e.g., 1000-digit inputs).

Assume that the dividend length is $2k$ and the divisor length is k :

- **Schoolbook Division:** $\mathcal{O}(k^2)$

The method uses, as a basic step, the division of a $(k + 1)$ -digit integer by a k -digit integer. The main problem is to **guess efficiently the quotient**;

- **Recursive division:** $\mathcal{O}(k^{\log 3} + k \log k)$

It is based on a similar idea to that of Karatsuba's multiplication algorithm.

- 1 Euclidean algorithm: $\mathcal{O}((\log a)(\log b))$
- 2 Binary gcd algorithm: $\mathcal{O}((\log ab)^2)$.
- 3 Lehmer's or Sorenson's algorithm: $\mathcal{O}(k^2 / \log k)$
where operands have length at most k .
- 4 Schönhage's algorithm (the fastest): $\mathcal{O}(k(\log k)^2 \log \log k)$
where operands have length at most k . This algorithm is
based on FFT arithmetic and it is efficient for very large
inputs.

Extended version of gcd algorithms: same complexity as for the corresponding gcd algorithm.

Arithmetic in \mathbb{Z}_m is usually called **modular arithmetic**. The basic modular operations (addition, subtraction, multiplication, exponentiation) are obtained from the basic operations on integers plus a **modular reduction**.

Modular addition and subtraction can be easily implemented taking into account the following remarks:

- $a + b < 2m$,
- if $a \geq b$, then $0 \leq a - b < m$, and
- if $a < b$ then $a + m - b < m$,

for all $a, b \in \mathbb{Z}_m$.

Therefore, the complexity of modular addition/subtraction is $\mathcal{O}(k)$, where k is the length of m .

Time complexity: Modular arithmetic

Things are more involved in case of modular multiplication or exponentiation, where efficient modular reduction techniques are required.

Three main techniques for modular reduction are mostly used:

- reduction by division
- Barrett reduction
- Montgomery reduction

	Barrett	Montgomery	Recursive Division
Restrictions on input	$a < \beta^{2k}$	$a < m\beta^k$	None
Pre-computation	$\lfloor \beta^{2k}/m \rfloor$	$-m_0^{-1} \bmod \beta$	Normalization
Post-computation	None	None	Unnormalization
Multiplication	$k(k+4)$	$k(k+1)$	$2K(k) + \mathcal{O}(k \log k)$

Figure: Complexity of multiplication under the three reduction methods

Modular exponentiation problem: Compute $a^e \bmod m$, where $m \geq 2$, $a \in \mathbb{Z}_m$, and $e > 0$.

- The **naive method**: $\mathcal{O}(e \cdot (\log m)^2)$

Compute $a^e \bmod m$ by performing $e - 1$ multiplications modulo m .

- **Exponentiation by squaring**: $\mathcal{O}(\log e \cdot (\log m)^2)$

Decompose e in base 2, $(e_{k-1}, e_{k-2}, \dots, e_1, e_0)_2$, and

$$a^e \bmod m = (\dots ((a^{e_{k-1}})_m^2 \otimes_m a^{e_{k-2}})_m^2 \dots a^{e_1})_m^2 \otimes_m a^{e_0}$$

The number of multiplications is

$$(k - 1) + e_{k-2} + \dots + e_0 < 2|e|_2$$

There is a large variety of exponentiation algorithms.

Algorithm 1: Exponentiation by Squaring

input : $m \geq 2$, $a \in \mathbb{Z}_m$, and $e > 0$;

output: $z = a^e \bmod m$;

begin

$y := a$;

$z := 1$;

while $e > 1$ **do**

$f := e \text{ div } 2$;

if $e > 2f$ **then**

$z := z \cdot y \bmod m$;

$y := y \cdot y \bmod m$;

$e := f$

$z := z \cdot y \bmod m$

Odd integer $n > 1$ and integer $a > 0$:

- The **naive method**: $\mathcal{O}((\log |a|)(\log n))$
- Using the asymptotically fastest gcd algorithm:
 $\mathcal{O}(\log n \cdot (\log n)^2 \cdot \log \log n)$

For practical input sizes, the most efficient algorithms seem to be variants of the binary gcd (adapted to compute the Jacobi symbol)

Factoring a positive integer n means finding two positive integers $a, b > 1$ such that $n = ab$.

