



SAPIENZA
UNIVERSITÀ DI ROMA

Algorithms

Volume 1 — Basic algorithms

The representation of numbers and pseudo-code, the language in which all the algorithms of the course are written; the first algorithms: sorting (insertion, exchange and selection sort), linear and binary search, numerical algorithms (change of base, primality test, sieve of Eratosthenes, greatest common divisor). Every algorithm with its pseudo-code and examples executed step by step.

Fabio Furini

fabiofurini.github.io

Department of Computer, Control and Management Engineering — Sapienza University of Rome

Online version and downloadable material:

fabiofurini.github.io/algorithms/
github.com/fabiofurini/algorithms

Academic year 2026–2027

Contents

1	Representation of numbers	3
	Introduction	3
1.1	Representation of natural numbers	3
1.2	Representation of integer numbers	10
1.3	Representation of real numbers	18
2	Pseudo-code	23
	Introduction	23
2.1	Introduction to pseudo-code	24
2.2	Variables and assignments	25
2.3	Basic operations	27
2.4	Basic constructs	28
2.5	Arrays	37
2.6	Functions	40
2.7	Sets	46
2.8	Style conventions	48
3	Sorting algorithms	49
	Introduction	49
3.1	The sorting problem	49
3.2	Insertion Sort	52
3.3	Exchange Sort	59
3.4	Selection Sort	64
3.5	Comparison of the three algorithms	69
4	Linear and binary search	71
4.1	The search problem	71
4.2	Linear search	72
4.3	Binary search	75
5	Numerical algorithms	82
	Introduction	82
5.1	Change of base	82



5.2	Primality test	88
5.3	Sieve of Eratosthenes	94
5.4	Greatest common divisor	99

Representation of numbers

Introduction

An algorithm manipulates data, and before we can describe how an algorithm works we have to agree on how its data are written down. For numbers, this question has a precise answer: a computer stores every number as a finite sequence of symbols, and only finitely many numbers can be written with a given number of symbols. This simple fact has important consequences: every representation has a **representation interval**, outside of which numbers cannot be stored, and the representation of real numbers is, in general, only approximate.

In this chapter we study three representations:

- the representation of **natural numbers** in a positional numeral system with base b , in particular the binary system used by computers;
- the representation of **integer numbers**, first in sign-magnitude and then in two's complement, which is the representation used by modern computers;
- the representation of **real numbers** in floating point, according to the IEEE 754 standard.

For each representation we compute the largest and the smallest number that can be represented with a given number of digits, and the minimum number of digits needed to represent a given number.

1.1 Representation of natural numbers

1.1.1 Positional numeral systems

We are used to writing numbers in the decimal system, where the same digit has a different value depending on where it appears: in 555 the three digits are equal, but they stand for 500, 50 and 5. This is the idea of a positional numeral system.

Definition 1.1: positional numeral system

A **positional numeral system** is a way to encode the natural numbers using a **sequence of digits**, where each digit (single symbol) contributes differently depending on its position.

Definition 1.2: base of a positional numeral system

The number of different digits in a positional numeral system is called the **base** b of the system.

- The base b of a positional numeral system must be greater than or equal to two.
- The unary system is not a positional numeral system: natural numbers are represented by repeating the digit 1 as many times as necessary (e.g., 3 is represented as 111).
- Some of the principal positional numeral systems are:

Positional numeral system	base b	different digits
Binary system	2	0, 1
Octal system	8	0, 1, 2, 3, 4, 5, 6, 7
Decimal system	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Hexadecimal system	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

- In the hexadecimal system the six letters A, B, C, D, E, F are digits, with the values 10, 11, 12, 13, 14, 15.

Definition 1.3: sequence of digits

A **sequence of k digits** representing a natural number in base b is denoted as:

$$(d_{k-1} \ d_{k-2} \ \dots \ d_2 \ d_1 \ d_0)_b \quad (1.1)$$

where the positions are counted from right to left, starting at 0 and increasing up to $k - 1$. The **Least Significant Digit** (LSD) is d_0 . The **Most Significant Digit** (MSD) is d_{k-1} .

The digit in position i is multiplied by the i -th power of the base. Summing all these contributions we obtain the number represented by the sequence.

Given a sequence of k digits in a positional numeral system in base b , the natural number it represents in base ten is given by the following sequence of digits:

$$\left(\sum_{i=0}^{k-1} d_i b^i \right)_{10} \quad (1.2)$$

Example 1.1: sequences of digits

The number one hundred written in the four positional numeral systems of the table above. The value of each sequence is computed with formula (1.2), position by position, starting from the least significant digit d_0 .

Sequence of digits	b	k	decimal representation
$(100)_2$	2	3	$(0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2)_{10} = (4)_{10}$
$(1100100)_2$	2	7	$(0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6)_{10}$ $= (4 + 32 + 64)_{10} = (100)_{10}$
$(144)_8$	8	3	$(4 \cdot 8^0 + 4 \cdot 8^1 + 1 \cdot 8^2)_{10} = (4 + 32 + 64)_{10} = (100)_{10}$
$(100)_{10}$	10	3	$(0 \cdot 10^0 + 0 \cdot 10^1 + 1 \cdot 10^2)_{10} = (100)_{10}$
$(64)_{16}$	16	2	$(4 \cdot 16^0 + 6 \cdot 16^1)_{10} = (4 + 96)_{10} = (100)_{10}$

The same sequence of digits 100 represents different numbers in different bases, and the same number needs more digits when the base is smaller.

Example 1.2: all the sequences of four binary digits

With $k = 4$ binary digits we can write $2^4 = 16$ different sequences, which represent the natural numbers from 0 to 15:

Sequence of digits	b	k	decimal representation
$(0000)_2$	2	4	$(0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)_{10} = (0)_{10}$
$(0001)_2$	2	4	$(0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10} = (1)_{10}$
$(0010)_2$	2	4	$(0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)_{10} = (2)_{10}$
$(0011)_2$	2	4	$(0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = (3)_{10}$
$(0100)_2$	2	4	$(0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)_{10} = (4)_{10}$
$(0101)_2$	2	4	$(0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10} = (5)_{10}$
$(0110)_2$	2	4	$(0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)_{10} = (6)_{10}$
$(0111)_2$	2	4	$(0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = (7)_{10}$
$(1000)_2$	2	4	$(1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)_{10} = (8)_{10}$
$(1001)_2$	2	4	$(1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10} = (9)_{10}$
$(1010)_2$	2	4	$(1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)_{10} = (10)_{10}$
$(1011)_2$	2	4	$(1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = (11)_{10}$
$(1100)_2$	2	4	$(1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0)_{10} = (12)_{10}$
$(1101)_2$	2	4	$(1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10} = (13)_{10}$
$(1110)_2$	2	4	$(1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0)_{10} = (14)_{10}$
$(1111)_2$	2	4	$(1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = (15)_{10}$

Typically, for sequences of digits in base ten, the parentheses are omitted, and if no parentheses are present, the base is assumed to be ten.

1.1.2 Representation intervals

The table of Example 1.2 stops at 15: with four binary digits there is no way of writing 16. In general, a fixed number k of digits can only represent the natural numbers up to a certain bound, which we now compute.

Observation 1.4

Given k digits in base b , the **largest natural number** that can be represented is:

$$b^k - 1 \quad (1.3)$$

and the associated **representation interval** for the natural numbers is:

$$[0, b^k - 1] \quad (1.4)$$

Proof. The largest natural number in base b with k digits consists of only the largest digit $(b - 1)$ repeated k times:

$$\left((b-1)_{k-1} \ (b-1)_{k-2} \ \dots \ (b-1)_1 \ (b-1)_0 \right)_b$$

Accordingly, using the sum of the geometric progression $\sum_{i=0}^{k-1} b^i = \frac{b^k - 1}{b - 1}$ (valid since $b \geq 2$), the largest natural number in decimal is:

$$\sum_{i=0}^{k-1} (b-1) b^i = (b-1) \sum_{i=0}^{k-1} b^i = (b-1) \frac{b^k - 1}{b - 1} = b^k - 1$$

The smallest natural number is 0, represented by k digits equal to 0. Every natural number between 0 and $b^k - 1$ can be represented: the b^k different sequences of k digits represent b^k different numbers, all in the interval $[0, b^k - 1]$, which contains exactly b^k natural numbers. $\square$

Digits in the binary system are called **bits** and 8 bits are called a **byte**.

Example 1.3: representation intervals in binary

For $b = 2$ the representation interval with k bits is $[0, 2^k - 1]$. The last three rows correspond to 1, 2, 4 and 8 bytes.

k	b	$b^k - 1$	representation interval
1	2	$2^1 - 1 = 1$	$[0, 1]$
2	2	$2^2 - 1 = 3$	$[0, 3]$
3	2	$2^3 - 1 = 7$	$[0, 7]$
4	2	$2^4 - 1 = 15$	$[0, 15]$
8	2	$2^8 - 1 = 255$	$[0, 255]$
16	2	$2^{16} - 1 = 65,535$	$[0, 65,535]$
32	2	$2^{32} - 1 = 4,294,967,295$	$[0, 4,294,967,295]^1$
64	2	$2^{64} - 1$	$[0, 2^{64} - 1]^2$

¹ Representation interval of **unsigned int** in C and C++ (4 bytes).

² Representation interval of **unsigned long long int** in C and C++ (8 bytes); $2^{64} = 18,446,744,073,709,551,616$.

The observation can also be read in the opposite direction: given a natural number n , how many digits do we need to write it?

Observation 1.5

The minimum number of digits to write a natural number $n \geq 1$ in base b is:

$$\lfloor \log_b n \rfloor + 1 \quad (1.5)$$

Proof. By Observation 1.4, the number n can be written with k digits if and only if:

$$n \leq b^k - 1 \quad \Longleftrightarrow \quad n < b^k \quad \Longleftrightarrow \quad \log_b n < k$$

where the first equivalence holds because n and b^k are integers, and the second because the logarithm in base $b \geq 2$ is an increasing function. The smallest integer k strictly greater than $\log_b n$ is $\lfloor \log_b n \rfloor + 1$. Accordingly, $\lfloor \log_b n \rfloor + 1$ is the minimum number of digits to write a natural number n in base b . $\square$

Example 1.4: minimum number of digits of one hundred

We compute the minimum number of digits of $n = 100$ in base 2 and in base 10.

- Step 1 (base 2): since $2^6 = 64 \leq 100 < 128 = 2^7$, we have $6 \leq \log_2 100 < 7$ (indeed $\log_2 100 \approx 6.644$), hence:

$$\lfloor \log_2 100 \rfloor + 1 = 6 + 1 = 7$$

- Step 2 (base 10): since $10^2 = 100$, we have $\log_{10} 100 = 2$, hence:

$$\lfloor \log_{10} 100 \rfloor + 1 = 2 + 1 = 3$$

Both results agree with Example 1.1: $(1100100)_2$ has 7 digits and $(100)_{10}$ has 3 digits.

Example 1.5: minimum number of digits in binary

For the powers of two $n = 2^m$ we have $\log_2 n = m$, so the minimum number of bits is $m + 1$: one digit 1 followed by m digits 0.

n	b	$\log_b n$	$\lfloor \log_b n \rfloor + 1$
$2 = 2^1$	2	$\log_2 2 = 1$	$\lfloor \log_2 2 \rfloor + 1 = 2$
$4 = 2^2$	2	$\log_2 4 = 2$	$\lfloor \log_2 4 \rfloor + 1 = 3$
$8 = 2^3$	2	$\log_2 8 = 3$	$\lfloor \log_2 8 \rfloor + 1 = 4$
$16 = 2^4$	2	$\log_2 16 = 4$	$\lfloor \log_2 16 \rfloor + 1 = 5$
$32 = 2^5$	2	$\log_2 32 = 5$	$\lfloor \log_2 32 \rfloor + 1 = 6$
$64 = 2^6$	2	$\log_2 64 = 6$	$\lfloor \log_2 64 \rfloor + 1 = 7$
$128 = 2^7$	2	$\log_2 128 = 7$	$\lfloor \log_2 128 \rfloor + 1 = 8$
$256 = 2^8$	2	$\log_2 256 = 8$	$\lfloor \log_2 256 \rfloor + 1 = 9$
$512 = 2^9$	2	$\log_2 512 = 9$	$\lfloor \log_2 512 \rfloor + 1 = 10$
$1,024 = 2^{10}$	2	$\log_2 1,024 = 10$	$\lfloor \log_2 1,024 \rfloor + 1 = 11$
$2,048 = 2^{11}$	2	$\log_2 2,048 = 11$	$\lfloor \log_2 2,048 \rfloor + 1 = 12$
$4,096 = 2^{12}$	2	$\log_2 4,096 = 12$	$\lfloor \log_2 4,096 \rfloor + 1 = 13$
$8,192 = 2^{13}$	2	$\log_2 8,192 = 13$	$\lfloor \log_2 8,192 \rfloor + 1 = 14$
$16,384 = 2^{14}$	2	$\log_2 16,384 = 14$	$\lfloor \log_2 16,384 \rfloor + 1 = 15$
$32,768 = 2^{15}$	2	$\log_2 32,768 = 15$	$\lfloor \log_2 32,768 \rfloor + 1 = 16$
$65,536 = 2^{16}$	2	$\log_2 65,536 = 16$	$\lfloor \log_2 65,536 \rfloor + 1 = 17$
$131,072 = 2^{17}$	2	$\log_2 131,072 = 17$	$\lfloor \log_2 131,072 \rfloor + 1 = 18$
$262,144 = 2^{18}$	2	$\log_2 262,144 = 18$	$\lfloor \log_2 262,144 \rfloor + 1 = 19$
$524,288 = 2^{19}$	2	$\log_2 524,288 = 19$	$\lfloor \log_2 524,288 \rfloor + 1 = 20$
$1,048,576 = 2^{20}$	2	$\log_2 1,048,576 = 20$	$\lfloor \log_2 1,048,576 \rfloor + 1 = 21$

Key points on natural numbers.

- The sequence $(d_{k-1} \dots d_1 d_0)_b$ represents the natural number $\sum_{i=0}^{k-1} d_i b^i$.
- With k digits in base b we represent exactly the b^k natural numbers of the interval $[0, b^k - 1]$.
- A natural number $n \geq 1$ needs at least $\lfloor \log_b n \rfloor + 1$ digits in base b .

1.2 Representation of integer numbers

Integer numbers include both positive and negative values, plus zero. With k bits we still have only 2^k different sequences, but now we have to use them to represent the sign as well. Two approaches are studied: **sign-magnitude**, which is the most natural, and **two's complement**, which is the standard in modern computers because of its arithmetic advantages. In this section the base is always $b = 2$.

1.2.1 Sign-magnitude representation

Definition 1.6: sign-magnitude representation

In the **sign-magnitude representation**, an integer number n is represented using exactly $k \geq 2$ bits, where:

- the most significant bit represents the **sign**: 0 if $n \geq 0$ and 1 if $n < 0$;
- the remaining $k - 1$ bits represent the **magnitude** $|n|$ (absolute value) in base 2.

Observation 1.7

Given k bits, the **largest integer number** that can be represented in sign-magnitude is:

$$2^{k-1} - 1 \quad (1.6)$$

and the associated **representation interval** for the integer numbers is:

$$[-(2^{k-1} - 1), 2^{k-1} - 1] \quad (1.7)$$

Proof. The magnitude is a natural number written with $k - 1$ bits. By Observation 1.4 with $b = 2$ and $k - 1$ digits, the largest magnitude consists of the digit 1 repeated $k - 1$ times:

$$\sum_{i=0}^{k-2} 1 \cdot 2^i = \frac{2^{k-1} - 1}{2 - 1} = 2^{k-1} - 1$$

With sign bit 0 we obtain the largest integer $2^{k-1} - 1$, and with sign bit 1 the smallest integer $-(2^{k-1} - 1)$. Every integer in between is represented, since its absolute value is at most $2^{k-1} - 1$. $\square$

Example 1.6: sign-magnitude with 4 bits

With $k = 4$ bits, the first bit is the sign and the last three bits are the magnitude:

Binary	k	Sign	Decimal
$(0000)_2$	4	0 (non-negative)	$+(0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0) = +0$
$(0001)_2$	4	0 (non-negative)	$+(0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = 1$
$(0010)_2$	4	0 (non-negative)	$+(0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = 2$
$(0011)_2$	4	0 (non-negative)	$+(0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) = 3$
$(0100)_2$	4	0 (non-negative)	$+(1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0) = 4$
$(0101)_2$	4	0 (non-negative)	$+(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = 5$
$(0110)_2$	4	0 (non-negative)	$+(1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = 6$
$(0111)_2$	4	0 (non-negative)	$+(1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) = 7$
$(1000)_2$	4	1 (negative)	$-(0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0) = -0$
$(1001)_2$	4	1 (negative)	$-(0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = -1$
$(1010)_2$	4	1 (negative)	$-(0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = -2$
$(1011)_2$	4	1 (negative)	$-(0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) = -3$
$(1100)_2$	4	1 (negative)	$-(1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0) = -4$
$(1101)_2$	4	1 (negative)	$-(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = -5$
$(1110)_2$	4	1 (negative)	$-(1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = -6$
$(1111)_2$	4	1 (negative)	$-(1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) = -7$

The representation interval is $[-7, 7]$, as stated by Observation 1.7 with $2^{4-1} - 1 = 7$.

The table of Example 1.6 shows the two weak points of sign-magnitude.

- Zero has **two representations**: $(0000)_2 = +0$ and $(1000)_2 = -0$. Hence only $2^k - 1$ different integers are represented with 2^k sequences.
- The addition of two integers cannot be performed as the addition of the two sequences seen as natural numbers. For example, $3 + (-1) = 2$, but adding $(0011)_2$ and $(1001)_2$ as natural numbers gives $3 + 9 = 12 = (1100)_2$, which in sign-magnitude represents -4 . The sign bits must be handled separately, and this complicates arithmetic operations.

For these reasons sign-magnitude is not the standard representation of integers in computers.

1.2.2 Two's complement representation

Two's complement solves both problems: zero has a unique representation, and addition works exactly as for natural numbers.

Definition 1.8: two's complement representation

In the **two's complement representation** (base 2), an integer n is represented using exactly k bits as follows:

- if $n \geq 0$, use the standard binary representation of n with k bits;
- if $n < 0$, the representation of n is the binary representation of the natural number $2^k + n$ with k bits.

