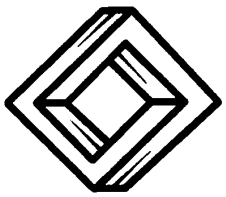
Publicaciones Electrónicas
Sociedad Matemática Mexicana

Theory of Algorithms
-Course Handouts-

Guillermo Morales-Luna

www.sociedadmatematicamexicana.org.mx

Serie: Textos. Vol. 11 (2009)



Guillermo Morales-Luna

Theory of Algorithms

– Course Handouts –

November 30, 2009

Cinvestav-IPN

Foreword

This course follows general lines in *Algorithm Design* as suggested by the *Association of Computing Machinery* (ACM). It is addressed to both the professional engineer or technician, working on software implementation of general computing procedures, and to students in Computer Engineering or Computer Science, who are prospective instructors or developers.

We follow closely classical texts in the area. Indeed, textbooks that are also continuous reference books. In chronological order, a short list contains [Aho et al(1974)Aho, Hopcroft, and Ullman], [Brassard and Bratley(1988)] (with a version in Spanish, [Brassard and Bratley(1990)]), [Manber(1989)], [Sedgewick and Flajolet(1996)], [Papadimitriou and Steiglitz(1998)], [Hopcroft et al(2001)Hopcroft, Motwani, and Ullman], [Cormen et al(2003)Cormen, Leiserson, Rivest, and Stein] and [Levitin(2007)]. Textbooks with emphasis in implementational matters are [Gonnet and Baeza-Yates(1991)], [Wood(1993)] and [Cattaneo and Italiano(1999)]. In the Algorithms Design a good background in Mathematics is always necessary. Here the suggested lectures are [Graham et al(1994)Graham, Knuth, and Patashnik] and [Knuth(1999)]. For the interested reader in the Computing limitations, the classical reference is [Garey and Johnson(1979)].

The current exposition has been taught in a series of courses lectured at the Information Technology Laboratory, Cinvestav-IPN, in Tamaulipas.

Mexico, July 2009 *Guillermo Morales-Luna*

Contents

Chapter 1
Basic Analysis

Abstract We introduce here the most basic concepts in measuring algorithms complexities. Initially we distinguish behaviors according to the computational costs due to their inputs. Then we introduce some mathematical functions of extended use in measuring algorithmic performances, and the growth orders of maps. We study the sums, with either finite or countable number of summands, and the convergence properties. We outline some policies in order to get good algorithm implementations. Then we discuss the tradeoff among time and space in computing algorithms. Finally we sketch the most fundamental criteria of growth estimation of functions determined by recurrence relation. The current chapter is a introduction to the mathematics of the analysis of algorithms.

1.1 Best, average and worst case behaviors

1.1.1 General definitions

Let Σ be a finite alphabet. Σ^* denotes the *dictionary* over Σ, or the collection of words of finite length with symbols in that alphabet. For any word $\sigma \in \Sigma^*$, the number of symbols conforming the word is called the *length* of σ and it is written as either $|\sigma|$ or $\text{len}(\sigma)$. The *empty word* $\lambda \in \Sigma^*$ is the unique word with zero length, $\text{len}(\lambda) = 0$.

In the *functional connotation*, an *algorithm* may be seen as a map $A : \Sigma^* \to \Sigma^*$, transforming a word $\sigma \in \Sigma^*$ either into a word $\tau \in \Sigma^*$ or producing no result. In the first case we write $\tau = A(\sigma)$, and it is said that τ is the *output* corresponding to the *input* σ. In the second case we write $A(\sigma) \uparrow$ and we say that the algorithm A *diverges* for the input σ. The *domain* of the algorithm A, $\text{dom}(A)$, is the collection of those words producing a definite output under A.

In the *procedural connotation*, an algorithm A determines a sequence of primitive actions for each input $\sigma \in \Sigma^*$. If $A(\sigma) \uparrow$ it may happen that either the corresponding

sequence of primitive actions is infinite or it is undefined (at some step, it is not defined the next step). If $\sigma \in \mathrm{dom}\,(A)$, let $t_A(\sigma)$ be the number of primitive actions performed by A from its starting at σ to its reaching of an *ending state*, producing thus the output $\tau = A(\sigma)$. The map $t_A : \Sigma^* \to \mathbb{N}$, $\sigma \mapsto t_A(\sigma)$, is called the *running time* function, and has the same domain as the algorithm A, $\mathrm{dom}\,(t_A) = \mathrm{dom}\,(A)$. On the other side, any device, or engine, running the algorithm A, initially shall occupy some memory locations in order to store the input $\sigma \in \Sigma^*$ and the current initial state, and thereafter, at each step it must occupy memory locations in order to store the current configuration, e.g. partial results, current instruction, pointers to necessary data in the computation, and extra control data. If $\sigma \in \mathrm{dom}\,(A)$, let $s_A(\sigma)$ be the maximum number of memory locations occupied by A at any computing step going from the starting at σ to an ending state producing the output $\tau = A(\sigma)$. The map $s_A : \Sigma^* \to \mathbb{N}$, $\sigma \mapsto s_A(\sigma)$, is called the *running space* function, and has the same domain as the algorithm A, $\mathrm{dom}\,(s_A) = \mathrm{dom}\,(A)$.

For any natural number let Σ^n be the collection of words over Σ of length n, $\Sigma^n = \{\sigma \in \Sigma^* | \, \mathrm{len}\,(\sigma) = n\}$.

For any algorithm A and for each non-negative integer $n \in \mathbb{N}$, let $\mathrm{dom}_n(A) = \Sigma^n \cap \mathrm{dom}\,(A)$ be the collection of input words of length n producing an output under A. Let us define the following maps:

Time. $T_A : \mathbb{N} \to \mathbb{N}$, $n \mapsto \max\{t_A(\sigma) | \, \sigma \in \mathrm{dom}_n(A)\}$.
Space. $S_A : \mathbb{N} \to \mathbb{N}$, $n \mapsto \max\{s_A(\sigma) | \, \sigma \in \mathrm{dom}_n(A)\}$.

Thus, $T_A(n)$ is the greatest number of steps that can be achieved by an input of length n within algorithm A. $S_A(n)$ is analogous with respect to space. Both maps are dependent on the algorithm A and on lengths of words, not just on words.

The maps T_A and S_A determine the *complexity measures* of algorithm A, the first gives the *time complexity* and the other the *space complexity*.

1.1.2 A particular example

Any non-negative integer number $x \in \mathbb{N}$ can be written in binary using $\ell(x) = \lceil \log_2(x+1) \rceil$ bits. $\ell(x)$ is the *length*, or the *size*, of integer x. Any integer may be represented by a string of length $\ell(x)+1$, where the extra bit can be considered as a *sign*: 0 for non-negative integers and 1 for negative integers. Explicitly this representation is

$$\rho_0 : x \mapsto \rho_0(x) = (|x|)_2 \star \left[\frac{1 - \mathrm{sgn}\,(x)}{2} \right].$$

For instance, if $x = 2009$, its base-2 representation is $(x)_2 = 11111011001$, thus $\rho_0(2009) = 111110110010$ while $\rho_0(-2009) = 111110110011$. Hence the size of an integer is proportional to the logarithm of its absolute value in base 2.

Now, let $\rho_1 : (0+1)^* \to (0+1)^*$ be the map that in each word changes each 1 by 10 and leaves each 0 as 0 itself. For instance,

$$\rho_1(\rho_0(2009)) = 10101010100101000100$$
$$\rho_1(\rho_0(-2009)) = 10101010100101000101 0$$

Clearly, for each integer $x \in \mathbb{Z}$, $\text{len}\,(\rho_1(\rho_0(x))) = (\ell(|x|)+1)+w(\rho_1(\rho_0(x)))$ where $w(\rho_1(\rho_0(x)))$ is the number of 1's in $\rho_1(\rho_0(x))$. Consequently, $\text{len}\,(\rho_1(\rho_0(x))) \leq 2(\ell(|x|)+1)$.

Let $\rho_2 : \mathbb{Z}^* \to (0+1)^*$ be the map that to each finite sequence of integers $x = (x_0,\ldots,x_{m-1})$ it associates the bit-string

$$\rho_2(x) = \rho_1(\rho_0(x_0)) \star 11 \star \rho_1(\rho_0(x_1)) \star 11 \star \cdots \star 11 \rho_1(\rho_0(x_{m-1})).$$

ρ_2 is an injective map. Clearly, $\forall x = (x_0,\ldots,x_{m-1}) \in: \mathbb{Z}^*$:

$$\begin{aligned}
\text{len}\,(\rho_2(x)) &= 2(m-1) + \sum_{j=0}^{m-1} \text{len}\,(\rho_1(\rho_0(x_j))) \\
&\leq 2(m-1) + m \max_{j\in[\![0,m-1]\!]} \text{len}\,(\rho_1(\rho_0(x_j))) \\
&\leq 2(m-1) + m\, 2\left(1 + \max_{j\in[\![0,m-1]\!]} \ell(|x_j|)\right) \\
&= 2m\left(2 + \max_{j\in[\![0,m-1]\!]} \ell(|x_j|)\right) - 2. \qquad (1.1)
\end{aligned}$$

Thus the length $\text{len}\,(\rho_2(x))$ can be upperly bounded by a polynomial expression depending on x, or rather on its dimension m and its entries x_j.

Alternatively we may fix a given size to represent positive integers and use a convention for negative integers (as two's complement). This has been, as is well known, the standard approach in industrial implementations, and can be checked in detail in several texts [Fernandez et al(2003)Fernandez, Garcia, and Garzon].

1.1.3 Estimating computational costs

Let us consider the sorting algorithm depicted at table 1.1.

If the input array A has dimension n, then the number of operations can be estimated as follows:

`InsertionSort`
Input. An integer array $A \in \mathbb{Z}^*$
Output. The same array ordered non-decreasingly

1. For each $j \in [\![2, \text{len}(A)]\!]$ Do
   a. $CurVal := A[j]$;
   b. $i := j-1$;
   c. While $(i > 0) \wedge (A[i] > CurVal)$ Do
      i. $A[i+1] := A[i]$;
      ii. $i--$
   d. $A(i+1) := CurVal$

Table 1.1 Algorithm `InsertionSort`.

Sta	Cost	Meaning	NR
1.	c_1	cost of control operations in this greater loop	n
1.a	c_2	assignment cost: done once at each greater iteration	$n-1$
1.b	c_3	assignment cost: done once at each greater iteration	$n-1$
1.c	c_4	cost of control operations in this lesser loop	$\sum_{j=2}^{n} j$
1.c.i	c_5	assignment cost: done once at each lesser iteration	$\sum_{j=2}^{n} (j-1)$
1.c.ii	c_6	assignment cost: done once at each lesser iteration	$\sum_{j=2}^{n} (j-1)$
1.d	c_7	assignment cost: done once at each greater iteration	$n-1$

(**Sta**: Statement **NR**: Number of repetitions)

We may assume $c_2 = c_3 = c_5 = c_6 = c_7$, since those values are just assignment costs to variables. Thus the full time cost is

$$T(n) = c_1 n + c_2 \left(3(n-1) + 2\sum_{j=2}^{n} (j-1) \right) + c_4 \sum_{j=2}^{n} j.$$

However, the exact cost depends on the input array.

Best case The minimum cost appears when the input array is already non-decreasingly ordered, because the lesser loop is skipped. In this case, $T_b(n) = c_1 n + 3c_2(n-1)$.

Worst case The maximum cost appears when the input array is given decreasingly ordered, because the lesser loop is always performed. In this case

$$\begin{aligned} T_w(n) &= c_1 n + c_2 \left(3(n-1) + 2\sum_{j=2}^{n} (j-1) \right) + c_4 \sum_{j=2}^{n} j \\ &= \left(c_2 + \frac{c_4}{2} \right) n^2 + \left(c_1 + 2c_2 + \frac{c_4}{2} \right) n - (3c_2 + c_4) \end{aligned}$$

Average or typical cases It is expected that the lesser loop is realized half the number of possibilities at each greater iteration. Hence,

$$\begin{aligned} T_a(n) &= c_1 n + c_2 \left(3(n-1) + 2\sum_{j=2}^{n} \frac{j-1}{2} \right) + c_4 \sum_{j=2}^{n} \frac{j}{2} \\ &= \frac{1}{2} \left(c_2 + \frac{c_4}{2} \right) n^2 + \left(c_1 + \frac{5c_2}{2} + \frac{c_4}{4} \right) n - (3c_2 + c_4) \end{aligned}$$

We see that the worst case and the averages cases produce a quadratic cost function. The ratio of them is such that $\lim_{n\to+\infty} \frac{T_a(n)}{T_w(n)} = \frac{1}{2}$, thus in the limit the worst case is twice as expensive as the average case. However $\lim_{n\to+\infty} \frac{T_b(n)}{T_a(n)} = 0$ thus in the limit the cost of the best case is negligible with respect to the average case.

The best cases of an algorithm correspond to lower computational costs but also to particular assumptions on the inputs that may be hard to be achieved. The worst cases correspond to higher computational costs and may also appear in extremely uncommon, bad luck indeed, cases. The average cases may computational costs lying between the worst and the best cases. For some algorithms these may be closer to the worst cases, as for the shown `InsertionSort`, but for other algorithms they can be closer to the best cases.

For practical purposes the estimation of time-complexities is useful in order to know the size of instances suitable to be treated in reasonable times. Let us consider the *nanosecond* (ns) as an unit of time (typically the time required to execute a primitive instruction in a physical conventional computer). Another time unit u is a multiple of ns, let $\text{len}(u)$ denote the length of u, as a multiple of ns, expressed in base 2. The following table summarizes the values of $\text{len}(u)$:

u	second sec	minute min	hour hr	day day	week wk	year yr	decade dec	century ct
$\text{len}(u)$	30	36	42	47	50	55	59	62

Let us consider, for instance, the following increasing functions:

$$\begin{array}{lll} f_1(n) = \log(n) & f_4(n) = n \cdot \log(n) & f_7(n) = n^{10} \\ f_2(n) = \sqrt{n} & f_5(n) = n^2 & f_8(n) = 2^n \\ f_3(n) = n & f_6(n) = n^3 & f_9(n) = n! \end{array}$$

f_3 is the identity map and it is, obviously, a linear map. The maps f_1 and f_2 are sublinear and the maps f_k, $k \in [\![4,9]\!]$, are superlinear. For each of these superlinear increasing maps, f_k, let D_k be the interval consisting of those values $n \in \mathbb{N}$ such that $f_k(n)$, measured in ns, lies between one second and one century. Then:

f_k	D_k
$f_4(n) = n \cdot \log(n)$	$4 \times 10^7 \leq n \leq 6 \times 10^{16}$
$f_5(n) = n^2$	$3 \times 10^4 \leq n \leq 2 \times 10^9$
$f_6(n) = n^3$	$1 \times 10^3 \leq n \leq 2 \times 10^6$
$f_7(n) = n^{10}$	$8 \leq n \leq 72$
$f_8(n) = 2^n$	$30 \leq n \leq 63$
$f_9(n) = n!$	$12 \leq n \leq 20$

Observe that the polynomial map f_7 initially restricts n to be very small compared with the initial value of the exponential map f_8. However at the end of the shown time scale, f_7 allows a greater value for n than f_8. This is just an illustration that although initially a polynomial map may seem to have a greater growth than an exponential map, eventually the exponential growth will win the race!

1.2 Asymptotic analysis

1.2.1 Growth orders

Let $f : \mathbb{N} \to \mathbb{N}$ be a map on the non-negative integers. Let us define the following function classes:

$$\begin{aligned} O(f) &= \{g : \mathbb{N} \to \mathbb{N} | \exists c > 0, n_0 \in \mathbb{N} : (n \geq n_0 \Rightarrow g(n) \leq c \cdot f(n))\}, \\ \Theta(f) &= \{g : \mathbb{N} \to \mathbb{N} | [g \in O(f)] \wedge [f \in O(g)]\}, \\ \Omega(f) &= \{g : \mathbb{N} \to \mathbb{N} | \exists c > 0, n_0 \in \mathbb{N} : (n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n))\} \end{aligned}$$

If $\mathscr{O}$ is any of the symbols O, Θ, Ω we will write $g(n) = \mathscr{O}(f(n))$ to mean $g \in \mathscr{O}(f)$.

Remark 1.1. For any two maps $f, g : \mathbb{N} \to \mathbb{N}$ the following conditions are direct:

1. $g(n) = O(f(n)) \Leftrightarrow f(n) = \Omega(g(n))$,
2. $g(n) = \Theta(f(n)) \Leftrightarrow [g(n) = O(f(n))] \wedge [g(n) = \Omega(f(n))]$.

(here, the symbol "$\wedge$" is read *and*).

Let us also define:

$$\begin{aligned} o(f) &= \{g : \mathbb{N} \to \mathbb{N} | \forall c > 0 \exists n_0 \in \mathbb{N} : (n \geq n_0 \Rightarrow g(n) \leq c \cdot f(n))\}, \\ \omega(f) &= \{g : \mathbb{N} \to \mathbb{N} | f \in o(g)\}, \end{aligned}$$

in other words,

$$g(n) = o(f(n)) \Leftrightarrow \lim_{n \to +\infty} \frac{g(n)}{f(n)} = 0,$$

in this case it is said that *the growth order of g is strictly lower than that of f*, and

$$g(n) = \omega(f(n)) \Leftrightarrow \lim_{n\to+\infty} \frac{g(n)}{f(n)} = +\infty,$$

and in this case it is said that *the growth order of g is strictly greater than that of f*.

Remark 1.2. The following conditions are immediate:

Transitivity: For any symbol $\mathscr{O} \in \{O, \Theta, \Omega, o, \omega\}$ the following implication holds:

$$g(n) = \mathscr{O}(f(n)) \wedge f(n) = \mathscr{O}(h(n)) \Rightarrow g(n) = \mathscr{O}(h(n)).$$

Reflexivity: For any symbol $\mathscr{O} \in \{O, \Theta, \Omega\}$ there holds the relation:

$$f(n) = \mathscr{O}(f(n)).$$

Simmetry: $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$.

Consequently, the relation $g(n) = \Theta(f(n))$ defines an *equivalence relation* in the space of functions. Each equivalence class is a *growth order*.

A map f is *monotone function* if it is either an *increasing function* or a *decreasing function*. It is increasing if $\forall n_0, n_1 : (n_1 < n_2 \Rightarrow f(n_1) < f(n_2))$.
It is decreasing if $\forall n_0, n_1 : (n_1 < n_2 \Rightarrow f(n_1) > f(n_2))$.
It is a *non-decreasing function* if $\forall n_0, n_1 : (n_1 \leq n_2 \Rightarrow f(n_1) \leq f(n_2))$.
It is a *non-increasing function* if $\forall n_0, n_1 : (n_1 \leq n_2 \Rightarrow f(n_1) \geq f(n_2))$.
Thus a constant function is non-decreasing and non-increasing.

1.2.2 Polynomials

A *one-variable polynomial* has the form $p(X) = \sum_{i=0}^{m} a_i X^i$. The integer m is called the *degree* of the polynomial, ∂p, and the real numbers $a_0, \ldots, a_m$ are the *coefficients* of the polynomial. The coefficient a_m corresponding to the power of the degree is called the *leading coefficient*.

A *several-variables polynomial* has the form $p(\mathbf{X}) = \sum_{\mathbf{i}\in A}^{\mathbf{m}} a_{\mathbf{i}} \mathbf{X}^{\mathbf{i}}$, where $A \subset \mathbb{N}^k$ is a finite set and if $\mathbf{X} = (X_1, \ldots, X_k)$ and $\mathbf{i} = (i_1, \ldots, i_k)$ then $\mathbf{X}^{\mathbf{i}} = \prod_{j=1}^{k} X_j^{i_j}$.

Here we will consider just one-variable polynomials. Those of degree 1 are called *linear polynomials*, those of degree 2 *quadratic polynomials* and those of degree 3 *cubic polynomials*. For any two polynomials $p_1(X), p_2(X)$:

$$p_1(n) = O(p_2(n)) \Leftrightarrow \partial p_1 \leq \partial p_2,$$
$$p_1(n) = \Theta(p_2(n)) \Leftrightarrow \partial p_1 = \partial p_2,$$
$$p_1(n) = o(p_2(n)) \Leftrightarrow \partial p_1 < \partial p_2.$$

Let us define the *class of functions polynomially bounded* as $n^{O(1)} = \bigcup_{m \geq 0} O(n^m)$.

1.2.3 Exponentials and logarithms

1.2.3.1 Exponentials

Let us recall that for any non-zero real number $a \neq 0$,

$$a^0 = 1 \qquad \& \qquad \forall n \in \mathbb{N} : a^{n+1} = a^n \cdot a.$$

For negative exponents, say $n_1 = -n$ with $n \in \mathbb{N}$, it is defined $a^{n_1} = \frac{1}{a^n}$. If the *base number* a is positive, for any rational number $\frac{p}{q}$, with $q > 0$, it is defined $a^{\frac{p}{q}} = x$ where x is the real positive number x such that $x^q = a^p$. By continuity, a^z is well defined for any $z \in \mathbb{R}$.

The following *power rules* hold:

$$a^n a^m = a^{n+m} \qquad \& \qquad (a^n)^m = a^{nm}. \tag{1.2}$$

Besides, for any non-negative base number a,

$$a^n \underset{n \to +\infty}{\longrightarrow} \begin{cases} 0 & \text{if } a < 1 \\ 1 & \text{if } a = 1 \\ \infty & \text{if } a > 1 \end{cases} \tag{1.3}$$

Conventional base numbers are, for instance, $a = 10$ or $a = 2$, but the most important base number in mathematics is e, the *base of natural logarithms*, which is defined, among a great variety of different ways, as

$$e = \lim_{n \to +\infty} \left(1 + \frac{1}{n}\right)^n \tag{1.4}$$

and an approximation to its value is $e \approx 2.7182818284590452354\ldots$ A well known series expansion of the *exponential function* is

$$\forall x \in \mathbb{R} : \; e^x = \sum_{m \geq 0} \frac{x^m}{m!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots \tag{1.5}$$

1.2.3.2 Logarithms

For any positive base number $a > 0$, the *logarithm in base a* of $x > 0$ is the real number $\log_a x = y$ such that $a^y = x$. In symbols:

$$\log_a x = y \;\Leftrightarrow\; a^y = x \tag{1.6}$$

Then, from the power rules (1.7),

$$\begin{array}{ll} \log_a(xy) & = \log_a x + \log_a y \\ \log_a(x^y) & = y \cdot \log_a x \\ \log_b x & = \frac{\log_a x}{\log_a b} \end{array} \tag{1.7}$$

The first equation in (1.7) asserts that, for each base $a > 0$, *the logarithm function is a homomorphism among the multiplicative structure of the real positive numbers and the additive structure of the whole real line*.

On the other hand, let us consider the "reciprocal map", $t \mapsto 1/t$, defined over the positive real numbers, and its integral,

$$F : x \mapsto \int_1^x \frac{dt}{t}.$$

Clearly, $\forall x, y \in \mathbb{R}^+$:

$$\begin{aligned} F(xy) &= \int_1^{xy} \frac{dt}{t} \\ &= \int_1^x \frac{dt}{t} + \int_x^{xy} \frac{dt}{t} \\ &= \int_1^x \frac{dt}{t} + \int_1^y \frac{d(xt_1)}{(xt_1)} \\ &= \int_1^x \frac{dt}{t} + \int_1^y \frac{dt}{t} \\ &= F(x) + F(y) \end{aligned}$$

hence, F is also a homomorphism among the multiplicative structure of the real positive numbers and the additive structure of the real line. Necessarily, there might be a base $e \in \mathbb{R}$ such that $F(x) = \log_e x$, and this is indeed the number defined at (1.4). The *natural logarithm* is the map $x \mapsto \ln x = \log_e x$.

For any positive base $a > 0$:

$$a^x = e^{\ln a^x} = e^{x \ln a} \tag{1.8}$$

and this, together with eq. (1.5) gives a procedure to compute a^x.

As another procedure for calculation, we recall that for any real number with absolute value lower than 1, $x \in I =]-1, 1[$:

$$\ln(1+x) = \sum_{m\geq 1} (-1)^{1+m} \frac{x^m}{m} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \frac{x^6}{6} \cdots . \tag{1.9}$$

Right member in last equation is known as the *Mercator series*.

For each positive real number $y \in \mathbb{R}^+$ let $n_y = \max\{m \in \mathbb{Z} | e^m \leq y\}$ be the *characteristic* of the natural logarithm of y. We may write $y = e^{n_y}(1 + (y\,e^{-n_y} - 1))$, thus

$$\ln y = n_x + \ln x_y = n_x + \sum_{m\geq 1} (-1)^{1+m} \frac{x_y^m}{m} \tag{1.10}$$

where $x_y = y\,e^{-n_y} - 1$, and $x_y \in I$. The value $\ln x_y$ is the *mantissa* of the natural logarithm of y.

The Mercator series has a convergence rate extremely low, thus in practice it is necessary to apply some transforms on it in order to speed-up convergence.

1.2.3.3 Comparison with polynomial maps

For any positive base number $a > 0$, and any positive integer m, from relations (1.8) and (1.5) we have

$$\lim_{n\to+\infty} \frac{n^m}{a^n} = 0 \tag{1.11}$$

hence $n^m = o(a^n)$. Thus, whenever $a > 0$ and $f(n) = n^{O(1)}$ necessarily $f(n) = o(a^n)$. Consequently, *any exponential function grows faster than any polynomially bounded map*.

Let $(\log_a n)^{O(1)} = \bigcup_{m\geq 0} O((\log_a n)^m)$ be the *class of functions bounded polylogarithmically*. In order to compare this class with the polynomials, let us write $n_1 = \log_a n$. Then, from relation (1.11),

$$\frac{(\log_a n)^m}{n} = \frac{(\log_a n)^m}{a^{\log_a n}} = \frac{n_1^m}{a^{n_1}} \underset{n\to+\infty}{\longrightarrow} 0.$$

Hence $(\log_a n)^m = o(n)$.

Consequently, *any polylogarithmic map grows slower than the identity map, hence slower than any other polynomial function*.

1.2.4 Factorial

The map $n \mapsto n!$ (factorial-n) is defined recursively:

$$0! = 1 \qquad \& \qquad \forall n \in \mathbb{N} : (n+1)! = n! \cdot (n+1).$$

As an approximation we have $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1+\Theta\left(\frac{1}{n}\right)\right)$ and in a more precise way,

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

Hence $n! = o(n^n)$, $n! = \omega(2^n)$, and $\log(n!) = \Theta(n\log n)$.

1.3 Sums

Finite sums If $A = (a_1, \ldots, a_n)$ is a finite array of numbers, its *sum* is

$$\sum_{i=1}^{n} a_i = a_1 + \cdots + a_n.$$

Series If $A = (a_n)_{n\geq 0}$ is a sequence of numbers, its *series* is

$$\sum_{n=1}^{+\infty} a_n = \sum_{n\geq 0} a_n = \lim_{n\to+\infty} \sum_{i=1}^{n} a_i.$$

Let us review some estimation criteria.

1.3.1 Telescopic sums

If $A = (a_0, a_1, \ldots, a_n)$ is a finite array of numbers, then $\sum_{i=1}^{n}(a_i - a_{i-1}) = a_n - a_0$ (the middle terms cancel each other). This is called a *telescopic sum*.