Factoring a composite integer is believed to be a hard problem (due to our failure so far to find a fast and practical factoring algorithm):

- **RSA-200**, a 663 bit integer, was factored on May 9, 2005, by a team of the German Federal Agency for Information Security. The work began in late 2003. **The sieving effort is estimated to have taken the equivalent of 55 years on a single 2.2 GHz Opteron CPU.**
- **RSA-768**, a 232 digit number, was factored on Dec 12, 2009, and it took almost two years. **On a single core 2.2 GHz AMD Opteron processor with 2 GB RAM, sieving would have taken about 1500 years.**

Time complexity: Factorization

Algebraic
Foundations of
Computer Science
(AFCS)

Prof.Dr. F.L.
Tiplea

Asymptotic
notation

Complexity of
arith. operations

Course readings

- 1 Factorization by trial division: requires $\pi(\sqrt{n})$ trial divisions (without counting primality testing)
- 2 Number Field Sieve: $\mathcal{O}(e^{(1.923+o(1))\sqrt[3]{(\ln m)(\ln \ln m)^2}})$
- 3 Quadratic Sieve: $\mathcal{O}(e^{(1+o(1))\sqrt{(\ln m)(\ln \ln m)}})$

A **primality test** is an algorithm for determining whether a positive integer is prime.

Since 1960s there were many attempts to find efficient algorithm for primality testing:

- 1 Pratt, 1975: the primality problem is in $NP \cap co - NP$;
- 2 Miller, 1976: the primality problem can be solved by deterministic polynomial time algorithms assuming *Extended Riemann Hypothesis* (ERH);
- 3 Solovay and Strassen, 1977: developed a probabilistic polynomial time algorithm;
- 4 Adleman, Pomerance, and Rumely: deterministic algorithm running in $(\log n)^{O(\log \log \log n)}$ time;
- 5 Agrawal, Kayal, and Saxena, 2002: **primality problem can be solved by deterministic algorithms running in $O((\log n)^{10.5})$.**

Time complexity: Primality

Algebraic
Foundations of
Computer Science
(AFCS)

Prof.Dr. F.L.
Tiplea

Asymptotic
notation

Complexity of
arith. operations

Course readings

The simplest primality test is to check whether there are integers less than or equal to \sqrt{n} which divides n .

There are ways to speed up this test but it still remains very slow for large integers that have large prime factors.

Although the algorithm proposed by Agrawal, Kayal, and Saxena works in polynomial time, its complexity is still high. In practice, the most efficient primality tests are **probabilistic**:

- Fermat primality test;
- Solovay-Strassen primality test;
- Miller-Rabin primality test.



Fermat primality test is based on the congruence

$$F(n, a) : a^{n-1} \equiv 1 \pmod{n}$$

for any prime n and integer a such that $(a, n) = 1$.

Definition 3

Let $n > 2$ be an odd composite integer and $1 \leq a < n$.

-  a is called a **Fermat witness for n** if $\neg F(n, a)$.
-  a is called a **Fermat liar for n** if $F(n, a)$.

There are odd composite integers n such that all $a \in \mathbb{Z}_n^*$ are Fermat liars. These are called **Carmichael numbers**.

Algorithm 2: Fermat primality test

input : $n > 2$ odd;**output**: “ n is composite” or “ n is prime”;**begin** choose randomly a , $1 < a < n$; **if** $(a, n) > 1$ **then** | “ n is composite” **else** **if** $a^{n-1} \not\equiv 1 \pmod n$ **then** | “ n is composite” **else** | “ n is prime”

Complexity: $\mathcal{O}((\log n)^3)$

Given a positive integer $n \geq 2$, denote

$$\mathcal{F}_n = \{a \mid 1 \leq a < n \wedge F(n, a)\}.$$

Theorem 4

Let $n \geq 2$ be an odd composite integer. Then, $\mathcal{F}_n \subseteq \mathbb{Z}_n^*$ and, if there exists $a \in \mathbb{Z}_n^*$ such that $\neg F(n, a)$, then \mathcal{F}_n is a proper subgroup of \mathbb{Z}_n^* .