In two's complement, the most significant bit indicates the sign:

- 0: non-negative number;
- 1: negative number.

To read a sequence whose most significant bit is 1, compute its value r as a natural number: the integer represented is $r - 2^k$.

- Zero has a unique representation: $(00 \dots 0)_2$.
- Addition and subtraction are performed in the same way as for natural numbers, modulo 2^k .
- This is the standard representation used in modern computers.

Example 1.7: two's complement with 4 bits

With $k = 4$ bits, the sequences with most significant bit 0 represent $0, 1, \dots, 7$; the sequences with most significant bit 1 represent their natural value minus $2^4 = 16$:

Binary	k	Sign	Decimal
$(0000)_2$	4	non-negative	0
$(0001)_2$	4	non-negative	1
$(0010)_2$	4	non-negative	2
$(0011)_2$	4	non-negative	3
$(0100)_2$	4	non-negative	4
$(0101)_2$	4	non-negative	5
$(0110)_2$	4	non-negative	6
$(0111)_2$	4	non-negative	7
$(1000)_2$	4	negative	$8 - 16 = -8$
$(1001)_2$	4	negative	$9 - 16 = -7$
$(1010)_2$	4	negative	$10 - 16 = -6$
$(1011)_2$	4	negative	$11 - 16 = -5$
$(1100)_2$	4	negative	$12 - 16 = -4$
$(1101)_2$	4	negative	$13 - 16 = -3$
$(1110)_2$	4	negative	$14 - 16 = -2$
$(1111)_2$	4	negative	$15 - 16 = -1$

Figure 1.1 arranges the 16 sequences of Example 1.7 on a circle: moving one step clockwise adds 1 to the natural value of the sequence, modulo $2^4 = 16$. The integers represented grow from -8 to 7 , and the only jump is between $(0111)_2 = 7$ and $(1000)_2 = -8$.

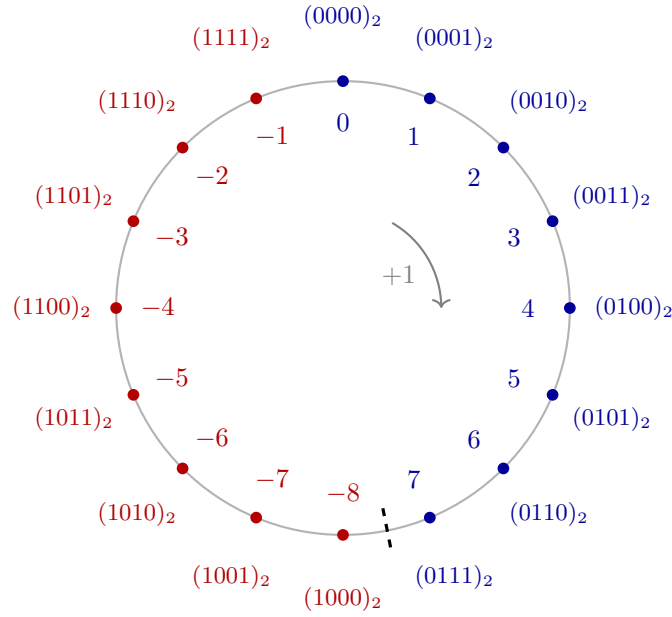


Figure 1.1: The 16 sequences of 4 bits on a circle, with the integers they represent in two's complement: non-negative integers in blue (most significant bit 0), negative integers in red (most significant bit 1). The dashed mark between $(0111)_2 = 7$ and $(1000)_2 = -8$ is the only point where the integer represented jumps.

Computing $2^k + n$ by a subtraction is not the only way to obtain the representation of a negative integer: the following observation gives a procedure that works directly on the bits.

Observation 1.9

Let n be an integer with $-2^{k-1} \leq n < 0$. The two's complement representation of n with k bits is obtained from the binary representation of $|n|$ with k bits in two steps:

1. invert all the bits (each 0 becomes 1 and each 1 becomes 0);
2. add 1 to the result, as a natural number.

Proof. Let $(d_{k-1} \dots d_1 d_0)_2$ be the binary representation of $|n|$ with k bits. Inverting the bits gives the digits $1 - d_i$, hence the natural number:

$$\sum_{i=0}^{k-1} (1 - d_i) 2^i = \sum_{i=0}^{k-1} 2^i - \sum_{i=0}^{k-1} d_i 2^i = (2^k - 1) - |n|$$

Adding 1 we obtain $2^k - |n| = 2^k + n$, which is the two's complement representation of n by Definition 1.8. Since $1 \leq |n| \leq 2^{k-1}$, we have $2^{k-1} \leq 2^k + n \leq 2^k - 1$, so the result still has k bits. $\square$

Example 1.8: two's complement of -5 with 8 bits

We compute the representation of $n = -5$ with $k = 8$ bits in both ways.

- Step 1: binary representation of $|n| = 5$ with 8 bits: $(00000101)_2$.
- Step 2: invert all the bits: $(11111010)_2$, whose natural value is $250 = (2^8 - 1) - 5$.
- Step 3: add 1: $(11111010)_2 + 1 = (11111011)_2$.
- Check with Definition 1.8: $2^8 + (-5) = 256 - 5 = 251 = 128 + 64 + 32 + 16 + 8 + 2 + 1 = (11111011)_2$.

1.2.3 Representation interval**Observation 1.10**

Given k bits, the largest integer that can be represented in two's complement is:

$$2^{k-1} - 1 \quad (1.8)$$

the smallest integer that can be represented is:

$$-2^{k-1} \quad (1.9)$$

and therefore the representation interval is:

$$[-2^{k-1}, 2^{k-1} - 1] \quad (1.10)$$

Proof. With k bits, there are 2^k possible binary strings, whose natural values are $0, 1, \dots, 2^k - 1$.

The strings whose most significant bit is 0 have natural values $0, 1, \dots, 2^{k-1} - 1$ and represent the non-negative integers:

$$0, 1, 2, \dots, 2^{k-1} - 1$$

The strings whose most significant bit is 1 have natural values $r = 2^{k-1}, \dots, 2^k - 1$ and represent the integers $r - 2^k$, i.e., the negative integers:

$$-2^{k-1}, -2^{k-1} + 1, \dots, -1$$

Hence the largest representable integer is $2^{k-1} - 1$, the smallest representable integer is -2^{k-1} , and the interval is:

$$[-2^{k-1}, 2^{k-1} - 1].$$

□

Compared with sign-magnitude, the interval contains one more integer, -2^{k-1} : the sequence

$(10 \dots 0)_2$, which was -0 in sign-magnitude, now represents -2^{k-1} . All the 2^k sequences represent different integers.

Example 1.9: representation intervals in two's complement

k	-2^{k-1}	$2^{k-1} - 1$	representation interval
1	-1	0	$[-1, 0]$
2	-2	1	$[-2, 1]$
3	-4	3	$[-4, 3]$
4	-8	7	$[-8, 7]$
8	-128	127	$[-128, 127]$
16	-32,768	32,767	$[-32,768, 32,767]$
32	-2,147,483,648	2,147,483,647	$[-2,147,483,648, 2,147,483,647]^1$
64	-2^{63}	$2^{63} - 1$	$[-2^{63}, 2^{63} - 1]^2$

¹ Representation interval of `int` in C and C++ (4 bytes).

² Representation interval of `long long int` in C and C++ (8 bytes); $2^{63} = 9,223,372,036,854,775,808$.

1.2.4 Minimum number of bits

Observation 1.11

The minimum number of bits needed to represent an integer n in two's complement is the smallest integer $k \geq 1$ such that

$$-2^{k-1} \leq n \leq 2^{k-1} - 1. \quad (1.11)$$

Proof. By Observation 1.10, an integer represented with k bits in two's complement belongs to the interval

$$[-2^{k-1}, 2^{k-1} - 1],$$

and every integer of this interval can be represented with k bits. Therefore, the minimum number of bits required to represent n is exactly the smallest k for which

$$-2^{k-1} \leq n \leq 2^{k-1} - 1.$$

□

As for the natural numbers, the condition of Observation 1.11 can be turned into an explicit formula with logarithms.

Corollary 1.12

The minimum number of bits needed to represent an integer n in two's complement is:

$$k = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = -1 \\ \lfloor \log_2 n \rfloor + 2 & \text{if } n > 0 \\ \lceil \log_2 |n| \rceil + 1 & \text{if } n < 0 \end{cases}$$

Proof. With $k = 1$ the interval is $[-1, 0]$, which gives the first case.

- If $n > 0$, the condition of Observation 1.11 reduces to $n \leq 2^{k-1} - 1$, i.e., $n < 2^{k-1}$, i.e., $\log_2 n < k - 1$. The smallest integer $k - 1$ strictly greater than $\log_2 n$ is $\lfloor \log_2 n \rfloor + 1$, hence $k = \lfloor \log_2 n \rfloor + 2$.
- If $n < 0$, the condition reduces to $-2^{k-1} \leq n$, i.e., $|n| \leq 2^{k-1}$, i.e., $\log_2 |n| \leq k - 1$. The smallest integer $k - 1$ greater than or equal to $\log_2 |n|$ is $\lceil \log_2 |n| \rceil$, hence $k = \lceil \log_2 |n| \rceil + 1$ (for $n = -1$ this gives $k = 1$, in agreement with the first case).

□

Example 1.10: minimum number of bits in two's complement

The last column shows the representation of n with the minimum number of bits.

n	minimum number of bits	representation
-1	1	1
0	1	0
1	2	01
-2	2	10
2	3	010
-4	3	100
4	4	0100
-8	4	1000
8	5	01000
-16	5	10000

For instance, for $n = 8$ Corollary 1.12 gives $\lfloor \log_2 8 \rfloor + 2 = 3 + 2 = 5$, and for $n = -16$ it gives $\lceil \log_2 16 \rceil + 1 = 4 + 1 = 5$.

1.2.5 Addition and overflow

The main advantage of two's complement is that integers are added exactly as natural numbers: we add the two sequences of k bits column by column, from right to left, with carries, and we discard the carry that goes out of the most significant position.

Observation 1.13

Let n_1 and n_2 be two integers represented with k bits in two's complement. Add their representations as natural numbers and keep only the k least significant bits of the sum.

- If $n_1 + n_2 \in [-2^{k-1}, 2^{k-1} - 1]$, the result is the two's complement representation of $n_1 + n_2$.
- Otherwise, the sum cannot be represented with k bits and we have an **overflow**. An overflow occurs if and only if n_1 and n_2 have the same sign and the result has the opposite sign bit.

Proof. Let r_1 and r_2 be the natural values of the two representations: by Definition 1.8, $r_j = n_j$ or $r_j = 2^k + n_j$, so $r_1 + r_2$ differs from $n_1 + n_2$ by 0, 2^k or 2^{k+1} . Keeping the k least significant bits means subtracting 2^k when $r_1 + r_2 \geq 2^k$ (the discarded carry). Hence the result r satisfies $r \in [0, 2^k - 1]$ and $r - (n_1 + n_2)$ is a multiple of 2^k .

If $n_1 + n_2 \in [-2^{k-1}, 2^{k-1} - 1]$, its representation $\bar{r}$ also lies in $[0, 2^k - 1]$ and $\bar{r} - (n_1 + n_2)$ is 0 or 2^k . Then $r - \bar{r}$ is a multiple of 2^k with $|r - \bar{r}| \leq 2^k - 1$, so $r = \bar{r}$.

If n_1 and n_2 have different signs, $n_1 + n_2$ lies between n_1 and n_2 , which belong to the interval: no overflow. If both are non-negative, $n_1 + n_2 \in [0, 2^k - 2]$, no carry is discarded and $r = n_1 + n_2$: an overflow occurs if and only if $n_1 + n_2 \geq 2^{k-1}$, i.e., if and only if the sign bit of r is 1. If both are negative, $r_1 + r_2 = 2^{k+1} + n_1 + n_2 \geq 2^k$, so $r = 2^k + n_1 + n_2$: an overflow occurs if and only if $n_1 + n_2 < -2^{k-1}$, i.e., if and only if $r < 2^{k-1}$, i.e., the sign bit of r is 0. $\square$

In the examples, the first row of each addition contains the carries: the carry written above a column is the one coming from the column on its right. The leftmost column receives the carry out of the most significant position, which is discarded.

Example 1.11: addition without overflow

We compute $3 + (-1)$ with $k = 4$ bits, interval $[-8, 7]$.

- Step 1: representations: $3 = (0011)_2$ and $-1 \rightarrow 2^4 - 1 = 15 = (1111)_2$.
- Step 2: column addition:

carry	1	1	1	1	
		0	0	1	1
		1	1	1	1
sum	1	0	0	1	0

- Step 3: discard the carry out: the result is $(0010)_2 = 2 = 3 + (-1)$.

Example 1.12: addition with overflow

We compute $6 + 3$ with $k = 4$ bits, interval $[-8, 7]$. Since $6 + 3 = 9 > 7$, an overflow is expected.

- Step 1: representations: $6 = (0110)_2$ and $3 = (0011)_2$.
- Step 2: column addition:

carry	0	1	1	0
		0	1	1
		0	0	1
sum	0	1	0	0

- Step 3: the result $(1001)_2$ has sign bit 1 and represents $9 - 16 = -7$, although both operands are positive: an overflow has occurred.

Key points on integer numbers.

- Sign-magnitude with k bits: interval $[-(2^{k-1} - 1), 2^{k-1} - 1]$, two representations of zero, addition requires a special treatment of the sign.
- Two's complement with k bits: a negative n is represented by $2^k + n$ (equivalently: invert the bits of $|n|$ and add 1); interval $[-2^{k-1}, 2^{k-1} - 1]$; a unique zero.
- In two's complement, addition is the addition of natural numbers modulo 2^k ; an overflow occurs only when two operands with the same sign give a result with the opposite sign.

1.3 Representation of real numbers

Natural and integer numbers are represented exactly, as long as they lie in the representation interval. Real numbers are different: between any two real numbers there are infinitely many others, and most real numbers (for instance $\frac{1}{3}$ or $\sqrt{2}$) have infinitely many digits in any base. With a finite number of bits, a real number can therefore be represented, in general, only **approximately**. In this section we study the floating-point representation used by computers.

1.3.1 Digits after the point

Positional numeral systems extend to the digits after the point: the digits on the right of the point are multiplied by the negative powers of the base.

Definition 1.14: sequence of digits with a fractional part

A sequence of k digits before the point and h digits after the point in base b is denoted as:

$$(d_{k-1} \dots d_1 d_0 . d_{-1} d_{-2} \dots d_{-h})_b \quad (1.12)$$

and represents the number:

$$\sum_{i=-h}^{k-1} d_i b^i \quad (1.13)$$

Example 1.13: binary numbers with a fractional part

- $(101.11)_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 4 + 1 + 0.5 + 0.25 = 5.75$
- $(0.011)_2 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.25 + 0.125 = 0.375$
- The decimal number 0.1 has no finite binary representation: its binary digits after the point repeat forever,

$$0.1 = (0.0001100110011\dots)_2$$

Writing a number with a fixed number of digits before and after the point (**fixed point**) wastes digits: very large numbers need many digits before the point, very small numbers many digits after it. The floating-point representation lets the point “float”, as in the scientific notation 6.02×10^{23} .

Definition 1.15: normalized binary scientific notation

Every real number $x \neq 0$ can be written in the **normalized binary scientific notation**:

$$x = (-1)^s \cdot (1.f_1 f_2 f_3 \dots)_2 \cdot 2^e \quad (1.14)$$

where $s \in \{0, 1\}$ is the **sign**, the sequence $(1.f_1 f_2 f_3 \dots)_2$ is the **significand** (or **mantissa**), whose bits $f_1, f_2, f_3, \dots$ after the point form the **fraction**, and the integer e is the **exponent**.

The significand is always between 1 (included) and 2 (excluded): its first digit is always 1, so it does not need to be stored. For example, $5.75 = (101.11)_2 = (1.0111)_2 \cdot 2^2$: the point moves two positions to the left and the exponent is 2.

1.3.2 The IEEE 754 standard

Computers store floating-point numbers according to the **IEEE 754** standard (IEEE: Institute of Electrical and Electronics Engineers). We describe its two most common formats, focusing on the 32-bit one.

Definition 1.16: IEEE 754 single precision

In the **IEEE 754 single precision** format, a real number is stored with 32 bits divided into three fields:

- the **sign** s : 1 bit;
- the **biased exponent** E : 8 bits, storing the natural number $E = e + 127$, where e is the exponent and 127 is the **bias**;
- the **fraction** f : 23 bits, storing the bits $f_1 f_2 \dots f_{23}$ of the significand after the point.

For $1 \leq E \leq 254$ (**normalized numbers**), the number represented is:

$$x = (-1)^s \cdot (1.f_1 f_2 \dots f_{23})_2 \cdot 2^{E-127} \quad (1.15)$$

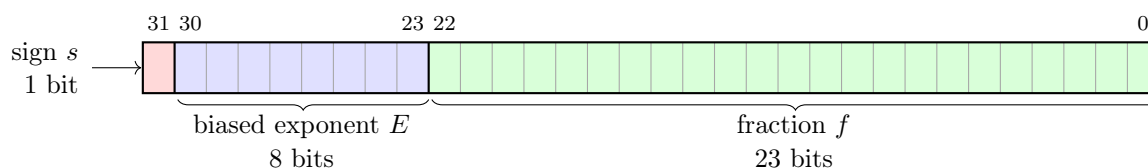


Figure 1.2: The 32 bits of the IEEE 754 single precision format, numbered from 31 (most significant) to 0 (least significant).

- The exponent is stored with a bias instead of in two's complement, so that E is a natural number between 0 and 255.
- The two extreme values of E are reserved. $E = 0$ is used for zero (all fraction bits equal to 0; there are two zeros, $+0$ and -0 , as in sign-magnitude) and for very small numbers called *subnormal*. $E = 255$ is used for $+\infty$ and $-\infty$ (fraction equal to 0, e.g., the result of $1/0$) and for *NaN*, “Not a Number” (fraction different from 0, e.g., the result of $0/0$).
- The **double precision** format has the same structure with 64 bits: 1 bit of sign, 11 bits of exponent with bias 1023, and 52 bits of fraction.