For instance:

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \sum_{i=1}^{n}\left(\frac{1}{i} - \frac{1}{i+1}\right) = 1 - \frac{1}{n}.$$

1.3.2 Sums of integer powers

Let $s_{mn} = \sum_{i=1}^{n} i^m$ be the addition of the m-powers of the first $(n+1)$ non-negative integers. From Newton's Binomial Formula, for each integer i,

$$(i+1)^m = \sum_{k=0}^{m} \binom{m}{k} i^k$$

thus

$$(i+1)^m - i^m = \sum_{k=0}^{m-1} \binom{m}{k} i^k \tag{1.12}$$

Summing-up these values gives a telescopic sum, hence

$$(n+1)^m - 1^m = \sum_{k=0}^{m-1} \binom{m}{k} s_{kn} \tag{1.13}$$

The following equation system results,

$$\begin{aligned}
(n+1)-1 &= s_{0n} \\
(n+1)^2-1 &= s_{0n}+2s_{1n} \\
(n+1)^3-1 &= s_{0n}+3s_{1n}+3s_{2n} \\
(n+1)^4-1 &= s_{0n}+4s_{1n}+6s_{2n}+4s_{3n} \\
(n+1)^5-1 &= s_{0n}+5s_{1n}+10s_{2n}+10s_{3n}+5s_{4n} \\
(n+1)^6-1 &= s_{0n}+6s_{1n}+15s_{2n}+20s_{3n}+15s_{4n}+6s_{5n} \\
\vdots &\quad\quad \vdots
\end{aligned}$$

and by solving for the first values of s_{mn} we get

$$\begin{aligned}
s_{0n} &= n \\
s_{1n} &= \frac{1}{2}n(1+n) = \frac{1}{2}s_{0n}(1+n) \\
s_{2n} &= \frac{1}{6}n(1+n)(1+2n) = \frac{1}{3}s_{1n}(1+2n) \\
s_{3n} &= \frac{1}{4}n^2(1+n)^2 = s_{1n}^2 \\
s_{4n} &= \frac{1}{30}n(1+n)(1+2n)(-1+3n+3n^2) = \frac{1}{5}s_{2n}(-1+3n+3n^2) \\
s_{5n} &= \frac{1}{12}n^2(1+n)^2(-1+2n+2n^2) = \frac{1}{3}s_{3n}(-1+2n+2n^2) \\
&\vdots \quad \vdots
\end{aligned}$$

Clearly, $\sum_{i=1}^{n} i^m = O\left(n^{m+1}\right)$.

1.3.3 Power sums

For each non-zero real number $x \neq 0$ and any integer $n \geq 0$,

$$x^{n+1} - 1 = \sum_{i=0}^{n} (x^{i+1} - x^i) = \sum_{i=0}^{n} x^i (x-1),$$

thus $\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x}$.

Consequently, whenever $|x| < 1$ we should have $\sum_{n\geq 0} x^n = \frac{1}{1-x}$.

By deriving each term and multiplying by x we obtain $\sum_{n\geq 0} nx^n = \frac{x}{(1-x)^2}$, clearly just for $|x| < 1$.

1.4 Empirical measurements of performance

Empirical analysis is exposed in greater detail at conventional books on Algorithms, e.g. [Wood(1993)], [Sedgewick and Flajolet(1996)] or [Cattaneo and Italiano(1999)].

In the formal analysis of algorithms several features are considered: correctness, time-efficiency, space-efficiency, clarity and optimality among others. However, on implementing them, several other features influence the performance of an algorithm, e.g. the platform and operating system, the compiler, the programming language and access time to storage devices, among many other "real world computing" features. An implementation of an algorithm may show discrepancies with respect to the expected performance due to a careless direct transcription of the algorithm without considering the implementation issues.

Although mostly the empirical analysis is put aside in courses of algorithms it is important to realize this analysis. It is required to have a clear understanding of the theoretical basis of the algorithm, to distinguish the performance parameters to be measured, to choose an appropriate hardware, to implement time measuring processes, to select non-biased input data in order to perform representative results, to interpret the results and to correlate them with the theoretical analysis.

For instance, consider `LinearSearch`: Given a numeric array A of length n and a query number x it produces as output the least index j such that $A[j] = x$, if x does indeed appear in A, or the value 0 otherwise. `LinearSearch` tests left-to-right whether each entry in A coincides with x, and it stops at the first successful test giving it as output, or outputs the value zero if all the attempts have failed.

If the array A have values uniformly distributed and x indeed appears at A then we may expect that in the average with $(n+1)/2$ tests `LinearSearch` will finish correctly. If p denotes the probability that x appears at A, then the expected number of tests is the weighted average:

$$T(n) = p\frac{n+1}{2} + (1-p)n = \left(1 - \frac{p}{2}\right)n + \frac{p}{2}. \tag{1.14}$$

An interesting exercise consists in implementing `LinearSearch` and to check whether the number of tests fits to the expected value given by (1.14).

Among the main tasks in implementation there are the following:

- Data structures and basic algorithms engineering

- Program design and program coding
- Algorithm verification
- Debugging
- Algorithm testing
- Performance analysis
- Performance tuning

In fact, empirical analysis is an initial form of *computer system performance evaluation*.

1.5 Time and space tradeoff

In general, running-time and storage-space are inversely proportional in algorithms: storage can be reduced at the cost of longer times, or equivalently, at the cost of producing slower programs. Conversely, quicker programs would cause greater storages. This phenomenon is known as *time-space tradeoff*.

For instance, frequently used computable values may be precalculated, then stored in a given space, called a *lookup table*, in order to be recalled each time they are needed. Since calculation times are avoided, programs may run faster but, obviously, storage is increased. Conversely, instead of storing values, they can be calculated each time they are required. Thus, storage will be reduced, but computing times will increase. The storage *cachés* are used typically as lookup tables and they are built in faster memory devices. Also, *hash tables* associate short identifications to information pieces to order to save time in accessing data. A hash table should be easily computable and it shall distribute uniformly the keys of information pieces.

It can be shown that sorting procedures with smallest storages, say proportional to the number n of elements to be sorted, may run in time proportional to $n \log n$, and although some procedures may run in linear time with n, they would require greater storages.

In Complexity Theory, time-space tradeoff is an area of intense research. Namely, the research in time-space tradeoff lower bounds seeks optimal programs implementing a given procedure.

Many algorithms require a *precomputing step* and a further *computation step* (for instance in accessing data into a database a precomputing step may consist in sorting the database, and the computing step in accessing to data through a logarithmic search). Such an algorithm split can be regarded as a space-for-time tradeoff.

1.6 Recurrence relations in recursive algorithms analysis

1.6.1 A divide and conquer algorithm

Divide_and_conquer (latin, *Divide_et_impera*) is a typical recursive methodology to deal with computational problems. An instance is divided into simpler instances, each of them is solved, and then all the solutions of the simpler instances are collected or merged in order to recover a solution of the original global problem.

For instance, `Merge-Sort` proceeds as follows:

Divide an array of dimension n into two $n/2$-dimensional subarrays.
Recursively sort each array by this procedure.
Conquer by merging the sorted subarrays.

The following pseudocode specifies in a more specific way the algorithm.

`MergeSort`
Input. An n-dimensional array A.
p, r extreme indexes of the A's segment to be sorted.
Output. The sorted segment $A[p \cdots r]$.

1. If $p < r$ Then
   Divide Step
   a. $q := \lceil \frac{p+q}{2} \rceil$;
   b. `MergeSort`(A, p, q) ;
   c. `MergeSort`$(A, q+1, r)$;
      Merge Step
   d. $Ptr_1 := p$; $Ptr_2 := q+1$; $Ptr_3 := 1$;
   e. While $Ptr_1 \leq q$ and $Ptr_2 \leq r$ Do
      i. If $A[Ptr_1] \leq A[Ptr_2]$ Then $\{ B[Ptr_3] := A[Ptr_1]$; Ptr_1++ $\}$
         Else $\{ B[Ptr_3] := A[Ptr_2]$; Ptr_2++ $\}$
      ii. Ptr_3++
   f. If $Ptr_1 > q$ Then $B := B \star A[Ptr_2 \cdots r]$
      Else $B := B \star A[Ptr_1 \cdots q]$;
   g. Let $A[p \cdots r] := B$

Let us denote by $T(n)$ the number of real number comparisons realized by `MergeSort` within an input of n values. Clearly,

$$T(n) = \begin{cases} 0 & \text{si } n = 1, \\ 2T\left(\frac{n}{2}\right) + n & \text{si } n > 1. \end{cases} \tag{1.15}$$

For any n enough big, the following equations hold:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right)+n \\ &= 2\left(2T\left(\frac{n}{2^2}\right)+\frac{n}{2}\right)+n \\ &= 2^2T\left(\frac{n}{2^2}\right)+2n \\ &= 2^2\left(2T\left(\frac{n}{2^3}\right)+\frac{n}{2^2}\right)+2n \\ &= 2^3T\left(\frac{n}{2^3}\right)+3n \\ &\vdots \\ &= 2^\kappa T\left(\frac{n}{2^\kappa}\right)+\kappa n \end{aligned}$$

Thus, if $n=2^k$ is a power of two, $k=\log_2 n$, then for $\kappa=k$,

$$T(n)=2^kT(1)+kn=2^k\cdot 0+kn=n\log_2 n,$$

and the solution of (1.15) is $T(n)=O(n\log n)$.

We see thus that recurrence equations as (1.15) are useful in the analysis and design of algorithms. In the current section we are going to sketch some solving methods.

1.6.2 Substitution method

Given a recurrence relation, one may guess a bounding function and then to prove the correctness of that bound.

For instance, let us consider:

$$T(n)=\begin{cases} n & \text{si } n=1, \\ 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+n & \text{si } n>1. \end{cases}$$

According to the above example, we may guess that the solution has order $O(n\log n)$. So let us prove that there is a constant c such that $T(n)\leq cn\log n$.

We may proceed by induction on n.

Let us assume as induction hypothesis: $T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)\leq c\left\lfloor\frac{n}{2}\right\rfloor\log\left\lfloor\frac{n}{2}\right\rfloor$. Then,

$$\begin{aligned} T(n) &\leq 2c\left\lfloor\frac{n}{2}\right\rfloor\log\left\lfloor\frac{n}{2}\right\rfloor+n \\ &\leq cn\log\frac{n}{2}+n \\ &\leq cn\log n-cn\log 2+n \\ &\leq cn\log n-cn+n \\ &\leq cn\log n \end{aligned}$$

whenever $c > 1$.

As a second example, let us consider

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1$$

and let us conjecture that its solution is $O(n)$. Let us prove that $T(n) \leq cn - b$ for some constants c, b.

As induction hypothesis let us assume $T(m) \leq cn - b$, for any $m < n$. Then

$$\begin{aligned} T(n) &\leq \left(c\left\lceil \frac{n}{2} \right\rceil - b\right) + \left(c\left\lfloor \frac{n}{2} \right\rfloor - b\right) + 1 \\ &\leq cn - 2b + 1 \\ &\leq cn - b \end{aligned}$$

whenever $c, b > 1$.

Sometimes it is convenient to do a *change of variables*.

For instance, let us consider $T(n) = 2T(\lceil \sqrt{n} \rceil) + \log_2 n$.

Let $m = \log_2 n$. Then $n = 2^m$ and the recurrence relation can be written as $T(2^m) = 2T\left(\left\lceil 2^{\frac{m}{2}} \right\rceil\right) + m$. Hence it has the form $S(m) = 2S\left(\left\lceil \frac{m}{2} \right\rceil\right) + m$, which has solution $S(m) = O(m \log m)$. Thus the solution of the original recurrence is $T(n) = O(\log n \log\log n)$.

1.6.3 Iteration method

Let us introduce this procedure following an example. Let

$$T(n) = 3T\left(\frac{n}{4}\right) + n.$$

By substituting consecutive expressions of the recurrence we get:

$$\begin{aligned}
T(n) &= 3T\left(\frac{n}{4}\right)+n \\
&= 3\left(3T\left(\frac{1}{4}\cdot\frac{n}{4}\right)+\frac{n}{4}\right)+n \\
&= 3^2T\left(\frac{n}{4^2}\right)+\left(\frac{3}{4}+1\right)n \\
&= 3^2\left(3T\left(\frac{1}{4}\cdot\frac{n}{4^2}\right)+\frac{n}{4^2}\right)+\left(\frac{3}{4}+1\right)n \\
&= 3^3T\left(\frac{n}{4^3}\right)+\left(\left(\frac{3}{4}\right)^2+\frac{3}{4}+1\right)n \\
&\vdots\ \vdots \\
&= 3^kT\left(\frac{n}{4^k}\right)+\left(\left(\frac{3}{4}\right)^{k-1}+\cdots+\frac{3}{4}+1\right)n \\
&\vdots\ \vdots \\
&\leq 3^{\log_4 n}\Theta(1)+\left(\sum_{k\geq 0}\left(\frac{3}{4}\right)^k\right)n
\end{aligned}$$

Since $3^{\log_4 n} = n^{\log_4 3}$ and $\sum_{k\geq 0}\left(\frac{3}{4}\right)^k = \frac{1}{1-\frac{3}{4}} = 4$, then

$$T(n) = \Theta(n^{\log_4 3})+4n = 4n+o(n) = O(n).$$

1.6.4 Master method

Theorem 1.1 (Master's –). *Let $a,b>0$ and $f:\mathbb{N}\to\mathbb{R}$ be a map. The solution of the recurrence*

$$T(n) = aT\left(\frac{n}{b}\right)+f(n)$$

satisfies the following relations which are dependent of f:

1. *If $f(n) = O\left(n^{\log_b a-\varepsilon}\right)$ for some $\varepsilon>0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.*
2. *If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a}\log n\right)$.*
3. *If $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ for some $\varepsilon>0$ and*

$$\exists c\in]0,1[, n_0\in\mathbb{N}:\ (n\geq n_0 \Rightarrow af\left(\frac{n}{b}\right)\leq cf(n)),$$

then $T(n) = \Theta(f(n))$.

In other words, the growth of the solution is determined by the comparison among f and $n^{\log_b a}$:

- If f "is below, a little bit more" than $n^{\log_b a}$, then the solution grows as $n^{\log_b a}$.
- If f "grows" as $n^{\log_b a}$, then the solution grows as $n^{\log_b a}\log n$.
- If f "is above, a little bit less" than $n^{\log_b a}$, then the solution grows as f.

The proof of the theorem is out of the scope of our presentation, it can be found in full detail in [Cormen et al(2003)Cormen, Leiserson, Rivest, and Stein].

Example 1.1. Let us consider $T(n) = 9T\left(\frac{n}{3}\right) + n$. Since $n^{\log_3 9} = n^2$, and obviously, $n = O\left(n^{\log_3 9 - \varepsilon}\right)$ whenever $\varepsilon \leq 1$. Thus by case 1. in the Master's Theorem $T(n) = \Theta\left(n^2\right)$.

Example 1.2. Let $T(n) = T\left(\frac{2}{3}n\right) + 1$. Since $n^{\log_{\frac{3}{2}} 1} = n^0 = 1$, and

$$f(n) = 1 = \Theta(1) = \Theta(n^{\log_{\frac{3}{2}} 1}),$$

by case 2. in the Master's Theorem $T(n) = \Theta(\log n)$.

Example 1.3. Let $T(n) = 3T\left(\frac{n}{4}\right) + n\log n$. Since $n^{\log_4 3} = O\left(n^{0.793}\right)$, and $f(n) = \Omega\left(n^{\log_4 3 + \varepsilon}\right)$, where ε is greater than 0.2, for $c = \frac{3}{4}$, we have $3 \cdot \left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \leq \frac{3}{4} \cdot n\log n$. Thus by case 3. in the Master's Theorem $T(n) = \Theta(n\log n)$.

Example 1.4. Let $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$. On one hand $n^{\log_2 2} = n$. But,

$$\frac{n\log n}{n^{\log_2 2}} = \log n = o(n^{\varepsilon}),$$

for any $\varepsilon > 0$. Thus there is no $\varepsilon > 0$ such that $f(n) = \Omega\left(n^{\log_2 2 + \varepsilon}\right)$. Consequently, no consequence can be drawn from the case 3. in the Master's Theorem.

There are other solving methods as the *method of generator maps*, and the *method of powers*.

Problems

1.1. Design an algorithm of complexity $O(n)$ to evaluate polynomials of degree n.

Hint. $\sum_{i=0}^{n} a_i x^i = x\sum_{i=1}^{n} a_i x^{i-1} + a_0$.

1.2. Which is the greatest value of $n > 1$ such that any algorithm with running time $8n^2$ runs slower than an algorithm with time 2^n in the same computer?

1.3. Which is the greatest value of $n > 1$ such that any algorithm with running time 2^n runs faster than an algorithm with time $100n^2$ in the same computer?

In the following exercises, for any two maps $g_1, g_2 : \mathbb{N} \to \mathbb{N}$ let us write $g_1 \preceq g_2$ (g_2 *eventually uppers* g_1) whenever $g_1 = O(g_2)$.

1.4. Compare the following two maps:

$$(\log_2 n)! \quad , \quad (\log_2 n)^2$$

1.5. Compare the following two maps:

$$\left(\sqrt{2}\right)^{\log_2 n} \quad , \quad 2^{2^n}$$

1.6. Compare the following two maps:

$$2^{2^n} \quad , \quad 2^{\log_2 n}$$

1.7. Compare the following two maps:

$$2^{\log_2 n} \quad , \quad 2^{\sqrt{2\log_2 n}}$$

1.8. Compare the following two maps:

$$n^2 \quad , \quad n!$$

1.9. Compare the following two maps:

$$n! \quad , \quad \left(\tfrac{3}{2}\right)^n$$

1.10. Compare the following two maps:

$$\left(\tfrac{3}{2}\right)^n \quad , \quad n^{\frac{1}{\log_2 n}}$$

1.11. Compare the following two maps:

$$n^{\frac{1}{\log_2 n}} \quad , \quad n2^n$$

1.12. Compare the following two maps:

$$n^3 \quad , \quad (\log_2 n!)$$

1.13. Compare the following two maps:

$$(\log_2 n!) \quad , \quad \ln\ln n$$

1.14. Compare the following two maps:

$$\ln\ln n \quad , \quad \ln n$$

1.15. Compare the following two maps:

$$\ln n \quad , \quad n\ln n$$

1.16. Compare the following two maps:

$$n^{\log_2 \log_2 n} \quad , \quad 1$$

1.17. Compare the following two maps:

$$1 \quad , \quad 2^{\log_2 n}$$

1.18. Compare the following two maps:

$$2^{\log_2 n} \quad , \quad \sqrt{\log_2 n}$$

1.19. Compare the following two maps:

$$\sqrt{\log_2 n} \quad , \quad 2^n$$

1.20. Compare the following two maps:

$$2^{2^{n+1}} \quad , \quad (n+1)!$$

1.21. Compare the following two maps:

$$(n+1)! \quad , \quad 4^{\log_2 n}$$

1.22. Compare the following two maps:

$$4^{\log_2 n} \quad , \quad e^n$$

1.23. Compare the following two maps:

$$e^n \quad , \quad (\log_2 n)^{\log_2 n}$$

1.24. Show that for any two constants $a, b > 0$ we have $(n+a)^b = \Theta(n^b)$.

1.25. Express the sum $\sum_{i=1}^{n}(2i-1)$ as a "single" formula.

1.26. Show that the solution of $T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1$ is $O(\log n)$.

1.27. Find tight bounds (upper and lower) to the recurrence $T(n) = 3T(n/3+5) + n/2$.

1.28. Find tight bounds (upper and lower) to the recurrence $T(n) = T(n-1) + 1/n$.

1.29. Calculate the sum $\displaystyle\sum_{i=1}^{n} i\binom{n}{i}$.

1.30. Calculate the sum $\displaystyle\sum_{i=1}^{n} \binom{n}{i}$.

1.31. Calculate the sum $\sum_{i=1}^{n} ia^i$.

1.32. Solve the recurrence

$$\begin{aligned} T(0) &= c \\ T(n) &= aT(n-1)+bn. \end{aligned}$$

1.33. Solve the recurrence

$$\begin{aligned} T(0) &= c \\ T(n) &= aT(n-1)+bn^k. \end{aligned}$$

1.34. Solve the recurrence

$$\begin{aligned} T(0) &= c \\ T(n) &= T\left(\frac{n}{2}\right)+bn\log n. \end{aligned}$$

1.35. Write a program to approximate e according to relation (1.4).

1.36. Write a program to approximate e^x according to relation (1.5).

1.37. Write a program to approximate a^x according to relation (1.8).

1.38. Write a program to approximate $\ln(1+x)$ according to relation (1.9).

1.39. Let the integers p_n, q_n be defined such that

$$\frac{p_n}{q_n} = \sum_{k=1}^{n} \frac{1}{k},$$

without common divisors among them. Find recursive relations to calculate the sequences of numbers $(p_n)_{n\geq 1}$ and $(q_n)_{n\geq 1}$.

1.40. Write a program to calculate the sequences $(p_n)_{n\geq 1}$ and $(q_n)_{n\geq 1}$ introduced above. Decide whether the *harmonic series* $\sum_{k\geq 1} \frac{1}{k}$ is convergent.

1.41. Design an algorithm to check experimentally the equation

$$\sum_{i=1}^{n} \frac{1}{i} = \log(n)+O(1).$$

1.42. Design an algorithm to check experimentally the equation

$$\sum_{i=1}^{n} \frac{1}{i^k} = k+(1-k)\frac{1}{n^{k-1}}+O(1).$$

Chapter 2
Algorithmic Strategies

Abstract We present here general strategies for algorithms design with representative examples. The brute-force algorithms are variants of exhaustive search for effective solutions and they depend on the enumeration processes of the search space. Greedy algorithms tend to build a solution based on immediate gains of some utility function. Divide-and-conquer are typical for algorithm optimization, and they are intended to reduce time complexities of algorithms. Backtracking is a search procedure based on a stacking device and depth-first traversing of tree structured search spaces. Heuristics is at present time an important area of development in Computer Engineering. The algorithms on syntactical pattern matching are illustrative of some procedures to reduce times on query processing through a preprocessing. Finally, in numerical approximation although there are no exact solutions they provide effectively and efficiently approximations to the solutions with different levels of closeness.

2.1 Brute-force algorithms

In section 1.1.1 we have seen several connotations of *algorithm*. Also, the notion of *problem* has several connotations.

A *decision problem* can be formally stated as a set of words Σ^* and a partition $\{Y,N,U\}$ of Σ^* into three sets: the *Yes-answered instances*, the *No-answered instances*, and the *Undefined-answer instances*. And for a given *instance* $x \in \Sigma^*$ the *goal* of the decision problem is to decide whether $x \in Y$ or $x \in N$ or $x \in U$.

A *search problem* instead can be seen as a set $P \in \Sigma^* \times \Sigma^*$. If $(x,y) \in P$ then it is said that y is a *solution* for the *instance* x. For any instance $x \in \Sigma^*$ let $P(x) = \{y \in \Sigma^* | (x,y) \in P\}$ be the set of solutions corresponding to x. The goal of a search problem is that, given an instance $x \in \Sigma^*$ it should find $y \in \Sigma^*$ such that $y \in P(x)$, if such a y does exist, or recognize that $P(x)$ is empty, otherwise.

Example 2.1 (Factorization). Given an integer number it is required to obtain the integers dividing that number.

As a decision problem, let $Y = \{(x,y) \in \mathbb{N}^2 | x \neq 0 \ \& \ \exists z \in \mathbb{N} : \ yz = x\}$ be the set of Yes-answers, let the undefined-answer set consists of pairs in which the dividend is zero $U = \{(0,y) | y \in \mathbb{N}\}$ and let $N = \mathbb{N}^2 - (Y \cup U)$ be the set of No-answers.

As a search problem, for any instance $x \in \mathbb{N}$, $x \neq 0$, a solution is a divisor of x, i.e. an integer $y \in \mathbb{N}$ such that $\exists z \in \mathbb{N} : \ yz = x$.

Example 2.2 (Map evaluation). Given a map $f : A \to A$, where A is a non-empty set, and a point $x \in A$, the value $y = f(x)$ shall be produced.

As a decision problem, the graph $Y = \{(x,y) \in A^2 | \ f(x) \downarrow \ \& \ y = f(x)\}$ is the set of Yes-answered instances, the undefined instances are in the set $U = \{(x,y) \in A^2 | \ f(x) \uparrow\}$ and $N = A^2 - (Y \cup U)$ is the set of No-answers.

As a search problem, for any instance $x \in A$, a solution is $y \in A$ such that $y = f(x)$ if $f(x) \downarrow$. Otherwise, no solution exists.

A *brute-force algorithm* tries to solve a search problem in a characteristic way: Given an instance $x \in \Sigma^*$, following an effective enumeration of the (image) space Σ^* it tests each element $y \in \Sigma^*$ as a solution corresponding to x and it stops the first time it gets a successful value.

Since the enumeration of the search space is relevant, these algorithms are also called *generate-and-test algorithms*.

A brute-force algorithm may be easy to implement and it is effective because through an exhaustive search, whenever there is a solution it will find it. The time complexity of the algorithm is proportional to the size of the search space. Consequently an approach to optimize the algorithm is to narrow or prune the search space in order to avoid the unfeasible part of that space.

Brute-force algorithms are suitable for problems in which there is no criteria to guide the search of solutions, or when there is no any solving heuristic. Indeed the brute-force strategy maybe considered as the simplest meta-heuristic to solve the problem. Brute-force algorithms are plausible only for problems with small search spaces.

Example 2.3 (Factorization). Let us continue the example 2.1. Given a positive integer $x \in \mathbb{N}$, which in base 2 is written with, say k bits, according to the Prime Number Theorem the number of primes less than x is approximately

$$\frac{x}{\ln x} \sim \frac{2^k}{k} = O(2^k). \tag{2.1}$$

Thus a naive brute-force algorithm has exponential time complexity.

However, if x is has a proper factor then it has a prime factor less than $\sqrt{x}$ which can be written with $\frac{k}{2}$ bits. The number of such prospective primes is approximately

$$\frac{2^{\frac{k}{2}}}{\frac{k}{2}} = \frac{2^{\frac{k+2}{2}}}{k}. \tag{2.2}$$

Although it also is determining an exponential time complexity, the gained speed-up is important because the ratio of relations (2.1) and (2.2) is $2^{-\frac{k-1}{2}}$ and this value tends to 0 as k grows.

Example 2.4 (Eight queens puzzle). Let us consider the following puzzle: *Locate 8 queens in a chessboard in such a way that no queen is attacking any other queen.*

Let us assume first that the queens are enumerated as $q_1, \ldots, q_8$. Then the number of ways to locate these queens in the 64-cells chessboard is

$$c_0 = 64 \cdot 63 \cdot \cdots \cdot 58 \cdot 57 = \frac{64!}{56!} = 178,462,987,637,760$$

However, if we make abstraction of the queens numbering, then the number of ways to select 8 cells in order to put in there the queens is

$$c_1 = \binom{64}{8} = \frac{c_0}{8!} = 4,426,165,368$$

But, since the queens cannot attack each other, at each column j in the chessboard there must be a queen in a row, say i_j, and no two such row values can coincide. Thus the map $\pi : j \mapsto i_j$ is a permutation $[\![1,8]\!] \to [\![1,8]\!]$. Namely, the search space can be restricted to the space of permutations of 8 elements, and its cardinality is

$$c_2 = 8! = 40,320$$

The brute-force algorithm based on this last search space consists in an enumeration of the permutations $[\![1,8]\!] \to [\![1,8]\!]$.

2.2 Greedy algorithms

These algorithms deal mostly with optimization problems: Given a real function defined in a domain $f : D \to \mathbb{R}$ it is required to calculate an extreme value $x_0 \in D$ such that $f(x_0) \leq f(x)$, $\forall x \in D$, if it is a *minimization problem* or $f(x_0) \geq f(x)$, $\forall x \in D$, if it is a *maximization problem*. A *greedy algorithm* proceeds by local approximations: at any current point, it moves in the direction pointing to the greatest gain in optimizing the objective function. The main disadvantage with this approach is that it does not always succeed in its goal: the procedure may fall into a local extreme point rather than in a global extreme point. However, for certain settings in the problem, greedy algorithms may be effective, and most probably, also efficient.