Fermat primality test gives a wrong answer with probability less than $1/2$.

Solovay-Strassen primality test is based on the congruence

$$E(n, a) : \left(\frac{a}{n}\right) \cdot a^{\frac{n-1}{2}} \bmod n = 1$$

for any odd prime n and integer a such that $(a, n) = 1$.

Definition 5

Let $n > 2$ be an odd composite integer and $1 \leq a < n$.

- a is called an **Euler witness for n** if $\neg E(n, a)$.
- a is called an **Euler liar for n** if $E(n, a)$.

There are no odd composite integers n without Euler witnesses (in other words, there are no Carmichael numbers in this case).

Algorithm 3: Solovay-Strassen primality test

input : $n > 2$ odd;

output: “ n is composite” or “ n is prime”;

begin

choose randomly a , $1 < a < n$;

if $\left(\frac{a}{n}\right) = 0$ then
“ n is composite”

e|se

if $\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}$ then
| “n is prime”

else

“ n is composite”

Complexity: $\mathcal{O}((\log n)^3)$

Given a positive integer $n \geq 2$, denote

$$\mathcal{E}_n = \{a \mid 1 \leq a < n \wedge E(n, a)\}.$$

Theorem 6

Let $n \geq 2$ be an odd composite integer. Then,

- $\mathcal{E}_n \subseteq \mathcal{F}_n$;
- \mathcal{E}_n is a proper subgroup of \mathbb{Z}_n^* .

Solovay-Strassen primality test gives a wrong answer with probability less than $1/2$.

Miller-Rabin primality test is based on the following theorem:

Theorem 7

Let $n \geq 2$ be an odd integer, $n = 2^s t + 1$, where $s, t \geq 1$ and t is odd. Then, n is prime if and only if

$$M(n, a) : a^t \equiv 1 \pmod{n} \vee (\exists i < s)(a^{2^i t} \equiv -1 \pmod{n}),$$

for any integer a such that $(n, a) = 1$.

Definition 8

Let $n > 2$ be an odd composite integer and $1 \leq a < n$.

-  a is called an **Miller-Rabin witness for n** if $\neg M(n, a)$.
-  a is called an **Miller-Rabin liar for n** if $M(n, a)$.

Algorithm 4: Miller-Rabin primality test

input : $n > 2$ odd;

output: “ n is composite” or “ n is prime”;

begin

 decompose $n := 2^s t + 1$, where t is odd;

 choose randomly a , $1 < a < n$;

if $(a, n) > 1$ **then**

 | “ n is composite”

else

$r := a^t \bmod n$;

if $r \in \{1, n-1\}$ **then** “ n is prime”;

if $(\exists 1 \leq i \leq s-1)(r^{2^i} \equiv -1 \bmod n)$ **then**

 | “ n is prime”

else

 | “ n is composite”

Complexity: $\mathcal{O}((\log n)^3)$

Given a positive integer $n \geq 2$, denote

$$\mathcal{M}_n = \{a \mid 1 \leq a < n \wedge M(n, a)\}.$$

Theorem 9

Let $n \geq 2$ be an odd composite integer. Then, $\mathcal{M}_n \subseteq \mathcal{F}_n$ and

$$|\mathcal{M}_n| \leq \frac{n-1}{4}.$$

Miller-Rabin primality test gives a wrong answer with probability at most $1/4$.

- F.L. Țiplea: *Fundamentele Algebrice ale Informaticii*, Ed. Polirom, Iași, 2006, **pag. 172–178**.
- F.L. Țiplea, S. Iftene, C. Hrițcu, I. Goriac, R.M. Gordân, E. Erbiceanu: *MpNT: A Multi-precision Number Theory Package. Number-theoretic Algorithms (I)*, Technical Report, “Al.I.Cuza” University of Iași, 2003.