Format	total bits	sign	exponent	fraction	bias
single precision (C++ <code>float</code>)	32	1	8	23	127
double precision (C++ <code>double</code>)	64	1	11	52	1023

Example 1.14: from a real number to its IEEE 754 single precision representation

We represent $x = 5.75$ in single precision.

- Step 1 (binary): $5.75 = 4 + 1 + 0.5 + 0.25 = (101.11)_2$.
- Step 2 (normalization): $(101.11)_2 = (1.0111)_2 \cdot 2^2$, hence $s = 0$, $e = 2$ and the fraction starts with 0111.
- Step 3 (biased exponent): $E = e + 127 = 129 = 128 + 1 = (10000001)_2$.
- Step 4 (fraction): the bits after the point, completed with zeros up to 23 bits: 01110000000000000000000.

The 32 bits are:

$$\underbrace{0}_s \quad \underbrace{10000001}_E \quad \underbrace{01110000000000000000000}_f$$

Example 1.15: from an IEEE 754 single precision representation to a real number

We read the 32 bits:

$$\underbrace{1}_s \quad \underbrace{10000010}_E \quad \underbrace{01000000000000000000000}_f$$

- Step 1 (sign): $s = 1$, the number is negative.
- Step 2 (exponent): $E = (10000010)_2 = 128 + 2 = 130$, hence $e = E - 127 = 3$.
- Step 3 (significand): $(1.01)_2 = 1 + 0.25 = 1.25$.
- Step 4 (value): $x = (-1)^1 \cdot 1.25 \cdot 2^3 = -10$.

When the binary digits of a number do not fit in the 23 bits of the fraction, the number is **rounded** to the nearest representable one.

Example 1.16: the number 0.1 is not representable exactly

By Example 1.13, $0.1 = (0.00110011\dots)_2 = (1.100110011001\dots)_2 \cdot 2^{-4}$, whose fraction never ends. In single precision the fraction is rounded to 23 bits, and the bits stored are:

$$\underbrace{0}_s \quad \underbrace{01111011}_{E=123} \quad \underbrace{10011001100110011001101}_f$$

The number actually stored is 0.100000001490116119384765625, not 0.1. In double precision the error is smaller but still present: the stored value is 0.1000000000000000055511151231..., and for this reason the computation $0.1 + 0.2$ in double precision gives 0.30000000000000004 and not the double precision value of 0.3.

1.3.3 Range and precision

The finite number of bits of the exponent limits the **range** of the representable numbers, and the finite number of bits of the fraction limits their **precision**.

Observation 1.17

In IEEE 754 single precision:

- the largest finite number is $(2 - 2^{-23}) \cdot 2^{127} \approx 3.4028235 \times 10^{38}$;
- the smallest positive normalized number is $2^{-126} \approx 1.1754944 \times 10^{-38}$;
- the distance between 1 and the next representable number is $2^{-23} \approx 1.1920929 \times 10^{-7}$ (called the **machine epsilon**).

Proof. The largest finite number has $s = 0$, the largest normalized biased exponent $E = 254$, i.e., $e = 127$, and all 23 fraction bits equal to 1. Its significand is:

$$(1.\underbrace{11\dots1}_{23})_2 = \sum_{i=0}^{23} 2^{-i} = 2 - 2^{-23}$$

hence the number is $(2 - 2^{-23}) \cdot 2^{127}$. The smallest positive normalized number has $E = 1$, i.e., $e = -126$, and all fraction bits equal to 0, hence it is $1 \cdot 2^{-126}$. Finally, $1 = (1.00\dots0)_2 \cdot 2^0$, and the next representable number is obtained by setting to 1 the last fraction bit, which has weight 2^{-23} when $e = 0$: it is $1 + 2^{-23}$. $\square$

The machine epsilon measures the **relative precision** of the format: single precision corresponds to about 7 significant decimal digits, double precision (machine epsilon $2^{-52} \approx 2.2 \times 10^{-16}$) to about 16. As a consequence, two expressions that are mathematically equal may give different floating-point results, and floating-point numbers should not be compared with the equality operator without care.

Key points on real numbers.

- A floating-point number is stored as a sign, an exponent and a fraction: $x = (-1)^s \cdot (1.f)_2 \cdot 2^{E-\text{bias}}$.
- IEEE 754 single precision: 1 + 8 + 23 bits, bias 127; double precision: 1 + 11 + 52 bits, bias 1023.
- The exponent bits determine the range, the fraction bits the precision; most real numbers (for instance 0.1) are stored with a rounding error.

Pseudo-code

Introduction

In the previous chapter we have seen how the data of an algorithm, the numbers, are represented. In this chapter we fix the language used in these notes to describe the algorithms themselves: the **pseudo-code**. Pseudo-code is precise enough to describe every step of an algorithm without ambiguity, and simple enough to be read without knowing any programming language.

The chapter introduces the elements of pseudo-code one at a time, from the simplest to the most structured:

- variables, declarations and assignments, and the basic operations on them;
- the basic constructs that control the order of execution: conditional statements, loops, and the loop control statements **break** and **continue**;
- arrays with one, two and more indices;
- functions, which give a name to an algorithm, with its input and its output;
- sets.

Each construct is followed by examples, executed step by step on small instances. Executing an algorithm by hand, while keeping track of the values of its variables, is the best way to understand what it does.

2.1 Introduction to pseudo-code

Definition 2.1: algorithm

An **algorithm** is a finite sequence of unambiguous instructions that, starting from some data given as **input**, produces a result given as **output** in a finite number of steps.

Definition 2.2: pseudo-code

Pseudo-code is a high-level description of an algorithm that uses a mixture of natural language and programming language-like structures.

- Pseudo-code is not meant to be executed by a computer, but rather to communicate the logic of an algorithm to humans.
- Pseudo-code allows us to focus on the logic and structure of an algorithm without worrying about the syntax of a specific programming language.
- **Convention:** in these notes, all reserved words (keywords) of pseudo-code are written in bold, e.g., **if**, **while**, **for**, **return**, etc.

Advantages of pseudo-code:

- Language-independent: it can be understood by programmers of different languages
- Easy to write and modify
- Focuses on the logic of the algorithm rather than on implementation details
- Facilitates communication between developers and non-technical stakeholders
- Useful for planning before actual coding

Key principles:

- Use clear and descriptive variable names
- Keep statements simple and concise
- Use indentation to show structure and nesting
- Use standard constructs (**if-then-else**, **while**, **for**, etc.)
- Include comments when necessary to clarify complex operations

Recommended practices:

- **Indentation:** use consistent indentation to show the block structure
- **Variables:** use descriptive names in italics or regular text
- **Comments:** add comments (in natural language) to explain complex steps
- **Input/Output:** clearly specify the input parameters and the output (return values)
- **Line numbers:** useful for referencing specific lines
- **Consistency:** maintain the same notation and style throughout

2.2 Variables and assignments

The data of an algorithm are stored in **variables**: named containers whose value can change during the execution. A variable is first created with a declaration and then modified with assignments.

2.2.1 Variables

- Variables must be declared before use
- The declaration operator is “:=”
- The declaration specifies the variable name and its initial value

```
1 variable := initial_value
```

Example 2.1: variable declaration

Seven variables declared with their initial values: integer, rational, $-\infty$ and logical (**true/false**) values are all allowed.

```
1 n := 5
2 m := 10
3 x := -3/5
4 sum := 0
5 count := 1
6 max :=  $-\infty$ 
7 status := true
```

2.2.2 Assignment

- The assignment operator is “ $\leftarrow$ ”
- It assigns the value of an expression to a previously declared variable: first the expression on the right is evaluated, then its value replaces the old value of the variable

```
1 variable  $\leftarrow$  expression
```

Example 2.2: assignment

With the variables of Example 2.1:

```
1  $n \leftarrow 10$ 
2  $sum \leftarrow n + m$ 
3  $count \leftarrow count + 1$ 
```

After the three assignments $n = 10$, $sum = 10 + 10 = 20$ and $count = 1 + 1 = 2$. In the last assignment the same variable appears on both sides: the old value 1 is used to compute the new value 2.

Example 2.3: exchanging the values of two variables

To exchange the values of two variables x and y we need a third, temporary variable tmp : the assignment $x \leftarrow y$ alone would overwrite the old value of x .

```
1  $tmp := x$ 
2  $x \leftarrow y$ 
3  $y \leftarrow tmp$ 
```

Step-by-step execution with $x = 3$ and $y = 8$:

statement	x	y	tmp
–	3	8	–
$tmp := x$	3	8	3
$x \leftarrow y$	8	8	3
$y \leftarrow tmp$	8	3	3

2.3 Basic operations

Expressions combine variables and values with the following operations.

Mathematical operations:

- $+$, $-$, $\times$, $/$: basic arithmetic
- $a \bmod b$: modulo, i.e., the remainder of the integer division of a by b
- $\lfloor x \rfloor$: floor (greatest integer $\leq x$)
- $\lceil x \rceil$: ceiling (smallest integer $\geq x$)
- $|x|$: absolute value
- $\max(a, b)$, $\min(a, b)$: maximum and minimum

Comparison operators:

- $=$, $\neq$: equal, not equal
- $<$, $\leq$, $>$, $\geq$: less than, less than or equal to, greater than, greater than or equal to

Logical operators:

- **and**, **or**, **not**: logical operations

Input/Output:

- **read**(variable): read input from the user
- **print**(expression): output a value

Example 2.4: evaluating expressions

- $17 \bmod 5 = 2$, since $17 = 3 \cdot 5 + 2$; in particular, an integer i is a multiple of 3 if and only if $i \bmod 3 = 0$.
- $\lfloor 7/2 \rfloor = \lfloor 3.5 \rfloor = 3$ and $\lceil 7/2 \rceil = \lceil 3.5 \rceil = 4$.
- $|-3| = 3$, $\max(-3, 2) = 2$ and $\min(-3, 2) = -3$.
- $(5 > 3)$ **and** $(2 > 4)$ is **false**, while $(5 > 3)$ **or** $(2 > 4)$ is **true**.

2.4 Basic constructs

The statements of pseudo-code are executed one after the other, from top to bottom. Two kinds of constructs change this order: **conditional statements**, which execute a block only when a condition holds, and **loops**, which execute a block several times.

2.4.1 Conditional statements

2.4.1.1 If-then

- Executes a block of statements only if a condition is true

```

1 if condition then
2   | statements
3 end

```

Example 2.5: if-then

Checking whether a number x is positive:

```

1 positive := false
2 if  $x > 0$  then
3   | positive ← true
4   | print( $x$  is positive)
5 end

```

Step-by-step execution:

Instance 1: $x = 5$

x	$x > 0?$	<i>positive</i>
–	–	false
5	Yes	true

Output: 5 is positive

Instance 2: $x = -3$

x	$x > 0?$	<i>positive</i>
–	–	false
-3	No	false

Output: nothing is printed.

2.4.1.2 If-then-else

- Executes one block if the condition is true, and another block if it is false

```

1 if condition then
2   | statements when true
3 else
4   | statements when false
5 end

```

Example 2.6: if-then-else

Computing the absolute value of a number x :

```

1 result := 0
2 if  $x \geq 0$  then
3   | result  $\leftarrow x$ 
4 else
5   | result  $\leftarrow -x$ 
6 end

```

Step-by-step execution:

Instance 1: $x = 7$

x	$x \geq 0?$	<i>result</i>
–	–	0
7	Yes	7

Result: $result = 7$

Instance 2: $x = -4$

x	$x \geq 0?$	<i>result</i>
–	–	0
-4	No	4

Result: $result = 4$

2.4.2 Loops

2.4.2.1 While loop

- Repeats a block of statements while a condition is true
- The condition is checked before each iteration
- If the condition is initially false, the loop body is never executed

```

1 while condition do
2   | statements
3 end

```

Example 2.7: while loop

Computing the sum of the numbers from 1 to n :

```

1 sum := 0
2 i := 1
3 while  $i \leq n$  do
4   |  $sum \leftarrow sum + i$ 
5   |  $i \leftarrow i + 1$ 
6 end

```

Step-by-step execution:

Instance: $n = 5$. Each row shows the value of i when the condition is checked, the outcome of the check, and the values of sum and i at the end of the iteration.

i	$i \leq n?$	sum	i (after)
–	–	0	1
1	Yes	1	2
2	Yes	3	3
3	Yes	6	4
4	Yes	10	5
5	Yes	15	6
6	No	15	–

Result: $sum = 15$

In the previous example the number of iterations is known in advance. The while loop is necessary when it is not, as in the next example.

Example 2.8: while loop

Computing how many times a number x must be halved before it becomes smaller than or equal to a threshold $\varepsilon > 0$:

```

1 count := 0
2 while  $x > \varepsilon$  do
3   |  $x \leftarrow x/2$ 
4   |  $\text{count} \leftarrow \text{count} + 1$ 
5 end

```

Step-by-step execution:

Instance 1: $x = 10, \varepsilon = 1$

x	count	$x > \varepsilon?$	x (after)	count (after)
10	0	Yes	5	1
5	1	Yes	2.5	2
2.5	2	Yes	1.25	3
1.25	3	Yes	0.625	4
0.625	4	No $\Rightarrow$ exit	–	4

Result: $\text{count} = 4$

Instance 2: $x = 100, \varepsilon = 10$

x	count	$x > \varepsilon?$	x (after)	count (after)
100	0	Yes	50	1
50	1	Yes	25	2
25	2	Yes	12.5	3
12.5	3	Yes	6.25	4
6.25	4	No $\Rightarrow$ exit	–	4

Result: $\text{count} = 4$

2.4.2.2 For loop

- Repeats a block of statements for a specified number of times
- Typically used when the number of iterations is known in advance

```

1 for  $\text{variable} := \text{start\_value}$  to  $\text{end\_value}$  do
2   | statements
3 end

```

The for loop is a compact way of writing a particular while loop: the loop variable is declared

with the start value, the condition compares it with the end value, and it is increased by 1 at the end of each iteration.

The loop on the left is equivalent to the loop on the right:

```

1 for  $i := s$  to  $e$  do
2   | statements
3 end

```

```

1  $i := s$ 
2 while  $i \leq e$  do
3   | statements
4   |  $i \leftarrow i + 1$ 
5 end

```

In particular, the body is executed $e - s + 1$ times if $s \leq e$, and it is never executed if $s > e$.

Example 2.9: for loop

Computing the sum of the numbers from 1 to n with a for loop (compare with Example 2.7):

```

1  $sum := 0$ 
2 for  $i := 1$  to  $n$  do
3   |  $sum \leftarrow sum + i$ 
4 end

```

Step-by-step execution:

Instance: $n = 5$

i	$i \leq n?$	sum
–	–	0
1	Yes	1
2	Yes	3
3	Yes	6
4	Yes	10
5	Yes	15
6	No	15

Result: $sum = 15$

The for loop can optionally use a **step** value, to increase the loop variable by values other than 1.

Example 2.10: for loop with stepComputing the sum of the first n even numbers $(2, 4, \dots, 2n)$:

```

1 sum := 0
2 for i := 2 to 2n step 2 do
3   | sum ← sum + i
4 end

```

Step-by-step execution:**Instance:** $n = 4$ (even numbers: 2, 4, 6, 8)

i	$i \leq 2n?$	sum
–	–	0
2	Yes	2
4	Yes	6
6	Yes	12
8	Yes	20
10	No	20

Result: $sum = 20$ **2.4.3 Loop control statements**

Sometimes a loop must be interrupted before its natural end, or a single iteration must be skipped. Two statements serve these purposes.

2.4.3.1 Break statement

- The **break** statement immediately exits the current loop
- Execution continues with the first statement after the loop
- Useful when a termination condition is met before the natural completion of the loop

```

1 while condition do
2   | statements
3   | if exit_condition then
4     | statements
5     | break
6   | end
7   | statements
8 end

```

```

1 for variable := start_value to end_value do
2   statements
3   if exit_condition then
4     statements
5     break
6   end
7   statements
8 end

```

Example 2.11: break statement

Computing the sum of the numbers from 1 to n , stopping as soon as the sum exceeds a threshold t .

With a while loop:

```

1 sum := 0
2 i := 1
3 while  $i \leq n$  do
4    $sum \leftarrow sum + i$ 
5   if  $sum > t$  then
6     break
7   end
8    $i \leftarrow i + 1$ 
9 end

```

With a for loop:

```

1 sum := 0
2 for  $i := 1$  to  $n$  do
3    $sum \leftarrow sum + i$ 
4   if  $sum > t$  then
5     break
6   end
7 end

```

Step-by-step execution (both loops):

Instance: $n = 6, t = 10$

i	sum	$sum > t?$
–	0	–
1	1	No
2	3	No
3	6	No
4	10	No
5	15	Yes $\Rightarrow$ break

Result: $sum = 15$; the iteration with $i = 6$ is never executed.

2.4.3.2 Continue statement

- The **continue** statement skips the rest of the current iteration
- Execution proceeds immediately with the next iteration of the loop (in a while loop, with the check of the condition)
- Useful when certain iterations should be skipped based on a condition

```

1 while condition do
2   statements
3   if skip_condition then
4     statements
5     continue
6   end
7   statements
8 end

```

```

1 for variable := start_value to end_value do
2   statements
3   if skip_condition then
4     statements
5     continue
6   end
7   statements
8 end

```

Example 2.12: continue statement

Computing the sum of the numbers from 1 to n , skipping the multiples of 3.

With a while loop:

```

1 sum := 0
2 i := 1
3 while i ≤ n do
4   if i mod 3 = 0 then
5     | i ← i + 1
6     | continue
7   end
8   sum ← sum + i
9   i ← i + 1
10 end

```

With a for loop:

```

1 sum := 0
2 for i := 1 to n do
3   if i mod 3 = 0 then
4     | continue
5   end
6   sum ← sum + i
7 end

```

In the while loop the variable i must be increased before **continue**: otherwise the condition $i \bmod 3 = 0$ would remain true forever and the loop would never end. In the for loop the increase is automatic.

Step-by-step execution (both loops):

Instance: $n = 7$

i	$i \bmod 3 = 0?$	sum
–	–	0
1	No	1
2	No	3
3	Yes	3
4	No	7
5	No	12
6	Yes	12
7	No	19

Result: $sum = 19 = 1 + 2 + 4 + 5 + 7$

Key points on the basic constructs.