2.2.1 Activities-selection problem

Let us assume a set of *activities* $\mathscr{A} = \{A_i\}_{i\in I}$,each with a *duration interval* $[b_i, e_i[$, $c_i < a_i$, where b_i is the *beginning time* and e_i is its *ending time*, and let us assume that there is just one server, or processor, able to realize the activities, one at any time. A set of activities $\mathscr{A}_J = \{A_i\}_{i\in J\subset I}$ is *consistent* if

$$\forall j_1, j_2 \in J:\ (j_1 \neq j_2\ \Rightarrow\ [b_{j_1}, e_{j_1}[\cap[b_{j_2}, e_{j_2}[= \emptyset).$$

A consistent set $\mathscr{A}_{J_M}$ is *maximal* if

$$\forall \mathscr{B} : [\mathscr{A}_{J_M} \subseteq \mathscr{B} \subseteq \mathscr{A}\ \&\ \mathscr{A}_{J_M} \neq \mathscr{B}\ \Longrightarrow\ \mathscr{B} \text{ is inconsistent.}].$$

Problem `Activities_selection`.
Instance: A set $\mathscr{A} = \{A_i = [b_i, e_i[\,|\,1 \leq i \leq n\}$ of n activities.
Solution: A maximal consistent set $\mathscr{A}_{J_M} \subseteq \mathscr{A}_{J_M}$.

A "natural" greedy algorithm proceeds as follows:

`Greedy_activities_selection`
Input. A set $\mathscr{A} = \{A_i = [b_i, e_i[\,|\,1 \leq i \leq n\}$ of n activities.
Output. A maximal consistent set $\mathscr{A}_{J_M} \subseteq \mathscr{A}_{J_M}$.

1. Sort the activities in increasing order according to the ending times $(e_i)_{1\leq i\leq n}$.
2. Let the first activity be the current activity.
3. Store the current activity in the output set.
4. While there is an activity whose beginning time is greater than the ending time of the current activity do
   a. Choose as next activity the first activity according to the test.
   b. Update the current activity as the next activity.
   c. Store the current activity in the output set.
5. Output the built collection.

Remark 2.1. The following assertions hold:

1. There is a maximal consistent set whose first activity is the first activity according to ending times.
2. The problem `Activities_selection` is indeed solved by the algorithm `Greedy_activities_selection`.
3. The time complexity of `Greedy_activities_selection` is $\Theta(n)$ if the input array is given already sorted, otherwise the time complexity is $\Theta(n\log n)$.

Proof. If $\mathscr{A}_{J_M}$ is maximal and its first activity A_{M1} is not the first activity A_1 of the whole collection, then we may substitute A_{M1} by A_1 within $\mathscr{A}_{J_M}$, and this substitution produces a maximal consistent set as claimed. □

2.2.2 Knapsack problems

In these problems there is a collection of items, each having a volume and producing an utility, and there is also a knapsack with a fixed capacity. The goal of the problems is to put into the knapsack a collection of items in such a way that the knapsack's capacity is not exceeded by the sum of the put item volumes and the total utility of the put items is maximal. In (0-1)-*knapsack*, or *integer knapsack*, it is allowed just to put a given item entirely, while in *fractional knapsack* it is allowed to put fractions of items into the knapsack, with proportional volume and utility.

Problem `(0-1)-Knapsack`.
Instance: A collection $\mathscr{C} = \{C_i = (t_i, u_i) | 1 \leq i \leq n\}$ of n items (as pairs volume-utility) and a total capacity T.
Solution: A subset $I \subset [\![1,n]\!]$ such that $\sum_{i\in I} t_i \leq T$ and $\sum_{i\in I} u_i$ is maximal.
Problem `Fractional_Knapsack`.
Instance: A collection $\mathscr{C} = \{C_i = (t_i, u_i) | 1 \leq i \leq n\}$ of n items (as pairs volume-utility) and a total capacity T.
Solution: A subset $I \subset [\![1,n]\!]$ and a sequence of fractions $Q = (q_i)_{i\in I} \subset \mathbb{Q}^+$ such that $\sum_{i\in I} q_i t_i \leq T$ and $\sum_{i\in I} q_i u_i$ is maximal.

This last problem is suitable to be treated by a greedy algorithm. Namely:

`Greedy_Fractional_Knapsack`
Input. A collection $\mathscr{C} = \{C_i = (t_i, u_i) | 1 \leq i \leq n\}$ of n items (as pairs volume-utility) and a total capacity T.
Output. A subset $I \subset [\![1,n]\!]$ and a sequence of fractions $Q = (q_i)_{i\in I} \subset \mathbb{Q}^+$ such that $\sum_{i\in I} q_i t_i \leq T$ and $\sum_{i\in I} q_i u_i$ is maximal.

1. Let $R := \left(\frac{u_i}{t_i}\right)_{1\leq i\leq n}$ be the sequence of utility–volume ratios.
2. Sort $\mathscr{C}$ non-increasingly with respect to R.
3. $I := \emptyset$; $Q := \emptyset$;
4. $T' := T$; $U := 0$; $i := 1$;
5. While $T' > 0$ and $i \leq n$ Do
   a. $q_i = \min\left\{\frac{T'}{t_i}, 1\right\}$;
   b. $I := I \cup \{i\}$; $Q := Q \cup \{q_i\}$;
   c. $T' := T' - q_i t_i$; $U := U + q_i u_i$
6. Output (U, I, Q).

2.3 Divide-and-conquer

Given a problem, say $P(x)$, with input size $s(x) = n$, it is split as a smaller problems, $P_0(x_0), \ldots, P_{a-1}(x_{a-1})$, with instances of *reduction ratio* b, $s(x_j) = \frac{n}{b}$, with $a, b \in \mathbb{Z}^+$, producing corresponding solutions $y_0, \ldots, y_{a-1}$. Then they are merged in order

to obtain $y = M(y_0, \ldots, y_{a-1})$ as a solution of the original problem $P(x)$. In fig. 2.1 we sketch this procedure with $a = b = 2$. In section 1.6.1 we have already given an

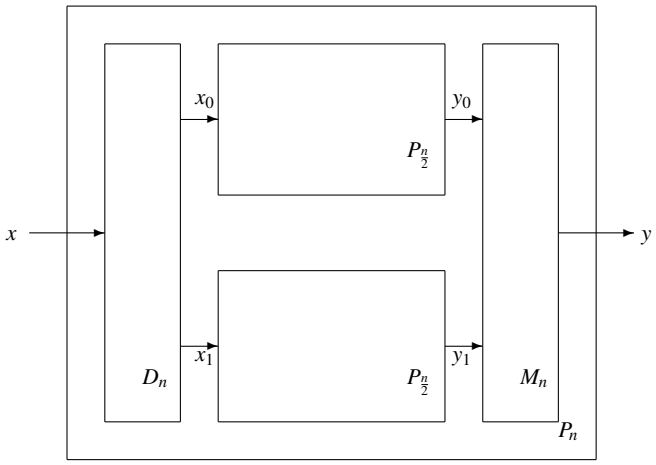


Fig. 2.1 Divide-and-conquer scheme: D_n is an instance divider or splitter, P_k is a solver for instances of size k, and M_n is a merger of solutions of size n instances.

example of a divide-and-conquer algorithm.

If $T(n)$ is a complexity measure of P for instances of size n, e.g. the number of primitive function calculations, then, we get

$$\forall n:\ T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + (d(n) + m(n)) & \text{if } n > 0 \\ t_0 & \text{if } n = 0 \end{cases} \tag{2.3}$$

where $d(n), m(n)$ are the respective measures of the divider D_n and the merger M_n. Either the substitution method or the Master's Theorem 1.1 gives a solution of the recurrence (2.3).

For instance, for $a = b = 2$, whenever $d(n) + m(n) = \Theta(n)$, the solution of the recurrence (2.3) is $T(n) = O(n \log n)$. Hence *the complexity of a divide-and-conquer algorithm is, for $a = b = 2$, $n \log n$*, thus it is superlinear but subquadratic (in fact it is closer to a linear behavior: $n \log n - n = o(n^2 - n \log n)$).

We will illustrate this algorithmic paradigm through several typical examples.

2.3.1 Raising to an integer power

Given a positive base $x > 0$, it is required to compute the map $\mathrm{Pow} : \mathbb{N} \to \mathbb{R}$, $n \mapsto x^n$. Clearly,

$$\forall n,m:\ \begin{cases} n=2m & \Longrightarrow \mathrm{Pow}(x,n)=\mathrm{Pow}(x,m)\cdot\mathrm{Pow}(x,m) \\ n=2m+1 & \Longrightarrow \mathrm{Pow}(x,n)=\mathrm{Pow}(x,m)\cdot\mathrm{Pow}(x,m)\cdot x \end{cases}$$

and this gives an immediate divide-and-conquer algorithm to compute Pow.

2.3.2 Integer multiplication

Given two n-bit integers $x=\sum_{j=0}^{n-1}2^j x_j$ and $y=\sum_{j=0}^{n-1}2^j y_j$ it is required to compute the product $z=x\cdot y$.

Formally, we have

$$z=\sum_{k=0}^{2n-2}2^k\left(\sum_{i+j=k}x_i y_j\right), \tag{2.4}$$

and due to bit-carry the result may consist of $2n$ bits. The whole evaluation of expression (2.4) entails $O(n^2)$ bit-products.

Alternatively, let us write

$$x_{low}=\sum_{j=0}^{\frac{n}{2}-1}2^j x_j\ \ ;\ \ x_{high}=\sum_{j=0}^{\frac{n}{2}-1}2^j x_{\frac{n}{2}+j}\ \ ;\ \ y_{low}=\sum_{j=0}^{\frac{n}{2}-1}2^j y_j\ \ ;\ \ y_{high}=\sum_{j=0}^{\frac{n}{2}-1}2^j y_{\frac{n}{2}+j}$$

Then, $x=2^{\frac{n}{2}}x_{high}+x_{low}$ and $y=2^{\frac{n}{2}}y_{high}+y_{low}$. Consequently

$$z=2^n x_{high}\cdot y_{high}+2^{\frac{n}{2}}\left(x_{low}\cdot y_{high}+x_{high}\cdot y_{low}\right)+x_{low}\cdot y_{low}.$$

On the other hand,

$$(x_{high}+x_{low})\cdot(y_{high}+y_{low})=x_{high}\cdot y_{high}+\left(x_{low}\cdot y_{high}+x_{high}\cdot y_{low}\right)+x_{low}\cdot y_{low}.$$

Thus,

$$z=2^n z_2+2^{\frac{n}{2}}(z_3-z_1-z_2)+z_1 \tag{2.5}$$

where

$$z_1=x_{low}\cdot y_{low}\ \ ;\ \ z_2=x_{high}\cdot y_{high}\ \ ;\ \ z_3=(x_{high}+x_{low})\cdot(y_{high}+y_{low}).$$

The relation (2.5) is the basis of a divide-and-conquer procedure.

Let $K(n)$ denote the number of bit-products required by this procedure. Then, the recurrence (2.3) is reduced in this case to the following:

$$\forall n:\ K(n)=\begin{cases} 3K\left(\frac{n}{2}\right) & \text{if } n>0 \\ 1 & \text{if } n=1 \end{cases} \tag{2.6}$$

By the Master's Theorem 1.1 we see that $K(n)=O(n^{\log_2 3})=O(n^{1.58496\cdots})$.

The outlined procedure is called *Karatsouba's multiplication method*.

2.3.3 Closest pair

Given a finite set $X = ((x_i, y_i))_{i=1}^n$ of points in the real plane, it is asked to find the pair $\{(x_{i_0}, y_{i_0}), (x_{i_1}, y_{i_1})\}$ of points in X realizing the minimum distance.

A brute-force approach would check all pairs, hence it has complexity $O(n^2)$.

A divide-and-conquer algorithm proceeds as follows:

1. Sort X according to the abscissae: $i < j \Rightarrow x_i \leq x_j$.
2. Divide X into two sets X_{left}, X_{right}, of almost equal cardinality, such that the abscissae of points in X_{left} are all not greater than the abscissa of any point in X_{right}.
3. Find the pairs $\{(x_{j_0}, y_{j_0}), (x_{j_1}, y_{j_1})\}$ and $\{(x_{k_0}, y_{k_0}), (x_{k_1}, y_{k_1})\}$ realizing respectively, the minimum distances d_{left} and d_{right} in X_{left} and X_{right}.
4. The solution $\{(x_{i_0}, y_{i_0}), (x_{i_1}, y_{i_1})\}$ of the original problem P may have either its both points at X_{left}, or its both points at X_{right} or have its points in different sets. In the first two possibilities, the solution shall coincide with the solution of the set giving the least distance. In order to deal with last situation:
   a. Let $d = \min\{d_{left}, d_{right}\}$ and let $\ell : x = x_\lambda$ be a boundary line between X_{left} and X_{right}.
   b. Let $L = \{(x, y) \in X | \ |x - x_\lambda| \leq d\}$ be the collection of points whose distance to ℓ is bounded by d.
   c. Sort L according to the y-coordinate.
   d. For each point in L check whether it has a neighbor closer than d.
5. Output the found closest pair.

In an implementation, the first sorting can be made as a preprocessing step, hence it can be assumed that the whole list X is already sorted.

2.3.4 Convex hull

Given a finite set $X = ((x_i, y_i))_{i=1}^n$ of points in the real plane, it is asked to find the *convex hull* of X.

The *convex hull* of a set X is the smallest convex set containing X, it is indeed the smallest convex polygon having as vertices some points of X and covering the whole set X.

A divide-and-conquer algorithm proceeds as follows:

1. Sort X according to the abscissae: $i < j \Rightarrow x_i \leq x_j$.
2. Divide X into two sets X_{left}, X_{right}, of almost equal cardinality, such that the abscissae of points in X_{left} are all not greater than the abscissa of any point in X_{right}.
3. Find the respective convex hulls Y_{left}, Y_{right} of X_{left}, X_{right}.

4. Let T_m be a common tangent line of Y_{left} and Y_{right} such that both of them lie completely at the same side of T_m. Let (x_{lm}, y_{lm}) and (x_{rm}, y_{rm}) be the vertices at Y_{left} and Y_{right} passing through T_m.
5. Let T_M be a common tangent line of Y_{left} and Y_{right} such that both of them lie completely at the same side of T_M, opposite side of T_m. Let (x_{lM}, y_{lM}) and (x_{rM}, y_{rM}) be the vertices at Y_{left} and Y_{right} passing through T_M.
6. The points (x_{lm}, y_{lm}) and (x_{lM}, y_{lM}) split Y_{left} into two segments, in a clockwise traversing, $[(x_{lm}, y_{lm}), (x_{lM}, y_{lM})]$ and $[(x_{lM}, y_{lM}), (x_{lm}, y_{lm})]$, this last one is closer to Y_{right}.
7. Similarly, the points (x_{rm}, y_{rm}) and (x_{rM}, y_{rM}) split Y_{right} into two segments, in a clockwise traversing, $[(x_{rm}, y_{rm}), (x_{rM}, y_{rM})]$ and $[(x_{rM}, y_{rM}), (x_{lm}, y_{lm})]$, the first one is closer to Y_{left}.
8. Suppress $[(x_{lM}, y_{lM}), (x_{lm}, y_{lm})]$ and add the edge $[(x_{lM}, y_{lM}), (x_{rM}, y_{rM})]$. Suppress $[(x_{rm}, y_{rm}), (x_{rM}, y_{rM})]$ and add the edge $[(x_{rm}, y_{rm}), (x_{lm}, y_{lm})]$.
9. Output the resulting polygon.

The search of the tangents T_m and T_M can be done by clockwise traversing the vertexes of Y_{left} and by counterclockwise traversing the vertexes of Y_{right}.

2.3.5 Strassen method for matrix multiplication

Given two matrices $A = [a_{ij}]_{1 \le i,j \le n}$, $B = [b_{ij}]_{1 \le i,j \le n}$ it is required to calculate the matrix product $C = A \cdot B$.

By definition we have

$$\forall i, j \in [\![1, n]\!] : \; c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Let $T_d(n)$ be the number of products necessary to compute the whole matrix multiplication. Since for each entry there are involved n products, we have $T_d(n) = n^3$.

The *Strassen method* reduces the number of multiplications. Let us partition the factor matrices as

$$A = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \quad \& \quad B = \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right].$$

Then the matrix product is

$$C = \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right]$$

and it can be calculated according to the following scheme:

$$\begin{array}{l|l} P_1 = (A_{11}+A_{22})(B_{11}+B_{22}) & \\ P_2 = (A_{21}+A_{22})B_{11} & \\ P_3 = A_{11}(B_{12}-B_{22}) & C_{11} = P_1+P_4-P_5+P_7 \\ P_4 = A_{22}(B_{21}-B_{11}) & C_{12} = P_3+P_5 \\ P_5 = (A_{11}+A_{12})B_{22} & C_{21} = P_2+P_4 \\ P_6 = (A_{21}-A_{11})(B_{11}+B_{12}) & C_{22} = P_1+P_3-P_2+P_6 \\ P_7 = (A_{12}-A_{22})(B_{21}+B_{22}) & \end{array} \tag{2.7}$$

The relations (2.7) entail a divide-and-conquer algorithm: The current problem is split into 7 subproblems of half size, thus for $a = 7$ and $b = 2$, the recurrence relation (2.3) becomes, for $T_s(n)$ denoting the number of products necessary to compute the whole matrix multiplication by relations (2.7):

$$\forall n:\ T_s(n) = \begin{cases} 7T_s\left(\frac{n}{2}\right) & \text{if } n > 0 \\ 1 & \text{if } n = 1 \end{cases}$$

By the Master's Theorem 1.1 we see that $T_s(n) = O(n^{\log_2 7}) = O(n^{2.80735\cdots})$.

Consequently, $T_s(n) = o(T_d(n))$. Indeed, for $n \sim 36$, the Strassen method will realize half the number of products than the matrix multiplication by the sole definition.

2.4 Backtracking

2.4.1 A general approach

Let us recall that a *tree* is a connected graph $T = (V,E)$, where V is the set of *vertexes* and E is the set of *edges*, such that any two vertices are connected by just one path and there is a distinguished vertex $\mathfrak{v} \in V$ called the *root*. For any vertex $v \in V$, the *parent* of v is the first vertex, say u, visited by the unique path going from v to the root $\mathfrak{v}$, and it is written $u = \texttt{parent}\,(v)$. In this case it is also said that v is a *child* of u. The vertexes with no children are called *leaves*, and the vertexes which are not leaves are called *inner*. An *ordered tree* is a tree such that for each inner tree the collection of its children is totally ordered. Thus an ordered tree can be specified as a list of registers in $V \times V^*$: if $(v;[v_0,\ldots,v_{k-1}])$ appears in that list it means that v is a node with children $v_0,\ldots,v_{k-1}$.

A *depth-first traversing* in an ordered tree is such that at any vertex, the traversing firstly visits the children, in their own ordering, and thereafter it visits that vertex, i.e. for any $(v;v_0,\ldots,v_{k-1})$ the visiting sequence is $v_0 \to \cdots \to v_{k-1} \to v$.

`Depth-first`
Input. An ordered tree as a list $(v, \texttt{children}(v))_{v\in V}$.
Output. A numbering $v \mapsto i \in \mathbb{N}$ according to the depth-first traversing

1. Let *stack* := $[(\upsilon, \texttt{on})]$; *counter* := 0 ;
2. While *stack* $\neq []$ Do
   a. $(v, signal) := \texttt{pop}(stack)$;
   b. If *signal* = `on` Then
      i. If $\texttt{children}(v) \neq []$ Then
         A. $\texttt{push}((v, \texttt{off}), stack)$;
         B. For each $u \in \texttt{children}(v)$ Do $\texttt{push}((u, \texttt{on}), stack)$
      Else
         A. *counter*++ ; $i[v] :=$ *counter* ;
   Else
      i. *counter*++ ; $i[v] :=$ *counter* ;
3. Output $[\, i[v] |\ v \in V \,]$

A *backtracking procedure* is basically a brute-force procedure in which the search space, or *domain space*, has a structure of an ordered tree and the ordering of testing is a depth-first traversing of the domain space.

2.4.2 Ordering of combinations

Let us recall that, for $r, n \in \mathbb{N}$, $r \leq n$, an *r-combination* in n indexes is a one-to-one map $[\![1, r]\!] \to [\![1, n]\!]$. There are $\frac{n!}{(n-r)!}$ such combinations. Let D_n be the union of all r-combinations in n indexes, with $r = 0, 1, \ldots, n$. Thus $\text{card}\,(D_n) = n! \sum_{j=0}^{n} \frac{1}{j!}$.

The collection D_n has a natural ordering structure by the *prefix* relation:

$$\forall \mathbf{c}, \mathbf{d} : \ \ [\mathbf{c} \leq_{pre} \mathbf{d} \implies \exists \mathbf{e} : \mathbf{d} = \mathbf{ce}] .$$

The diagram of this ordering is an ordered tree, and its leaves correspond exactly to the permutations $[\![1, n]\!] \to [\![1, n]\!]$, namely, there are $n!$ leaves in the tree.

A depth-first traversing of this tree gives an enumeration of the whole set D_n.

Let us show now some examples of some particular backtracking procedures.

2.4.3 Traveling salesman problem

Suppose given n cities in a region and a distance d_{ij} associated to each pair $\{c_i, c_j\}$ of cities. The *traveling salesman problem* (TSP) consists in selecting the minimum-distance traversing of all the cities without passing twice through any city.

Let C_n be the set of cities. If the salesman visits r cities, since he is not allowed to visit a city twice, then his traversing may be seen as an r-combination in the set C_n. Then using an effective enumeration of the cities, the set C_n may be identified with the set $[\![1,n]\!]$ and the collection of allowed traversings with the collection D_n of all r-combinations in n indexes, $r \leq n$. For any combination $[c_0,\ldots,c_{r-1}]$, its *weight* is

$$w[c_0,\ldots,c_{r-1}] = \sum_{j=1}^{r-1} d(c_{j-1},c_j).$$

The *backtracking algorithm* to solve TSP consists in the calculation of

$$\min\{w(\mathbf{c})|\ \mathbf{c} \in D_n\}$$

following a depth-first traversing of the diagram of D_n.

2.4.4 Queens problem

This is a generalization of the problem at example 2.4. Suppose given an $(n \times n)$-chessboard. The goal is to place n queens in such a way that no queen is threatening another queen.

A solution is indeed a permutation $\pi : [\![1,n]\!] \to [\![1,n]\!]$ with the connotation that at each column i, $\pi(i)$ gives the row at which the i-th queen should be placed in that column. Since the queens may move along diagonals, any solution must also satisfy

$$\forall i_1,i_2 :\ [i_1 \neq i_2 \implies (\pi(i_1)-i_1 \neq \pi(i_2)-i_2)\ \&\ (\pi(i_1)+i_1 \neq \pi(i_2)+i_2)] \quad (2.8)$$

The collection D_n introduced in section 2.4.2 of combinations in n indexes can be put in correspondence with the partial placements of queens. Thus, a *backtracking algorithm* to solve the Queens Problem consists in pruning all nodes in which condition (2.8) has failed and interrupting the whole process the first time that condition (2.8) succeeds at a leaf of the tree D_n.

2.4.5 Graham's scan for convex hull

As an alternative for *convex hull* calculation, there is a kind of backtracking algorithm called *Graham's scan*. The algorithm proceeds as follows:

`Graham's scan`
Input. A set $X = ((x_i, y_i))_{i=1}^{n}$ of points in the real plane.
Output. The convex hull $\mathrm{co}(X)$.

1. Let x_0 be the point in X with least x-coordinate.
2. Sort $X - \{x_0\}$ according to the angle of the segment joining each point with x_0 and the parallel line to the "y"-axis passing through x_0. Let us assume that after sorting, $X - \{x_0\} = \{x_1, \ldots, x_{n-1}\}$.
3. Initially, let $CH := [x_1, x_0]$, and ToBeTested $:= [x_2, \ldots, x_{n-1}]$
4. While ToBeTested $\neq$ [] Do
   a. Let x_{middle}, x_{prev} be the first two elements at CH ;
   b. $x_{new} :=$ `pop`(ToBeTested) ;
   c. While
      the angle $(\overline{x_{middle}, x_{prev}}\ , \overline{x_{middle}, x_{new}})$ is a right turn
      Do
      i. `drop`(x_{middle}, CH) ;
      ii. let x_{middle}, x_{prev} be the first two elements at CH ;
   d. `push`(x_{new}, CH) ;
5. Output CH.

The step 4.c is properly a backtracking procedure till a turn to the left is got in the three current tested points. In order to decide whether going from a vector $x_0 = (u_0, v_0)$ to a vector $x_1 = (u_1, v_1)$ makes a left turn or a right turn it must be checked

$$d(x_0, x_1) = \det \begin{vmatrix} u_0 & v_0 \\ u_1 & v_1 \end{vmatrix} = u_0 v_1 - u_1 v_0 :$$

$$[d(x_0, x_1) > 0 \implies \text{left turn}] \ \ \& \ \ [d(x_0, x_1) < 0 \implies \text{right turn}] .$$

2.5 Heuristics

Probably the best definition of *heuristic algorithm* is "common sense procedure". Such an algorithm may work fine for most, or just for some, cases, but there lacks a formal proof of its effectiveness. *Heuristics* are used when no effective or tractable solving strategy is known, and they consists of implementations of "rules of thumb". Namely, when it is unknown an efficient algorithm to solve a problem, and the expectation that a pathological instance would occur is rather small, the heuristic may provide a good solving alternative. Although a heuristic may solve some instances, or at least give very good solving approximations, there may be other instances in which the same heuristic fails to get its goal. For many practical problems, a heuristic algorithm may be the only way to get good solutions in a reasonable amount of time. However, success guarantee does not exist within this algorithmic paradigm.

In ordinary life, heuristics are used in a natural way. For instance, a rule for weather prediction may be: "Tomorrow's weather will be as is today". Clearly, in

many cases it is quite effective and its computational cost is negligible! When packing items of odd-shape into a box, usually one tries to put the largest items first and to use the available space to place smaller items. Similarly, TSP may be solved in a greedy way: At each city, the traveller continues his route through the closest non-yet-visited city.

General rules hinting particular ways to build heuristics for specific classes of problems are called *metaheuristics*. The greedy algorithms, as reviewed in section 2.2 provide a metaheuristic. *Simulated annealing*, *tabu search*, *genetic algorithms*, and *Lagrangian relaxation*, among others, give proper metaheuristics [Michalewicz and Fogel(2004)].

Example 2.5 (Assignment problem). Let us assume there are *n tasks* and *n processors* and a *cost matrix* $C = (c_{ij})_{i,j\in[\![0,n-1]\!]}$: c_{ij} is the cost to assign the j-th task to the i-th processor. It is sought the assignment $\alpha : [\![0,n-1]\!] \to [\![0,n-1]\!]$ of least cost $c_\alpha = \sum_{i=0}^{n-1} c_{i\alpha(i)}$.

The first approach is to proceed randomly: For each processor i select randomly a task among the not-yet-assigned tasks and let $\alpha(i)$ be this chosen task.

This is a quite economical assignment: it runs in linear time. But since it does not take care of the cost matrix, the probability to find a minimum-cost assignment is rather small.

As an alternative, for each $k \in [\![0,n-1]\!]$, let m be the least cost appearing at the current cost matrix and let i,j be such that $c_{ij} = m$. Let $\alpha(i) = j$ and suppress the i-th row and the j-th column in the cost matrix.

Evidently in the more advanced stages of the algorithm, the "freedom of choice" is narrower. Thus it may happen that the criteria to broke the ties, when choosing the current minima, may affect the algorithm effectiveness.

Let us remark finally that in the shown heuristics no mathematical formulation has been done. The goal may be stated as an integer programming problem which has an effective solution (although computationally expensive, since Integer Programming is NP-hard [Aho et al(1974)Aho, Hopcroft, and Ullman]).

2.5.1 Simulated annealing

This technique is used to locate extreme values of a given real function defined over a certain domain. Let us assume that the extrema sought values are minima. The most general discrete algorithms proceed through a *local search*: Beginning at an initial point, at each current point the algorithm updates it by choosing a neighbor point according to a determined criterion in search of an extreme value i.e. the algorithm descends following a pick direction, and this step is repeated until some halting conditions are fulfilled.

