afcs
ALGEBRAIC FOUNDATIONS of COMPUTER SCIENCE
lectures by Ferucio Laurentiu Tiplea

Computational Introduction to Number Theory

Part III

Prof.dr. Ferucio Laurențiu Țiplea

Spring 2022

Department of Computer Science
“Alexandru Ioan Cuza” University of Iași
Iași 700506, Romania

e-mail: `ferucio.tiplea@uaic.ro`

Outline

Asymptotic notation

Asymptotic notation

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

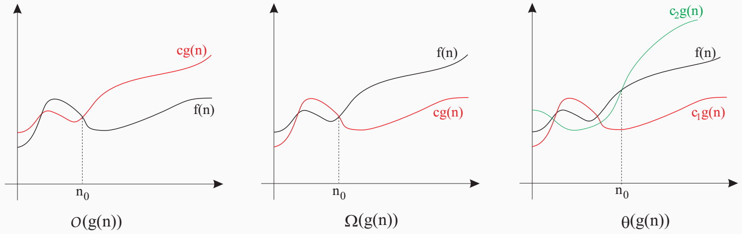
$$\begin{aligned}
\mathcal{O}(g) &= \{f : \mathbb{N} \to \mathbb{R}_+ | (\exists c \in \mathbb{R}_+^*)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(f(n) \leq cg(n))\} \\
\Omega(g) &= \{f : \mathbb{N} \to \mathbb{R}_+ | (\exists c \in \mathbb{R}_+^*)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(cg(n) \leq f(n))\} \\
\Theta(g) &= \{f : \mathbb{N} \to \mathbb{R}_+ | (\exists c_1, c_2 \in \mathbb{R}_+^*)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0) \\
&\qquad\qquad (c_1 g(n) \leq f(n) \leq c_2 g(n))\} \\
o(g) &= \{f : \mathbb{N} \to \mathbb{R}_+ | (\forall c \in \mathbb{R}_+^*)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(f(n) \leq cg(n))\}
\end{aligned}$$

Definition 1

Let $f, g : \mathbb{N} \to \mathbb{R}_+$ 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



- $f(n) = \mathcal{O}(g(n))$
  - $g(n)$ 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 $<$).

Basic properties of asymptotic notation

It is a good exercise for you to prove the following proposition.

Proposition 2

Let $f, g, h, k : \mathbb{N} \to \mathbb{R}_+$. 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* $lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Useful inequalities

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).

Asymptotic notation: examples

Example 3

Prove the following properties:

1. If $f(x) = a_0 + a_1 x + \cdots + a_k x^k$ is a polynomial of degree k with real coefficients and $f(x) \geq 0$ for any $x \in \mathbb{N}$, then $f(n) = \Theta(n^k)$.
2. $log_c\, n = \Theta(log\, n)$, for any real constant $c > 1$.
3. $log\, n = \mathcal{O}(n^\epsilon)$, for any real number ϵ such that $0 < \epsilon < 1$.
4. $log^k\, n = \mathcal{O}(n)$, for any natural number $k \geq 1$.
5. $n! = \Omega(2^n)$ and $n! = o(n^n)$.
6. $log(n!) = \Theta(nlog\, n)$.
7. If $f : \mathbb{N} \to \mathbb{R}_+$ satisfies $f(n) \geq 1$ for any $n \geq n_0$ and some $n_0 \in \mathbb{N}$, then
$$f(n) = \Theta(2^{\lceil log_2 f(n) \rceil}).$$
8. $4^n \neq \mathcal{O}(2^n)$.

Operations with classes of functions

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

1. $f + \mathcal{A} = \{f + g | g \in \mathcal{A}\}$;
2. $\mathcal{A} + \mathcal{B} = \{f + g | f \in \mathcal{A}, g \in \mathcal{B}\}$;
3. $f\mathcal{A} = \{f \cdot g | g \in \mathcal{A}\}$. If f is the constant c function, then we will write $c\mathcal{A}$ instead of $f\mathcal{A}$;
4. $\mathcal{A}\mathcal{B} = \{fg | f \in \mathcal{A}, g \in \mathcal{B}\}$;
5. $\mathcal{O}(\mathcal{A}) = \bigcup_{f \in \mathcal{A}} \mathcal{O}(f)$.

Basic properties

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

It is a good exercise for you to prove the following proposition.

Proposition 4

Let $f, g : \mathbb{N} \to \mathbb{R}_+$ 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. $\mathcal{O}(f(n)g(n)) = f(n)\mathcal{O}(g(n))$.

Complexity of the basic arithmetic operations

Programming with large integers

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:

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

MpNT is a large integer library developed at “Alexandru Ioan Cuza” University of Iasi.

Base representation of integers

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

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

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

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 5

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

Time complexity: addition and subtraction

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

1. addition with carry: $\mathcal{O}(k)$
2. subtraction with borrow: $\mathcal{O}(k)$

Time complexity: multiplication

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

1. 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;
2. 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;
3. 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).