- **if-then** and **if-then-else** choose which block to execute according to a condition.
- **while** repeats a block as long as a condition is true, checking it before each iteration;
for repeats a block for each value of a loop variable from a start value to an end value.

- **break** exits the loop; **continue** skips to the next iteration.
- Executing a construct step by step on a small instance, with a table of the values of the variables, shows exactly what it does.

2.5 Arrays

Many algorithms work on a collection of values of the same kind, for instance a list of n numbers. Instead of declaring n different variables, we use an array.

- Arrays are indexed collections of variables and must be declared before use
- The declaration operator is “new array” followed by the index range

```
1 array_name := new array[start..end]
```

Example 2.13: array declaration

Creating an array of size n , indexed from 1 to n :

```
1 a := new array[1..n]
```

- The notation $a[i]$ denotes the variable at index i
- Arrays are typically indexed starting from 0 or 1: in the pseudo-code of these notes the indices start from 1, so an array of n elements is $a[1..n]$; in many programming languages, such as C++, the indices start from 0
- Each variable in the array must be initialized before access and use
- To initialize all the elements to the same value, use a **for** loop:

```
1 for i := start to end do
2   | array_name[i] ← initial_value
3 end
```

Example 2.14: array assignment

Initializing and accessing the variables of an array $a[1..n]$ (with $n \geq 5$):

```

// Initialize all variables in the array to zero
1 for i := 1 to n do
2   | a[i] ← 0
3 end

// Access and modify
4 x := a[5]
5 a[3] ← 42

```

With $n = 5$ the array is $[0, 0, 0, 0, 0]$ after the loop, $x = 0$, and the array is $[0, 0, 42, 0, 0]$ at the end.

2.5.1 Two-dimensional arrays

- Two-dimensional arrays are indexed collections of variables with two indices and must be declared before use
- The declaration operator is “new array” followed by the two index ranges
- Two-dimensional arrays are typically indexed starting from 0 or 1 (from 1 in these notes)

```

1 array_name := new array[start1..end1][start2..end2]

```

Example 2.15: two-dimensional array declaration

Creating a two-dimensional array of size $m \times n$ (indexed from 1 to m and from 1 to n):

```

1 b := new array[1..m][1..n]

```

- The notation $b[i][j]$ denotes the variable at row i and column j
- Each variable in the array must be initialized before access and use
- To initialize all the elements to the same value, use nested **for** loops:

```

1 for i := start1 to end1 do
2   | for j := start2 to end2 do
3     | array_name[i][j] ← initial_value
4   | end
5 end

```

Example 2.16: two-dimensional array assignment

Initializing and accessing the variables of a two-dimensional array $b[1..m][1..n]$ (with $m \geq 2$ and $n \geq 3$):

```
// Initialize all variables in the two-dimensional array to zero
1 for i := 1 to m do
2   |   for j := 1 to n do
3   |   |   b[i][j] ← 0
4   |   end
5 end

// Access and modify
6 x := b[2][3]
7 b[1][1] ← 42
```

The outer loop runs over the rows and, for each row i , the inner loop runs over all the columns: the elements are visited in the order $b[1][1], b[1][2], \dots, b[1][n], b[2][1], \dots, b[m][n]$, for a total of $m \cdot n$ assignments.

2.5.2 Multi-dimensional arrays

- Multi-dimensional arrays are indexed collections of variables with multiple indices and must be declared before use
- The declaration operator is “new array” followed by the index ranges for each dimension

```
1 array_name := new array[start1..end1][start2..end2]...[start_k..end_k]
```

Example 2.17: multi-dimensional array declaration

Creating a 3-dimensional array of size $p \times q \times r$ (indexed from 1 to p , from 1 to q , and from 1 to r):

```
1 t := new array[1..p][1..q][1..r]
```

- The notation $t[i_1][i_2] \dots [i_k]$ denotes the variable at indices $i_1, i_2, \dots, i_k$
- Multi-dimensional arrays are typically indexed starting from 0 or 1 (from 1 in these notes)
- Each variable in the array must be initialized before access and use
- To initialize all the elements to the same value, use nested **for** loops (one per dimension):

```

1 for  $i_1 := start_1$  to  $end_1$  do
2   for  $i_2 := start_2$  to  $end_2$  do
3     :
4     for  $i_k := start_k$  to  $end_k$  do
5       | array_name[ $i_1$ ][ $i_2$ ]  $\cdots$  [ $i_k$ ]  $\leftarrow$  initial_value
6     end
7   end
8 end

```

Example 2.18: multi-dimensional array assignment

Initializing and accessing a 3-dimensional array $t[1..p][1..q][1..r]$ (with $p \geq 2$, $q \geq 2$ and $r \geq 3$):

```

// Initialize all variables in the 3-dimensional array to zero
1 for  $i := 1$  to  $p$  do
2   for  $j := 1$  to  $q$  do
3     for  $k := 1$  to  $r$  do
4       |  $t[i][j][k] \leftarrow 0$ 
5     end
6   end
7 end

// Access and modify
8  $x := t[1][2][3]$ 
9  $t[2][1][3] \leftarrow 42$ 

```

2.6 Functions

The examples seen so far are fragments of pseudo-code that work on variables assumed to exist. A function turns such a fragment into a complete algorithm, with a name, an input and an output.

- A **function** is a named, reusable block of code that performs a specific task
- A function takes **input** data through its **parameters** and produces **output** data
- The **return** statement terminates the function and specifies its results
- Functions are typically used to implement **algorithms** or to perform specific computations that can be reused in different parts of a program

Syntax:

Input : description of the input parameters

Output: description of the output / return value

```

1 function FunctionName(parameter1, parameter2, ..., parameterk)
2   | statements
3   | return result
4 end

```

In the rest of these notes every algorithm is written as a function, inside a red box. The input states the **preconditions** on the data (for instance, that an array has at least one element): the function is guaranteed to work only when they hold.

Example 2.19: maximum value in an array

A function returning the maximum value in an array: the variable *max* starts from the first element and is updated whenever a larger element is found.

Algorithm: FindMaximum

Input : An array $a[1..n]$ of $n \geq 1$ integer numbers

Output: The maximum value in the array a

```

1 function FindMaximum( $a, n$ )
2   |  $max := a[1]$ 
3   | for  $i := 2$  to  $n$  do
4   |   | if  $a[i] > max$  then
5   |   |   |  $max \leftarrow a[i]$ 
6   |   | end
7   | end
8   | return  $max$ 
9 end

```

Step-by-step execution:

Instance 1: $a = [9, 12, 3, 16, 19, 11, 12]$, $n = 7$

i	$a[i]$	$a[i] > max?$	max
–	–	–	9
2	12	Yes	12
3	3	No	12
4	16	Yes	16
5	19	Yes	19
6	11	No	19
7	12	No	19

Return: $max = 19$

Instance 2: $a = [3, 7, 2, 9, 1]$, $n = 5$

i	$a[i]$	$a[i] > max?$	max
—	—	—	3
2	7	Yes	7
3	2	No	7
4	9	Yes	9
5	1	No	9

Return: $max = 9$

Example 2.20: counting the positive elements of an array

A function counting the number of positive elements in an array:

Algorithm: CountPositive

Input : An array $a[1..n]$ of n integer numbers

Output: The number of positive elements in the array a

```

1 function CountPositive( $a, n$ )
2    $count := 0$ 
3   for  $i := 1$  to  $n$  do
4     if  $a[i] > 0$  then
5        $count \leftarrow count + 1$ 
6     end
7   end
8   return  $count$ 
9 end

```

Step-by-step execution:

Instance 1: $a = [-2, 5, -1, 3, 0]$, $n = 5$

i	$a[i]$	$a[i] > 0?$	$count$
—	—	—	0
1	-2	No	0
2	5	Yes	1
3	-1	No	1
4	3	Yes	2
5	0	No	2

Return: $count = 2$

Instance 2: $a = [4, -3, 7, 2]$, $n = 4$

i	$a[i]$	$a[i] > 0?$	$count$
—	—	—	0
1	4	Yes	1
2	−3	No	1
3	7	Yes	2
4	2	Yes	3

Return: $count = 3$

Example 2.21: checking whether an array is sorted

A function checking whether an array is sorted in non-decreasing order. The **return** statement inside the loop terminates the function as soon as two consecutive elements in the wrong order are found.

Algorithm: IsSorted

Input : An array $a[1..n]$ of $n \geq 1$ integer numbers

Output: **true** if the values in the array a are sorted in non-decreasing order and **false** otherwise

```

1 function IsSorted( $a, n$ )
2   for  $i := 1$  to  $n - 1$  do
3     if  $a[i] > a[i + 1]$  then
4       return false
5     end
6   end
7   return true
8 end

```

Step-by-step execution:

Instance 1: $a = [2, 5, 7, 9]$, $n = 4$

i	$a[i]$	$a[i+1]$	$a[i] > a[i+1]?$
1	2	5	No
2	5	7	No
3	7	9	No

Return: **true**

Instance 2: $a = [3, 1, 4, 2]$, $n = 4$

i	$a[i]$	$a[i+1]$	$a[i] > a[i+1]?$
1	3	1	Yes

Return: **false**, after a single comparison.

Example 2.22: scalar product of two arrays

A function computing the scalar product $\sum_{i=1}^n a[i] \cdot b[i]$ of two arrays:

Algorithm: ScalarProduct

Input : Two arrays $a[1..n]$ and $b[1..n]$ of size $n \geq 1$

Output: The scalar product of the two arrays

```

1 function ScalarProduct(a, b, n)
2   prod := a[1] · b[1]
3   for i := 2 to n do
4     | prod ← prod + a[i] · b[i]
5   end
6   return prod
7 end

```

Step-by-step execution:

Instance 1: $a = [1, 2, 3]$, $b = [4, 5, 6]$, $n = 3$

i	$a[i]$	$b[i]$	$a[i] \cdot b[i]$	$prod$
–	–	–	–	$1 \cdot 4 = 4$
2	2	5	10	$4 + 10 = 14$
3	3	6	18	$14 + 18 = 32$

Return: $prod = 32$

Instance 2: $a = [2, 0, -1]$, $b = [3, 4, 5]$, $n = 3$

i	$a[i]$	$b[i]$	$a[i] \cdot b[i]$	$prod$
–	–	–	–	$2 \cdot 3 = 6$
2	0	4	0	$6 + 0 = 6$
3	–1	5	–5	$6 + (–5) = 1$

Return: $prod = 1$

Example 2.23: reversing an array

A function reversing the elements of an array: the element $a[i]$ is exchanged with its symmetric element $a[n - i + 1]$, with the temporary variable of Example 2.3. Only $\lfloor n/2 \rfloor$ exchanges are needed: with n odd, the middle element stays in place.

Algorithm: ReverseArray**Input** : An array $a[1..n]$ of n numbers**Output**: The same array a with the elements in reversed order

```

1 function ReverseArray( $a, n$ )
2   for  $i := 1$  to  $\lfloor n/2 \rfloor$  do
3      $tmp := a[i]$ 
4      $a[i] \leftarrow a[n - i + 1]$ 
5      $a[n - i + 1] \leftarrow tmp$ 
6   end
7   return  $a$ 
8 end

```

Step-by-step execution:**Instance 1:** $a = [1, 2, 3, 4, 5]$, $n = 5$, $\lfloor 5/2 \rfloor = 2$ iterations

i	exchange	array a (after the exchange)
–	–	$[1, 2, 3, 4, 5]$
1	$a[1] \leftrightarrow a[5]$	$[5, 2, 3, 4, 1]$
2	$a[2] \leftrightarrow a[4]$	$[5, 4, 3, 2, 1]$

Return: $a = [5, 4, 3, 2, 1]$ **Instance 2:** $a = [10, 20, 30, 40]$, $n = 4$, $\lfloor 4/2 \rfloor = 2$ iterations

i	exchange	array a (after the exchange)
–	–	$[10, 20, 30, 40]$
1	$a[1] \leftrightarrow a[4]$	$[40, 20, 30, 10]$
2	$a[2] \leftrightarrow a[3]$	$[40, 30, 20, 10]$

Return: $a = [40, 30, 20, 10]$

2.7 Sets

Arrays store values in positions. When only the membership of the values matters, and not their position or their repetitions, we use a set.

- A **set** is an unordered collection of *distinct* elements
- Sets must be declared before use with the **new set** operator; no size needs to be specified; when created, a set is empty

```
1 set_name := new set
```

The main operators on sets are:

- **add**: insert an element into the set (ignored if already present)

add x **to** set_name

- **remove**: delete an element from the set (ignored if not present)

remove x **from** set_name

- **in**: membership test, which evaluates to **true** if x belongs to the set

x **in** set_name

- **size**: cardinality (number of elements currently in the set)

size(set_name)

- **foreach**: loop over the elements of the set; by convention, in these notes the elements are visited in insertion order

```
1 foreach  $x$  in set_name do
  | // process  $x$ 
2 end
```

- It is possible to loop over the elements of a set in a specific order by specifying a sorting criterion:

```
1 foreach  $x$  in set_name sorted by criterion do
  | // process  $x$ 
2 end
```

Example 2.24: set declaration and usage

```

// Declare an empty set
1 s := new set

// Populate the set with the even numbers from 1 to n
2 for i := 1 to n do
3   | if i mod 2 = 0 then
4   |   | add i to s
5   | end
6 end

// Membership test
7 if 4 in s then
8   | print(4 is in s)
9 end

// Remove an element
10 remove 4 from s

// Iterate and print all remaining elements
11 foreach x in s do
12   | print(x)
13 end

// Print the cardinality
14 print(size(s))

// Print the elements in non-ascending order
15 foreach x in s sorted by value non-ascending do
16   | print(x)
17 end

// Print the elements in non-descending order
18 foreach x in s sorted by value non-descending do
19   | print(x)
20 end

```

Step-by-step execution:

Instance: $n = 8$

- Step 1: the for loop adds 2, 4, 6, 8, so $s = \{2, 4, 6, 8\}$.
- Step 2: 4 belongs to s : the output is 4 is in s .
- Step 3: after the removal, $s = \{2, 6, 8\}$.
- Step 4: the first **foreach** prints 2, 6, 8 (insertion order), then the cardinality 3 is printed.
- Step 5: the last two loops print 8, 6, 2 and 2, 6, 8.

2.8 Style conventions

We conclude with a summary of the notation used in these notes, compared with other common choices.

Different authors may use slightly different conventions:

- Declaration: $:=$ (used in these notes) or $=$ (some authors use the same symbol for both declaration and assignment)
- Assignment: $\leftarrow$ (used in these notes) or $=$
- Array indexing: $a[i]$ (used in these notes) or a_i
- Two-dimensional array indexing: $b[i][j]$ (used in these notes) or b_{ij}
- Comments: `// comment`, `# comment`, or `(* comment *)`

The important thing is to be consistent and clear!

Key points of the chapter.

- Variables are declared with $:=$ and modified with $\leftarrow$.
- Conditional statements and loops (**if**, **while**, **for**, with **break** and **continue**) control the order of execution.
- Arrays $a[1..n]$, $b[1..m][1..n]$, $\dots$ are indexed from 1 in pseudo-code; sets store distinct elements without positions.
- An algorithm is written as a **function** with its input (including the preconditions), its output and a **return** statement.

Sorting algorithms

Introduction

Sorting is one of the most common tasks in computing: a sorted list of names, of grades or of prices is much easier to search, to compare and to read than an unsorted one. In this chapter we state the sorting problem precisely, we introduce the criteria used to compare sorting algorithms, and we study three classical algorithms that solve it: **Insertion Sort**, **Exchange Sort** and **Selection Sort**.

- The three algorithms are written in the pseudo-code of the course, with arrays indexed from 1, and then implemented in C++, with arrays indexed from 0.
- For each algorithm we show its execution step by step on the same two arrays, $[5, 2, 4, 1, 3]$ and $[12, 11, 13, 5, 6]$, so that the three methods can be compared directly.
- For each algorithm we count the key comparisons and the moves of keys that it performs, and we establish whether it is in-place and whether it is stable.

3.1 The sorting problem

Definition 3.1: sorting problem

Given an array $a[1..n]$ of $n \in \mathbb{Z}_{\geq 1}$ natural numbers, the **sorting problem** consists in rearranging the elements of the array in **non-decreasing order**, i.e., so that

$$a[1] \leq a[2] \leq \dots \leq a[n]. \quad (3.1)$$

A **rearrangement** (or **permutation**) of the array is an array that contains the same elements, each with the same multiplicity, in a possibly different order. The output of the sorting problem is the rearrangement of the input array that satisfies (3.1).

- Each element $a[i]$ is called a **key**: it is the value on which the ordering is based, i.e., the value that is compared to decide the final arrangement.
- The order is *non-decreasing* rather than *increasing* because the array may contain equal keys, as in $[3, 1, 3]$, whose sorted rearrangement is $[1, 3, 3]$.

- The output must be a rearrangement of the input: an array such as $[0, 0, 0, 0, 0]$ satisfies (3.1), but it is not the sorted version of $[5, 2, 4, 1, 3]$.

Example 3.1: sorting an array

- Input array: $a = [5, 2, 4, 1, 3]$.
- Sorted array: $a = [1, 2, 3, 4, 5]$.

Before studying specific algorithms, we fix the operations that they may use and the criteria by which we compare them.

3.1.1 Comparison-based sorting

Definition 3.2: comparison-based sorting

A sorting algorithm is **comparison-based** if it accesses the keys only by comparing them two at a time and by copying them.

- The only operations permitted on the keys in a comparison-based sorting algorithm are:
 1. **compare two keys**: determine which of $a[i]$ and $a[j]$ is larger (or whether they are equal);
 2. **copy / assign a key**: move or store a key value, e.g., $tmp \leftarrow a[i]$.
- Forbidden operations on keys:
 1. arithmetic on keys (addition, multiplication, etc.);
 2. inspecting individual digits or bits of a key;
 3. any operation that exploits the internal representation of the key.
- All the sorting algorithms studied in this chapter (Insertion Sort, Exchange Sort, Selection Sort) belong to this class: they decide how to rearrange the elements solely on the basis of pairwise key comparisons.