Simulated annealing is a substantial variation of that strategy. For most occasions it proceeds as outlined but in order to avoid falling into local minima, instead of the sought global minima, at the first stages of the search it jumps with higher probabil-

ities to other parts in the domain in order to reinitiate the descent. At more advanced steps the jump probability is lower. In "physical terms", initially the system is rather hot and as time passes the system is getting colder in a controlled way. This is a motivation of the given name.

Let $C : \mathbb{R}^+ \to [0,1]$ be a decreasing function such that $\lim_{t \searrow 0} C(t) = 0$ and $\lim_{t \nearrow +\infty} C(t) = 1$. The map C is a *cooling strategy*. For instance, for any constant $k > 0$ the map $t \mapsto e^{-\frac{k}{t}}$ is a cooling strategy. In Fig. 2.2, there appears the graph of this map for $k = 2$ and $\frac{1}{k^3} \leq t \leq 20k$.

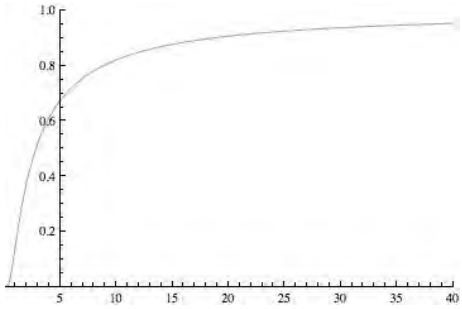


Fig. 2.2 The graph of $t \mapsto e^{-\frac{k}{t}}$ for $k = 2$ and $\frac{1}{k^3} \leq t \leq 20k$.

Let $f : D \to \mathbb{R}$ be a real map defined on a set D in which for each point $x \in D$ there is associated a finite set $N(x) \subset D$, called the *neighborhood* of x. The goal is to find a minimum value of f. Within this context, the objective map f is also called the *energy function* and the sought point a *minimal energy point*. Roughly, the algorithm is sketched in figure 2.3.

The steps 5.a.iii.A-B are the essence of simulated annealing: if the system is enough hot the updating may choose a worst current point.

2.5.2 Tabu search

As in simulated annealing, *tabu search* has an updating rule that sometimes may opt for worst updatings in the hope to escape from local minima. Let us assume that D is a domain such that for each $x \in D$ there is a neighborhood $N(x) \subset D$, and that $f : D \to \mathbb{R}$ is an objective function to be minimized. Now, there is a *tabu list* T whose aim is to track the local minima already found, and the paths leading to them, in order to avoid repetition of any path. The updating rule may be synthesized as:

$$x_{new} \in N(x_{old}) \;\&\; [f(x_{new}) < f(x_{old}) \text{ or } x_{new} \notin T] \implies x_{old} := x_{new}. \qquad (2.9)$$

In a schematic way, the procedure is shown in figure 2.4.

`SimulatedAnnealing`
Input. A cooling strategy $C : \mathbb{R}^+ \to [0,1]$.
An energy function $f : D \to \mathbb{R}$.
Output. A minimum energy point $x_0 \in D$.

1. Initially Pick $x_{old} \in D$; $v_{old} := f(x_{old})$;
2. t_0 :=(some small threshold) ; % Threshold for the system to get *cool*
3. $t :=$ (enough high value) ; % just to begin with a *hot* system
4. $cool := (t < t_0)$;
5. While (Not *cool*) Do
   a. Repeat
      i. Pick $x_{new} \in N(x_{old})$; $v_{new} := f(x_{new})$;
      ii. $\Delta f := v_{new} - v_{old}$;
      iii. If $\Delta f < 0$ then Update $\{x_{old} := x_{new}$; $v_{old} := v_{new}\}$
      Else
         A. Pick $r \in [0,1]$;
         B. If $r < C(t)$ then Update $\{x_{old} := x_{new}$; $v_{old} := v_{new}\}$
      Until No Updating has been done in several trials % Thermal equilibrium is got
   b. Decrease t ;
   c. $cool := (t < t_0)$;
6. Output $x_0 = x_{old}$

Fig. 2.3 Procedure `SimulatedAnnealing`.

`TabuSearch`
Input. An objective function $f : D \to \mathbb{R}$.
Output. An intended minimum point $x_0 \in D$: $f(x_0) \leq f(x)$, $\forall x \in D$.

1. Let $x_{mc} \in D$ be a (local) minimum of f ;
2. $x_c := x_{mc}$;
3. $T := [x_c]$;
4. Repeat
   a. Pick $x \in N(x_c) - T$;
   b. AppendTo$[T,x]$;
   c. If $f(x) < f(x_{mc})$ then $x_{mc} := x$;
   d. $x_c := x$;
   e. Update T

   Until some ending condition is fulfilled
5. Output x_{mc}

Fig. 2.4 Procedure `TabuSearch`.

The search at step 1. is done through any conventional procedure. Steps 4.c and 4.d implement the updating rule (2.9). In steps 4.b and 4.e some storage considerations may be implemented. In *short term memory*, 4.b can be skipped most times, and in step 4.e there can be stored just selected values of x_c. In *long term memory*, every tested point can be stored in order to avoid repetitions not just of paths but even of mere points.

2.5.3 Genetic algorithms

The *genetic algorithms* treat mainly with optimization problems. Given a real objective function $f : D \to \mathbb{R}$, the problem consists in finding an extreme value $x_0 \in D$, say a minimum, of the function f.

Here, at each stage of a genetic algorithm there is a whole collection of points with the corresponding function values. Naturally, in the collection those points giving lesser values are preferred. Hence some points are discarded and the remaining points are combined in order to increase the number of current points and the whole process is repeated until improvements in the objective function are either impossible (the minimum has already found) or implausible (a closer approximation to the minimum is computationally hard). Due to the biological inspiration of the algorithms, the jargon is this domain is quite particular. We will use it since currently it has become conventional. However the reader should be aware that at any time we will be dealing with points, values and maps and the biological terms are just naming (although in a metaphoric way) those objects or particular procedures.

The genetic algorithms involve characteristic subprocesses:

Initialization. Many individual solutions are randomly chosen within the domain D to form an *initial population*. These first elements either are uniformly distributed in order to form a representative and unbiased sample of points or may be *seeds* in likely regions to contain solutions of the problem. The size of the initial populations is chosen *ad-hoc*.

Selection. The current population is called a *generation*. A subset of the current population is selected in terms of *good-fitness*, in order to breed a new generation. The selection process may involve either the rating of all the individuals or just a randomly chosen sample of individuals, and in this case the stochastic procedures in selecting the individuals to be tested should guarantee the diversity of the generation in order to avoid premature convergence to poor solutions. Among well-studied selection procedures there are the so called *roulette wheel selection* and *tournament selection*.

Reproduction. There are to main operators: *crossover*, and *mutation*. For each prospective *child* in a new generation, a couple of *parents* is selected and they are combined in order to breed the child point. Of course, a couple of just two parents is an arbitrary decision, and couples with more than just two members may provide better performance. Also, individual children may mutate in order to get better fitness.

All the combinations are designed with the purpose to improve the average fitness of the current population.

Termination. Several ending conditions may be stated in terms of the reached number of generations, or the value of the average fitness, or the value of the best fitness, or the number of repetitions of some population features.

The sketch of the general procedure is the following:

`Genetic algorithm`
Input. An objective function $f : D \to \mathbb{R}$.
Output. An intended minimum point $x_0 \in D$: $f(x_0) \leq f(x), \forall x \in D$.

1. $ctr := 0$;
2. Initialize a population $P(ctr)$;
3. Evaluate the population $P(ctr)$;
4. While Termination conditions fail Do
   a. $ctr++$;
   b. Select $P(ctr)$ from $P(ctr-1)$;
   c. Crossover $P(ctr)$;
   d. Mutate $P(ctr)$;
   e. Evaluate $P(ctr)$;
5. Output a chosen $x_0 \in P(ctr)$

2.5.4 Lagrangian relaxation

Let $n, m \in \mathbb{N}$ be positive integers, and let $f : \mathbb{R}^n \to \mathbb{R}$ be a real continuous function. Let us consider the optimization problem

$$\begin{array}{ll} \text{Minimize}_{v \in \mathbb{R}^n} & f(v) \\ \text{subject to} & Ev = d \ \ \& \ \ \ell \leq v \leq u \end{array} \tag{2.10}$$

where $E \in \mathbb{R}^{m \times n}$, $d \in \mathbb{R}^m$, $\ell, u \in \mathbb{R}^n$. The problem (2.10) is called the *primal problem*. Let us consider the map

$$L : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R} \ \ , \ \ (v, w) \mapsto L(v, w) = f(v) + \langle w | Ev - d \rangle = f(v) + w^T (Ev - d),$$

the variable vector w consists of *Lagrange multipliers*. Also, let us consider

$$\eta : \mathbb{R}^m \to \mathbb{R} \ \ , \ \ w \mapsto \min\{L(v, w) |\ \ell \leq v \leq u\}.$$

Then the so called *dual problem* is:

$$\text{Maximize}_{w \in \mathbb{R}^m} \eta(w). \tag{2.11}$$

Remark 2.2. If $w_0 \in \mathbb{R}^m$ is a solution of the dual problem (2.11) and $v_0 \in \mathbb{R}^n$ is a solution of the primal problem (2.10) then $\eta(w_0) \leq f(v_0)$.

Proof. From the definitions of maps L and η we have:

$$w \in \mathbb{R}^m \;\&\; v \in \mathbb{R}^n \;\&\; \ell \leq v \leq u \implies \eta(w) \leq f(v) + w^T(Ev - d)$$

and consequently

$$w \in \mathbb{R}^m \;\&\; v \in \mathbb{R}^n \;\&\; \ell \leq v \leq u \;\&\; Ev = d \implies \eta(w) \leq f(v).$$

The remark follows. □

Remark 2.3. The map η is concave:

$$w_0, w_1 \in \mathbb{R}^m \;\&\; t \in [0,1] \implies (1-t)\eta(w_0) + t\eta(w_1) \leq \eta\left((1-t)w_0 + tw_1\right).$$

Thus η has an unique maximum. Hence, solving the dual problem is an easier task than solving the primal problem, and a solution of the dual problem entails a solution of the primal problem.

A particular case in the use of Lagrange multipliers is *Lagrangian relaxation*. Let us consider the *Linear Integer Problem*:

$$\begin{array}{ll} \text{Minimize}_{v \in \mathbb{Z}^n} & c^T v \\ \text{subject to} & Av = d \;\;\&\;\; Bv = e \end{array} \tag{2.12}$$

where $c \in \mathbb{Z}^n$, $A \in \mathbb{Z}^{m_0 \times n}$, $d \in \mathbb{Z}^{m_0}$, $B \in \mathbb{Z}^{m_1 \times n}$ and $e \in \mathbb{Z}^{m_1}$. We may think that the condition $Bv = e$ imposes *easy restrictions* (for instance, if B is the identity matrix and $e = 0$ the consition states $v \geq 0$), while the condition $Av = d$ imposes *hard restrictions*. Then the corresponding dual problem becomes

$$\max_{w \in \mathbb{Z}^{m_0}} \eta(w) \tag{2.13}$$

where

$$\eta : w \mapsto \min\{c^T v + w^T(Av - d) |\; v \in \mathbb{Z}^n \;\&\; Bv = e\}. \tag{2.14}$$

The dual problem (2.13) may be easier to solve, because due to the concavity of the corresponding objective map, it can be treated by "classical methods" in $\mathbb{R}^n$ and then to round-up to a solution in $\mathbb{Z}^n$, while the evaluation of η according to (2.14) involves just "easy" restrictions.

2.6 Pattern matching and syntactical algorithms

2.6.1 General notions

A *language* is a set of words and a *word* is a string of finite length over a given *alphabet A*. The zero-length word, i.e. the word containing no symbols, is the *empty word* λ. Let A^* denote the set of words. Thus a language is a set $L \subset A^*$. For instance, the *empty language* $\emptyset$ contains no words, while the *total language*, or *dictionary* A^*, contains all possible words. A language may be given as an exhaustive list consisting of the words in that language. Obviously this is feasible, although probably not possible, only when the language is finite.

Example 2.6. A database is a language. Each register in the data base is properly a word, which is, in turn a concatenation of "symbols" which are the values taken by the attributes in the database relational scheme. Thus the alphabet is the union of all attribute domains.

Example 2.7 (Chess). In chess, a match is a sequence of, let us say, positions, steps or configurations. Each configuration is a 64-length word over the alphabet P consisting of the figures, or pieces, white and black, and a "blank" figure to point that a cell may be empty. This alphabet has thus $16+16+1=33$ symbols. Let C denote the collection of configurations, $C \subset P^{64} \subset P^*$. Since each figure may appear in at most one cell, the number of configurations is bounded by

$$\operatorname{card}(C) \leq \sum_{j=2}^{32} \binom{64}{j} j! = \sum_{j=2}^{32} \frac{64!}{(64-j)!} \sim 10^{54}.$$

A match, as a finite sequence of configurations, is a word with symbols in C. Let M be the collection of legal chess matches. Then M is a language over C, $M \subset C^*$, and indeed a language over P in which each word has as length a multiple of 64.

If a language $L \subset A^*$ is infinite, then it can be specified by a collection of *syntactical rules* or *productions*:

$$\alpha \to \beta, \text{ with } \alpha \in (V+A)^* V (V+A)^* \text{ and } \beta \in (V+A)^*,$$

where V is a set of *metasymbols*, and the connotation

whenever there appears α, then it can be replaced by β.

Beginning from an initial metasymbol $S \in V$ the *language generated* by the system of productions (or *grammar*) is the set of words such that there is a transformation (or *parsing*) from the initial symbol to that word.

Example 2.8 (Triplets of equal length). Let $L_3 = \{a^k b^k c^k | k \geq 1\}$ be the language consisting of words formed by three strings of constant symbols a's, b's and c's of equal length.

A set of productions to generate this language is the following:

1. $S \to A$ initialize
2. $A \to aABC$ add a's ad for each a b and a c should be added
3. $A \to abC$ stop a's addition and balance it with a b and a c
4. $CB \to BC$ move the c's to the right of all b's
5. $bB \to bb$ change the provisional B's for terminal b's
6. $bC \to bc$ begin to change the C's
7. $cC \to cc$ change the provisional C's for terminal c's

The language generated by this grammar is L_3, indeed.

Example 2.9 (Chess). Let us get back to example 2.7. There is an *initial configuration*, the well known starting position in chess. Then a match is obtained through a finite sequence of *legal moves*. These are indeed also well known and they are formalized as productions in the language of configurations. The configurations obtained from the initial one are the ending configurations: checkmate to a king or draw configurations.

Two grammars are *equivalent* if they generate the same language.

The grammars can be classified according to the form of their productions, and this originates the so called *Chomsky hierarchy* (most conventional texts expose this hierarchy, e.g. [Hopcroft et al(2001)Hopcroft, Motwani, and Ullman]). Just to mention a few examples:

Regular. $T \to \beta U$, or $T \to \beta$, with $T,U \in V$, $\beta \in A^*$
Linear. $T \to \beta_l U \beta_r$, or $T \to \beta$, with $T,U \in V$, $\beta_l, \beta_r \in A^*$
Context-free. $T \to \beta$, with $T \in V$, $\beta \in (V+A)^*$
Unrestricted. No restrictions are imposed.

The levels of the hierarchy are nested. The most general, the unrestricted grammars, generate all *recursive* or *effectively recognized* languages. The languages receive the name of the grammars that generate them. The most popular programming languages are context-free languages (they are generated by context-free grammars) and regular languages are extremely important in communication protocols and in basic operations of operative systems. Each level in the Chomsky hierarchy has a particular kind of computing devices able to parse the languages in that level. The unrestricted grammars are parsed by *Turing machines*, context-free grammars are parsed by *pushdown automata*, linear grammars are parsed by two-state pushdown automata, and regular grammars are parsed by *regular* or *finite automata*.

Several important problems arose within the context of formal languages:

Word problem. Given a language $L \subset A^*$ and a word $x \in A^*$ decide whether $x \in L$.
Synthesis problem. Given a language $L \subset A^*$ find the simplest grammar that generates L.
Equivalence problem. Given two languages $L_0, L_1 \subset A^*$ decide whether $L_0 = L_1$.
Boolean composition problem. Given two languages $L_0, L_1 \subset A^*$ characterize the union $L_0 \cup L_1$, the meet $L_0 \cap L_1$, and the complement $A^* - L_0$.

The complexity of these problems depend on the level at the Chomsky hierarchy. The Word Problem is easily solvable within regular languages (given L find the minimum automaton recognizing it, and for any query word x apply it to the automaton: it arrives to a success state if and only if the query word lies indeed in the language), hence communication protocols as TCP/IP can efficiently be implemented; but the Word Problem is unsolvable for recursive languages.

The following problems have simple statements, the first has an existential flavour while the second is of an universal kind:

Substring problem. Given two words, a *text* and a *pattern*, decide whether the pattern occurs at the text.

Occurrences problem. Given a text and a pattern, find all occurrences of the pattern within the text.

In spite of their simple statements, the problems above have important applications in several areas of Computer Engineering: Image Processing, Word Processing, Database Management, Bioinformatics, Operative Systems, etc.

As concrete examples of syntactic algorithms, let us sketch some conventional procedures to solve the Substring Problem.

2.6.2 Research with an automaton

Let $y \in A^*$ be the text and let $x \in A^*$ be the pattern. Let us construct an automaton R recognizing the language A^*x consisting of all words that have x as a suffix. Then the pattern appears in the text if and only if when applying y into R a terminal state is visited at least once. In figure 2.5 we sketch the procedure.

2.6.3 Knuth-Morris-Pratt algorithm

Let $y \in A^*$ be the text and let $x \in A^*$ be the pattern:

$$y = \boxed{}$$
$$x = \boxed{}$$

Suppose that a prefix σ has a repetition:

$$x = \boxed{\sigma}\boxed{a}\boxed{}\boxed{\sigma}\boxed{b}\boxed{}$$

For the purpose of the current algorithm, let us number the symbols at any string from the index 0: $m = \operatorname{len}(z) \implies z = z_0 \cdots z_{m-1}$.

Suppose that the repeated prefix σ has length k. If the letter b appears at position j then it will be useful to remember that k symbols before b form a prefix of the

`ResearchWithAutomaton`
Input. The text $y \in A^*$ and the pattern $x \in A^*$.
Output. A Yes-No decision whether the pattern occurs at the text.

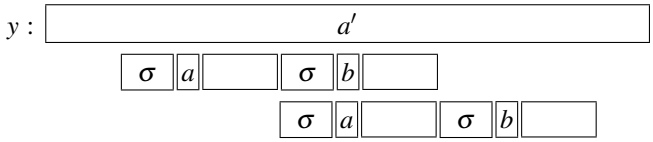
1. $n := \text{len}(x)$;
2. for each $i \in [\![0,n]\!]$ let x_i be the prefix of length i of x ;
3. let $R := (Q,A,t,x_0)$ be the automaton defined as follows:
   States: $Q = \{x_i\}_{i=0}^{n}$.
   Transitions: $\forall x_i \in Q$, $a \in A$,

$$\begin{cases} x_i a \in Q \Longrightarrow t(x_i,a) = x_i a \\ x_i a \notin Q \Longrightarrow t(x_i,a) = \text{longest suffix at } x_i a \text{ being a prefix of } x \end{cases}$$

   Initial state: $x_0 = \lambda$ (the empty word)
   Terminal state: $x_n = x$ (the whole pattern)
4. $m := \text{len}(y)$;
5. keep_on := False ;
6. $q_{\text{current}} := x_0$; $j := 1$;
7. While (Not keep_on)&&$(j \leq m)$ Do
   a. $y_j :=$ j-th symbol of y ;
   b. $q_{\text{current}} := t(q_{\text{current}}, y_j)$;
   c. keep_on := $(q_{\text{current}} == x)$;
   d. $j++$
8. If keep_on Then Output Yes Else Output No

Fig. 2.5 Procedure `ResearchWithAutomaton`.

pattern, because when matching the pattern with a current segment of the text, if a discrepancy occurs at b then the pattern x may be shifted in order to align the prefix σ, say $\sigma^{(1)}$, with its second occurrence, say $\sigma^{(2)}$, and to continue with the comparison process.



Thus, in a Track_x table, let us put $\text{Track}_x[j] := k$. For instance:

$$\begin{aligned} x &= \quad \text{G C A T G C T} \\ \text{Track}_x &= -1\ 0\ 0\ 0\ 0\ 1\ 2 \end{aligned}$$

The *Knuth-Morris-Pratt algorithm* (KMP) exploits the table Track_x. The procedure at figure 2.6 outlines the construction of the table.

The algorithm KMP shifts the pattern along the text in order to find an occurrence. The shift is not done by moving just one place, instead, whenever there is a discrepancy, the pattern moves to the first non-tested place in which the matching

`KMPTrackingTable`
Input. The pattern $x \in A^*$.
Output. The table Track_x.

1. $n := \text{len}(x)$;
2. $\text{Track}_x[0] := -1$; $\text{Track}_x[1] := 0$;
3. $j := 2$; $i := 0$;
4. While $j \leq n$ Do
   a. If $x[j-1] == x[i]$ Then
      i. $\text{Track}_x[j] := i+1$;
      ii. $j++$; $i++$
      Else
      i. If $i > 0$ Then $i := \text{Track}_x[i]$
         Else $\{\text{Track}_x[j] := 0$; $j++\}$;
5. Output Track_x.

Fig. 2.6 Procedure `KMPTrackingTable`.

may continue. This is done with the help of the tracking table. The procedure is shown at figure 2.7.

`KMP`
Input. The text $y \in A^*$ and the pattern $x \in A^*$.
Output. The first occurrence of x in y if it does occur.

1. $\text{Track}_x :=$ `KMPTrackingTable`(x) ;
2. $n_y := \text{len}(y)$; $n_x := \text{len}(x)$;
3. $j_y := 0$; $j_x := 0$;
4. While $j_x + j_y < n_y$ Do
   a. If $x[j_x] == y[j_y + j_x]$ Then
      i. j_x++ ;
      ii. If $j_x == n_x$ Then Output j_y
      Else
      i. $j_y := j_y + j_x - \text{Track}_x[j_x]$;
      ii. If $j_x > 0$ Then $j_x := \text{Track}_x[j_x]$;
5. Output "The pattern x does not appear in the text y"

Fig. 2.7 Procedure `KMP`: Decides whether a pattern occurs within a text.

2.6.4 Suffix trees

A *suffix tree* for a given text y of length m is a tree such that:

- Each internal node in the tree has at least two children.

- Each edge in the tree is labelled by a substring of y.
- No pair of edges exiting from one node may have a non-empty common prefix.
- The tree has m leaves: Each non-empty suffix of y is associated to one leaf in the tree. The path connecting the leaf to the root produces the associated suffix by concatenating the labels.

For instance, for $y =$`parangaracatirimicuaro` of length 22, table 2.1 displays its suffix tree (calculated with the Suffix Tree Demo developed by L. Allison[1]): each pair (j: label) represents the initial position of a suffix in the string and the label associated to the corresponding edge.

```
     |(1:parangaracatirimicuaro)|leaf
tree:|
     |     |     |     |(5:ngaracatirimicuaro)|leaf
     |     |     |(4:a)|
     |     |     |     |(10:catirimicuaro)|leaf
     |     |(3:r)|
     |     |     |(22:o)|leaf
     |(2:a)|
     |     |(5:ngaracatirimicuaro)|leaf
     |     |
     |     |(10:catirimicuaro)|leaf
     |     |
     |     |(12:tirimicuaro)|leaf
     |
     |     |     |(5:ngaracatirimicuaro)|leaf
     |     |(4:a)|
     |     |     |(10:catirimicuaro)|leaf
     |(3:r)|
     |     |(15:imicuaro)|leaf
     |     |
     |     |(22:o)|leaf
     |
     |(5:ngaracatirimicuaro)|leaf
     |
     |(6:garacatirimicuaro)|leaf
     |
     |      |(11:atirimicuaro)|leaf
     |(10:c)|
     |      |(19:uaro)|leaf
     |
     |(12:tirimicuaro)|leaf
     |
     |      |(14:rimicuaro)|leaf
     |(13:i)|
     |      |(16:micuaro)|leaf
     |      |
     |      |(18:cuaro)|leaf
     |
     |(16:micuaro)|leaf
     |
     |(19:uaro)|leaf
     |
     |(22:o)|leaf
```

Table 2.1 Suffix tree for the string `parangaracatirimicuaro` of length 22. It has 8 branching nodes.

If the string y is such that a proper prefix coincides with a proper prefix, say $\exists x,z \in A^*$: $y = xzx$, with $\text{len}(x) > 0$, then no suffix tree may exist for y. Thus, in order to avoid this problem, usually it is considered a symbol `$` not in A and the word y`$` instead of y.

Let us sketch a very basic procedure to build suffix trees. Let us assume that the string $y = u_1 \cdots u_m$ is given. Let, for each $i \leq m$, $y_i = u_i \cdots u_m$`$` be the suffix ranging

[1] `http://www.allisons.org/ll/AlgDS/Tree/Suffix/`

```
|      |          |(7:aedadysiglosdichososaquellosalosquelosantiguos)|leaf
|      |(2:ichos)|
|      |          |(25:osaquellosalosquelosantiguos)|leaf
|(1:d)|
|      |(10:adysiglosdichososaquellosalosquelosantiguos)|leaf
|      |
|      |(12:ysiglosdichososaquellosalosquelosantiguos)|leaf
tree:|
|      |         |(7:aedadysiglosdichososaquellosalosquelosantiguos)|leaf
|      |(3:chos)|
|      |         |(25:osaquellosalosquelosantiguos)|leaf
|(2:i)|
|      |       |(16:losdichososaquellosalosquelosantiguos)|leaf
|      |(15:g)|
|      |       |(50:uos)|leaf
|
|         |(7:aedadysiglosdichososaquellosalosquelosantiguos)|leaf
|(3:chos)|
|         |(25:osaquellosalosquelosantiguos)|leaf
|
|        |(7:aedadysiglosdichososaquellosalosquelosantiguos)|leaf
|(4:hos)|
|        |(25:osaquellosalosquelosantiguos)|leaf
|
|       |     |(8:edadysiglosdichososaquellosalosquelosantiguos)|leaf
|       |(7:a)|
|       |     |(28:quellosalosquelosantiguos)|leaf
|       |     |
|       |     |(36:losquelosantiguos)|leaf
|       |     |
|       |     |(46:ntiguos)|leaf
|(5:os)|
|       |(19:dichososaquellosalosquelosantiguos)|leaf
|       |
|       |(25:osaquellosalosquelosantiguos)|leaf
|       |
|       |(39:quelosantiguos)|leaf
|
|      |     |(8:edadysiglosdichososaquellosalosquelosantiguos)|leaf
|      |(7:a)|
|      |     |(28:quellosalosquelosantiguos)|leaf
|      |     |
|      |     |(36:losquelosantiguos)|leaf
|      |     |
|      |     |(46:ntiguos)|leaf
|(6:s)|
|      |(14:iglosdichososaquellosalosquelosantiguos)|leaf
|      |
|      |(19:dichososaquellosalosquelosantiguos)|leaf
|      |
|      |(25:osaquellosalosquelosantiguos)|leaf
|      |
|      |(39:quelosantiguos)|leaf
|
|      |(8:edadysiglosdichososaquellosalosquelosantiguos)|leaf
|(7:a)|
|      |(11:dysiglosdichososaquellosalosquelosantiguos)|leaf
|      |
|      |(28:quellosalosquelosantiguos)|leaf
|      |
|      |(36:losquelosantiguos)|leaf
|      |
|      |(46:ntiguos)|leaf
|
|      |(9:dadysiglosdichososaquellosalosquelosantiguos)|leaf
|(8:e)|
|      |       |(32:losalosquelosantiguos)|leaf
|      |(31:l)|
|      |       |(43:osantiguos)|leaf
|
|(12:ysiglosdichososaquellosalosquelosantiguos)|leaf
|
|       |(16:losdichososaquellosalosquelosantiguos)|leaf
|(15:g)|
|       |(50:uos)|leaf
|
|       |       |(19:dichososaquellosalosquelosantiguos)|leaf
|       |(17:os)|
|       |       |      |(36:losquelosantiguos)|leaf
|       |       |(35:a)|
|       |       |      |(46:ntiguos)|leaf
|       |       |
|       |       |(39:quelosantiguos)|leaf
|(16:l)|
|       |(32:losalosquelosantiguos)|leaf
|
|         |(32:losalosquelosantiguos)|leaf
|(28:quel)|
|         |(43:osantiguos)|leaf
|
|       |       |(32:losalosquelosantiguos)|leaf
|       |(30:el)|
|       |       |(43:osantiguos)|leaf
|(29:u)|
|       |(51:os)|leaf
|
|(46:ntiguos)|leaf
|
|(47:tiguos)|leaf
```

Table 2.2 $y =$`dichosaedadysiglosdichososaquellosalosquelosantiguos` of length 53. The tree has 22 branching nodes.

from the i-th to the last position. Let T_i be the suffix tree codifying just the suffixes $y_1 \ldots, y_i$. The goal is to build T_m. Let us proceed recursively.