Time complexity: division

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

1. 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;

2. Recursive division: $\mathcal{O}(k^{\log 3} + k \log k)$
   It is based on a similar idea to that of Karatsuba's multiplication algorithm.

Time complexity: GCD

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.

Time complexity: modular arithmetic

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:

1. $a + b < 2m$,
2. if $a \geq b$, then $0 \leq a - b < m$, and
3. 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:

1. reduction by division
2. Barrett reduction
3. Montgomery reduction

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

Figure 1: Complexity of multiplication under the three reduction methods

Time complexity: modular arithmetic

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

1. The naive method: $\mathcal{O}(e \cdot (\log m)^2)$
   Compute $a^e \ mod \ m$ by performing $e - 1$ multiplications modulo m.

2. Exponentiation by squaring: $\mathcal{O}(\log e \cdot (\log m)^2)$
   Decompose e in base 2, $(e_{k-1}, e_{k-2}, \ldots, e_1, e_0)_2$, and

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

   The number of multiplications is

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

There is a large variety of exponentiation algorithms.

Time complexity: modular arithmetic

Algorithm 1: Exponentiation by Squaring

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

```
1  begin
2      $y := a$;
3      $z := 1$;
4      while $e > 1$ do
5          $f := e \ div \ 2$;
6          if $e > 2f$ then
7              $z := z \cdot y \ mod \ m$;
8          $y := y \cdot y \ mod \ m$;
9          $e := f$
10     $z := z \cdot y \ mod \ m$
```

Time complexity: Jacobi symbol

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

1. The naive method: $\mathcal{O}((\log a)(\log n))$

2. 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).

Time complexity: factorization

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):

1. 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.

2. 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.

For more info, check RSA Factoring Challenge on Wikipedia.

Time complexity: factorization

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

Time complexity: primality

Attempts to find efficient algorithm for primality testing:

1. The simplest primality test: check for factors $\leq \sqrt{n}$. It is very slow for large integers that have large prime factors;
2. Pratt, 1975: the primality problem is in $NP \cap co-NP$;
3. Miller, 1976: primality can be solved by deterministic polynomial time algorithms assuming *Extended Riemann Hypothesis* (ERH);
4. Solovay and Strassen, 1977: PPT algorithm $\mathcal{O}(\log^3 n)$;
5. Adleman, Pomerance, and Rumely: $(\log n)^{\mathcal{O}(\log\log\log n)}$ time;
6. Agraval, Kayal, and Saxena, 2002: $\tilde{\mathcal{O}}(\log^{7.5} n)$;
7. Lenstra and Pomerance, 2003: $\tilde{\mathcal{O}}(\log^6 n)$.

soft-$\mathcal{O}$ notation $f(n) = \tilde{\mathcal{O}}(g(n))$: $\exists k$ s.t. $f(n) = \mathcal{O}(g(n)\log^k g(n))$

Probabilistic primality testing – still the most efficient!

Time complexity: probabilistic primality testing

Fermat primality test is based on the congruence

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

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

Definition 6

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

1. *a is called a Fermat witness for n if $\neg F(n,a)$.*
2. *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.

Time complexity: probabilistic primality testing

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 $a^{n-1} \not\equiv 1 \bmod n$ then
            "$n$ is composite"
        else
            "$n$ is prime"
```

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

Time complexity: probabilistic primality testing

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

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

Theorem 7

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 $1/2$.

Time complexity: probabilistic primality testing

Solovay-Strassen primality test is based on the congruence

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

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

Definition 8

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

1. *a is called an Euler witness for n if $\neg E(n,a)$.*
2. *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).

Time complexity: probabilistic primality testing

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"
    else
        if $\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \bmod n$ then
            "$n$ is prime"
        else
            "$n$ is composite"
```

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

Time complexity: probabilistic primality testing

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

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

Theorem 9

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

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

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

Time complexity: probabilistic primality testing

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

Theorem 10

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

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

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

Definition 11

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

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

Time complexity: probabilistic primality testing

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"
```

Time complexity: probabilistic primality testing

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

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

Theorem 12

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 $1/4$.

Reading and exercise guide

Reading and exercise guide

It is highly recommended that you do all the exercises marked in red from the slides.

Course readings:

1. For asymptotic notation, read pages 196-212 from textbook [1];
2. For the complexity of the basic arithmetic operations read the entire report [2].

References

[1] Ferucio Laurențiu Țiplea. *Algebraic Foundations of Computer Science*. “Alexandru Ioan Cuza” University Publishing House, Iași, Romania, second edition, 2021.

[2] C. Hrițcu I. Goriac R.M. Gordân E. Erbiceanu F.L. Țiplea, S. Iftene. A multi-precision number theory package. number-theoretic algorithms (I). Technical report, “Alexandru Ioan Cuza” University of Iași, 2003.