3.1.2 Performance metrics

- Since comparisons and assignments are the only operations permitted on the keys, we evaluate a comparison-based sorting algorithm by counting:
 - the **number of key comparisons** it performs;
 - the **number of key assignments** (copies) it performs.
- The number of assignments matters in practice. Each **swap** of two array elements requires three assignments:

$$tmp \leftarrow a[i]; \quad a[i] \leftarrow a[j]; \quad a[j] \leftarrow tmp,$$

so algorithms that swap less tend to be faster.

- Both numbers depend not only on n but also on the initial order of the keys. For this reason we compute them in the **best case** (the input that minimizes them) and in the **worst case** (the input that maximizes them), among all arrays with n elements.
- In the counts we use the sum of the first $n - 1$ positive integers:

$$\sum_{k=1}^{n-1} k = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}. \quad (3.2)$$

3.1.3 In-place sorting and memory usage

- Beyond time, we also care about how much **extra memory** an algorithm uses beyond the space already needed to store the n input keys.

Definition 3.3: in-place sorting

A sorting algorithm is **in-place** if the amount of extra memory it requires is **constant**, i.e., it does not grow with n .

- Equivalently, an in-place sorting algorithm sorts by rearranging the elements *within* the original array, using at most a fixed number of auxiliary variables (e.g., loop counters, a temporary variable for swapping).
- In-place algorithms typically rearrange the input array by shifting or swapping elements, so they do not require extra space for another array to hold the sorted output.

3.1.4 Stability

- In many applications the keys are part of larger **records** (e.g., a student identified by a grade and a name), and two different records may have the same key. We write a record as a pair (*key*, *label*). Only the keys are compared, and each label travels with its key.

Definition 3.4: stable sorting

A sorting algorithm is **stable** if records with equal keys appear in the output in the same relative order as in the input.

Example 3.2: stable and non-stable sorting

- Array of records: $a = [(3, A), (1, B), (3, C), (2, D), (1, E)]$.
 - Stable sorting: $a = [(1, B), (1, E), (2, D), (3, A), (3, C)]$. The records with key 1 keep the order B, E and the records with key 3 keep the order A, C , as in the input.
 - Non-stable sorting: $a = [(1, E), (1, B), (2, D), (3, C), (3, A)]$. The keys are sorted, but both pairs of records with equal keys are inverted.
- Stability matters when the records are sorted several times by different keys: for instance, sorting students first by name and then, with a stable algorithm, by grade, gives the students with the same grade in alphabetical order.

A comparison-based sorting algorithm is evaluated by the **number of key comparisons** and the **number of key assignments** it performs (in the best and in the worst case), by the **extra memory** it uses (in-place or not), and by whether it is **stable**.

3.2 Insertion Sort

- Insertion Sort works as many people sort a hand of playing cards: the cards already in the hand are kept sorted, and each new card is inserted in its correct place among them.
- At the beginning of iteration i , the sub-array $a[1..i-1]$ is sorted. The algorithm takes the next element $a[i]$, called *key*, shifts one position to the right all the elements of $a[1..i-1]$ that are greater than *key*, and writes *key* in the position left free (Figure 3.1).
- This is the pseudo-code of the Insertion Sort algorithm, which sorts an array $a[1..n]$ of n natural numbers in non-decreasing order.

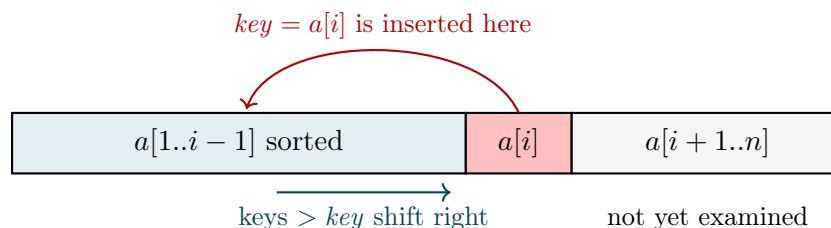


Figure 3.1: Iteration i of Insertion Sort: $key = a[i]$ is inserted in the sorted sub-array $a[1..i-1]$, after shifting one position to the right the keys greater than key .

Algorithm: InsertionSort

Input : An array $a[1..n]$ of n natural numbers

Output: The array a sorted in non-decreasing order

```

1 function InsertionSort( $a, n$ )
2    $i := 2$ 
3   while  $i \leq n$  do
4      $key := a[i]$                                 // element to insert
5      $j := i - 1$ 
6     while  $j \geq 1$  and  $a[j] > key$  do
7        $a[j + 1] \leftarrow a[j]$                     // shift element to the right
8        $j \leftarrow j - 1$ 
9     end
10     $a[j + 1] \leftarrow key$                         // insert key in its position
11     $i \leftarrow i + 1$ 
12  end
13 end

```

- Explanation of the algorithm:

- The outer loop (lines 3–11) iterates through the array starting from the second element ($i = 2$): a single element, $a[1..1]$, is already sorted.
- For each element $a[i]$ (the *key*), the inner loop (lines 6–8) scans the sorted portion $a[1..i-1]$ from right to left and shifts one position to the right all the elements greater than *key*.
- The inner loop stops when $j = 0$ (all the elements of $a[1..i-1]$ were greater than *key*) or when $a[j] \leq key$. The condition at line 6 is evaluated from left to right: when $j = 0$ the comparison $a[j] > key$ is not evaluated, since $a[0]$ does not exist.
- Line 10 writes *key* in the position $j + 1$ left free by the shifts.
- After each iteration of the outer loop, the sub-array $a[1..i]$ is sorted. When the algorithm stops, $i = n + 1$ and the whole array $a[1..n]$ is sorted.

- In the trace tables of this chapter, each row shows the state of the array *after* the operation described in the last column. The current key is highlighted in red, and the cells modified by a shift, an insertion or a swap are highlighted in blue.

Example 3.3: Insertion Sort on [5, 2, 4, 1, 3]

Input array: $a = [5, 2, 4, 1, 3]$, with $n = 5$. The outer loop executes the iterations $i = 2, 3, 4, 5$.

- Detailed trace (inner loop):

i	j	key	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
—	—	—	5	2	4	1	3	initial state
2	1	2	5	2	4	1	3	$key \leftarrow a[2] = 2; j \leftarrow 1$
	0	2	5	5	4	1	3	$a[1] = 5 > 2$: shift $a[2] \leftarrow 5; j \leftarrow 0$
	—	2	2	5	4	1	3	$j = 0$: exit; insert $a[1] \leftarrow 2$
3	2	4	2	5	4	1	3	$key \leftarrow a[3] = 4; j \leftarrow 2$
	1	4	2	5	5	1	3	$a[2] = 5 > 4$: shift $a[3] \leftarrow 5; j \leftarrow 1$
	—	4	2	4	5	1	3	$a[1] = 2 \leq 4$: exit; insert $a[2] \leftarrow 4$
4	3	1	2	4	5	1	3	$key \leftarrow a[4] = 1; j \leftarrow 3$
	2	1	2	4	5	5	3	$a[3] = 5 > 1$: shift $a[4] \leftarrow 5; j \leftarrow 2$
	1	1	2	4	4	5	3	$a[2] = 4 > 1$: shift $a[3] \leftarrow 4; j \leftarrow 1$
	0	1	2	2	4	5	3	$a[1] = 2 > 1$: shift $a[2] \leftarrow 2; j \leftarrow 0$
	—	1	1	2	4	5	3	$j = 0$: exit; insert $a[1] \leftarrow 1$
5	4	3	1	2	4	5	3	$key \leftarrow a[5] = 3; j \leftarrow 4$
	3	3	1	2	4	5	5	$a[4] = 5 > 3$: shift $a[5] \leftarrow 5; j \leftarrow 3$
	2	3	1	2	4	4	5	$a[3] = 4 > 3$: shift $a[4] \leftarrow 4; j \leftarrow 2$
	—	3	1	2	3	4	5	$a[2] = 2 \leq 3$: exit; insert $a[3] \leftarrow 3$

- State of the array after each iteration of the outer loop:

i	key	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
—	—	5	2	4	1	3	initial state
2	2	2	5	4	1	3	insert 2 at the beginning
3	4	2	4	5	1	3	insert 4 between 2 and 5
4	1	1	2	4	5	3	insert 1 at the beginning
5	3	1	2	3	4	5	insert 3 between 2 and 4

- The sorted array is $[1, 2, 3, 4, 5]$. The algorithm performs 9 key comparisons and 7 shifts.

Example 3.4: Insertion Sort on [12, 11, 13, 5, 6]

Input array: $a = [12, 11, 13, 5, 6]$, with $n = 5$.

- Detailed trace (inner loop):

i	j	key	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
—	—	—	12	11	13	5	6	initial state
2	1	11	12	11	13	5	6	$key \leftarrow a[2] = 11; j \leftarrow 1$
	0	11	12	12	13	5	6	$a[1] = 12 > 11$: shift $a[2] \leftarrow 12; j \leftarrow 0$
	—	11	11	12	13	5	6	$j = 0$: exit; insert $a[1] \leftarrow 11$
3	2	13	11	12	13	5	6	$key \leftarrow a[3] = 13; j \leftarrow 2$
	—	13	11	12	13	5	6	$a[2] = 12 \leq 13$: exit; insert $a[3] \leftarrow 13$
4	3	5	11	12	13	5	6	$key \leftarrow a[4] = 5; j \leftarrow 3$
	2	5	11	12	13	13	6	$a[3] = 13 > 5$: shift $a[4] \leftarrow 13; j \leftarrow 2$
	1	5	11	12	12	13	6	$a[2] = 12 > 5$: shift $a[3] \leftarrow 12; j \leftarrow 1$
	0	5	11	11	12	13	6	$a[1] = 11 > 5$: shift $a[2] \leftarrow 11; j \leftarrow 0$
	—	5	5	11	12	13	6	$j = 0$: exit; insert $a[1] \leftarrow 5$
5	4	6	5	11	12	13	6	$key \leftarrow a[5] = 6; j \leftarrow 4$
	3	6	5	11	12	13	13	$a[4] = 13 > 6$: shift $a[5] \leftarrow 13; j \leftarrow 3$
	2	6	5	11	12	12	13	$a[3] = 12 > 6$: shift $a[4] \leftarrow 12; j \leftarrow 2$
	1	6	5	11	11	12	13	$a[2] = 11 > 6$: shift $a[3] \leftarrow 11; j \leftarrow 1$
	—	6	5	6	11	12	13	$a[1] = 5 \leq 6$: exit; insert $a[2] \leftarrow 6$

- State of the array after each iteration of the outer loop:

i	key	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
—	—	12	11	13	5	6	initial state
2	11	11	12	13	5	6	insert 11 at the beginning
3	13	11	12	13	5	6	13 already in position
4	5	5	11	12	13	6	insert 5 at the beginning
5	6	5	6	11	12	13	insert 6 between 5 and 11

- The sorted array is $[5, 6, 11, 12, 13]$. In iteration $i = 3$ the key 13 is already in its position: the inner loop stops at the first comparison and no shift is performed. The algorithm performs 9 key comparisons and 7 shifts.

3.2.1 Properties of Insertion Sort

- The two examples show that the work done by Insertion Sort depends on the initial order of the keys: a key that is already in its position costs one comparison and no shift, while a key smaller than all the previous ones is shifted across the whole sorted portion.

Observation 3.5: comparisons and shifts of Insertion Sort

Given an array $a[1..n]$ of n natural numbers, **InsertionSort** performs at least $n - 1$ and at most $\frac{n(n-1)}{2}$ key comparisons, and at most $\frac{n(n-1)}{2}$ shifts. The lower bounds are attained when the array is already sorted (no shift is performed), the upper bounds when it is sorted in decreasing order with distinct keys.

Proof. Consider iteration i of the outer loop, with $i \in \{2, 3, \dots, n\}$, and let s_i and c_i be the numbers of shifts and of key comparisons it performs. The inner loop starts with $j = i - 1 \geq 1$, so $c_i \geq 1$. Each shift (lines 7–8) follows a comparison $a[j] > key$ that is true, and decreases j by 1; hence $s_i \leq i - 1$. The inner loop then stops in one of two ways:

- with $j = 0$, after $s_i = i - 1$ shifts and without a further comparison: $c_i = s_i = i - 1$;
- after a comparison $a[j] > key$ that is false, with $j \geq 1$: then $s_i = i - 1 - j \leq i - 2$ and $c_i = s_i + 1 \leq i - 1$.

In both cases $1 \leq c_i \leq i - 1$ and $0 \leq s_i \leq i - 1$. Summing over $i \in \{2, 3, \dots, n\}$ and using (3.2):

$$n - 1 \leq \text{key comparisons} \leq \sum_{i=2}^n (i - 1) = \frac{n(n - 1)}{2}, \quad \text{shifts} \leq \frac{n(n - 1)}{2}.$$

If the array is sorted, at every iteration $a[i - 1] \leq key$: the inner loop stops at the first comparison, with no shift, and the algorithm performs exactly $n - 1$ comparisons. If the keys are distinct and in decreasing order, $key = a[i]$ is smaller than all the keys of $a[1..i - 1]$: all of them are shifted and the loop stops with $j = 0$, so iteration i performs $i - 1$ comparisons and $i - 1$ shifts, and the totals are both $\frac{n(n-1)}{2}$. $\square$

- Every iteration also performs two assignments of keys (lines 4 and 10), so the number of key assignments is $2(n - 1)$ plus the number of shifts: between $2(n - 1)$ and $2(n - 1) + \frac{n(n-1)}{2}$.
- Insertion Sort never swaps two elements: a shift is a single assignment, which makes the method cheaper than a sequence of swaps that would move key one position at a time.

Observation 3.6: Insertion Sort is in-place and stable

`InsertionSort` is an in-place and stable sorting algorithm.

Proof. Besides the array, the algorithm uses only the three variables i , j and key , independently of n : it is in-place. A key is moved to the left only past keys that are *strictly* greater than it (condition $a[j] > key$ at line 6): the inner loop stops at the first key $a[j] \leq key$, so key is inserted after every equal key that precedes it. Hence two records with equal keys are never exchanged, and their relative order in the output is the same as in the input: the algorithm is stable. $\square$

3.2.2 C++ implementation

- This is a C++ implementation of the Insertion Sort algorithm. In C++ the array has indices from 0 to $n-1$: the outer loop starts from $i = 1$, and the inner loop stops when j becomes -1 . The operator `&&` evaluates its right operand only when the left one is true, so `a[j]` is never read with $j = -1$.

```

1 // Sorts an array in-place in non-decreasing order
2
3 void InsertionSort(int n, int *a)
4 {
5     int i, j, key;
6
7     for (i = 1; i < n; i++)
8     {
9         key = a[i]; // Element to insert
10        j = i - 1;
11
12        // Shift elements greater than key to the right
13        while (j >= 0 && a[j] > key)
14        {
15            a[j + 1] = a[j];
16            j = j - 1;
17        }
18
19        // Insert key in correct position
20        a[j + 1] = key;
21    }
22 }
```

Example 3.5: execution of the C++ code on [5, 2, 4, 1, 3]

The C++ function is called with $n = 5$ and the array $a = [5, 2, 4, 1, 3]$, stored in $a[0..4]$. The execution is the one of Example 3.3, with all the indices decreased by 1:

i	j	key	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Action
—	—	—	5	2	4	1	3	initial state
1	0	2	5	2	4	1	3	$key \leftarrow a[1] = 2; j \leftarrow 0$
	-1	2	5	5	4	1	3	$a[0] = 5 > 2$: shift $a[1] \leftarrow 5; j \leftarrow -1$
	—	2	2	5	4	1	3	$j = -1$: exit; insert $a[0] \leftarrow 2$
2	1	4	2	5	4	1	3	$key \leftarrow a[2] = 4; j \leftarrow 1$
	0	4	2	5	5	1	3	$a[1] = 5 > 4$: shift $a[2] \leftarrow 5; j \leftarrow 0$
	—	4	2	4	5	1	3	$a[0] = 2 \leq 4$: exit; insert $a[1] \leftarrow 4$
3	2	1	2	4	5	1	3	$key \leftarrow a[3] = 1; j \leftarrow 2$
	1	1	2	4	5	5	3	$a[2] = 5 > 1$: shift $a[3] \leftarrow 5; j \leftarrow 1$
	0	1	2	4	4	5	3	$a[1] = 4 > 1$: shift $a[2] \leftarrow 4; j \leftarrow 0$
	-1	1	2	2	4	5	3	$a[0] = 2 > 1$: shift $a[1] \leftarrow 2; j \leftarrow -1$
	—	1	1	2	4	5	3	$j = -1$: exit; insert $a[0] \leftarrow 1$
4	3	3	1	2	4	5	3	$key \leftarrow a[4] = 3; j \leftarrow 3$
	2	3	1	2	4	5	5	$a[3] = 5 > 3$: shift $a[4] \leftarrow 5; j \leftarrow 2$
	1	3	1	2	4	4	5	$a[2] = 4 > 3$: shift $a[3] \leftarrow 4; j \leftarrow 1$
	—	3	1	2	3	4	5	$a[1] = 2 \leq 3$: exit; insert $a[2] \leftarrow 3$

The function terminates with the sorted array $[1, 2, 3, 4, 5]$.