Clearly, T_1 consists of just one edge labelled by $y_1 = y\$$. The edge has as extremes the root vertex and a leaf, marked as 1.

Let us assume that T_j has been constructed already. Find the largest prefix z_{j+1} of y_{j+1} obtained as concatenation of labels in T_j from its root to a vertex in T_j. The right extreme of z_{j+1} should be between two nodes u and v of T_j. In other words, there is a suffix $w_l \in A^*$ of z_{j+1} and two vertexes u and v in T_j such that the label of the edge (u,v) is $w_l w_r$, with $w_r \in A^*$. Then split the edge (u,v) and add a new leaf. Explicitly, add two vertexes u', v', and the edges (u,u'), (u',v) and (u',v') with respective labels w_l, w_r, and the complementary suffix of z_{j+1} within y_{j+1}. Mark v' as the $(j+1)$-th leaf. The resulting tree is T_{j+1}.

The reader may check this algorithm with the example displayed at table 2.1.

Since there are m suffixes each of different length between 1 and m, we see that the mentioned algorithm requires $O(m^2)$ comparisons of symbols. An alternative algorithm [Ukkonen(1995)] requires $O(m)$ comparisons, thus the quadratic naive mentioned algorithm is far from the linear complexity of suffix trees construction.

However, the representation is very economical. Since the suffix tree has m leaves, then necessarily it should have $2m-1$ vertexes. On the other hand, the original string has, for each $i \leq m$, $m-i+1$ substrings of length i. Thus, there are $\frac{1}{2}m(m-1) = O(m^2)$ non-empty substrings. Hence a linear space is used to store a quadratic number of objects.

It is interesting to observe that the suffix trees for words of the form $ab^k a$ and $a^k b^k$ are remarkably similar.

The Substring Problem can be solved in linear time with suffix trees: Namely, give a text y and a pattern x, construct the suffix tree T_y of y, then in a breadth-first way locate the vertex v with an edge labelled with a word having the beginning of the pattern, and parse the pattern along the descendants of v. Reply in accordance with the parsing success.

The Occurrences Problem also can be solved in linear time with suffix trees: Namely, once a pattern has been decided to occur in the text, and the parsing has led to a vertex w in the suffix tree, then all occurrences of the pattern are realized as the leaves of the subtree rooted at w.

The *longest common subsequence* of two sequences x, y can also be efficiently solved. Namely, let $z = x\#y\$$ and construct the suffix tree T_z of z. Then the longest common subsequence is the longest pattern w of length at most $\min\{\operatorname{len}(x), \operatorname{len}(y)\}$ that occurs in a suffix of x of length greater than $\operatorname{len}(w)$ and in a suffix of y of length greater than $\operatorname{len}(w)$.

2.7 Numerical approximation algorithms

We will review here schematic procedures based on numerical approximation. The reader may get a detailed exposition at the classical book [Stoer and Bulirsch(1980)].

2.7.1 Roots of functions

2.7.1.1 A reminder of Differential Calculus

For any function $f : \mathbb{R}^n \to \mathbb{R}$ and any point $x \in \mathbb{R}^n$, the following notions are conventional:

Partial derivatives. For each $j \in [\![1,n]\!]$ the *partial derivative* of f at x is

$$\partial_j f(x) = \frac{\partial f}{\partial x_j}(x) = \lim_{t \to 0} \frac{1}{t} \left[f(x + t e_j) - f(x) \right], \tag{2.15}$$

where e_j is the j-th vector at the canonical basis of $\mathbb{R}^n$, i.e. it is the vector that has the value 0 at each entry, except at the j-th entry in which it has the value 1.

Gradient. The *gradient* of f at x is

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x) \ \cdots \ \frac{\partial f}{\partial x_n}(x) \right].$$

In other words, the gradient is the $(1 \times n)$-matrix whose entries are the partial derivatives. Naturally, the gradient can be identified with the vector whose components are the partial derivatives,

$$\nabla f(x) \approx \sum_{j=1}^{n} \frac{\partial f}{\partial x_j}(x) e_j.$$

Differential. The *differential* of f at x is the linear map $Df(x) : \mathbb{R}^n \to \mathbb{R}$ that is represented by the gradient of f at x:

$$Df(x) : y \mapsto \nabla f(x) \cdot y = \sum_{j=1}^{n} \frac{\partial f}{\partial x_j}(x) \, y_j.$$

Higher order derivatives. These are defined recursively. If $j_1, j_2, \ldots, j_m$ are $m \geq 2$ indexes in the integer interval $[\![1,n]\!]$ the corresponding *partial derivative of order* m of f at x is

$$\partial_{j_1 j_2 \cdots j_m} f(x) = \partial_{j_1} \left(\partial_{j_2 \cdots j_m} f(x) \right).$$

Hessian. The *Hessian* of f at x is the $(n \times n)$-matrix whose entries are the partial derivatives of order 2:

$$Hf(x) = \begin{bmatrix} \partial_{11} f(x) & \cdots & \partial_{1n} f(x) \\ \vdots & & \vdots \\ \partial_{n1} f(x) & \cdots & \partial_{nn} f(x) \end{bmatrix}.$$

Customarily, it is also written $D^{(2)} f(x) = Hf(x)$. The Hessian determines a *quadratic form*

$$D^{(2)}f(x):\mathbb{R}^n\to\mathbb{R}\ ,\ y\mapsto D^{(2)}f(x)(y)=y^T\ D^{(2)}f(x)\ y=\sum_{i,j=1}^{n}\partial_{ij}f(x)\ y_iy_j.$$

For any enough smooth function $f:\mathbb{R}^n\to\mathbb{R}$ and any point $x_0\in\mathbb{R}^n$ a local *first order approximation* is obtained as

$$f(x)=f(x_0)+Df(x)\,(x-x_0)+o\left(\|x-x_0\|\right) \tag{2.16}$$

and a local *second order approximation* is obtained as

$$f(x)=f(x_0)+Df(x)\,(x-x_0)+\frac{1}{2}D^{(2)}f(x)\,(x-x_0)+o\left(\|x-x_0\|^2\right). \tag{2.17}$$

In those formulas $\|\cdot\|:x\mapsto\|x-x_0\|=\sqrt{\sum_{j=1}^{n}|x_j|^2}$ is the Euclidean norm.

For $n=1$ and enough smooth functions $f:\mathbb{R}\to\mathbb{R}$, above relation (2.17), can be generalized to the *Taylor's Expansion Formula*: For any $k\geq 0$,

$$f(x)=\sum_{\kappa=0}^{k}\frac{1}{\kappa!}f^{(\kappa)}(x_0)\,(x-x_0)^{\kappa}+o\left(\|x-x_0\|^{\kappa}\right). \tag{2.18}$$

As is well known, Taylor's Expansion is used to approximate within arbitrary precision the values of transcendent functions.

2.7.1.2 The Fixed Point Theorem

A classical problem consists in finding the *roots* of a real valued function. If $f:\mathbb{R}^n\to\mathbb{R}$ is a real valued map defined on the n-dimensional real vector space, a *root of map* f is a point $x\in\mathbb{R}^n$ such that $f(x)=0$.

Let us assume along this section that the considered objective maps are enough differentiable in their (open) domains.

Remark 2.4. The problem to find the roots of a function is closely related to the problem to find unconstrained function extrema.

Suppose that $h:\mathbb{R}^n\to\mathbb{R}$ is a smooth objective function to be minimized over its domain $\mathbb{R}^n$. Then any minimum point $x_0\in\mathbb{R}^n$ is such that the gradient attains a null value, $\nabla h(x)=0$. Hence the problem to minimize h reduces to the problem to calculate a common root of the maps $f_j=\partial_j h:x\mapsto\partial_j h(x)$, for $j\in[\![1,n]\!]$.

Indeed, the converse is also valid. Suppose that $f:\mathbb{R}^n\to\mathbb{R}$ is a function whose roots are sought. Let $h:x\mapsto\|f(x)\|^2$, then the roots of f correspond to the points giving the minimal values of h.

Remark 2.5. The problem to find the simultaneous roots of n real functions is closely related to the problem to find fixed points of maps $\mathbb{R}^n\to\mathbb{R}^n$.

Given $f_1,\ldots,f_n:\mathbb{R}^n\to\mathbb{R}$, let

$$g : \mathbb{R}^n \to \mathbb{R}^n \ , \ x \mapsto g(x) = x + (f_1(x), \ldots, f_n(x)).$$

Evidently, for any $x \in \mathbb{R}^n$:

$$x = g(x) \iff \forall j \in [\![1,n]\!] : \ f_j(x) = 0.$$

A function $g : \mathbb{R}^n \to \mathbb{R}^n$ is a *contraction* if

$$\exists c \in [0,1[: \ \forall x, y \in \mathbb{R}^n \ \|f(x) - f(y)\| \leq c\|x - y\|. \tag{2.19}$$

Theorem 2.1 (Banach's Fixed Point -). *Any contraction $g : \mathbb{R}^n \to \mathbb{R}^n$ possesses a fixed point.*

Proof. Let $x_0 \in \mathbb{R}^n$ be an arbitrary point, and for each $k \geq 1$ let $x_k = g(x_{k-1})$. In other words, the sequence $(x_k)_{k=0}^{+\infty}$ consists of the consecutive iterations of the map g starting at an arbitrary point in the domain space.

Then, for any $k \in \mathbb{N}$, $k \geq 2$:

$$\|x_k - x_{k-1}\| = \|g(x_{k-1}) - g(x_{k-2})\| \leq c\|x_{k-1} - x_{k-2}\| \leq \cdots \leq c^{k-1}\|x_1 - x_0\|.$$

Consequently, for any $k \leq \ell$,

$$\begin{aligned} \|x_\ell - x_k\| &= \left\| \sum_{\kappa=k+1}^{\ell} (x_\kappa - x_{\kappa-1}) \right\| \\ &\leq \sum_{\kappa=k+1}^{\ell} \|x_\kappa - x_{\kappa-1}\| \\ &\leq \sum_{\kappa=k+1}^{\ell} c^{\kappa-1}\|x_1 - x_0\| \\ &\leq \|x_1 - x_0\| \, c^k \sum_{\kappa'\geq 0} c^{\kappa'} \\ &= \frac{\|x_1 - x_0\|}{1-c} \, c^k \end{aligned} \tag{2.20}$$

Since $0 \leq c < 1$, we have $c^k \longrightarrow_{k\to+\infty} 0$. The above inequalities entail that $(x_k)_{k=0}^{+\infty}$ is a Cauchy sequence in $\mathbb{R}^n$. Hence there exists a point x_∞ such that $x_k \longrightarrow_{k\to+\infty} x_\infty$. Necessarily, $x_\infty = g(x_\infty)$, and x_∞ is indeed a fixed point of g. □

Let us observe that, since the final inequality (2.20) is satisfied for all $\ell \geq k$, it will be satisfied by the limit point, e.g.

$$\forall k \in \mathbb{N} : \ \|x_\infty - x_k\| \leq \frac{\|x_1 - x_0\|}{1-c} \, c^k. \tag{2.21}$$

Thus, given a threshold $\varepsilon > 0$, if the iteration is continued till

$$k \geq \frac{\ln\left(\left(\frac{1-c}{\|x_1-x_0\|}\varepsilon\right)^{-1}\right)}{\ln(c^{-1})}$$

then the approximation error done by the iteration process will be bounded by ε.

2.7.1.3 Newton's method

Let us consider the problem to find a root of an enough smooth function $f : \mathbb{R}^n \to \mathbb{R}$.

Let us assume that we are proceeding in an approximate way through a sequence $(x_k)_{k=0}^{+\infty}$ of points in $\mathbb{R}^n$ converging to a root. Then at a sufficiently large n, we must expect, due to the continuity of f, that $f(x_{n+1}) \sim 0$, and the difference between two consecutive points in the sequence is negligible. Thus, the first order approximation (2.16) around point x_k becomes $0 = f(x_k) + Df(x_k)(x_{k+1} - x_k)$ thus $Df(x_k)(x_{k+1} - x_k) = -f(x_k)$.

Hence, as a non-deterministic method, let us consider a sequence $(x_k)_{k=0}^{+\infty}$ built as follows:

$$\left.\begin{array}{l}\text{Choose } x_0 \in \mathbb{R}^n.\\ \text{For each } k \geq 0 \text{ pick } x_{k+1} \in \mathbb{R}^n \text{ such that } Df(x_k)(x_{k+1} - x_k) = -f(x_k).\end{array}\right\} \quad (2.22)$$

Proposition 2.1 (Newton's method). *If the sequence $(x_k)_{k=0}^{+\infty}$ built according to rules (2.22) converges to a limit point $x_\infty \in \mathbb{R}^n$, then this point is a root of f.*

The special case for $n = 1$ is well determined and for $f : \mathbb{R} \to \mathbb{R}$ the *Newton-Raphson method* is the following:

$$\left.\begin{array}{l}\text{Choose } x_0 \in \mathbb{R}.\\ \forall k \geq 0 : \; x_{k+1} = x_k - \left(\frac{df}{dx}(x_k)\right)^{-1} f(x_k).\end{array}\right\} \quad (2.23)$$

Another special case is for $n \geq 1$ and $f : \mathbb{R}^n \to \mathbb{R}^n$. Let us write $f = (f_1, \ldots, f_n)$. Then at any point $x \in \mathbb{R}^n$ the *differential* is represented by the *Jacobian matrix*

$$Jf(x) = Df(x) = [\partial_j f_i]_{i,j \in [\![1,n]\!]}.$$

The *Newton-Raphson method* is in this case:

$$\left.\begin{array}{l}\text{Choose } x_0 \in \mathbb{R}^n.\\ \forall k \geq 0 : \; x_{k+1} = x_k - (Df(x_k))^{-1} f(x_k).\end{array}\right\} \quad (2.24)$$

Here a common root of all component functions is obtained, or, in an equivalent form, a root for the system of equations $f_i = 0$, $i \in [\![1,n]\!]$, is obtained.

Of course, some additional conditions on the objective functions should be imposed in order to guarantee convergence. For instance, it is important to have at any point that the Jacobian matrix is non-singular, and this is just a minor condition on

the whole context of the problem. However, whenever the method converges, the convergence is at least quadratic:

$$\|x_k - x_\infty\| = O(h^{2k}), \qquad \text{for some } h \in]0,1[.$$

2.7.2 Iterative methods to solve linear systems of equations

Let us recall that a system of linear equations

$$Ax = b \tag{2.25}$$

with $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, has an unique solution $x = A^{-1}b$ provided that A is non-singular, $\det A \neq 0$. In this case the calculation of the inverse matrix A^{-1} entails a cost of $O(n^3)$ real numbers multiplications. If the matrix A has a large order, this can be very costly.

As an alternative, suppose that $B \in \mathbb{R}^{n\times n}$ is a non-singular matrix. Then any solution of the original system of linear equations (2.25) satisfies $Bx+(A-B)x = b$, and consequently $x = (\mathrm{Id}_n - B^{-1}A)x + B^{-1}b$, where Id_n is the $(n\times n)$-identity matrix. Hence the solution of (2.25) is a fixed point of the operator

$$g_B : x \mapsto (\mathrm{Id}_n - B^{-1}A)x + B^{-1}b.$$

If $\sup_{x\in\mathbb{R}^n-\{0\}} \frac{\|(\mathrm{Id}_n - B^{-1}A)x\|}{\|x\|} < 1$ then g_B is a contraction, and the fixed point can be approximated by the sequence:

$$x_0 \in \mathbb{R}^n - \{0\} \quad , \quad \forall k \geq 0 : \; x_{k+1} = g_B(x_k).$$

The method depends on the chosen matrix B. In practice, this matrix is chosen in dependence of matrix A. Namely, let $D = \mathrm{diag}\,[a_{11} \; \cdots \; a_{nn}]$ be the diagonal $(n\times n)$-matrix whose non-zero values are those appearing at the diagonal of A, let $L = [\ell_{ij}]_{i,j\in[\![1,n]\!]}$ be the lower triangular matrix consisting of the lower triangular block of matrix A: $(i > j \;\Rightarrow\; \ell_{ij} = a_{ij})$; $(i \leq j \;\Rightarrow\; \ell_{ij} = 0)$, and let $U = [u_{ij}]_{i,j\in[\![1,n]\!]}$ be the upper triangular matrix consisting of the upper triangular block of matrix A: $(i \geq j \;\Rightarrow\; u_{ij} = 0)$; $(i < j \;\Rightarrow\; u_{ij} = a_{ij})$. The following procedures result:

Jacobi method. Let $B = D$. Then $\mathrm{Id}_n - B^{-1}A = -D^{-1}(L+U)$.
Gauss-Seidel method. Let $B = D - L$. Then $\mathrm{Id}_n - B^{-1}A = -(D+L)^{-1}U$.

The inversion of diagonal and triangular matrices can be computed efficiently. Thus, the iterative methods would be more efficient than the calculation of the inverse matrix.

2.7.3 Discretization of differential equations

A *differential equation* of order k has the form

$$F\left(x, y(x), y^{(1)}(x), \ldots, y^{(k)}(x)\right) = 0 \tag{2.26}$$

where $F : \mathbb{R}^{k+2} \to \mathbb{R}$ is a continuous map and for each $\kappa \leq k$, $y^{(\kappa)}(x) = \partial_\kappa y(x)$. A *solution* in a real interval $[a,b] \subset \mathbb{R}$ is a map $y : [a,b] \to \mathbb{R}$ such that for each point $x \in [a,b]$, eq. (2.26) holds.

The *Cauchy problem* consists in solving the differential equation restricted to an *initial condition*:

$$\text{Find a solution } y \text{ of eq. (2.26) such that } \left(y(a), y^{(1)}(a), \ldots, y^{(k)}(a)\right) = \mathbf{y}_0 \in \mathbb{R}^{k+1}.$$

Instead, a *problem with boundary conditions* is stated as

$$\text{Find a solution } y \text{ of eq. (2.26) such that } (y(a), y(b)) = \mathbf{y}_0 \in \mathbb{R}^2.$$

Let us introduce a *grid* in the domain:

$$a = x_0 < x_1 < \cdots < x_{m-1} < x_m = b$$

consisting of $m+1$ equally spaced points, in increasing order, with extremes in the real interval. $\forall i \in [\![0,n]\!]$, $x_i = a + i\Delta x$, with $\Delta x = \frac{1}{m}(b-a)$. Then, according to relation (2.15), the derivative of a solution y at a point x_i has as approximation:

$$dy(x_i) \approx \delta_1(y)(i) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = \frac{1}{\Delta x}(y_{i+1} - y_i). \tag{2.27}$$

Consecutively, if $\delta_{\kappa-1}(y)(i)$ is an approximation of the $(\kappa-1)$-th derivative of y at the i-th point, then an approximation of the κ-th derivative is

$$y^{(\kappa)}(x_i) = (y^{(\kappa-1)})^{(1)}(x_i) \approx \delta_1(y^{(\kappa-1)}(i) \approx \delta_1 \circ \delta_{\kappa-1}(y)(i).$$

Thus $\delta_\kappa = \delta_1 \circ \delta_{\kappa-1}$ and $\delta_\kappa = \delta_1^\kappa$, which entails

$$\delta_\kappa(y)(i) = \frac{1}{(\Delta x)^\kappa} \sum_{\iota=0}^{\kappa} \binom{\kappa}{\iota} (-1)^{\kappa-\iota} y_{i+\iota}. \tag{2.28}$$

We see that, for each κ, each point (x_j, y_j) appears in $\kappa+1$ expressions of the form (2.28). These possibilities give rise to several modifications in the discretization process.

A substitution of the approximations (2.28) into eq. (2.26) poses a linear system of equations, and the solution of that transformed system is indeed a method to approximate the solution of the original differential equation.

Problems

2.1. Write a program to solve the Eight Queens puzzle by Brute-Force as explained in example 2.4.

2.2. Write a program implementing `Activities_selection` as explained in section 2.2.1.

2.3. Write a program implementing `Greedy_Fractional_Knapsack` as explained in section 2.2.2.

2.4. Write a program implementing `Closest_Pair` as explained in section 2.3.3.

2.5. Write a program implementing `Convex_hull` as explained in section 2.3.4.

2.6. Write a program implementing `Depth_First` traversal as explained in section 2.4.1.

2.7. Write a program to solve the Eight Queens puzzle by a backtracking procedure as explained in section 2.4.4.

2.8. Write a program implementing `Graham_scan` as explained in section 2.4.5.

2.9. Write a program implementing the `Strassen_method` for matrix multiplication as explained in section 2.3.5.

2.10. Write a program implementing the `Traveling_salesman_problem` as explained in section 2.4.3.

2.11. Write a program implementing `Simulated_annealing` as explained in section 2.5.1. Test the implementation with $D = [0, 2\pi]$ and the map $f : x \mapsto \sin(x)\cos(15x)$.

2.12. Write a program implementing `Tabu_search` as explained in section 2.5.2. Test the implementation with $D = [0, 2\pi]$ and the map $f : x \mapsto \sin(x)\cos(35x)$.

2.13. Write a program implementing the research with an automaton as explained in section 2.6.2.

2.14. Write a program implementing the `Knuth-Morris-Pratt algorithm` as explained in section 2.6.3.

2.15. Write a program implementing the `Suffix-Tree Construction` as explained in section 2.6.4.

2.16. Write a program implementing the search of a fixed point according to Banach's Theorem (see theorem 2.1). Test the implementation with $[0, \frac{\pi}{2}]$ and the map $f : x \mapsto \cos(x)$.

2.17. Write a program implementing the `Newton_Raphson method` as explained in section 2.7.1.3. Test the implementation with $D = [0, 2\pi]$ and the map $f : x \mapsto \sin(x)\cos(15x)$.

2.18. Write a program implementing the iterative method to solve systems of linear equations as explained in section 2.7.2 using the Jacobi method.

2.19. Write a program implementing the iterative method to solve systems of linear equations as explained in section 2.7.2 using the Gauss-Seidel method.

Chapter 3
Fundamental Algorithms

Abstract In this chapter we expose succinctly classical algorithm that serve as prototypes of the general design lines in the previous chapter. In numerical algorithms we pose the interpolation problem and the solution given by Legendre polynomials and by solving systems of linear equations. The queries to databases entail localization of given registers. The sequential search solve this problem in linear time while the binary search works in logarithmic time. Typical sorting algorithms work on quadratic time, but by using a divide-and-conquer approach sorting can be done in $n \lg n$ time through Quicksort. Hash tables help to store information in compact forms, and binary trees are also storage devices structuring the information in such a way to reduce, to logarithmic costs, the most basic relational operations. The final part of the chapter is devoted to Graph Theory problems with wide applications. The final part of the chapter is devoted to Graph Theory problems with wide applications. Here, the breadth-first and depth-first traversal help to solve important problems as calculation of shortest-paths and of minimum spanning trees.

3.1 Simple numerical algorithms

The *interpolation problem* is the following:

Problem `Polynomial interpolation`.

Instance: A set of $n+1$ points in the plane $\{(x_i, y_i)\}_{i=0}^{n}$, with pairwise distinct abscissae.

Solution: The n degree polynomial $p(X) = \sum_{j=0}^{n} a_j X^j$ such that $p(x_i) = y_i$ for $i \in [\![0, n]\!]$.

For instance, for $n = 1$, the straight line joining (x_0, y_0) and (x_1, y_1) has parametric equation $r(t) = t(x_1, y_1) + (1-t)(x_0, y_0) = (x_0 + t(x_1 - x_0), y_0 + t(y_1 - y_0))$. Thus, any point $(x, y) = r(t)$ in that line should satisfy $\frac{x-x_0}{x_1-x_0} = \frac{y-y_0}{y_1-y_0}$, or

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0.$$

Thus the expression at the right hand side is the linear polynomial interpolating the two given points.

In order to solve the Interpolation Problem let us consider the

Lagrange method. Suppose that $\ell_i(X)$ is a n degree polynomial such that

$$\forall j \in [\![0,n]\!] : \ [j \neq i \implies \ell_i(x_j) = 0] \ \& \ [j = i \implies \ell_i(x_i) = 1] . \tag{3.1}$$

Then clearly the sought polynomial is

$$p(X) = \sum_{i=0}^{n} y_i \ell_i(X) \tag{3.2}$$

Relations (3.1) and (3.2) determine an algorithm to solve the Interpolation Problem.

The polynomials satisfying relation (3.1) are

$$\ell_i(X) = \frac{\prod_{j \in [\![0,n]\!]-\{i\}} (X - x_j)}{\prod_{j \in [\![0,n]\!]-\{i\}} (x_i - x_j)} = \prod_{j \in [\![0,n]\!]-\{i\}} \frac{X - x_j}{x_i - x_j}$$

Vandermonde method. Since it is required

$$\forall j \in [\![0,n]\!] : \ y_j = p(x_j) \tag{3.3}$$

then, as an equation system, these equations can be stated jointly as

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{n} a_j x_0^j \\ \sum_{j=0}^{n} a_j x_1^j \\ \vdots \\ \sum_{j=0}^{n} a_j x_n^j \end{bmatrix} = \begin{bmatrix} 1 & x_0 & \cdots & x_0^n \\ 1 & x_1 & \cdots & x_1^n \\ \vdots & \vdots & & \vdots \\ 1 & x_n & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}$$

thus we have a system $u = Va$, with solution $a = V^{-1}u$, provided that the matrix V is non-singular. One can see that the matrix V is indeed non- singular because

$$\det V = \prod_{0 \leq i < j \leq n} (x_j - x_i).$$

However this value could be very close to zero if the distance among consecutive abscissae is smaller than 1. This renders the problem rather ill-conditioned.

In section 2.7 some other numerical algorithms have been discussed.

3.2 Sequential and binary search algorithms

For a given finite sequence A of objects of the same type, and a query object x it is asked to decide whether x appears in A and if it appears indeed, then to locate its position in the sequence.

The *sequential search* consists in checking consecutively whether each element in the array coincides with the query object x. The procedure, shown at figure 3.1, is quite direct.

`SequentialSearch`
Input. An array $A[1:n]$, and a query value x.
Output. The position at A in which x occurs, or an indication that it does not occur, otherwise.

1. $p_{\text{curr}} := 0$;
2. Stop := False ;
3. While Not(Stop) && $p_{\text{curr}} <$ len(A) Do
   a. $p_{\text{curr}}++$;
   b. Stop := $(x == A[p_{\text{curr}}])$;
4. If Stop Then Output p_{curr}
   Else Output 'x does not occur in A'

Fig. 3.1 Procedure `SequentialSearch`.

Let us assume that for each entry $i \in [\![1,n]\!]$ at the array, p_i is the probability that the query object x coincides with $A[i]$. Hence $W = (p_i)_{i=1}^n$ is a probability distribution on A. Let χ be the random variable $\chi : x \mapsto i$, that for each query object gives the number of tests in `SequentialSearch` realized to find it in the array. The expected number of tests is thus

$$E(\chi) = \sum_{i=1}^{n} i\, p_i \in [1,n],$$

which is an average of the indexes weighted by the probabilities. If we perform a permutation of the entries in A, let $E(\chi,\pi) = \sum_{i=1}^n i\, p_{\pi(i)}$ be the corresponding expected number of tests for the permuted array. Then the minimum value corresponds to the permutation that sorts the probability distribution in a decreasing way, while the permutation arranging the probability increasingly gives the greatest expected values. This means that if the array is arranged with the most probable elements for query at the beginning then the expected number of tests will be diminished.

On the other side, if the objects in the array are in an ordered set and the array is sorted monotonically with respect to that order, then the search of a query element can be done in a manner much more efficient.

Indeed, *binary search* is a divide-and-conquer algorithm. Suppose that the array is sorted in an increasing way. Given a query object x, initially let us consider the

whole array A with inferior extreme index $i_m = 1$ and superior index $i_M = n$. At any stage of the algorithm, compare x with the element at the middle of the array in order to decide whether the element should appear at the inferior or at the superior segment of A and continue the search with that subsegment. The rule is thus the following: Let $i_{mid} := \texttt{Ceiling}\left(\frac{i_m+i_M}{2}\right)$,

$$\begin{aligned} x < A[i_{mid}] &\Longrightarrow \text{let } i_M := i_{mid} \\ x = A[i_{mid}] &\Longrightarrow \text{Output "Found } x \text{ at entry } i_{mid}\text{"} \\ x > A[i_{mid}] &\Longrightarrow \text{let } i_m := i_{mid} \end{aligned}$$

Thus according with the time estimations performed at section 2.3 the expected number of tests in finding x is $O(\log n)$.

3.3 Quadratic sorting algorithms

When sorting an array of items in an ordered set, the most intuitive algorithm is a round-robin comparison among its entries as shown in figure 3.2.

`Round-robinSort`
Input. An array $A = A[1:n]$.
Output. The sorted array A.

1. For $i = 1$ to $n-1$ do
   a. For $j = i+1$ to n do
      i. If $A[i] > A[j]$ Then switch $A[i]$ and $A[j]$;
2. Output A

Fig. 3.2 Procedure `Round-robinSort`.

The number of comparisons, realized at step 1.a.i., is

$$\sum_{i=1}^{n-1}(n-i) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2).$$

Thus it has a quadratic time complexity.

`InsertionSort` discussed at section 1.1.3 has also quadratic time complexity.

3.4 Quicksort type algorithms

Quicksort is the typical example of a divide-and conquer algorithm:

Divide Given an array A split it into two subarrays $A = A_1 \cup A_2$ such that each entry at A_1 is not greater than any entry at A_2.

Conquer Sort each subarray A_i, $i = 1,2$, using the current procedure.

No merge is required due to the "divide" condition. The procedure is sketched at figure 3.3.

`Quicksort`
Input. An array A and extreme indexes $1 \leq p \leq r \leq n$
Output. The subarray $A[p \cdots r]$ sorted.

1. If $p < r$ Then
   a. $q :=$ `SplittingIndex`(A, p, r) ;
   b. `Quicksort`(A, p, q) ;
   c. `Quicksort`$(A, q+1, r)$;

Fig. 3.3 Procedure `Quicksort`.

There are several criteria to choose the splitting index, at step 1.a., and they depend on the selection of a *pivot*. Namely, given a pivot x in the range of A, let $J_1 = \{j |\ A[j] < x\}$ and $J_2 = \{j |\ A[j] \geq x\}$. Clearly, when recognizing J_1, the elements of A may be switched if necessary in order to guarantee the condition in the "divide" step. The splitting index is thus $q = p + \text{card}\,(J_1)$.

Each criterion for *pivot selection* produces a variant of Quicksort. Among the criteria, there are the following:

- the first entry $A[1]$,
- the middle entry $A\left[\left\lceil \frac{p+r}{2} \right\rceil\right]$,
- the entry in "golden ratio" proportion $A\left[\lceil p + \gamma(r-p) \rceil\right]$, where $\gamma = \frac{\sqrt{5}-1}{2}$,
- the middle value of extremes $\frac{A[p]+A[r]}{2}$,
- the value in "golden ratio" proportion of the extremes $A[p] + \gamma(A[r] - A[p])$ if $A[r] \geq A[p]$, or $A[r] + \gamma(A[p] - A[r])$ otherwise,
- etc.

Quicksort will show better performances whenever the splitting index is closer to the middle number of current entries $\frac{p+r}{2}$ and the probabilistic distribution of the values at the input array will determine whether this condition is fulfilled. In best and average cases, the time complexity of Quicksort is $\Theta(n \log n)$.

3.5 Hash tables, including collision-avoidance strategies

A *hashing* is a map $h : A \to B$ from a set A of strings of large length into a set B of strings of short length. For instance, let A be the set of full names of citizens in a country, and let B be the set of social security numbers in the USA or the

certificado único de registro de población (CURP) in Mexico. Usually the names of persons are long strings, while the social security numbers or CURP's are short. The correspondence among citizens and these identifiers is a hash function. In database management, the key attributes in a database define a hash function: The whole register can be identified by its value in a key attribute. Conversely, whenever there is a hash function $h : R \to B$, from a database into a set B, then B can be seen as the domain of a key attribute defined by the values of h.

Although there cannot be a one-to-one correspondence $A \to B$, the maps that "seem" to be injective can be very useful. A *collision* in a hashing consists of a pair $a_1, a_2 \in A$ such that they have the same hash value, $h(a_1) = h(a_2)$.

For purposes of efficient information storage or digital signatures, hash functions with low collision probability in a well defined domain are very important. Let us recall some general constructions of hash functions and later some collision-avoidance strategies.

Suppose $\operatorname{card}(A) = n$ and $\operatorname{card}(B) = m$. We look up for a map $h : A \to B$ satisfying $\forall b \in B : \Pr\{a \in A | h(a) = b\} \approx 1/m$.

An elementary method is the following: let $(r_j)_{j=1}^n$ be a sequence of random numbers with uniform distribution in $[0,1[$ and let $h : a_j \mapsto \lfloor r_j \cdot m \rfloor$.

As a second elementary example, let us assume $A, B \subset \mathbb{N}$. Let p be a prime number close to m and let $h : k \mapsto k \bmod p$.

As a last elementary example, let $\cdot \bmod 1 : x \mapsto x - \lfloor x \rfloor$ be the *fractionary part map*, and let $\gamma \in]0,1[$ be a fixed real number, mostly it is taken the golden ratio $\gamma = \frac{1}{2}(\sqrt{5}-1)$. In this case, let $h : k \mapsto \lfloor m((k\gamma) \bmod 1) \rfloor$. Observe here, that m does not essentially influence in the definition of h. It appears just as a parameter, hence h actually may be considered as a whole family of hash functions.

Let $\mathscr{H} \subset [\![0, m-1]\!]^A$ be a finite family of hash functions. $\mathscr{H}$ is called an *universal hash family* if

$$\forall a_1, a_2 \in A \left[a_1 \neq a_2 \Rightarrow \operatorname{card}\{h \in \mathscr{H} | h(a_1) = h(a_2)\} = \frac{\operatorname{card}(\mathscr{H})}{m} \right] \tag{3.4}$$

Within an universal hash family, one can have a hash map $h : x \mapsto h_x(x)$, where $h_x \in \mathscr{H}$ is chosen randomly for each $x \in A$. Obviously, it is necessary to record the used map h_x for each x.

Remark 3.1. If $\mathscr{H}$ is universal, then for each $a \in A$ the expected value of the number of collisions involving a is $\frac{n-1}{m}$.

Proof. For each $u, v \in A$, $u \neq v$, let $c_{uv} = 1$ if $h(u) = h(v)$, or $c_{uv} = 0$ otherwise; and let $C_u = \sum_{v \neq u} c_{uv}$. Then $E(c_{uv}) = \frac{1}{m}$ and $E(C_u) = \sum_{v \neq u} E(c_{uv}) = \frac{n-1}{m}$. □

A direct method for *collision-avoidance* is the following: Suppose an $a \in A$ is given and let $b = h(a)$. If there has appeared $a_1 \in A$ such that $b = h(a_1)$ then consider $b_1 = b + 1$. If there has appeared $a_2 \in A$ such that $b_1 = h(a_2)$ then consider $b_2 = b_1 + 1$. And so on. The first time that a b_k is found such that it has not been taken by previous a's then let us associate this value to the current a. In a more general way,

let $f : \mathbb{N} \to \mathbb{Z}$ be a one-to-one map. Then if the value $b = h(a)$ has been taken by a previous a_1, the associate to a the value $b + f(k)$, where k is the minimum index such that $b + f(k)$ has not been assumed formerly.

As another direct possibility, a tagging could be used in order to distinguish different objects in A colliding into the same hash values.

3.6 Binary search trees

A *binary tree* is either an *empty tree* or a *leaf* or a *root* which is the parent of a *left* binary tree and a *right* binary tree. In this last case, the root of the left subtree is the *left child* and the root of the right subtree is the *right child* of the parent root. Thus in any binary tree any internal node is the parent of two children. For any node x let $\text{Left}(x)$ denote the left subtree and $\text{Right}(x)$ the right subtree that have x as parent.

If n is the number of vertexes in a binary tree and k is its height then $k = O(\log n)$ and $n = O(2^k)$, i.e. the number of vertexes is exponential with the height and the height is logarithmic with the number of vertexes. Besides in a binary tree, the number of leaves is around half the total number of vertexes.

An *inorder labeling* is a map $L : \{\text{vertexes}\} \to \mathbb{Z}$ such that

$$\forall x, \forall y : \begin{cases} y \in \text{Left}(x) & \Rightarrow L(y) < L(x) \\ y \in \text{Right}(x) & \Rightarrow L(y) > L(x) \end{cases} \tag{3.5}$$

The labels may be the values of a hashing function, thus a binary tree may be used as a data structure containing the keys of the database. The elementary access functions can be efficiently performed:

Find the minimum key. Just go to the leftmost leaf.

Find the maximum key. Just go to the rightmost leaf.

Find successor. Given a vertex x, if $\text{Right}(x)$ is not empty, the `successor` of x is the minimum element in $\text{Right}(x)$; otherwise, ascend in order to find the first ancestor in the left line, the `successor` of x is this ancestor.

Find predecessor. Given a vertex x, if $\text{Left}(x)$ is not empty, the `predecessor` of x is the maximum element in $\text{Left}(x)$; otherwise, ascend in order to find the first ancestor in the right line, the `predecessor` of x is this ancestor.

Search a query key. Compare the query key z with the root x. If $z < L(x)$ then search z in $\text{Left}(x)$, if $z = L(x)$ then exit successfully the search, and if $z > L(x)$ then search z in $\text{Right}(x)$. A search procedure in an empty tree is always a failure process.

The time complexity is proportional with the height, thus it is logarithmic with the length of the input array.

Insertion of a labelled node. Given a key z and a binary tree T, consider z as a tree consisting just of one leaf. Find the position, using tree search, at which z should be at T. If x is the node to act as parent of z then define the pointers as it was necessary.

Deletion of a labelled node. Given a vertex x for deletion in a binary tree T, let us consider three cases according to the number of children of x in T. If x has no children, i.e. it is a leaf, just suppress it and redefine the pointer to x of its parent. If x has one child, just overskip it, i.e. suppress it and redefine the pointers of its parent and its child. If x has two children then its right son y should be the left son of `successor`(y).

The binary trees entail thus procedures with expected logarithmic time complexities. However, the worst cases may maintain linear complexities.

An elaborated technique is given by the *red-black trees* that maintains all trees balanced, and consequently even the worst case cases have logarithmic complexity, but this technique is out of the scope of the current presentation.

3.7 Representations of graphs

A *graph* is a structure $G = (V,E)$ where V is a non-empty set of *vertexes* or *nodes* and E is a set of *edges*. If $E \subset V^{(2)}$, i.e. the edges are 2-sets of vertexes, the graph G is called *undirected*, while if$E \subset V^2$, i.e. the edges are ordered pairs of vertexes, the graph G is called *directed* or a *digraph*. In this case, if $(x,y) \in E$ is an edge, x is called the *tail* of the edge and y its *arrow*. In general, the sole term graph will refer to an undirected graph.

A *weighted graph* is $G = (V,E,w)$ where V is a set of vertexes, E is a set of edges and $w : E \to \mathbb{R}$ associates a *weight* $wa \in \mathbb{R}$ to each edge $a \in E$.

In a graph $G = (V,E)$, the *degree* or *valency*, $\partial(v)$, of a vertex $v \in V$ is the number of edges incident to v. A vertex of degree zero is an *isolated vertex*.

In a digraph $G = (V,E)$, the *in-degree* $\partial^+(v)$ of a vertex $v \in V$ is the number of edges in which v acts as arrow, while the *out-degree* $\partial^-(v)$ of v is the number of edges in which v acts as tail.

A *path* in a graph $G = (V,E)$ is a sequence $v_0, \ldots, v_k$ such that $\forall i \leq k$: $\{v_{i-1}, v_i\} \in E$, i.e. any pair of consecutive vertexes is an edge. A path is a *cycle* if its extreme points coincide $v_k = v_0$.

For any vertex $v \in V$, the *connected component* of v is the set $\mathrm{Con}_G(v)$ consisting of the extreme vertexes of paths starting at v. The graph is *connected* if for any pair of vertexes there is a path connecting them. The graph is connected if it is the connected component of any of its vertexes.

3.7.1 Adjacency-lists

Let $G = (V,E)$ be a graph. For each vertex $v \in V$, its *adjacency-list* is $\mathrm{Adj}(v) = [u]_{\{v,u\} \in E}$. Thus $\mathrm{Adj}(v) \in V^*$. If G is weighted, then the adjacency-list is $\mathrm{Adj}(v) =$

$[(u, w(\{v,u\}))]_{\{v,u\}\in E}$ and has type $(V \times \mathbb{R})^*$. The length of $\mathrm{Adj}(v)$ is the degree $\partial(v)$ of v. Thus the whole graph can be represented by a structure of type $(V^*)^*$.

For a directed graph $G = (V,E)$ its adjacency-list is $\mathrm{Adj}(v) = [u]_{(u,v)\in E}$. The length of $\mathrm{Adj}(v)$ is the in-degree $\partial^+(v)$. The calculation of out-degrees is rather expensive since it is necessary to check all adjacency-lists.

3.7.2 Adjacency-matrices

Let $G = (V,E)$ be a graph. The *adjacency-matrix* of G is

$$M_G = [m_{vu}]_{v,u\in V} \qquad , \qquad m_{vu} = \begin{cases} 1 & \text{if } \{v,u\} \in E \\ 0 & \text{if } \{v,u\} \notin E \end{cases}$$

Thus, $M_G \in \{0,1\}^{n\times n}$, where $n = \mathrm{card}\,(V)$ is the number of vertexes in the graph, and the number of 1's appearing in M_G is $2m$ where $m = \mathrm{card}\,(E)$ is the number of edges in the graph. Let us remark that M_g is a symmetric matrix and the entries at the diagonal are all zero. Thus, instead of storing the whole matrix M_G it is worth to store just the $\frac{1}{2}n(n-1)$ entries above the main diagonal.

The adjacency matrix of a digraph is defined similarly, but it is not necessarily symmetric.

3.8 Depth- and breadth-first traversals

Let $G = (V,E)$ be a graph. Given a vertex $v \in V$ it is required to calculate $\mathrm{Con}_G(v)$.

In order to solve this problem, a representation by adjacency-lists will be used, as well as a coloring of vertexes in three colors: white, gray and black.

Initially all vertex are painted in white. A vertex is *discovered* the first time it is painted non-white. The calculation process will keep an *invariant condition*:

> If $(u,v) \in E$ is an edge and u is black, then v must be either gray or black, i.e. no pair white-black is allowed to be adjacent. In other words, the adjacent vertexes to black colors shall be discovered already.

The invariant condition states that at any time the white part has not yet been discovered, the black region has been discovered and the gray region separates those former regions.

The procedure uses a *queue* working in a principle "First-In–First-Out". It is sketched in figure 3.4.

The steps 3. and 4.d essentially just paste the adjacency-lists of s and u.

The procedure determines a tree structure in the connected component of the source vertex s and it is indeed traversing the tree in a *breadth-first traversal*.

`BreadthFirst`
Input. A graph $G=(V,E)$ as a list of adjacency-lists and a *source* vertex $s \in V$.
Output. The connected component $\mathrm{Con}_G(v)$.

1. Initially paint the source point gray ;
2. let $d(s) := 0$;
3. enqueue all pairs $(s,v) \in E$;
4. Repeat
   a. dequeue an edge $\{v,u\}$ painted (gray,white) ;
   b. paint it (black, gray) ;
   c. declare v as the parent of u ; let $d_u := d_v + 1$;
   d. enqueue all pairs $(u,w) \in E$ with w white ;

   until the queue is empty ;
5. Output the black region

Fig. 3.4 Procedure `BreadthFirst`.

In contrast, a depth-first traversal proceeds using backtracking and a stack, working in a principle "First-In–Last-Out", as an auxiliary storage device. In section 2.4 we have seen several examples of depth-first traversals.

Let $G=(V,E)$ be a digraph. A *precedence map* is a function $\pi : V \to V$ such that

$$\forall v \in V : \ \pi(v) \neq \text{nil} \ \Rightarrow \ (\pi(v), v) \in E.$$

For a precedence map π, let $G_\pi = (V, E_\pi)$ be the digraph with edges $(\pi(v), v)$, $v \in V$. G_π is a subgraph of G, probably not-connected, hence it is a *forest*, and it is called a *depth-first forest*.

In order to build such a forest, let us use, as before, a vertex coloring with three colors: white, gray and black. Besides, as map d before, let us use two *time marks*: d_v is the discovering time of vertex v, while f_v is the moment when its adjacency-list has been exhausted and v is turning black. In figure 3.5 we show an algorithm for calculation of a depth-first forest.

`DepthFirst`
Input. A digraph $G=(V,E)$ as a list of adjacency-lists.
Output. A predecessor map and the corresponding depth-first forest.

1. For each $v \in V$ Do $\{color(v) := white \ ; \ \pi_v := \text{nil}\}$;
2. $time := 0$;
3. For each $v \in V$ Do If $color(v) := white$ Then `TraverseDepthFirst`(v) ;
4. Output maps π, d and f

Fig. 3.5 Procedure `DepthFirst`.

The algorithm `DepthFirst` has an initialization process and thereafter it calls the recursive program `TraverseDepthFirst` displayed in figure 3.6.

`TraverseDepthFirst`
Input. A digraph $G=(V,E)$ and a vertex $v \in V$.
Output. Depth-first traversal of the tree rooted at v.

1. $color(v) := gray$;
2. $time++$; $d_v := time$;
3. For each $u \in \mathrm{Adj}(v)$ Do
   a. If $color(u) := white$ Then $\{\pi_u := v$; `TraverseDepthFirst`$(u)\}$;
   b. $color(u) := black$;
4. $time++$; $f_v := time$

Fig. 3.6 Procedure `TraverseDepthFirst`.

Observe that steps 3. at `DepthFirst` and 3. at `TraverseDepthFirst` are realizing a backtracking procedure. In `TraverseDepthFirst`, the steps 2. and 4., the maps d and f are defined, while the predecessor map π is actually defined at step 3.a.

The whole process has time complexity $\Theta(\operatorname{card}(V)+\operatorname{card}(E))$.

3.9 Topological sort

Let $G=(V,E)$ be a directed acyclic graph, a *dag* for short. A *topological sort* is an enumeration $e : V \to [\![1,n]\!]$ such that

$$\forall (u,v) \in E :\ e(u) < e(v).$$

If we write $v_i = v$ whenever $i = e(v)$, then above condition means

$$\forall i,j \in [\![1,n]\!] :\ [(v_i,v_j) \in E \implies i<j.] \tag{3.6}$$

This condition entails that the graph may be drawn in a horizontal line and all vertices are going form left to right. Obviously, a digraph with cycles cannot posses a topological sort.

In order to construct a topological sort we may use the `DepthFirst` procedure shown at figure 3.5. Namely, apply this algorithm on $G=(V,E)$ in order to obtain the predecessor map π, the initial-times map d and the ending-times map f of a depth-first traversal of the graph G. Then a sorting of the set of vertexes V according to f defines a topological sort in G.

Consequently, the topological sort can be obtained in time complexity $\Theta(\operatorname{card}(V)+\operatorname{card}(E))$.

3.10 Shortest-path algorithms

Let $G = (V,A)$ be a graph, and for any two vertexes $v,u \in V$ let

$$\delta(v,u) = \min\{n | \exists c = [u_0,\ldots,u_n] \text{ path } : u_0 = v, u_n = u\}.$$

$\delta(v,u)$ is the distance of the *shortest path* connecting v with u in G. $\delta(v,u) = \infty$ if there is no a path from v to u.

Lemma 3.1. *The following propositions hold in a graph:*

1. *If $(v,u) \in E$ the for any vertex w: $\delta(v,w) \leq \delta(u,w)+1$.*
2. *If d_u is the value calculated at* `BreadthFirst`(G,v)*, then*

$$\forall u \in V - \{v\} : \; d_u \geq \delta(v,u).$$

3. *Let us assume that at a given moment, the queue is $[u_1,\ldots,u_k]$ in* `BreadthFirst`*. Then,*

$$\begin{aligned} d_{u_k} &\leq d_{u_1} + 1 \\ \forall i < k : \; d_{u_i} &\leq d_{u_{i+1}} \end{aligned}$$

From here, we have:

Theorem 3.1 (`BreadthFirst` **correctness).** *In any graph $G = (V,E)$, the algorithm* `BreadthFirst`(G,v) *discovers all vertexes connected with v in G, for each $u \in Con_G(v)$, the value d_u will equal $\delta(v,u)$ and the shortest path from v to u can be recovered from the tree structure introduced in $Con_G(v)$.*

The most used procedure for shortest paths calculation is *Dijkstra's algorithm*. Given a directed weighted graph $G = (V,E,w)$, with non-negative weights, and a source vertex $s \in V$ it gives the distance of the shortest path from s to any other vertex in V (if a vertex is not connected with s by a path then this distance is ∞). The algorithm proceeds inductively: if a vertex x is not yet reached from s, then the shortest path connecting x with s is the path passing through an already reached intermediate point y minimizing the addition of the shortest path from s to y and the weight of the connection between y and x. The algorithm is displayed in figure 3.7, which has quadratic time complexity.

3.11 Transitive closure

The *Floyd-Warshall algorithm*, in contrast with Dijkstra's algorithm, produces the shortest path among every pair of vertexes.

In a weighted digraph $G = (V,E,w)$, with non-negative weights, $n = \text{card}\,(V)$, for a $(n \times n)$-matrix $D = [d_{ij}]_{i,j \leq n}$, representing current distances among pairs of points, a *triangle operation* involving three vertexes is

`DijkstraAlgorithm`
Input. A weighted digraph $G = (V,E,w)$, with non-negative weights, and a source vertex $s \in V$.
Output. The distance of the shortest path from s to any other vertex in V.

1. Reached $:= \{s\}$; $\rho(s) := 0$;
2. For each $x \in V - \{s\}$ Do $\rho(x) := w(s,x)$;
3. While Reached $\neq V$ Do
   a. let $x :=$ arg_min$\{\rho(y) |\ y \in V -$ Reached$\}$;
   b. Add x into Reached ;
   c. For each $y \in V -$ Reached Do $\rho(y) := \min\{\rho(y), \rho(x) + w(x,y)$;
4. Output the array ρ

Fig. 3.7 Procedure `DijkstraAlgorithm`.

$$\texttt{Triangle}(D;i,j,k) \ \equiv \ (d_{ik} := \min\{d_{ik}, d_{ij} + d_{jk}\}) \quad \text{whenever } i,k \neq j, \quad (3.7)$$

with the connotation: *if the weight diminishes, replace the segment from v_i to v_k by two segments through the intermediate point v_j.*

If the triangle operation is iterated over all j's and realized for all other pairs i,k, beginning with the weights at the graph, then the distances will take the values of the shortest paths. The algorithm is displayed in figure 3.8, which has time complexity $O(n(n-1)^2)$.

`FloydWarshallAlgorithm`
Input. A weighted digraph $G = (V,E,w)$, with non-negative weights.
Output. A matrix $D \in \mathbb{R}^{n \times n}$ giving the distances of the shortest paths among vertex pairs in V.

1. For $i = 1$ To n Do For $k = 1$ To n Do
   If $i == k$ Then $d_{ii} := \infty$ Else $d_{ik} := w(v_i, v_k)$;
2. For each $j = 1$ To n Do
   a. For each $i \in [\![1,n]\!] - \{j\}$ Do
      i. For each $k \in [\![1,n]\!] - \{j\}$ Do `Triangle`$(D;i,j,k)$;
3. Output the array D

Fig. 3.8 Procedure `FloydWarshallAlgorithm`.

By keeping track of the changes made by the triangle operation at step 2.a.i, as defined by relation (3.7), then it is possible to recover the actual shortest path among any pair of vertexes.

Given a digraph $G = (V,E)$, its *transitive closure* is the graph $G^* = (V,E^*)$ such that

$$\forall i,j \in V : \ [(v_i,v_j) \in E^* \Longleftrightarrow \exists [u_0,\ldots,u_k] \text{ path} : u_0 = v_i \ \& \ u_k = v_j] \quad (3.8)$$

The transitive closure is thus the graph of paths in G.

If M_G is the adjacency matrix of G, then the Boolean power $M_G^2 = M_G \cdot M_G$ represents the graph of paths of length 2, $M_G^3 = M_G^2 \cdot M_G$ represents the graph of paths of length 3, and so on. Since there exists a power $k \in \mathbb{N}$ such that $M_G^{k+1} = M_G^k$ we have $M_{G^*} = M_G^k$. This provides an effective, although expensive, procedure to compute the transitive closure of G.

The Floyd-Warshall algorithm can be used as an alternative to compute G^*. Namely, let us assign a weight $w(v_i, v_j) = 1$ to each edge $(v_i, v_j) \in E$ and an infinite weight to pairs not belonging to E. Then, after applying the Floyd-Warshall algorithm, we have:

$$\forall i, j \in V : \ [(v_i, v_j) \in E^* \Longleftrightarrow d_{ij} < n] .$$

3.12 Minimum spanning tree

Let $G = (V, E, w)$ be a weighted graph. A *tree* $T = (U, F)$ within G is a subgraph, $U \subseteq V$, $F \subseteq E$, connected without cycles. A *spanning forest* of G is a collection of trees $\{(U_i, F_i)\}_{i=1}^k$ within G such that $\{U_i\}_{i=1}^k$ is a partition of V, i.e. $V = \bigcup_{i=1}^k U_i$ and $[i \neq j \Rightarrow U_i \cap U_j = \emptyset]$. A *spanning tree* of G is a subtree $T = (U, F)$ such that $U = V$, i.e. $\{T\}$ is a spanning tree. The *weight* of a spanning tree is the sum of the weight of edges in F, $w(T) = \sum_{\{v,u\} \in E} w(v, u)$.

The *minimum spanning tree* problem (MST) consists in finding the spanning tree of minimum weight:

$$\text{Find } T_m = \text{arg\_min}\{w(T) |\ T \text{ spanning tree in } G\}.$$

This is a combinatorial problem whose search space is extremely huge, $O(n^{n^2})$. Thus any brute-force approach is unrealistic.

However, a clever remark allows to solve this problem efficiently:

Proposition 3.1. *If $\{(U_i, F_i)\}_{i=1}^k$ is a spanning forest in G and $\{v, u\} \in E$ is a lowest weight edge with only one extreme in the first tree U_1, then among the spanning trees containing $F = \bigcup_{i=1}^k F_i$ there is one optimal, i.e. with least weight, containing the $\{v, u\}$.*

Although it is not hard to prove this proposition, we skip the proof (the interested reader may consult [Papadimitriou and Steiglitz(1998)]).

The resulting algorithm is direct: Starting with the forest consisting of trees formed by the vertices with no edges, $\{(\{v\}, \emptyset)\}_{v \in V}$, choose the edge with minimum weight and initiate a tree with this edge. Thereafter to having the current tree, find the minimum weight edge with just an extreme in this tree and extend the tree. The algorithm is displayed in figure 3.9, which has time complexity $O(n^2)$.