Example 3.6: execution of the C++ code on $[12, 11, 13, 5, 6]$

The C++ function is called with $n = 5$ and the array $a = [12, 11, 13, 5, 6]$, stored in $a[0..4]$:

i	j	key	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Action
—	—	—	12	11	13	5	6	initial state
1	0	11	12	11	13	5	6	$key \leftarrow a[1] = 11; j \leftarrow 0$
	-1	11	12	12	13	5	6	$a[0] = 12 > 11$: shift $a[1] \leftarrow 12; j \leftarrow -1$
	—	11	11	12	13	5	6	$j = -1$: exit; insert $a[0] \leftarrow 11$
2	1	13	11	12	13	5	6	$key \leftarrow a[2] = 13; j \leftarrow 1$
	—	13	11	12	13	5	6	$a[1] = 12 \leq 13$: exit; insert $a[2] \leftarrow 13$
3	2	5	11	12	13	5	6	$key \leftarrow a[3] = 5; j \leftarrow 2$
	1	5	11	12	13	13	6	$a[2] = 13 > 5$: shift $a[3] \leftarrow 13; j \leftarrow 1$
	0	5	11	12	12	13	6	$a[1] = 12 > 5$: shift $a[2] \leftarrow 12; j \leftarrow 0$
	-1	5	11	11	12	13	6	$a[0] = 11 > 5$: shift $a[1] \leftarrow 11; j \leftarrow -1$
	—	5	5	11	12	13	6	$j = -1$: exit; insert $a[0] \leftarrow 5$
4	3	6	5	11	12	13	6	$key \leftarrow a[4] = 6; j \leftarrow 3$
	2	6	5	11	12	13	13	$a[3] = 13 > 6$: shift $a[4] \leftarrow 13; j \leftarrow 2$
	1	6	5	11	12	12	13	$a[2] = 12 > 6$: shift $a[3] \leftarrow 12; j \leftarrow 1$
	0	6	5	11	11	12	13	$a[1] = 11 > 6$: shift $a[2] \leftarrow 11; j \leftarrow 0$
	—	6	5	6	11	12	13	$a[0] = 5 \leq 6$: exit; insert $a[1] \leftarrow 6$

The function terminates with the sorted array $[5, 6, 11, 12, 13]$.

3.3 Exchange Sort

- Insertion Sort builds the sorted prefix by inserting each new key among the previous ones. Exchange Sort follows the opposite idea: it fills the positions $1, 2, \dots, n-1$ one at a time, each with the smallest of the remaining keys.
- In iteration i , the algorithm compares $a[i]$ with every element $a[j]$ with $j > i$ and swaps the two elements whenever $a[i] > a[j]$. At the end of the iteration $a[i]$ is the minimum of $a[i..n]$ (Figure 3.2).

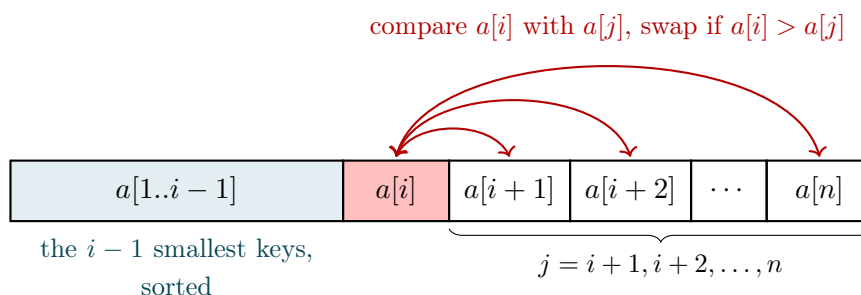


Figure 3.2: Iteration i of Exchange Sort: $a[i]$ is compared with every $a[j]$, $j > i$, and the two elements are swapped whenever $a[i] > a[j]$; at the end, $a[i] = \min(a[i..n])$.

- This is the pseudo-code of the Exchange Sort algorithm, which sorts an array $a[1..n]$ of n natural numbers in non-decreasing order.

Algorithm: ExchangeSort

Input : An array $a[1..n]$ of n natural numbers

Output: The array a sorted in non-decreasing order

```

1 function ExchangeSort( $a, n$ )
2    $i := 1$ 
3   while  $i \leq n-1$  do
4      $j := i+1$ 
5     while  $j \leq n$  do
6       if  $a[i] > a[j]$  then
7          $tmp := a[i]$ 
8          $a[i] \leftarrow a[j]$ 
9          $a[j] \leftarrow tmp$ 
10      end
11       $j \leftarrow j+1$ 
12    end
13     $i \leftarrow i+1$ 
14  end
15 end

```

// swap $a[i]$ and $a[j]$

- Explanation of the algorithm:

- The outer loop iterates with index i from 1 to $n - 1$.
- For each i , the inner loop compares $a[i]$ with every element $a[j]$ for $j \in \{i + 1, i + 2, \dots, n\}$.
- Whenever $a[i] > a[j]$, the two elements are swapped (lines 7–9), so $a[i]$ always holds the smallest key met so far in $a[i..n]$.
- After iteration i of the outer loop, $a[i]$ is the minimum of $a[i..n]$ and the sub-array $a[1..i]$ contains the i smallest keys of the array, sorted. After iteration $n - 1$, the last element $a[n]$ is the largest key, and the whole array is sorted.

Example 3.7: Exchange Sort on $[5, 2, 4, 1, 3]$

Input array: $a = [5, 2, 4, 1, 3]$, with $n = 5$. The outer loop executes the iterations $i = 1, 2, 3, 4$.

- Detailed trace (inner loop):

i	j	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
–	–	5	2	4	1	3	initial state
1	2	2	5	4	1	3	$a[1] = 5 > a[2] = 2$: swap $a[1] \leftrightarrow a[2]$
	3	2	5	4	1	3	$a[1] = 2 \leq a[3] = 4$: no swap
	4	1	5	4	2	3	$a[1] = 2 > a[4] = 1$: swap $a[1] \leftrightarrow a[4]$
	5	1	5	4	2	3	$a[1] = 1 \leq a[5] = 3$: no swap
2	3	1	4	5	2	3	$a[2] = 5 > a[3] = 4$: swap $a[2] \leftrightarrow a[3]$
	4	1	2	5	4	3	$a[2] = 4 > a[4] = 2$: swap $a[2] \leftrightarrow a[4]$
	5	1	2	5	4	3	$a[2] = 2 \leq a[5] = 3$: no swap
3	4	1	2	4	5	3	$a[3] = 5 > a[4] = 4$: swap $a[3] \leftrightarrow a[4]$
	5	1	2	3	5	4	$a[3] = 4 > a[5] = 3$: swap $a[3] \leftrightarrow a[5]$
4	5	1	2	3	4	5	$a[4] = 5 > a[5] = 4$: swap $a[4] \leftrightarrow a[5]$

- State of the array after each iteration of the outer loop:

i	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
–	5	2	4	1	3	initial state
1	1	5	4	2	3	$a[1] = \min(a[1..5]) = 1$ (2 swaps)
2	1	2	5	4	3	$a[2] = \min(a[2..5]) = 2$ (2 swaps)
3	1	2	3	5	4	$a[3] = \min(a[3..5]) = 3$ (2 swaps)
4	1	2	3	4	5	$a[4] = \min(a[4..5]) = 4$ (1 swap)

- The sorted array is $[1, 2, 3, 4, 5]$. The algorithm performs 10 key comparisons and 7 swaps.

Example 3.8: Exchange Sort on $[12, 11, 13, 5, 6]$

Input array: $a = [12, 11, 13, 5, 6]$, with $n = 5$.

- Detailed trace (inner loop):

i	j	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
–	–	12	11	13	5	6	initial state
1	2	11	12	13	5	6	$a[1] = 12 > a[2] = 11$: swap $a[1] \leftrightarrow a[2]$
	3	11	12	13	5	6	$a[1] = 11 \leq a[3] = 13$: no swap
	4	5	12	13	11	6	$a[1] = 11 > a[4] = 5$: swap $a[1] \leftrightarrow a[4]$
	5	5	12	13	11	6	$a[1] = 5 \leq a[5] = 6$: no swap
2	3	5	12	13	11	6	$a[2] = 12 \leq a[3] = 13$: no swap
	4	5	11	13	12	6	$a[2] = 12 > a[4] = 11$: swap $a[2] \leftrightarrow a[4]$
	5	5	6	13	12	11	$a[2] = 11 > a[5] = 6$: swap $a[2] \leftrightarrow a[5]$
3	4	5	6	12	13	11	$a[3] = 13 > a[4] = 12$: swap $a[3] \leftrightarrow a[4]$
	5	5	6	11	13	12	$a[3] = 12 > a[5] = 11$: swap $a[3] \leftrightarrow a[5]$
4	5	5	6	11	12	13	$a[4] = 13 > a[5] = 12$: swap $a[4] \leftrightarrow a[5]$

- State of the array after each iteration of the outer loop:

i	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
–	12	11	13	5	6	initial state
1	5	12	13	11	6	$a[1] = \min(a[1..5]) = 5$ (2 swaps)
2	5	6	13	12	11	$a[2] = \min(a[2..5]) = 6$ (2 swaps)
3	5	6	11	13	12	$a[3] = \min(a[3..5]) = 11$ (2 swaps)
4	5	6	11	12	13	$a[4] = \min(a[4..5]) = 12$ (1 swap)

- The sorted array is $[5, 6, 11, 12, 13]$. The algorithm performs 10 key comparisons and 7 swaps.

3.3.1 Properties of Exchange Sort

- Unlike Insertion Sort, Exchange Sort compares every pair of positions $i < j$ exactly once, whatever the initial order of the keys: only the number of swaps depends on the input.

Observation 3.7: comparisons and swaps of Exchange Sort

Given an array $a[1..n]$ of n natural numbers, **ExchangeSort** performs exactly $\frac{n(n-1)}{2}$ key comparisons and at most $\frac{n(n-1)}{2}$ swaps. No swap is performed when the array is already sorted, and $\frac{n(n-1)}{2}$ swaps are performed when it is sorted in decreasing order with distinct keys.

Proof. In iteration i of the outer loop, $i \in \{1, 2, \dots, n-1\}$, the inner loop executes line 6 once for each $j \in \{i+1, i+2, \dots, n\}$, i.e., $n-i$ times. By (3.2), the total number of key comparisons is

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

Each comparison causes at most one swap, so there are at most $\frac{n(n-1)}{2}$ swaps.

If the array is sorted, $a[i] \leq a[j]$ for every $i < j$: the first comparison causes no swap, the array does not change, and the same holds for all the following comparisons.

If the keys are distinct and in decreasing order, let $b[1..n]$ be the input array, so $b[1] > b[2] > \dots > b[n]$. In iteration 1, before the comparison with $j \in \{2, 3, \dots, n\}$ the current value of $a[1]$ is $b[j-1]$ (the last key swapped into position 1), which is greater than $a[j] = b[j]$: every comparison causes a swap, which moves $b[j-1]$ to position j . At the end of iteration 1, $a[1] = b[n]$ and $a[2..n] = [b[1], b[2], \dots, b[n-1]]$, which is again in decreasing order: for instance, $[6, 5, 4, 3, 2, 1]$ becomes $[1, 6, 5, 4, 3, 2]$. Repeating the argument on $a[i..n]$ for $i = 2, 3, \dots, n-1$, all the $\frac{n(n-1)}{2}$ comparisons cause a swap. $\square$

- On an array in decreasing order with distinct keys, Exchange Sort performs $\frac{n(n-1)}{2}$ swaps, i.e., $\frac{3n(n-1)}{2}$ key assignments, while Insertion Sort moves the keys with $\frac{n(n-1)}{2}$ shifts, one assignment each.

Observation 3.8: Exchange Sort is in-place but not stable

ExchangeSort is an in-place sorting algorithm, but it is not stable.

Proof. Besides the array, the algorithm uses only the three variables i , j and tmp : it is in-place. To show that it is not stable, it is enough to exhibit one input on which the relative order of two equal keys changes. Consider the array of records $[(2, A), (2, B), (1, C)]$. For $i = 1$: the comparison with $j = 2$ finds $2 > 2$ false, so no swap; the comparison with $j = 3$ finds $2 > 1$, so $a[1]$ and $a[3]$ are swapped, giving $[(1, C), (2, B), (2, A)]$. For $i = 2$: $2 > 2$ is false, so no swap. The output is $[(1, C), (2, B), (2, A)]$: the record $(2, A)$, which preceded $(2, B)$ in the input, now follows it. $\square$

3.3.2 C++ implementation

- This is a C++ implementation of the Exchange Sort algorithm, with indices from 0 to $n-1$.

```

1 // Sorts an array in-place in non-decreasing order
2
3 void ExchangeSort(int n, int *a)
4 {
5     for (int i = 0; i < n-1; i++)
6     {
7         for (int j = i+1; j < n; j++)
8         {
9             if (a[i] > a[j])
10            {
11                int tmp = a[i];
12                a[i] = a[j];
13                a[j] = tmp;
14            }
15        }
16    }
17 }
```

Example 3.9: execution of the C++ code on [5, 2, 4, 1, 3]

The C++ function is called with $n = 5$ and the array $a = [5, 2, 4, 1, 3]$, stored in $a[0..4]$. The table shows only the comparisons that cause a swap, with the state of the array after each swap:

i	j	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Action
–	–	5	2	4	1	3	initial state
0	1	2	5	4	1	3	$a[0] = 5 > a[1] = 2$: swap $a[0] \leftrightarrow a[1]$
	3	1	5	4	2	3	$a[0] = 2 > a[3] = 1$: swap $a[0] \leftrightarrow a[3]$
1	2	1	4	5	2	3	$a[1] = 5 > a[2] = 4$: swap $a[1] \leftrightarrow a[2]$
	3	1	2	5	4	3	$a[1] = 4 > a[3] = 2$: swap $a[1] \leftrightarrow a[3]$
2	3	1	2	4	5	3	$a[2] = 5 > a[3] = 4$: swap $a[2] \leftrightarrow a[3]$
	4	1	2	3	5	4	$a[2] = 4 > a[4] = 3$: swap $a[2] \leftrightarrow a[4]$
3	4	1	2	3	4	5	$a[3] = 5 > a[4] = 4$: swap $a[3] \leftrightarrow a[4]$

The function terminates with the sorted array [1, 2, 3, 4, 5].

Example 3.10: execution of the C++ code on [12, 11, 13, 5, 6]

The C++ function is called with $n = 5$ and the array $a = [12, 11, 13, 5, 6]$, stored in $a[0..4]$. Again, only the comparisons that cause a swap are shown:

i	j	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Action
–	–	12	11	13	5	6	initial state
0	1	11	12	13	5	6	$a[0] = 12 > a[1] = 11$: swap $a[0] \leftrightarrow a[1]$
	3	5	12	13	11	6	$a[0] = 11 > a[3] = 5$: swap $a[0] \leftrightarrow a[3]$
1	3	5	11	13	12	6	$a[1] = 12 > a[3] = 11$: swap $a[1] \leftrightarrow a[3]$
	4	5	6	13	12	11	$a[1] = 11 > a[4] = 6$: swap $a[1] \leftrightarrow a[4]$
2	3	5	6	12	13	11	$a[2] = 13 > a[3] = 12$: swap $a[2] \leftrightarrow a[3]$
	4	5	6	11	13	12	$a[2] = 12 > a[4] = 11$: swap $a[2] \leftrightarrow a[4]$
3	4	5	6	11	12	13	$a[3] = 13 > a[4] = 12$: swap $a[3] \leftrightarrow a[4]$

The function terminates with the sorted array [5, 6, 11, 12, 13].

3.4 Selection Sort

- Exchange Sort may swap many times in the same iteration only to bring the minimum of $a[i..n]$ into position i . Selection Sort reaches the same result with at most one swap per iteration: it first *selects* the position of the minimum of $a[i..n]$, and then swaps it with $a[i]$ (Figure 3.3).

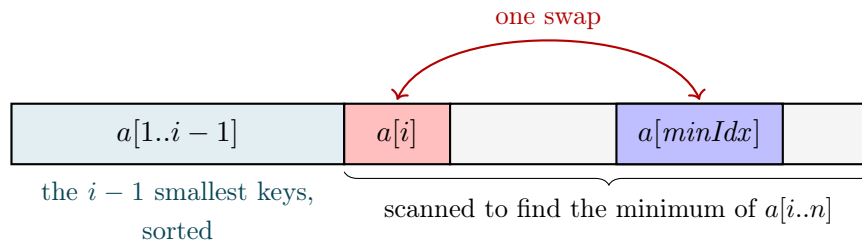


Figure 3.3: Iteration i of Selection Sort: the inner loop finds the position $minIdx$ of the minimum of $a[i..n]$, and then $a[i]$ and $a[minIdx]$ are swapped (if $minIdx \neq i$).

- This is the pseudo-code of the Selection Sort algorithm, which sorts an array $a[1..n]$ of n natural numbers in non-decreasing order.

Algorithm: SelectionSort

Input : An array $a[1..n]$ of n natural numbers

Output: The array a sorted in non-decreasing order

```

1 function SelectionSort( $a, n$ )
2    $i := 1$ 
3   while  $i \leq n - 1$  do
4      $minIdx := i$                                 // index of the minimum in  $a[i..n]$ 
5      $j := i + 1$ 
6     while  $j \leq n$  do
7       if  $a[j] < a[minIdx]$  then
8          $minIdx \leftarrow j$                         // new minimum found
9       end
10       $j \leftarrow j + 1$ 
11    end
12    if  $minIdx \neq i$  then
13       $tmp := a[i]$                                 // swap  $a[i]$  and  $a[minIdx]$ 
14       $a[i] \leftarrow a[minIdx]$ 
15       $a[minIdx] \leftarrow tmp$ 
16    end
17     $i \leftarrow i + 1$ 
18  end
19 end

```

- Explanation of the algorithm:

- The outer loop iterates with index i from 1 to $n - 1$.
- For each i , the inner loop scans $a[i + 1..n]$ and keeps in $minIdx$ the index of the smallest key met so far. Since the comparison at line 7 is strict, in case of ties $minIdx$ is the index of the *first* minimum.
- After the inner loop, if $minIdx \neq i$, the minimum $a[minIdx]$ is swapped with $a[i]$: at most **one swap** per iteration of the outer loop, unlike Exchange Sort.
- After iteration i of the outer loop, the sub-array $a[1..i]$ is sorted and contains the i smallest keys of the array.

Example 3.11: Selection Sort on $[5, 2, 4, 1, 3]$

Input array: $a = [5, 2, 4, 1, 3]$, with $n = 5$. The outer loop executes the iterations $i = 1, 2, 3, 4$.