`PrimMST`
Input. A weighted graph $G = (V, E, w)$, as a distance matrix $W = [w_{ij}]_{ij}$.
Output. A spanning tree of minimal weight.

1. let $\{v, u\} \in E$ be the minimal weight edge ;
2. $U := \{v\}$; $F := \emptyset$;
3. For each $u \in V - U$ Do $\text{closest}_U(u) := v$;
4. While $U \neq V$ Do
   a. let $v := \text{arg\_min}\{d(u, \text{closest}_U(u)) |\ u \in V - U\}$;
   b. $U := U \cup \{v\}$; $F := F \cup \{\{v, \text{closest}_U(v)\}\}$;
   c. For each $u \in V - U$ Do
      If $d(u, \text{closest}_U(u)) > d(u, v)$ Then $\text{closest}_U(u) := v$;
5. Output (U, F)

Fig. 3.9 Procedure `PrimMST`.

Chapter 4
Distributed Algorithms

Abstract This is a short introduction to distributive algorithms. The area is a topic that requires a complete textbook by itself. However we would illustrate some basic procedures for analysis and design of distributive algorithms. First, we sketch the most used connectivity topologies and the formal tools to analyze distributive procedures. Then we proceed to examine concurrency and the typical access for data input and output into a shared memory, and we compare several models. Finally we overview routing algorithms and the solution of Graph Theory and Flowing problems within the context of distribution.

4.1 Distributed systems

A *distributed system* consists of a set of processors with a communication platform forming a network. The connection may be physical or just logical: A set of processes are connected through a message passing mechanism.

In this chapter we will sketch basic notions of distributed computing, and we refer the interested reader to extensive texts in this area, e.g. [Schnitger(2007)].

4.1.1 Networks

A *network* is properly a graph $G = (V, E)$. The graph is undirected if the communications along the edges is *bidirectional* and the graph is directed if just one-way communications are allowed. Let us recall that the *degree* of a node is the number of its adjacent neighbors. The *degree* of a graph is the maximum degree of its vertexes, the *diameter* of G is the maximum distance among any two vertexes on the graph and the *bisection width* is the minimal number of edges to be removed in order to split G into two disjoint subgraphs $G_0 = (V_0, E_0)$, $G_1 = (V_1, E_1)$ such that $V = V_0 \cup V_1$ and $|\operatorname{card}(V_1) - \operatorname{card}(V_0)| \leq 1$.

4.1.1.1 Tree networks

Let T_0 be the tree consisting of just one vertex. This vertex is the root as well as the leaf of T_0 and it has no edges.

For $k \geq 1$, let T_k be the tree consisting of a root with exactly two children: each being the root of a tree isomorphic to T_{k-1}.

Thus T_k is the *complete binary tree network*. In T_k each vertex is able to communicate with any neighbor vertex: either its parent or its children. It has height k, $2^{k+1}-1$ vertexes and 2^k leaves. Each vertex is either a leaf or it has exactly two children. As a graph, it has diameter $2k$ and its bisection width is 1. As a network, the root of T_k is a bottleneck since in order to connect two vertexes in different copies of T_{k-1} the root necessarily should be visited.

For instance, in order to select the minimum value among an array of n values using a circuit T_k let us assign $\frac{n}{2^k}$ values to each leaf. Initially, in parallel, each leaf selects the minimum value among its assigned values. Consecutively, each node receives the minima of its two children and then it sends the minimum resulting value to its parent. The root will have the minimum value of the whole array.

This procedure entails k communication steps.

4.1.1.2 Mesh networks

Let us consider the lattice $\mathbb{Z}_m^d$ with the neighboring notion given as differences lying in the canonical basis of $\mathbb{R}^d$: x and y are neighbors if and only if $\exists i \in [\![0,d-1]\!]$, $\varepsilon \in \{-1,+1\}$: $x-y=\varepsilon e_i$. As a graph, $\mathbb{Z}_m^d$ has m^d vertexes, the degree of each vertex is at most $2d$ and its diameter is $d(m-1)$. The bisection width is upperly bounded by m^{d-1}: along any coordinate $i \in [\![0,d-1]\!]$ let E_x, $x \in \mathbb{Z}_m^{d-1}$, be the edge $\{y,z\}$ such that the i-th entry of y is $y_i = \lfloor \frac{m}{2} \rfloor$, the i-th entry of z is $z_i = \lfloor \frac{m}{2} \rfloor - 1$, and the remaining entries of y and z are given by x. For even m, the removal of all edges E_x, $x \in \mathbb{Z}_m^{d-1}$, bisects the graph.

For instance, let us consider the *odd-even transposition sort* (OETS) within a linear ($d=1$) array of m processors. Given an array of n numerical values initially each processor is provided with a $\frac{n}{m}$-length subarray. Then, in parallel, each processor sorts its subsequence using `Quicksort`. Consecutively in m steps, let us proceed as follows: at odd indexed steps, each even numbered processor p_{2i} sends its sorted sequence to its left neighbor p_{2i-1}, which merges both subsequences, keeps the smaller half and sends back the larger half to p_{2i}; at even indexed steps, each even numbered processor p_{2i} sends its sorted sequence to its right neighbor p_{2i+1}, which merges both subsequences, keeps the larger half and sends back the smaller half to p_{2i}.

OETS is accomplishing its task in $O\left(\frac{n}{m}\log\frac{n}{m}+n\right)$ computing steps and m communication steps involving messages of length $\frac{n}{m}$.

4.1.1.3 Hypercubes

Let $\mathbb{Z}_2^d$ be the d-dimensional *hypercube*. Its edges are of the form $\{x, x+e_i\}$ where $w \in \mathbb{Z}_2^d$ and e_i is a vector in the canonical basis and addition is assumed modulo 2. $\mathbb{Z}_2^d$ is a regular graph of degree d and its diameter is also d.

For each $k \leq d$, there are $\binom{d}{k} 2^{d-k}$ subgraphs isomorphic to $\mathbb{Z}_2^k$ within $\mathbb{Z}_2^d$. Hence, there are $d\,2^{d-1}$ edges in $\mathbb{Z}_2^d$.

If $A \subset \mathbb{Z}_2^d$ let $C(A)$ be the set of edges to be suppressed in order to disconnect A and its complement $A^c = \mathbb{Z}_2^d - A$. Let us observe:

$$\operatorname{card}(A) \leq \operatorname{card}(A^c) \implies \operatorname{card}(A) \leq \operatorname{card}(C(A)). \tag{4.1}$$

Namely, let us prove implication (4.1) by induction on d.

For $d = 1$, the hypothesis in (4.1) entails $\operatorname{card}(A) \leq 1$, hence the implication is obvious.

Let $d > 1$. Let us assume (4.1) for $d-1$. Let $A \subset \mathbb{Z}_2^d$ be such that $\operatorname{card}(A) \leq \operatorname{card}(A^c)$. Let A_0 consist of those points in A whose first coordinate has value 0, and let A_1 consist of those points in A whose first coordinate has value 1. Clearly, $\{A_0, A_1\}$ is a partition of A and each subset A_ε in an own copy, say Z_ε, of the $(d-1)$-dimensional hypercube $\mathbb{Z}_2^{d-1}$, for $\varepsilon \in \{0,1\}$.

If $\operatorname{card}(A_0) \leq 2^{d-2}$ and $\operatorname{card}(A_1) \leq 2^{d-2}$ then, by the induction hypothesis, $\operatorname{card}(A_0) \leq \operatorname{card}(C(A_0))$ and $\operatorname{card}(A_1) \leq \operatorname{card}(C(A_1))$. Obviously, $\operatorname{card}(C(A_0)) + \operatorname{card}(C(A_1)) \leq \operatorname{card}(C(A))$. Hence $\operatorname{card}(A) \leq \operatorname{card}(C(A))$.

Otherwise, let us assume that $\operatorname{card}(A_0) > 2^{d-2}$. Then $\operatorname{card}(A_1) < 2^{d-2}$. By the induction hypothesis, in $\mathbb{Z}_2^{d-1}$ or properly Z_1, $\operatorname{card}(A_1) \leq \operatorname{card}(C(A_1))$.

For each $x \in \mathbb{Z}_2^{d-1}$, $(0x, 1x)$ is an edge in the hypercube, thus in order to split A and its complement in $\mathbb{Z}_2^d$ we must exclude the edges such that

$$[0x \in A_0 \,\&\, 1x \notin A_1] \;\&\; [0x \notin A_0 \,\&\, 1x \in A_1]$$

as well as the splitting edges of A_1 in Z_1. Hence

$$\begin{aligned} C(A) &\geq (\operatorname{card}(A_0) - \operatorname{card}(A_0 \cap A_1)) + (\operatorname{card}(A_0^c) - \operatorname{card}(A_0^c \cap A_1)) + \operatorname{card}(A_1) \\ &= 2^{d-1} - \operatorname{card}(A_1) + \operatorname{card}(A_1). \\ &= 2^{d-1} \end{aligned}$$

Thus, $\operatorname{card}(A) \leq \operatorname{card}(C(A))$. □

As a consequence of the implication (4.1) we have:

Proposition 4.1. *The bisection width of the hypercube $\mathbb{Z}_2^d$ is 2^{d-1}.*

Thus any pair among the 2^d nodes in $\mathbb{Z}_2^d$ can communicate within a route of length at most d, in other words, the communicating length is logarithmic with respect to the number of nodes, as is the case in the complete binary networks. However here there are no bottle-necks. Instead, the "connectedness degree" is rather high in the hypercube, as seen in proposition 4.1.

Definition 4.1 (Routing procedures). Let $G = (V, E)$ be a graph. For any two nodes $u, v \in V$ let P_{uv} be the class of paths in G beginning in u and ending in v. A *routing* is a map $R : (u, v) \mapsto R(u, v) \in P_{uv}$ that associates to each pair (u, v) a path in G going from u to v. For any permutation $\pi \in S_V$ a π-*routing* is a map $\rho_\pi : u \mapsto \rho_\pi(u) \in P_{u,\pi(u)}$ giving a path, for each vertex u, going from u into $\pi(u)$. If μ_u is the message that node u should send to $\pi(u)$ then ρ_π is the route that message μ_u will follow.

In the hypercube $G = \mathbb{Z}_2^d$ a direct routing procedure acts according to the "Bit-Fixing Strategy", and it is sketched in figure 4.1.

`BitFixingRouting`
Input. A permutation $\pi \in S_V$
Output. A π-routing $\rho_\pi : u \mapsto \rho_\pi(u) \in P_{u,\pi(u)}$

1. For each $u \in \mathbb{Z}_2^d$ do (in parallel)
   a. let $\rho_\pi(u) :=$ `BitFixingStrategy`$(u, \pi(u))$
2. Output ρ_π

Fig. 4.1 Procedure `BitFixingRouting`.

In the Bit Fixing Strategy, in order to go from a node u into another node v in the hypercube $\mathbb{Z}_2^d$, the bits of u and v are compared from left to right: if there is a discrepancy then the current point is moved along that discrepancy's direction.

`BitFixingStrategy`
Input. Two nodes $u, v \in V$
Output. A path joining u with v

1. let $crpt := u$; $path := [crpt]$;
2. For each $i \in [\![1, d]\!]$ do
   a. If $crpt[[i]] \neq v[[i]]$ then
      i. $crpt := crpt + e_i$;
      ii. `AppendTo`$[path, crpt]$;
3. Output *path*

Fig. 4.2 Procedure `BitFixingStrategy`.

Let $(u_0, v_0), (u_1, v_1)$ be two pairs of vertexes. Let $\rho_0 \in P_{u_0 v_0}$, $\rho_1 \in P_{u_1 v_1}$ be paths joining the pairs. We say that there is a *path crossing* at step $i \leq d$ if $\rho_0[[i]] = \rho_1[[i]]$.

Clearly `BitFixingStrategy` will produce paths with a crossing if the pairs can be expressed as $u_0 = u_0^p w u_0^s$, $v_0 = v_0^p w v_0^s$, $u_1 = u_1^p w u_1^s$, $v_1 = v_1^p w v_1^s$, for some non-empty subwords u's and v's, and $v_0^p w u_0^s = v_1^p w u_1^s$. In this case, this common value determines a path crossing. At this crossing, `BitFixingStrategy` will produce as many skips as the length of the infix w. From here it follows that any two paths leaving a coincidence will never meet again.

`BitFixingRouting` may be quite expensive since it should manage many concurrences. For instance, let us assume that d is an even integer, $d = 2f$, and let us consider the permutation $\pi : (u,v) \mapsto (v,u)$. In the hypercube $\mathbb{Z}_2^d$ there are $2^f = 2^{\frac{d}{2}} = \sqrt{2^d}$ nodes of the form $(u,0)$. Thus when `BitFixingRouting` is applied to them, at the f-th step they will pass through $(0,0)$, the origin is thus a path crossing for all the paths and this vertex is becoming a bottleneck.

In order to avoid this difficulty, let us proceed with the random algorithm sketched in figure 4.3.

`RandomMiddle`
Input. A permutation $\pi \in S_V$
Output. A π-routing $\rho_\pi : u \mapsto \rho_\pi(u) \in P_{u,\pi(u)}$

1. For each $u \in \mathbb{Z}_2^d$ do (in parallel)
   a. select randomly a node $\alpha(u)$;
   b. as in `BitFixingStrategy` choose a path L_0 from u to $\alpha(u)$;
   c. as in `BitFixingStrategy` choose a path L_1 from $\alpha(u)$ to $\pi(u)$;
   d. let us take the concatenation $\rho_\pi(u) := L_0 * L_1$;
2. Output ρ_π

Fig. 4.3 Procedure `RandomMiddle`.

Since each vertex v in the hypercube has $\binom{n}{k}$ neighbors at distance k, $k \in [\![0,d]\!]$, the expected length of any path connecting two points is

$$\sum_{k=0}^{d} k \frac{\binom{d}{k}}{2^d} = \frac{1}{2^d} \sum_{k=1}^{d} k \frac{d!}{k!(d-k)!} = \frac{d}{2^d} \sum_{k=0}^{d-1} \frac{(d-1)!}{k!(d-1-k)!} = \frac{d}{2}.$$

Let R be a routing strategy. For each edge $a = \{u,v\}$, let W_a be the number of paths, connecting pairs $\{w, \pi(w)\}$, that pass through the edge a. Then, for each edge a, the expected value of W_a is

$$E(W_a) = \sum_{w} \frac{\frac{d}{2}}{d2^{d-1}} = \frac{d}{2} \frac{2^d}{d2^{d-1}} = 1.$$

Now, let for each two vertexes $u, v \in \mathbb{Z}_2^d$,

$$H_{uv} = \begin{cases} 1 \text{ if } R(u) \text{ and } R(v) \text{ share an edge} \\ 0 \text{ otherwise} \end{cases}$$

Then, for each u, $E(\sum_v H_{uv}) \leq \frac{d}{2} E(W_a) = \frac{d}{2}$. Clearly, $\sum_v H_{uv}$ is the number of "delays" that the message μ_u should wait due to "edge collisions". According to Chernov's inequalities [Schnitger(2007)], for any $\beta > 0$,

$$\Pr\left[\sum_v H_{uv}) \geq (1+\beta)\frac{d}{2}\right] \leq e^{-\frac{\beta^2}{3}\frac{d}{2}}.$$

Hence, for $\beta = 3$, the probability that a message μ_u is delayed at least $2d$ time units is upper bounded by $\exp\left(-\frac{3}{2}d\right)$. Consequently, the probability that some μ_u requires at least $2d$ time units is upper bounded by

$$2^d \exp\left(-\frac{3}{2}d\right) = \exp\left(d\left(\ln 2 - \frac{3}{2}\right)\right) \leq \exp\left(-\frac{d}{2}\right).$$

Thus, with probability at least $1 - 2\exp\left(-\frac{d}{2}\right)$ the first stage 1.b of procedure `RandomMiddle` is done in $3d$ time units and the whole procedure is done in $6d$ time units.

4.1.1.4 Butterflies

Let $B_d = \mathbb{Z}_2^d \times [\![0,n]\!]$ be the Cartesian product of the hypercube $\mathbb{Z}_2^d$ and the set consisting of the first $n+1$ natural numbers. Let

$$R_x = \{x\} \times [\![0,n]\!] \ , \ C_j = \mathbb{Z}_2^d \times \{j\}$$

be the *x-th row* and the *j-th column*, or *level*, of B_d. Let

$$E_{hd} = \{\{(x,j),(x,j+1)\} |\ j \in [\![0,n-1]\!], x \in \mathbb{Z}_2^d\} \quad : \textit{horizontal} \text{ edges}$$
$$E_{cd} = \{\{(x,j),(y,j+1)\} |\ j \in [\![0,n-1]\!], x \in \mathbb{Z}_2^d, y = x + e_j\} : \textit{crossing} \text{ edges}$$

The graph $(B_d, E_{hd} \cup E_{cd})$ is called the *d-dimensional butterfly network*. In figure 4.4 we sketch the graphs of the butterflies for $d \in [\![1,4]\!]$.

The butterfly can be realized as transitions in the hypercube made in parallel. Also, it can be seen that a d-dimensional butterfly contains two subgraphs isomorphic to the $(d-1)$-dimensional butterfly.

A *wrapped butterfly* is obtained by substituting $[\![0,n]\!]$ by $\mathbb{Z}_n$ and letting addition of indexes modulo n (thus the last level $j = n$ in the butterfly is identified with the initial level $j = 0$). Let WB_d denote the d-dimensional wrapped butterfly network.

We have:

- The butterfly network B_d is a graph of order 4 (each inner node is the extreme of 4 edges, and the nodes at the extreme levels have 2 neighbors). The number of nodes in B_d is $(d+1)\ 2^d$ and the number of edges is $d\ 2^{d+1}$. The diameter of B_d is $2d$ and its bisection width is $\Theta(2^d)$.
- The wrapped butterfly network WB_d is a regular graph of order 4. The number of nodes in WB_d is $d\ 2^d$ and the number of edges is $d\ 2^{d+1}$. The diameter of WB_d is $\lfloor \frac{3d}{2} \rfloor$ and its bisection width is 2^d.

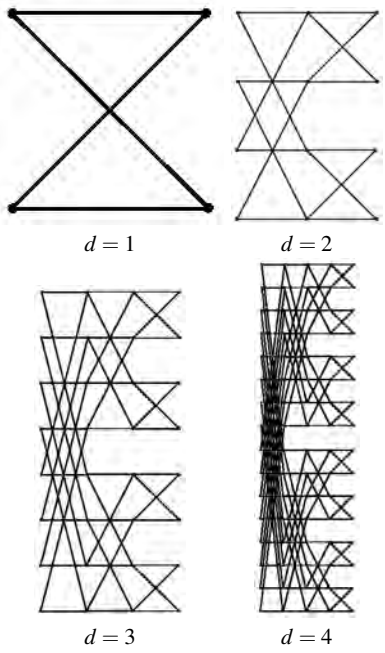


Fig. 4.4 d-dimensional butterfly networks for $d \in [\![1,4]\!]$.

4.1.2 Transition systems

The *communication protocols* involve proof methods for *safety* and *liveness properties*, as well as *event dependency*, message transmission and network routing as well as avoidance of store-and-forward packet deadlocks.

A *transition system* is a triplet $\mathscr{T} = (C, T, C_0)$ where C is a non-empty set of *configurations*, $T \subset C \times C$ is a *transition relation* (usually, if $(c_b, c_c) \in T$ it is written $c_b \longrightarrow c_c$) and $C_0 \subset C$ is a subset of *initial* configurations. A *partial execution* is a sequence $[c_0, c_1, \ldots, c_k]$ of configurations such that $c_0 \in C_0$ and $\forall i > 0$, $c_i \rightarrow c_{i+1}$. In this case, we write $c_0 \stackrel{*}{\rightarrow} c_k$. An *execution* is a maximal partial execution. If $c_0 \stackrel{*}{\rightarrow} c_k$ through an execution then c_k is called an *ending configuration*. A configuration $c \in C$ is *reachable* if $\exists c_0 \in C_0$: $c_0 \stackrel{*}{\rightarrow} c$.

A property Φ on states is a *safety requirement* if it holds on any reachable state at an execution and it is a *liveness requirement* if it holds on some reachable state at an execution.

For two properties Φ, Ψ on states, we write $\{\Phi\} \rightarrow \{\Psi\}$ if the following implication holds:

$$[(c_b \rightarrow c_c) \ \& \ \Phi(c_b)] \implies \Psi(c_c).$$

An *invariant requirement* is a property Φ that holds in the initial configurations and holds in any configuration in a partial execution, $\forall c \in C_0$, $\Phi(c)$, and besides, $\{\Phi\} \to \{\Phi\}$. Thus an invariant property holds in each reachable configuration at any execution.

4.1.3 Distributed systems

A *local algorithm*, or *process*, is a transition system whose configurations are called *states* and whose transitions are called *events*. The events are of three types: *internal*, *send* and *receive*. Thus, if S_p is the set of states and M the set of *messages*, then

$$\begin{aligned} \to_{i,p} &\subset S_p \times S_p \\ \forall \mu \in M : \to_{s,\mu,p} &\subset S_p \times S_p \\ \forall \mu \in M : \to_{r,\mu,p} &\subset S_p \times S_p \end{aligned}$$

Let $\to_p = \to_{i,p} \cup \bigcup_{\mu \in M} \left(\to_{s,\mu,p} \cup \to_{r,\mu,p} \right)$, i. e.,

$$\forall c,d \in S_p : \ \left[(c \to d) \iff (c \to_{i,p} d) \ \text{ or } \exists \mu \in M \left(c \to_{s,\mu,p} d \text{ or } c \to_{s,\mu,p} d \right) \right].$$

A *distributed system* is a collection of process. The collection of *states* is the Cartesian product of the processes states. The *communication subsystem* is a multiset of messages: each pair of processes has associated, at each step, a message. Formally, let P be a collection of processes:

States. $C = \prod_{p \in P} C_p \times \prod_{p \in P} M_p$, where for each $p \in P$, $M_p = M^*$ is the collection of finite sequences of messages.

Transitions. These are defined according to the type of events at each process. Namely,

$$\left((c_q)_{q \in P}, \left(\mu_q^*\right)_{q \in P} \right) \to \left((d_q)_{q \in P}, \left(\nu_q^*\right)_{q \in P} \right) \tag{4.2}$$

where

$$c_p \to_{i,p} d_p \implies \begin{cases} (\forall q \in P - \{p\} : \ d_q = c_q) \,\& \\ \left(\forall q \in P : \ \nu_q^* = \mu_q^*\right) \end{cases} \tag{4.3}$$

$$c_p \to_{s,\mu,p} d_p \implies \begin{cases} (\forall q \in P - \{p\} : \ d_q = c_q) \,\& \\ \left(\forall q \in P - \{p\} : \ \nu_q^* = \mu_q^*\right) \,\& \\ \left(\nu_p^* = \mu_p^* - \{\mu\}\right) \end{cases} \tag{4.4}$$

$$c_p \to_{r,\mu,p} d_p \implies \begin{cases} (\forall q \in P - \{p\} : \ d_q = c_q) \,\& \\ \left(\forall q \in P - \{p\} : \ \nu_q^* = \mu_q^*\right) \,\& \\ \left(\nu_p^* = \mu_p^* \cup \{\mu\}\right) \end{cases} \tag{4.5}$$

It is said that the events at the left sides of (4.3-4.5) *are applicable* at the configuration on the left side of (4.2), and each *produces* the configuration at the right side. As described above, the distributed system is called *asynchronous*.

Proposition 4.2 (Asynchronous message passing). *Let γ be a configuration and let e_p, e_q be local events at different processes, both applicable in γ. Then e_q is applicable in $e_p(\gamma)$, e_p is applicable in $e_q(\gamma)$ and $e_q(e_p(\gamma)) = e_p(e_q(\gamma))$.*

Instead, in a *synchronous distributed system* the *set of states* is $C = \prod_{p\in P} C_p$ and the *transitions* have the form $(c_q)_{q\in P} \to (d_q)_{q\in P}$ where

$$c_p \to_{i,p} d_p \implies \forall q \in P - \{p\} :\ d_q = c_q \tag{4.6}$$

$$\left.\begin{array}{c} c_{p_s} \to_{s,\mu,p_s} d_{p_s} \\ c_{p_r} \to_{r,\mu,p_r} d_{p_r} \end{array}\right\} \implies \forall q \in P - \{p_s, p_r\} :\ d_q = c_q. \tag{4.7}$$

In event (4.7) the local events at the left side are called *corresponding*.

Let $\Gamma = (\gamma_\kappa)_{\kappa=0}^{k}$ be an execution in a synchronous distributed system. For each $\kappa > 0$ let e_κ be the event such that $\gamma_\kappa = e_\kappa(\gamma_{\kappa-1})$. Let $\overline{\Gamma} = (e_\kappa)_{\kappa=1}^{k}$ be the corresponding sequence of events.

Let $\preceq_\Gamma$ be the smallest ordering relation in the collection of local events satisfying the following conditions:

- If e' and e'' are events in the same process and e' is applied before than e'' in the execution Γ, then $e' \preceq_\Gamma e''$.
- If e' is a *send* event and e'' is the corresponding *receive* event in the execution Γ, then $e' \preceq_\Gamma e''$.

"$\prec_\Gamma$" is called the *causal order* determined by Γ.

Two events e' and e'' are *concurrent*, $e' \| e''$, if neither $e' \preceq_\Gamma e''$ nor $e'' \preceq_\Gamma e'$.

Let $\pi : [\![1,k]\!] \to [\![1,k]\!]$ be a permutation *consistent with the clausal order*:

$$\forall i, j \in [\![1,k]\!] :\ \left[e_{\pi(i)} \preceq_\Gamma e_{\pi(i)} \implies i \le j\right].$$

Let $\overline{\Delta} = \left(d_\kappa = e_{\pi(\kappa)}\right)_{\kappa=1}^{k}$. Then $\overline{\Delta}$ determines an execution Δ in the distribution system.

Proposition 4.3. *Δ and Γ have the same length and the last configuration of Δ coincides with the last configuration of Γ.*

Δ and Γ are thus *equivalent executions*. This notion defines, indeed, an *equivalent relation* on the whole collection of executions.

A *logical clock* is a map $\omega : \{\text{events}\} \to \mathbb{N}$ that is increasingly monotone with respect to a clausal ordering $\preceq$ and the usual ordering in $\mathbb{N}$.

Let us see some examples of distributed algorithms.

Example 4.1 (Unidirectional leader election). Let us assume a ring of processors. There are n processors, $p_0, \ldots, p_{n-1}$, and each, p_i, is able to send a message to its successor, p_{i+1} (where addition in the indexes is taken modulus n).

The goal of the algorithm is that some processor eventually will claim `'I am the leader'`, and, if required, any other will claim `'I am not the leader'`. The setting is the following:

Messages. Processors UID's (these are labels in $\mathbb{N}$) and an *empty message*, to represent that no message is sent.
States. $\{\perp, \texttt{leader}\}$.
Initial states. Each processor knows its own UID and is in state $\perp$.

Each process acts as follows:

1. Let $v :=$ Receive ;
2. If $v >$UID Then Send(v)
   Else
   a. If $v =$UID Then `leader`
      Else
      i. Do Nothing

Clearly, the elected leader would be that with greatest UID and the elected processor will realize that within n rounds. This process involves $O(n^2)$ message transmissions.

Example 4.2 (Bidirectional leader election). Also in this case, let us assume a ring of n processors, $p_0, \ldots, p_{n-1}$, and each, p_i, is now able to send a message to both its successor, p_{i+1}, and its predecessor, p_{i-1} (indexes are taken modulus n).

As before, the elected leader would be that with greatest UID. The messages should be pairs

1. For each round $k \geq 1$ Do
   a. each processor sends its UID to both sides with the aim to reach processors at distance 2^k and return to the original processor,
      i. If the traveling UID is less than the UID of a visited processor, the traveling UID is lost,
      ii. Once the traveling UID returns to its original processor, a new round begins.
   b. If a traveling UID arrives to its original processor in the "on-going" direction, then that node claims itself as the leader.