- Detailed trace (inner loop):

i	j	$minIdx$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
–	–	–	5	2	4	1	3	initial state
1	–	1	5	2	4	1	3	$minIdx := 1$ ($a[1] = 5$)
	2	2	5	2	4	1	3	$a[2] = 2 < a[1] = 5$: $minIdx \leftarrow 2$
	3	2	5	2	4	1	3	$a[3] = 4 \geq a[2] = 2$: no update
	4	4	5	2	4	1	3	$a[4] = 1 < a[2] = 2$: $minIdx \leftarrow 4$
	5	4	5	2	4	1	3	$a[5] = 3 \geq a[4] = 1$: no update
	–	–	1	2	4	5	3	$minIdx = 4 \neq 1$: swap $a[1] \leftrightarrow a[4]$
2	–	2	1	2	4	5	3	$minIdx := 2$ ($a[2] = 2$)
	3	2	1	2	4	5	3	$a[3] = 4 \geq a[2] = 2$: no update
	4	2	1	2	4	5	3	$a[4] = 5 \geq a[2] = 2$: no update
	5	2	1	2	4	5	3	$a[5] = 3 \geq a[2] = 2$: no update
	–	–	1	2	4	5	3	$minIdx = 2$: no swap
	–	–	1	2	4	5	3	
3	–	3	1	2	4	5	3	$minIdx := 3$ ($a[3] = 4$)
	4	3	1	2	4	5	3	$a[4] = 5 \geq a[3] = 4$: no update
	5	5	1	2	4	5	3	$a[5] = 3 < a[3] = 4$: $minIdx \leftarrow 5$
	–	–	1	2	3	5	4	$minIdx = 5 \neq 3$: swap $a[3] \leftrightarrow a[5]$
	–	–	1	2	3	5	4	
4	–	4	1	2	3	5	4	$minIdx := 4$ ($a[4] = 5$)
	5	5	1	2	3	5	4	$a[5] = 4 < a[4] = 5$: $minIdx \leftarrow 5$
	–	–	1	2	3	4	5	$minIdx = 5 \neq 4$: swap $a[4] \leftrightarrow a[5]$

- State of the array after each iteration of the outer loop:

i	$minIdx$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
–	–	5	2	4	1	3	initial state
1	4	1	2	4	5	3	swap $a[1] \leftrightarrow a[4]$: $a[1] = \min(a[1..5]) = 1$
2	2	1	2	4	5	3	no swap: $a[2] = \min(a[2..5]) = 2$
3	5	1	2	3	5	4	swap $a[3] \leftrightarrow a[5]$: $a[3] = \min(a[3..5]) = 3$
4	5	1	2	3	4	5	swap $a[4] \leftrightarrow a[5]$: $a[4] = \min(a[4..5]) = 4$

- The sorted array is $[1, 2, 3, 4, 5]$. The algorithm performs 10 key comparisons and 3 swaps.

Example 3.12: Selection Sort on [12, 11, 13, 5, 6]

Input array: $a = [12, 11, 13, 5, 6]$, with $n = 5$.

- Detailed trace (inner loop):

i	j	$minIdx$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
–	–	–	12	11	13	5	6	initial state
1	–	1	12	11	13	5	6	$minIdx := 1$ ($a[1] = 12$)
	2	2	12	11	13	5	6	$a[2] = 11 < a[1] = 12$: $minIdx \leftarrow 2$
	3	2	12	11	13	5	6	$a[3] = 13 \geq a[2] = 11$: no update
	4	4	12	11	13	5	6	$a[4] = 5 < a[2] = 11$: $minIdx \leftarrow 4$
	5	4	12	11	13	5	6	$a[5] = 6 \geq a[4] = 5$: no update
	–	–	5	11	13	12	6	$minIdx = 4 \neq 1$: swap $a[1] \leftrightarrow a[4]$
2	–	2	5	11	13	12	6	$minIdx := 2$ ($a[2] = 11$)
	3	2	5	11	13	12	6	$a[3] = 13 \geq a[2] = 11$: no update
	4	2	5	11	13	12	6	$a[4] = 12 \geq a[2] = 11$: no update
	5	5	5	11	13	12	6	$a[5] = 6 < a[2] = 11$: $minIdx \leftarrow 5$
	–	–	5	6	13	12	11	$minIdx = 5 \neq 2$: swap $a[2] \leftrightarrow a[5]$
3	–	3	5	6	13	12	11	$minIdx := 3$ ($a[3] = 13$)
	4	4	5	6	13	12	11	$a[4] = 12 < a[3] = 13$: $minIdx \leftarrow 4$
	5	5	5	6	13	12	11	$a[5] = 11 < a[4] = 12$: $minIdx \leftarrow 5$
	–	–	5	6	11	12	13	$minIdx = 5 \neq 3$: swap $a[3] \leftrightarrow a[5]$
4	–	4	5	6	11	12	13	$minIdx := 4$ ($a[4] = 12$)
	5	4	5	6	11	12	13	$a[5] = 13 \geq a[4] = 12$: no update
	–	–	5	6	11	12	13	$minIdx = 4$: no swap

- State of the array after each iteration of the outer loop:

i	$minIdx$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	Action
–	–	12	11	13	5	6	initial state
1	4	5	11	13	12	6	swap $a[1] \leftrightarrow a[4]$: $a[1] = \min(a[1..5]) = 5$
2	5	5	6	13	12	11	swap $a[2] \leftrightarrow a[5]$: $a[2] = \min(a[2..5]) = 6$
3	5	5	6	11	12	13	swap $a[3] \leftrightarrow a[5]$: $a[3] = \min(a[3..5]) = 11$
4	4	5	6	11	12	13	no swap: $a[4] = \min(a[4..5]) = 12$

- The sorted array is [5, 6, 11, 12, 13]. The algorithm performs 10 key comparisons and 3 swaps.

3.4.1 Properties of Selection Sort

Observation 3.9: comparisons and swaps of Selection Sort

Given an array $a[1..n]$ of n natural numbers, **SelectionSort** performs exactly $\frac{n(n-1)}{2}$ key comparisons and at most $n - 1$ swaps. No swap is performed when the array is already sorted, and $n - 1$ swaps are performed, for instance, on the array $[2, 3, \dots, n, 1]$.

Proof. As for Exchange Sort, in iteration $i \in \{1, 2, \dots, n - 1\}$ the comparison at line 7 is executed once for each $j \in \{i + 1, i + 2, \dots, n\}$, so the total number of key comparisons is $\sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2}$ by (3.2). Each of the $n - 1$ iterations of the outer loop performs at most one swap (lines 13–15), so there are at most $n - 1$ swaps.

If the array is sorted, the minimum of $a[i..n]$ is $a[i]$ and, by the strict comparison at line 7, $minIdx$ remains equal to i : no swap is performed. On $[2, 3, \dots, n, 1]$, iteration 1 finds the minimum 1 in position n and swaps it with $a[1] = 2$, giving $[1, 3, 4, \dots, n, 2]$; in general, at the beginning of iteration i the array is $[1, 2, \dots, i - 1, i + 1, \dots, n, i]$, the minimum i of $a[i..n]$ is in position $n \neq i$, and the swap gives $[1, 2, \dots, i, i + 2, \dots, n, i + 1]$. Hence each of the $n - 1$ iterations performs a swap. $\square$

- Selection Sort performs at most $3(n - 1)$ key assignments, the smallest worst-case number of key assignments among the three algorithms of this chapter, while its number of comparisons, $\frac{n(n-1)}{2}$, does not depend on the input.

Observation 3.10: Selection Sort is in-place but not stable

SelectionSort is an in-place sorting algorithm, but it is not stable.

Proof. Besides the array, the algorithm uses only the four variables i , j , $minIdx$ and tmp : it is in-place. Consider the array of records $[(2, A), (2, B), (1, C)]$. For $i = 1$, the minimum is $(1, C)$, with $minIdx = 3$, and the swap of $a[1]$ and $a[3]$ gives $[(1, C), (2, B), (2, A)]$. For $i = 2$, the comparison $2 < 2$ is false, so $minIdx = 2$ and no swap is performed. In the output $(2, A)$ follows $(2, B)$, while it preceded it in the input: the swap moved $(2, A)$ past the equal key $(2, B)$. $\square$

3.4.2 C++ implementation

- This is a C++ implementation of the Selection Sort algorithm, with indices from 0 to $n - 1$.

```

1 // Sorts an array in-place in non-decreasing order
2
3 void SelectionSort(int n, int *a)
4 {
5     for (int i = 0; i < n-1; i++)
6     {
7         int minIdx = i;
8         for (int j = i+1; j < n; j++)
9         {
10             if (a[j] < a[minIdx])
11             {
12                 minIdx = j;
13             }
14         }
15         if (minIdx != i)
16         {
17             int tmp = a[i];
18             a[i] = a[minIdx];
19             a[minIdx] = tmp;
20         }
21     }
22 }

```

Example 3.13: execution of the C++ code on [5, 2, 4, 1, 3]

The C++ function is called with $n = 5$ and the array $a = [5, 2, 4, 1, 3]$, stored in $a[0..4]$. The execution is the one of Example 3.11, with all the indices decreased by 1:

i	j	$minIdx$	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Action
—	—	—	5	2	4	1	3	initial state
0	—	0	5	2	4	1	3	$minIdx := 0$ ($a[0] = 5$)
	1	1	5	2	4	1	3	$a[1] = 2 < a[0] = 5$: $minIdx \leftarrow 1$
	2	1	5	2	4	1	3	$a[2] = 4 \geq a[1] = 2$: no update
	3	3	5	2	4	1	3	$a[3] = 1 < a[1] = 2$: $minIdx \leftarrow 3$
	4	3	5	2	4	1	3	$a[4] = 3 \geq a[3] = 1$: no update
	—	—	1	2	4	5	3	$minIdx = 3 \neq 0$: swap $a[0] \leftrightarrow a[3]$
1	—	1	1	2	4	5	3	$minIdx := 1$ ($a[1] = 2$)
	2	1	1	2	4	5	3	$a[2] = 4 \geq a[1] = 2$: no update
	3	1	1	2	4	5	3	$a[3] = 5 \geq a[1] = 2$: no update
	4	1	1	2	4	5	3	$a[4] = 3 \geq a[1] = 2$: no update
	—	—	1	2	4	5	3	$minIdx = 1$: no swap
2	—	2	1	2	4	5	3	$minIdx := 2$ ($a[2] = 4$)
	3	2	1	2	4	5	3	$a[3] = 5 \geq a[2] = 4$: no update
	4	4	1	2	4	5	3	$a[4] = 3 < a[2] = 4$: $minIdx \leftarrow 4$
	—	—	1	2	3	5	4	$minIdx = 4 \neq 2$: swap $a[2] \leftrightarrow a[4]$
3	—	3	1	2	3	5	4	$minIdx := 3$ ($a[3] = 5$)
	4	4	1	2	3	5	4	$a[4] = 4 < a[3] = 5$: $minIdx \leftarrow 4$
	—	—	1	2	3	4	5	$minIdx = 4 \neq 3$: swap $a[3] \leftrightarrow a[4]$

The function terminates with the sorted array $[1, 2, 3, 4, 5]$.

Example 3.14: execution of the C++ code on $[12, 11, 13, 5, 6]$

The C++ function is called with $n = 5$ and the array $a = [12, 11, 13, 5, 6]$, stored in $a[0..4]$. Here we show only the state of the array after each iteration of the outer loop:

i	$minIdx$	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Action
–	–	12	11	13	5	6	initial state
0	3	5	11	13	12	6	swap $a[0] \leftrightarrow a[3]$: $a[0] = \min(a[0..4]) = 5$
1	4	5	6	13	12	11	swap $a[1] \leftrightarrow a[4]$: $a[1] = \min(a[1..4]) = 6$
2	4	5	6	11	12	13	swap $a[2] \leftrightarrow a[4]$: $a[2] = \min(a[2..4]) = 11$
3	3	5	6	11	12	13	no swap: $a[3] = \min(a[3..4]) = 12$

The function terminates with the sorted array $[5, 6, 11, 12, 13]$.

3.5 Comparison of the three algorithms

- Table 3.1 collects the results of the six Observations of this chapter, for an array of n keys. The “moves” are the shifts for Insertion Sort and the swaps for the other two algorithms.

algorithm	key comparisons		moves		in-place	stable
	best	worst	best	worst		
Insertion Sort	$n - 1$	$\frac{n(n-1)}{2}$	0 shifts	$\frac{n(n-1)}{2}$ shifts	yes	yes
Exchange Sort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	0 swaps	$\frac{n(n-1)}{2}$ swaps	yes	no
Selection Sort	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	0 swaps	$n - 1$ swaps	yes	no

Table 3.1: Key comparisons, moves of keys, memory and stability of the three sorting algorithms of the chapter, for an array of n keys.

Example 3.15: the three algorithms on the same arrays

On the two arrays used throughout the chapter, the three algorithms perform the following numbers of key comparisons and moves.

algorithm	$[5, 2, 4, 1, 3]$		$[12, 11, 13, 5, 6]$	
	comparisons	moves	comparisons	moves
Insertion Sort	9	7 shifts	9	7 shifts
Exchange Sort	10	7 swaps	10	7 swaps
Selection Sort	10	3 swaps	10	3 swaps

- Insertion Sort, Exchange Sort and Selection Sort are comparison-based and in-place, and in the worst case all three perform $\frac{n(n-1)}{2}$ key comparisons.
- Insertion Sort adapts to the input: on an array that is already sorted it performs only $n - 1$ comparisons and no shift. It is the only stable algorithm of the three.
- Exchange Sort and Selection Sort always perform $\frac{n(n-1)}{2}$ comparisons, but Selection Sort performs at most $n - 1$ swaps, against the $\frac{n(n-1)}{2}$ swaps of Exchange Sort in the worst case.

Linear and binary search

4.1 The search problem

- Looking for a value in a collection of data is one of the most frequent operations performed by a computer: a name in a list of contacts, a code in a table of products, a word in a dictionary.
- In this chapter the collection is an array of natural numbers, and we present two algorithms that solve the problem:
 - **linear search**, which works on every array;
 - **binary search**, which works only on arrays sorted in non-decreasing order, but which is much faster than linear search on long arrays.

Definition 4.1: search problem

Given an array $a[1..n]$ of n natural numbers and a target natural number m , the **search problem** consists in determining if m is one of the elements of the array, i.e., if there exists an index $k \in \{1, 2, \dots, n\}$ such that

$$a[k] = m.$$

An algorithm for the search problem returns such an index k if it exists, and the value -1 otherwise.

- The value -1 is never a valid index of the array $a[1..n]$ (nor of a C++ array, whose indices start from 0): for this reason it can be used to signal that the target value has not been found.

4.2 Linear search

Given an array $a[1..n]$ of n natural numbers and a target natural number m , we consider the problem of determining if m is one of the elements of the array.

- The simplest idea is to compare the target m with the elements of the array one after the other, from the first to the last, and to stop as soon as an element equal to m is found.
- This is the pseudo-code of the Linear Search algorithm that determines if a target value m is present in an array $a[1..n]$ of n natural numbers:

Algorithm: LinearSearch

Input : An array $a[1..n]$ of n natural numbers and a target value m

Output: Index of m in a , or -1 if not found

```

1 function LinearSearch( $a, n, m$ )
2    $i := 1$ 
3   while  $i \leq n$  do
4     if  $a[i] = m$  then
5       return  $i$                                 // Element found at index  $i$ 
6     end
7      $i \leftarrow i + 1$ 
8   end
9   return  $-1$                                     // Element not found
10 end

```

- Explanation of the algorithm:
 - The algorithm scans the array sequentially from index 1 to n .
 - At each iteration, it compares the element $a[i]$ with the target m (line 4).
 - If $a[i] = m$, the element is found and index i is returned (line 5).
 - If the loop completes without a match, the element is not in the array and -1 is returned (line 9).
- If m appears more than once in the array, the algorithm stops at the first match: the returned index is the **smallest** index k such that $a[k] = m$.

Example 4.1: Linear Search

Consider the array $a = [4, 2, 7, 1, 5, 9, 3]$ with $n = 7$.

- Search for $m = 5$:

iteration	i	$a[i]$	action
1	1	4	$a[i] \neq m$
2	2	2	$a[i] \neq m$
3	3	7	$a[i] \neq m$
4	4	1	$a[i] \neq m$
5	5	5	$a[i] = m \Rightarrow \text{Found!}$

Accordingly, element 5 is found at index 5.

- Search for $m = 6$:

iteration	i	$a[i]$	action
1	1	4	$a[i] \neq m$
2	2	2	$a[i] \neq m$
3	3	7	$a[i] \neq m$
4	4	1	$a[i] \neq m$
5	5	5	$a[i] \neq m$
6	6	9	$a[i] \neq m$
7	7	3	$a[i] \neq m$
—	8	—	$i > n \Rightarrow \text{loop ends, Not found!}$

Accordingly, element 6 is not in the array (return -1).

- The work done by linear search is measured by the number of comparisons $a[i] = m$ at line 4: in the example, 5 comparisons are needed to find $m = 5$, and all the $n = 7$ elements are compared to conclude that $m = 6$ is not in the array.

Observation 4.2: number of comparisons of linear search

Given an array $a[1..n]$ of n natural numbers and a target value m :

- if k is the smallest index such that $a[k] = m$, then **LinearSearch** performs exactly k comparisons at line 4 and returns k ;
- if m is not in the array, then **LinearSearch** performs exactly n comparisons at line 4 and returns -1 .

Proof. The variable i takes the values $1, 2, 3, \dots$ in this order, and at each value $i \leq n$ exactly one comparison $a[i] = m$ is performed.

- If k is the smallest index with $a[k] = m$, then $a[i] \neq m$ for $i \in \{1, 2, \dots, k-1\}$, so the algorithm does not stop at these values of i ; for $i = k$ the comparison is true and the algorithm returns k . The comparisons are those with $i = 1, 2, \dots, k$, i.e., k comparisons.
- If m is not in the array, every comparison is false, so the loop is executed for

$i = 1, 2, \dots, n$ and stops when $i = n + 1$. The comparisons are n , and the algorithm returns -1 at line 9.

□

Linear search makes no assumption on the order of the elements. Its price is that, when the target is absent (or it is in the last position), all the n elements of the array must be examined.