Each message consists of the UID and two indexes: The first tracks the number of visited processors in the "on-going" direction, while the second tracks the number of visited processors in the "returning" direction.

We may see that the leader will be elected within $1 + \lceil \log_2 x \rceil$ rounds and the total number of sent messages is $8n(1 + \lceil \log_2 x \rceil) = O(n \log n)$. Indeed initially each processor sends at most 4 messages, and at each new round k, the processors thats end messages are those that have not been defeated, and at each group of $2^{k-1} + 1$ only one goes to the next round, hence they will remain just $\lfloor \frac{n}{2^{k-1}+1} \rfloor$ processors, hence the number of messages in that round will be $4 \cdot 2^k \cdot \lfloor \frac{n}{2^{k-1}+1} \rfloor \leq 8n$.

Example 4.3 (Breadth-first search). Let $G = (V, E)$ be a directed graph and let $s \in V$ be a *source* vertex. A breadth-first traversal of G is a spanning tree rooted at s such that any node at distance d from s is at height d in the tree.

Initially the source vertex is declared as the only *marked* node and it sends a *search* message to its neighbors. Thereafter, when a non-marked node receives a search message then it is declared as a marked node, it declares the transmitting node as its own parent, and it sends a search message to its neighbors.

Clearly, the algorithm produces a tree which is indeed a breadth-first traversal similar as the traversal produced by the procedure at figure 3.4. At each round, any vertex with distance (i.e. the length of the shortest path connecting it with s) not exceeding the round index has been marked and an assigned parental. The algorithm has time complexity $O(k)$ where k is the diameter of the graph and sends $O(e)$ messages, where e is the number or edges in the graph.

4.2 Concurrency

A distributed system, consisting of several processors, may involve some *shared resources*, as memory or input-output devices. The *concurrency problems* are related to (simultaneous) access to shared resources.

4.2.1 Exclusive and concurrent access

Let $\mathscr{C}$ be a class of processors, e.g. PRAM's, Turing machines or while-programs, see for instance, [Kfoury et al(1991)Kfoury, Arbib, and Moll]. Let us consider now as the only shared resource a shared memory. Thus, besides the common read-write instructions into local memory devices, let us consider also general instructions, allowing processors to access the shared memory, `read*` and `write*` into registers of the shared memory. A *conflict* appears when two processors try to access the same register in order to commit a read or write action. Let us define the following program classes:

EREW-$\mathscr{C}$ (Exclusive Read-Exclusive Write) : Programs in $\mathscr{C}$ with neither read nor write conflicts.

CREW-$\mathscr{C}$ (Concurrent Read-Exclusive Write) : Programs in $\mathscr{C}$ with no write conflicts.

ERCW-$\mathscr{C}$ (Exclusive Read-Concurrent Write) : Programs in $\mathscr{C}$ with no read conflicts.

CRCW-$\mathscr{C}$ (Concurrent Read-Concurrent Write) : Programs in $\mathscr{C}$ with no read-write restrictions.

For any subclass $\mathscr{D} \subseteq \mathscr{C}$ and any two functions $p, t : \mathbb{N} \to \mathbb{N}$, let $\mathscr{D}(p(n), t(n))$ denote the class of problems in $\mathscr{D}$ running in time $O(t(n))$ requiring $O(p(n))$ processors and

an unbounded shared memory. Thus, for instance $\bigcup_{k,\ell\in\mathbb{N}}$ CRCW-$\mathscr{C}(n^k,[\log(n)]^\ell)$ is the class of programs running in polylogarithmic time in a distributive system involving a shared memory and a polynomial number of processors. Due to definition,

$$\text{EREW-}\mathscr{C}(p(n),t(n)) \subseteq \text{CREW-}\mathscr{C}(p(n),t(n)) \subseteq \text{CRCW-}\mathscr{C}(p(n),t(n)) \text{ and}$$
$$\text{EREW-}\mathscr{C}(p(n),t(n)) \subseteq \text{ERCW-}\mathscr{C}(p(n),t(n)) \subseteq \text{CRCW-}\mathscr{C}(p(n),t(n)).$$

A program $P \in \mathscr{C}$ *recognizes* a word $\sigma \in (0+1)^*$ if on input σ the corresponding computation of P ends with the value 1 in the first shared register. The *recognized language* of P is $L(P) \subset (0+1)^*$, obviously, consisting of all words that are recognized by P. In this way, any program class $\mathscr{D} \subseteq \mathscr{C}$ can be seen as a language class: $[L(P) \in \mathscr{D} \iff P \in \mathscr{D}]$.

A distributive system with shared memory can be simulated with a hypercube using message-passing. Namely, suppose $P \in$ CRCW-$\mathscr{C}(p(n),t(n))$, and let $k(n) = \lceil \log_2 p(n) \rceil$. Then the processors for P can be included as vertexes of the $k(n)$-dimensional hypercube. Using a hash function let us map the shared memory as the union of extended local memories of processors in the hypercube. Then the program P can be simulated by this hypercube. Hence:

An algorithm in CRCW-$\mathscr{C}(p(n),t(n))$ *can be simulated by a* $\lceil \log_2 p(n) \rceil$*-dimensional hypercube running in time* $O(t(n) \cdot \lceil \log_2 p(n) \rceil)$.

In order to provide some examples and to compare the introduced subclasses let us consider three elementary problems:

Problem `ZeroCounting`.
Instance: A word $\sigma = s_0 \cdots s_{n-1} \in 0^+1^+$.
Solution: The position i such that $s_i = 0$ and $s_{i+1} = 1$.

Problem `Boolean-OR`.
Instance: A word $\sigma \in (0+1)^*$.
Solution: $\max_{1 \le i \le \text{len}(\sigma)} s_i$.

Problem `Parity`.
Instance: A word $\sigma \in (0+1)^*$.
Solution: $\left[\sum_{i=1}^{\text{len}(\sigma)} s_i\right] \mod 2$.

A recursive EREW distributive algorithm, `EREWZeroCount`, to solve the problem `ZeroCounting` is outlined in fig. 4.5.

The time complexity of `EREWZeroCount` is determined by the recurrence $t(n) = t(\frac{n}{2}) + 1$, thus $t(n) = O(\log n)$. The number of required processors is linear with n. But it is a rough measure.

As a new complexity measure, let $w : n \mapsto \sum_i t_i(n)p_i(n)$ be the *work* of the distributed algorithm, where i runs over the steps of the algorithm, t_i is the time required at the i-th step and p_i is the number of involved processors.

The work complexity of `EREWZeroCount` is determined by the recurrence $w(n) = w(\frac{n}{2}) + \frac{n}{2}$, thus $w(n) = O(n)$.

`EREWZeroCount`
Input. A word $\sigma = s_0 \cdots s_{n-1} \in 0^+1^+$.
Output. The position ZC $= i$ such that $s_i = 0$ and $s_{i+1} = 1$.

1. $n := \text{len}(\sigma)$;
2. If $n \leq 2$ then output 0
   Else
   a. For $i = 1$ To $\lceil \frac{n-1}{2} \rceil - 1$ Do (in parallel)
      i. $t_i :=$ `'0'`
      ii. If $s_{2i} ==$ `'1'` && $s_{2i+1} ==$ `'1'` Then $t_i :=$ `'1'` ;
   b. $j_0 :=$ `EREWZeroCount`(τ) ;
   c. If $s_{2j_0+1} ==$ `'1'` Then $i := 2j_0$ Else $i := 2j_0 + 1$;
3. Output i

Fig. 4.5 Procedure `EREWZeroCount`.

A recursive EREW distributive algorithm to solve in a more general setting the problem `Boolean-OR` is sketched in fig. 4.6, with the procedure `EREWMaxArray`.

`EREWMaxArray`
Input. A numeric array S
Output. The max-array $[\max_{j \in [\![0,i]\!]} S[[j]]]_{i \in [\![0, \text{len}(S)-1]\!]}$

1. $n := \text{len}(S)$;
2. If $n == 1$ then output S
   Else
   a. For $i = 0$ To $\lceil \frac{n-1}{2} \rceil - 1$ Do (in parallel)
      $\{ T[[i]] := \max\{S[[2i]], S[[2i+1]]\} \}$;
   b. $N :=$ `MaxArray`(T) ;
   c. For $i = 0$ To $n-1$ Do (in parallel)
      i. Case odd i: $M[[i]] := \max\{N[[(i-1)/2]], S[[i]]\}$;
      ii. Case even i: $M[[i]] := N[[i/2]]$;
3. Output M

Fig. 4.6 Procedure `EREWMaxArray`.

The time complexity of `EREWMaxArray` is determined by the recurrence $t_0(n) = t_0(n/2) + 1$, thus $t_0(n) = O(\log n)$, while the number of required processors is determined by the recurrence $p_0(n) = p_0(n/2) + n$, hence $p_0(n) = O(n)$.

It is worth to remark here that the same algorithm may be used to compute partial sums in an array. Indeed, by changing each max operation by addition, we obtain the partial sums. This modification may be used to compute the multiplication of two $(n \times n)$-matrices with n^3 processors. Namely, for each index triplet (i, j, k) let an own processor compute $A[[i,k]] * B[[k, j]]$; then use the above mentioned modification to calculate, for each pair (i, j), $C[[i, j]] := \sum_k A[[i,k]] * B[[k, j]]$.

Instead, a first CREW distributive algorithm to calculate the maximum element in an array of non-negative real numbers is sketched in fig. 4.7.

```
CREWFirstMax
Input. A numeric array S
Output. The pairs such that (i,S[[i]]) such that S[[i]] = max_{j∈⟦0,len(S)−1⟧} S[[j]]

1. n := len(S) ;
2. If n == 1 Then Output (0,S[[0]])
   Else
   a. For i = 0 To n − 1 Do (in parallel) A[[i]] := 1 ;
   b. For Each pair (i, j) with 0 ≤ i < j ≤ n − 1 Do (in parallel)
            { If S[[i]] < S[[j]] Then GT[[i, j]] := 0 Else GT[[i, j]] := 1 } ;
   c. For Each pair (i, j) with 0 ≤ i < j ≤ n − 1 Do (in parallel)
            { A[[i]] := min(A[[i]], GT[[i, j]]) } ;
   d. For i = 0 To n − 1 Do (in parallel)
            { If A[[i]] == 1 Then Output (i,S[[i]])
```

Fig. 4.7 Procedure `CREWFirstMax`.

The time complexity of `CREWFirstMax` is constant, $t_1(n) = O(1)$, while the number of required processors is quadratic, $p_1(n) = O\left(\binom{n}{2}\right) = O(n^2)$.

As a second CREW algorithm, let us proceed recursively as in fig. 4.8.

```
CREWSecondMax
Input. A numeric array S
Output. The pairs such that (i,S[[i]]) such that S[[i]] = max_{j∈⟦0,len(S)−1⟧} S[[j]]

1. For i = 0 To √n − 1 Do (in parallel)
   a. S_i := S[[j| √n i ≤ j < √n (i+1)]] ;
   b. T[[i]] := CREWSecondMax(S_i) ;
2. mx := CREWFirstMax(T) ;
3. Output mx
```

Fig. 4.8 Procedure `CREWSecondMax`.

The time complexity of `CREWSecondMax` is determined by the recurrence $t_2(n) = t_2(\sqrt{n}) + O(1)$, thus $t_2(n) = O(\log\log n)$. Here we see that the cycles 1. and 2. in the algorithm may use the same processors. Thus, it is rather arbitrary to count the number of involved processors. For `CREWSecondMax` the work is determined by the recurrence $w_2(n) = \sqrt{n}\, w_2(\sqrt{n}) + O(n)$, hence $w_2(n) = O(n \log\log n)$.

As a third CREW algorithm, let us section the given numeric array into pieces, let us select the maxima of the pieces by `EREWMaxArray` and then let us select the maximum in the maxima array by `CREWSecondMax`. See fig. 4.9.

The time complexity of `CREWThirdMax` is

`CREWThirdMax`
Input. A numeric array S
Output. The pairs such that $(i, S[[i]])$ such that $S[[i]] = \max_{j \in [\![0, \text{len}(S)-1]\!]} S[[j]]$

1. For $i = 0$ To $\frac{n}{\log\log n} - 1$ Do (in parallel)
   a. $S_i := S[[j \mid (\log\log n)\, i \leq j < (\log\log n)\,(i+1)]]$;
   b. $T[[i]] := \texttt{EREWMaxArray}(S_i)$;
2. mx := $\texttt{CREWSecondMax}(T)$;
3. Output mx

Fig. 4.9 Procedure `CREWThirdMax`.

$$\begin{aligned} t_3(n) &= t_0(\log\log n) + t_2\left(\frac{n}{\log\log n}\right) \\ &= O(\log\log\log n) + O(\log\log n) \\ &= O(\log\log n). \end{aligned}$$

The work is

$$\begin{aligned} w_3(n) &= w_0(\log\log n) + w_2\left(\frac{n}{\log\log n}\right) \\ &= O(\log\log\log n) + O(n) \\ &= O(n). \end{aligned}$$

Given an increasing numeric array $(X[[i]])_{i \in [\![1,n]\!]}$ let us add as extreme values $X[[0]] = -\infty$ and $X[[n+1]] = +\infty$. For any value $y \in \mathbb{R}$ let $R(y,X) = i$ if and only if $X[[i]] \leq y < X[[i+1]]$. Clearly $X[[0]] < y < X[[n+1]]$.

Let us consider the following problem:
Problem `Rank`.
Instance: An increasing array $(X[[i]])_{i \in [\![1,n]\!]}$ and a value $y \in \mathbb{R}$.
Solution: The *rank* $R(y,X)$.

A CREW distributed algorithm to solve `Rank` is displayed at pseudocode 4.10.

Let us remark here that the step 3. at pseudocode 4.10 is typical in the simulation of EREW procedures by CREW procedures.

Remark 4.1. p values on a p-processor EREW procedure can be reduced to a CREW procedure within $O(\log p)$ time.

Suppose that the array of values is $[s_i]_{i=0}^{p-1}$. Let us construct a binary tree as follows: Initially let us correspond each leaf with a value, $v_{0i} \leftrightarrow \{s_i\}$, for $i \in [\![0, \ell_0 - 1]\!]$, with $\ell_0 = p$. Consecutively, for each $k > 0$ let v_{ki} be the parent of the left child $v_{k-1,2i}$ and the right child $v_{k-1,2i+1}$, for $i \in [\![0, \ell_k - 1]\!]$, with $\ell_k = \frac{\ell_{k-1}}{2}$, and let us associate as label of the parent the concatenation of the labels of its children. Clearly the constructed tree has height $\log_2 p$ and the label of the root is the whole data array.

`CREWRank`
Input. An increasing numeric array X, a value $y \in \mathbb{R}$ and a fixed number of processors $p \in \mathbb{Z}^+$.
Output. $R(y,X)$

1. $n_p := \frac{n+1}{p}$;
2. $i_c := 0$;
3. Repeat $\lceil \log_p(n+2) \rceil$ times
   a. For $i = 1$ To p Do (in parallel)
      i. If $X[[i_c + (i-1)n_p]] \leq y \leq X[[i_c + in_p - 1]]$ Then
         $\{ i_c := i_c + (i-1)n_p \; ; n_p := \frac{n_p}{p} \}$;
   b. For $i = 1$ To p Do (in parallel)
      i. If $X[[i_c + in_p - 1]] < y < X[[i_c + in_p]]$ Then
         Output $(i_c + in_p - 1)$

Fig. 4.10 Procedure `CREWRank`.

Thus, for instance, the distributive algorithm, `EREWZeroCount`, in fig. 4.5 gives rise to a corresponding `CREWZeroCount` with time complexity at least $\Omega(\log_2 p)$ and $O(p)$ processors.

We have that `CREWRank` runs in time $O(\frac{\log n}{\log p})$ with p processors, while in any EREW scheme its time is at least $\Omega(\log \frac{n}{p})$. Thus CREW is more powerful, in general, than EREW.

4.2.2 Dining philosophers problem

This is a typical problem illustrating the appearances of *deadlocks* when some resources are shared in a distributed environment.
Dining philosophers problem

> At any moment, a *dining philosopher* is either *thinking* or *eating*. Since the menu consists just of spaghetti, any philosopher should be provided with two forks in order to begin to eat.
>
> Let us assume that p philosophers are sit around a circular table with the spaghetti bowl at the center. In between two adjacent philosopher lies a fork. Thus in order to begin to eat, each philosopher should use the two forks adjacent to him.

A deadlock situation appears when each philosopher is granted to use his left fork. Each is waiting for his right fork to be free, and no philosopher is able to begin to eat!

The deadlock may be broken by introducing a waiter. A waiter may schedule an eating roll, thus he may assign fork pairs (and eventually to take care of forks cleaning after any use!) to dining philosophers.

A very well known algorithm for deadlock breaking without the participation of any external agents is the following:

1. Let us associate ID's to the philosophers in a one-to-one fashion with ID's in a totally ordered set.
2. Let us put a *dirty* fork in between two neighbor philosophers and let us associate it to the philosopher with least ID.
3. Whenever a philosopher wants to eat he sends requests for *clean* forks to both of his neighbors.
4. Whenever a philosopher receives a request for a clean fork:
   a. If he has a proper dirty fork, then he cleans it and allows the requesting neighbor to use the cleaned fork.
   b. If the proper fork is clean, then he keeps it and maintains his requesting neighbor in a waiting state.
   c. If he has no forks, he keeps on waiting.
5. When a philosopher gets two clean forks he begins to eat. When finishing, he remains with two dirty forks.

In general, in order to break deadlocks, within a distributed system any processor is provided with marked tokens and it may decide its access to shared resources by the current configuration of its own tokens.

4.3 Routing

4.3.1 Byzantine Generals problem

In a battle field, a General gives an order to either attack or retreat. The order is transmitted by the subordinates and all participants should agree on a value of the order: Either to attack or to retreat. The General may be a participant and all subordinates are participants as well. Among participants there are traitors that may alter, when transmitting it, the sense of the order. Even the General may be a traitor. The goal of the problem is thus to make agree all the non-traitor subordinates on the General's order.

In the language of distributed systems the problem is posed as follows: A source processor transmits a bit. Among the processors there are correct and faulty processors. The faulty processors may change the value of the message bit. The goal of the correct processors is to convene in the value of the original message bit emitted by the source processor.

Let S denote the source processor. For any processors X, Y let

$$\begin{aligned} \Psi_S(X,Y) &: \text{value claimed by } Y, \text{ up to the knowledge of } X, \text{ from } S \\ \Phi_S(X) &: \text{received value by } X \text{ from } S \\ \Xi_S(X) &: \text{transmitted value by } X \text{ allegedly arrived from } S \end{aligned}$$

Necessarily, $\Phi_S(X) = \Psi_S(X,S)$. Clearly,

$$\Phi_S(X) \neq \Xi_S(X) \implies X \in \text{ Faulty },$$

and for any $Y \neq X, S$:

$$\Phi_S(X) \neq \Psi_S(X,Y) \implies Y \in \text{ Faulty } \text{ or } S \in \text{ Faulty }.$$

In the above situation, for just three participants X, Y, S, it is not possible for X to decide which of the other two participants is faulty.

Let p denote the number of processors and let q be the number of faulty processors. If $3q < p$ then an agreement protocol does exist.

Let us consider the following subprocedures:

`OM`(0)
Input. S source processor and a bit b_S.
Output. b_S transmitted to neighbors of S

1. For each neighbor Y of S Do b_s is sent form S to Y ;
2. Y assumes b_S

`OM`(m)
Input. S source processor and a bit b_S.
Output. An agreed value of b_S transmitted from S

1. S sends b_S to all processors ;
2. For each processor X Do (in parallel)
   a. $b_X = \Xi_S(X)$;
   b. `OM`$(m-1)$ (X, b_X) (X acts as source and resends its bit value to all processors different than S and X itself)
3. For each processor X Do (in parallel)
   X assumes the most frequent value among $(\Phi_Y(X) |\ Y \neq X, S)$.

4.3.2 Routing algorithms

The `Routing problem` consists in select the "best" pat connecting two nodes in a network. Any solving procedure should be *correct* (any path selected among two nodes, should indeed connect them), *efficient* (the selected paths are optimal among the paths connecting the nodes) and *robust* (the paths should be selected without prior knowledge of the, possible varying, networks topology).

Any network may be realized as a graph $G = (V, E)$ (either undirected or directed). A *cycle* is a path whose extreme points coincide. A *simple path* is a path without internal cycles.

In static networks a general strategy may be the computation of spanning trees. For any node $v \in V$, let T_v be a spanning tree of the network rooted at v. Then for each node $u \in V$ in the connected component of v, let the branch connecting v with u be the selected route.

At any modification of the network, the spanning tree should be recomputed.

In case of a weighted network, whose edge weights are either costs of traffic loads, the Floyd-Warshall algorithm (shown in figure 3.8) computes the shortest distance path among any pair source-destination nodes. Let us sketch here a distributed version of the Floyd-Warshall algorithm.

For each node $v \in V$ let us consider the following data arrays:

N_v. Neighbors of v: $u \in N_v$ if and only if $\{v,u\} \in E$.
In this case, let $w_{vu} \in \mathbb{R}^+$ be the weight of the edge $\{v,u\}$.

R_v. Subset of nodes whose route from v has been decided.

D_v. Distances from v: $D_v[[u]]$ will be the distance of the route from v to w.

I_v. Initial visited nodes from v: $I_v[[u]] = w$ if and only if the route connecting v with u shall begin going through node w.

The distributive algorithm to build arrays D_v and I_v, $v \in V$ is sketched at pseudocode 4.11.

`DistributedFloydWarshall`
Input. A weighted graph $G = (V,E)$ with non negative weights modeling a network.
Output. The arrays D_v and I_v, $v \in V$.

1. For each $v \in V$ Do (in parallel)
   a. $R_v := \{v\}$;
   b. $D_v[[v]] := 0$; $I_v[[v]] := \perp$;
   c. For each $u \in V - \{v\}$ Do (in parallel)
      i. If $u \in N_v$ Then { $I_v[[u]] := u$; $D_v[[u]] := w_{vu}$ }
         Else { $I_v[[u]] := \perp$; $D_v[[u]] := +\infty$ } ;
2. For each $v \in V$ Do (in parallel)
   a. While $\neg(R_v == V)$ do
      i. Broadcast (I_v, D_v) ;
      ii. For each $w \in V - R_v$ Do (in parallel)
         A. Broadcast (I_w, D_w) ;
         B. For each $u \in V - \{v,w\}$ Do (in parallel)
            { pnw $:= D_v[[w]] + D_w[[u]]$;
              If pnw $< D_v[[u]]$ Then
                 { $I_v[[u]] := I_v[[w]]$; $D_v[[u]] :=$ pnw }
            } ;
      iii. $R_v := R_v \cup \{w\}$;
3. Output $\{(I_v, D_v) | \ v \in V\}$.

Fig. 4.11 Procedure `DistributedFloydWarshall`.

Alternatively we may proceed according to the pseudocode 4.12.

`ChandyMisra`
Input. A weighted graph $G = (V,E)$ with non negative weights modeling a network.
Output. The arrays D_v and I_v, $v \in V$.

1. For each $v \in V$ Do (in parallel)
   a. $D_v[[v]] := 0$;
   b. For each $u \in V - \{v\}$ Do (in parallel)
      i. If $u \in N_v$ Then $\{\ I_v[[u]] := u$; $D_v[[u]] := w_{vu}\ \}$
         Else $\{\ I_v[[u]] := \perp$; $D_v[[u]] := +\infty\ \}$;
2. For each $v \in V$ Do (in parallel) For each $u \in V - \{v\}$ Do (in parallel)
   a. For each $w \in V - \{v,u\}$ Do (in parallel)
      $\{$ pnw $:= D_v[[w]] + D_w[[u]]$;
      If pnw $< D_v[[u]]$ Then
      $\{\ I_v[[u]] := I_v[[w]]$; $D_v[[u]] :=$ pnw $\}$
      $\}$;
3. Output $\{(I_v, D_v) |\ v \in V\}$.

Fig. 4.12 Procedure `ChandyMisra`.

References

[Aho et al(1974)Aho, Hopcroft, and Ullman] Aho AV, Hopcroft JE, Ullman J (1974) The Design and Analysis of Computer Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, URL http://portal.acm.org/citation.cfm?id=578775

[Brassard and Bratley(1988)] Brassard G, Bratley P (1988) Algorithmics: theory & practice. Prentice-Hall, Inc., Upper Saddle River, NJ, USA

[Brassard and Bratley(1990)] Brassard G, Bratley P (1990) Algor´ítmica: Concepción y Análisis. Editorial Masson

[Cattaneo and Italiano(1999)] Cattaneo G, Italiano G (1999) Algorithm engineering. ACM Comput Surv p 3, DOI http://doi.acm.org/10.1145/333580.333582

[Cormen et al(2003)Cormen, Leiserson, Rivest, and Stein] Cormen TH, Leiserson CE, Rivest RL, Stein C (2003) Introduction to Algorithms, 2nd edn. McGraw-Hill Science / Engineering / Math, URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0072970545

[Fernandez et al(2003)Fernandez, Garcia, and Garzon] Fernandez JJ, Garcia I, Garzon E (2003) Educational issues on number representation and arithmetic in computers: an undergraduate laboratory. IEEE Transactions on Education 46(4):477–485, DOI 10.1109/TE.2003.815237

[Garey and Johnson(1979)] Garey MR, Johnson DS (1979) Computers and Intractability : A Guide to the Theory of NP-Completeness. Series of Books in the Mathematical Sciences, W.H. Freeman & Company, URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0716710455

[Gonnet and Baeza-Yates(1991)] Gonnet GH, Baeza-Yates R (1991) Handbook of algorithms and data structures: in Pascal and C (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[Graham et al(1994)Graham, Knuth, and Patashnik] Graham RL, Knuth DE, Patashnik O (1994) Concrete Mathematics: A Foundation for Computer Science (2nd Edition), 2nd edn. Addison-Wesley Professional, URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201558025

[Hopcroft et al(2001)Hopcroft, Motwani, and Ullman] Hopcroft JE, Motwani R, Ullman JD (2001) Introduction to automata theory, languages, and computation, 2nd edition. SIGACT News 32(1):60–65, DOI http://doi.acm.org/10.1145/568438.568455

[Kfoury et al(1991)Kfoury, Arbib, and Moll] Kfoury AJ, Arbib MA, Moll RN (1991) Programming Approach to Computability. Springer-Verlag New York, Inc., Secaucus, NJ, USA

[Knuth(1999)] Knuth DE (1999) Mathematics for the Analysis of Algorithms. Birkhauser Boston

[Levitin(2007)] Levitin AV (2007) Introduction to the Design and Analysis of Algorithms, 2-nd Edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[Manber(1989)] Manber U (1989) Introduction to Algorithms: A Creative Approach. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA

[Michalewicz and Fogel(2004)] Michalewicz Z, Fogel DB (2004) How to Solve It: Modern Heuristics. Springer, URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/3540224947

[Papadimitriou and Steiglitz(1998)] Papadimitriou CH, Steiglitz K (1998) Combinatorial Optimization : Algorithms and Complexity. Dover Publications, URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0486402584

[Schnitger(2007)] Schnitger G (2007) Parallel and Distributed Algorithms. Institut f ür Informatik, Johann Wolfgang Goethe-Universität, Frankfurt am Main, URL http://www.thi.informatik.uni-frankfurt.de/Parallele/index.html

[Sedgewick and Flajolet(1996)] Sedgewick R, Flajolet P (1996) An introduction to the analysis of algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[Stoer and Bulirsch(1980)] Stoer J, Bulirsch R (1980) Introduction to Numerical Analysis. Springer-Verlag, New York and Berlin

[Ukkonen(1995)] Ukkonen E (1995) On-line construction of suffix trees. Algorithmica 14(3):249–260

[Wood(1993)] Wood D (1993) Data Structures, Algorithms, and Performance. Addison-Wesley Publishing Co., Reading, MA, USA

Index