4.2.1 C++ implementation

- This is a C++ implementation of the Linear Search algorithm. In C++ the indices of an array of n elements go from 0 to $n - 1$: the index returned by the function is therefore the index of the pseudo-code minus 1.

```

1 // Returns the index of m in array a, or -1 if not found
2
3 int LinearSearch(int n, int *a, int m)
4 {
5     for (int i = 0; i < n; i++)
6     {
7         if (a[i] == m)
8         {
9             return i; // Element found at index i
10        }
11    }
12
13    return -1; // Element not found
14 }
```

- The function can be called, for instance, from the following main:

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int a[] = {4, 2, 7, 1, 5, 9, 3};
7     int n = 7;
8
9     cout << LinearSearch(n, a, 5) << endl; // prints 4
10    cout << LinearSearch(n, a, 6) << endl; // prints -1
11
12    return 0;
13 }
```

Example 4.2: execution of the C++ implementation of Linear Search

Consider the array of Example 4.1, stored in C++ with indices from 0 to 6:

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$
	4	2	7	1	5	9	3

- Search for $m = 5$:

iteration	i	comparison	action
1	0	$a[0] = 4 \neq 5$	continue
2	1	$a[1] = 2 \neq 5$	continue
3	2	$a[2] = 7 \neq 5$	continue
4	3	$a[3] = 1 \neq 5$	continue
5	4	$a[4] = 5 = m$	Found! Return 4

The function returns 4: the element is found at C++ index 4, i.e., at index 5 of the pseudo-code.

- Search for $m = 6$:

iteration	i	comparison	action
1	0	$a[0] = 4 \neq 6$	continue
2	1	$a[1] = 2 \neq 6$	continue
3	2	$a[2] = 7 \neq 6$	continue
4	3	$a[3] = 1 \neq 6$	continue
5	4	$a[4] = 5 \neq 6$	continue
6	5	$a[5] = 9 \neq 6$	continue
7	6	$a[6] = 3 \neq 6$	continue

The **for** loop terminates with $i = n = 7$: the element is not found and the function returns -1 .

4.3 Binary search

- Think of looking up a word in a paper dictionary: nobody reads it from the first page. We open it in the middle and, since the words are in alphabetical order, we know in which half to continue. Binary search applies the same idea to a sorted array.

Definition 4.3: sorted array

An array $a[1..n]$ of n natural numbers is **sorted in non-decreasing order** if

$$a[1] \leq a[2] \leq \dots \leq a[n].$$

Given an array $a[1..n]$ of n natural numbers sorted in non-decreasing order and a target natural number m , we consider the problem of determining if m is one of the elements of the array.

- Binary search keeps a portion $a[\text{left}..\text{right}]$ of the array, called the **search interval**, which is the only portion that may still contain m . At the beginning the search interval is the whole array.
- At each iteration the target m is compared with the element $a[\text{mid}]$ in the middle of the search interval:
 - if $a[\text{mid}] < m$, then, since the array is sorted, all the elements $a[\text{left}], \dots, a[\text{mid}]$ are smaller than m , and the search continues in the right half $a[\text{mid} + 1..\text{right}]$;
 - if $a[\text{mid}] > m$, then all the elements $a[\text{mid}], \dots, a[\text{right}]$ are greater than m , and the search continues in the left half $a[\text{left}..\text{mid} - 1]$.
- This is the pseudo-code of the Binary Search algorithm that determines if a target value m is present in an array $a[1..n]$ of n natural numbers sorted in non-decreasing order:

Algorithm: BinarySearch

Input : An array $a[1..n]$ of n natural numbers sorted in non-decreasing order and a target value m

Output: Index of m in a , or -1 if not found

```

1 function BinarySearch( $a, n, m$ )
2    $left := 1$                                 // Left boundary
3    $right := n$                                 // Right boundary
4   while  $left \leq right$  do
5      $mid := left + \lfloor (right - left) / 2 \rfloor$  // Middle index
6     if  $a[mid] = m$  then
7       return  $mid$                             // Element found
8     end
9     if  $a[mid] < m$  then
10       $left \leftarrow mid + 1$                     // Search in right half
11    end
12    else
13       $right \leftarrow mid - 1$                   // Search in left half
14    end
15  end
16  return  $-1$                                     // Element not found
17 end

```

- Explanation of the algorithm:
 - The algorithm maintains two indices, $left$ and $right$, defining the search interval $a[\text{left}..\text{right}]$, which contains $right - left + 1$ elements.
 - At each iteration, it computes the middle index mid (line 5) and compares $a[\text{mid}]$ with the target m .
 - If $a[\text{mid}] = m$, the element is found and its index is returned (line 7).

- If $a[mid] < m$, the search continues in the right half by updating $left \leftarrow mid + 1$ (line 10).
- If $a[mid] > m$, the search continues in the left half by updating $right \leftarrow mid - 1$ (line 13).
- If the search interval becomes empty ($left > right$), the element is not in the array and -1 is returned (line 16).
- Since $0 \leq \lfloor (right - left)/2 \rfloor \leq right - left$, the index mid always lies in the search interval: $left \leq mid \leq right$.

Example 4.3: Binary Search

Consider the sorted array $a = [3, 9, 11, 12, 12, 16, 19]$ with $n = 7$.

- Search for $m = 12$:

iteration	$left$	$right$	mid	$a[mid]$	action
1	1	7	4	12	$a[mid] = m \Rightarrow$ Found!

Accordingly, element 12 is found at index 4.

- Search for $m = 10$:

iteration	$left$	$right$	mid	$a[mid]$	action
1	1	7	4	12	$a[mid] > m \Rightarrow right \leftarrow 3$
2	1	3	2	9	$a[mid] < m \Rightarrow left \leftarrow 3$
3	3	3	3	11	$a[mid] > m \Rightarrow right \leftarrow 2$
–	3	2	–	–	$left > right \Rightarrow$ Not found!

Accordingly, element 10 is not in the array (return -1).

- Search for $m = 3$:

iteration	$left$	$right$	mid	$a[mid]$	action
1	1	7	4	12	$a[mid] > m \Rightarrow right \leftarrow 3$
2	1	3	2	9	$a[mid] > m \Rightarrow right \leftarrow 1$
3	1	1	1	3	$a[mid] = m \Rightarrow$ Found!

Accordingly, element 3 is found at index 1.

- Search for $m = 19$:

iteration	$left$	$right$	mid	$a[mid]$	action
1	1	7	4	12	$a[mid] < m \Rightarrow left \leftarrow 5$
2	5	7	6	16	$a[mid] < m \Rightarrow left \leftarrow 7$
3	7	7	7	19	$a[mid] = m \Rightarrow$ Found!

Accordingly, element 19 is found at index 7.

- The following figure shows the search interval at each iteration of the search for $m = 10$: the elements outside the search interval (grey) are discarded, and the element $a[mid]$ compared with m is highlighted in red.

index	1	2	3	4	5	6	7	
iteration 1	3	9	11	12	12	16	19	$a[4] = 12 > 10 \Rightarrow right \leftarrow 3$
iteration 2	3	9	11	12	12	16	19	$a[2] = 9 < 10 \Rightarrow left \leftarrow 3$
iteration 3	3	9	11	12	12	16	19	$a[3] = 11 > 10 \Rightarrow right \leftarrow 2$
loop ends	3	9	11	12	12	16	19	$left > right \Rightarrow \text{return } -1$

Figure 4.1: Search intervals of **BinarySearch** for $m = 10$ in $a = [3, 9, 11, 12, 12, 16, 19]$: blue cells form the search interval $a[left..right]$, the red cell is $a[mid]$, grey cells have been discarded.

- If m appears more than once in the array, binary search returns the index of **one** of its occurrences, not necessarily the first one. For instance, for $a = [1, 2, 2, 2, 2, 3]$ and $m = 2$, the first iteration computes $mid = 1 + \lfloor 6/2 \rfloor = 4$ and, since $a[4] = 2 = m$, the algorithm returns 4, while **LinearSearch** returns 2.

4.3.1 Number of iterations

- Each iteration that does not find m discards the element $a[mid]$ and at least half of the remaining elements of the search interval. This is why binary search is fast: the length of the search interval is (at least) halved at each iteration.

Observation 4.4: number of iterations of binary search

Given an array $a[1..n]$ of $n \geq 1$ natural numbers sorted in non-decreasing order and a target value m , the while loop of **BinarySearch** executes at most

$$\lfloor \log_2 n \rfloor + 1$$

iterations.

Proof. Let $L = right - left + 1 \geq 1$ be the number of elements of the search interval at the beginning of an iteration that does not return. Since $mid = left + \lfloor (L-1)/2 \rfloor$, the new search interval has:

- $mid - left = \lfloor (L-1)/2 \rfloor$ elements, if $right \leftarrow mid - 1$;
- $right - mid = (L-1) - \lfloor (L-1)/2 \rfloor = \lceil (L-1)/2 \rceil$ elements, if $left \leftarrow mid + 1$.

In both cases the new number of elements is at most $\lceil (L-1)/2 \rceil = \lfloor L/2 \rfloor \leq L/2$.

At the beginning $L = n$. Hence, after k iterations that do not return, the search interval

has at most $n/2^k$ elements. An iteration is executed only if the search interval is not empty, so iteration $k + 1$ is executed only if

$$1 \leq \frac{n}{2^k}, \quad \text{i.e.,} \quad k \leq \log_2 n.$$

Since k is an integer, $k \leq \lfloor \log_2 n \rfloor$, and the number $k + 1$ of any executed iteration is at most $\lfloor \log_2 n \rfloor + 1$. $\square$

- In Example 4.3, $n = 7$ and $\lfloor \log_2 7 \rfloor + 1 = 3$: the searches for $m = 10$, $m = 3$ and $m = 19$ execute exactly 3 iterations, so the bound is attained.
- The difference with linear search grows very quickly with n . The following table compares the number of comparisons of linear search when m is not in the array (Observation 4.2) with the maximum number of iterations of binary search:

n	linear search n comparisons	binary search at most $\lfloor \log_2 n \rfloor + 1$ iterations
1 000	1 000	10
1 000 000	1 000 000	20
1 000 000 000	1 000 000 000	30

Binary search requires the array to be sorted in non-decreasing order. On a sorted array of n elements it executes at most $\lfloor \log_2 n \rfloor + 1$ iterations, while linear search may need n comparisons. If the array is not sorted, binary search may wrongly discard the half that contains m , and linear search must be used.

4.3.2 C++ implementation

- This is a C++ implementation of the Binary Search algorithm. The initial search interval is $a[0..n - 1]$.

```

1 // Returns the index of m in array a, or -1 if not found
2 // The array a must be sorted in non-decreasing order
3
4 int BinarySearch(int n, int *a, int m)
5 {
6     int left = 0;
7     int right = n - 1;
8
9     while (left <= right)
10    {
11        int mid = left + (right - left) / 2;
12
13        if (a[mid] == m)
14        {
15            return mid; // Element found at index mid
16        }
17
18        if (a[mid] < m)

```

```

19     {
20         left = mid + 1; // Search in right half
21     }
22     else
23     {
24         right = mid - 1; // Search in left half
25     }
26 }
27
28 return -1; // Element not found
29 }

```

- The division between two `int` values with $right - left \geq 0$ discards the fractional part, so $(right - left)/2$ is exactly $\lfloor (right - left)/2 \rfloor$.
- The middle index is computed as $left + (right - left)/2$ instead of $(left + right)/2$: the two expressions have the same value, but the sum $left + right$ may exceed the largest value of an `int` when the array is very long, while $left + (right - left)/2$ never exceeds $right$.
- The function can be called, for instance, from the following `main`:

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int a[] = {3, 9, 11, 12, 12, 16, 19};
7     int n = 7;
8
9     cout << BinarySearch(n, a, 12) << endl; // prints 3
10    cout << BinarySearch(n, a, 10) << endl; // prints -1
11    cout << BinarySearch(n, a, 3) << endl; // prints 0
12    cout << BinarySearch(n, a, 19) << endl; // prints 6
13
14    return 0;
15 }

```

Example 4.4: execution of the C++ implementation of Binary Search

Consider the sorted array of Example 4.3, stored in C++ with indices from 0 to 6:

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$
	3	9	11	12	12	16	19

At the beginning $left = 0$ and $right = 6$.

- Search for $m = 12$:

iteration	$left$	$right$	mid	comparison and update
1	0	6	$0 + (6 - 0)/2 = 3$	$a[3] = 12 = m \Rightarrow$ Found! Return 3

- Search for $m = 10$:

iteration	$left$	$right$	mid	comparison and update
1	0	6	$0 + (6 - 0)/2 = 3$	$a[3] = 12 > 10 \Rightarrow right = 3 - 1 = 2$
2	0	2	$0 + (2 - 0)/2 = 1$	$a[1] = 9 < 10 \Rightarrow left = 1 + 1 = 2$
3	2	2	$2 + (2 - 2)/2 = 2$	$a[2] = 11 > 10 \Rightarrow right = 2 - 1 = 1$
–	2	1	–	$left > right \Rightarrow \mathbf{Return -1}$

- Search for $m = 3$:

iteration	$left$	$right$	mid	comparison and update
1	0	6	$0 + (6 - 0)/2 = 3$	$a[3] = 12 > 3 \Rightarrow right = 3 - 1 = 2$
2	0	2	$0 + (2 - 0)/2 = 1$	$a[1] = 9 > 3 \Rightarrow right = 1 - 1 = 0$
3	0	0	$0 + (0 - 0)/2 = 0$	$a[0] = 3 = m \Rightarrow \mathbf{Found! Return 0}$

- Search for $m = 19$:

iteration	$left$	$right$	mid	comparison and update
1	0	6	$0 + (6 - 0)/2 = 3$	$a[3] = 12 < 19 \Rightarrow left = 3 + 1 = 4$
2	4	6	$4 + (6 - 4)/2 = 5$	$a[5] = 16 < 19 \Rightarrow left = 5 + 1 = 6$
3	6	6	$6 + (6 - 6)/2 = 6$	$a[6] = 19 = m \Rightarrow \mathbf{Found! Return 6}$

Every C++ index is the corresponding index of Example 4.3 minus 1.

Key points of the chapter

- The search problem asks for an index k such that $a[k] = m$, or -1 if no such index exists.
- Linear search works on every array, returns the first occurrence of m , and performs up to n comparisons.
- Binary search works only on arrays sorted in non-decreasing order; it halves the search interval $a[left..right]$ at each iteration and executes at most $\lfloor \log_2 n \rfloor + 1$ iterations.
- In the pseudo-code the indices go from 1 to n , in C++ from 0 to $n - 1$: the returned indices differ by 1.

Numerical algorithms

Introduction

In this chapter we study four classical algorithms that work on natural numbers:

- the **change of base**, which writes a natural number given in base ten as a sequence of digits in another base b ;
- the **primality test**, which decides whether a natural number is prime;
- the **Sieve of Eratosthenes**, which finds all the prime numbers up to a given natural number;
- the **Euclidean algorithm**, which computes the greatest common divisor of two natural numbers.

All four algorithms are built on one elementary operation, the **division with remainder**: the change of base reads the digits as remainders, the primality test and the sieve look for remainders equal to zero, and the Euclidean algorithm replaces a pair of numbers with a smaller pair through a remainder. For each algorithm we give the pseudo-code (with arrays indexed from 1), an execution step by step, and a C++ implementation (with arrays indexed from 0), together with its execution on the same inputs.

5.1 Change of base

In the chapter «Representation of numbers» we saw that a sequence of k digits $(d_{k-1} \dots d_1 d_0)_b$ in base b represents the natural number $\sum_{i=0}^{k-1} d_i b^i$, and that $\lfloor \log_b n \rfloor + 1$ digits are needed to write a natural number $n \geq 1$ in base b . In this section we solve the inverse problem: given n in base ten, we compute its digits in base b .

5.1.1 Division with remainder

Definition 5.1: division with remainder

Given a natural number n and a natural number $b \geq 1$, the **division with remainder** (or Euclidean division) of n by b computes the two natural numbers q and r such that

$$n = q b + r \quad \text{with} \quad 0 \leq r < b.$$

The number n is called the **dividend**, b the **divisor**, q the **quotient** and r the **remainder**. The two numbers q and r are unique, and we write

$$q = \lfloor n/b \rfloor \quad \text{and} \quad r = n \bmod b.$$

The key fact behind the change of base is that one division by b extracts the least significant digit of n in base b , and leaves as quotient the number formed by the remaining digits.

Observation 5.2

Let $n \geq 1$ be a natural number whose representation in base $b \geq 2$ is $(d_{k-1} \dots d_1 d_0)_b$. Then

$$n \bmod b = d_0 \quad \text{and} \quad \lfloor n/b \rfloor = \sum_{i=1}^{k-1} d_i b^{i-1} = (d_{k-1} \dots d_1)_b,$$

where the quotient is 0 if $k = 1$.

Proof. Collecting the factor b from all the terms with $i \geq 1$, we obtain

$$n = \sum_{i=0}^{k-1} d_i b^i = \left(\sum_{i=1}^{k-1} d_i b^{i-1} \right) b + d_0.$$

Since d_0 is a digit in base b , we have $0 \leq d_0 < b$. By the uniqueness of the quotient and of the remainder of the division with remainder, the quotient of n divided by b is $\sum_{i=1}^{k-1} d_i b^{i-1}$ and the remainder is d_0 . $\square$

Dividing n by b , the remainder is the least significant digit d_0 , and the quotient is the number written with the other digits $(d_{k-1} \dots d_1)_b$. Dividing the quotient again by b gives d_1 , and so on: repeating the division until the quotient is 0 produces all the digits of n in base b , from the least significant to the most significant one, in exactly $\lfloor \log_b n \rfloor + 1$ divisions.

5.1.2 The algorithm

Given a natural number $n > 0$ represented in base 10 (decimal) and a target base $b \geq 2$, we consider the problem of converting n to base b .

- Pseudo-code of the algorithm to convert sequences of digits from base ten into base b :

Algorithm: ChangeOfBase

Input : A natural number $n \geq 1$ (in base 10) and a base $b \geq 2$

Output: An array $d[1..nd]$ of digits in base b and the number of digits nd

```

1 function ChangeOfBase( $n, b$ )
2    $nd := \lfloor \log_b n \rfloor + 1$                                 // Number of digits needed
3    $d := \text{new array}[1..nd]$ 
4    $k := 1$ 
5   while  $n > 0$  do
6      $q := \lfloor n/b \rfloor$                                     // Quotient
7      $r := n \bmod b$                                        // Remainder (digit in base  $b$ )
8      $d[k] \leftarrow r$ 
9      $n \leftarrow q$ 
10     $k \leftarrow k + 1$ 
11  end
12  return  $d, nd$ 
13 end

```

- **Explanation:**

- The algorithm calculates the number of digits nd needed to represent n in base b using the formula $\lfloor \log_b n \rfloor + 1$.
- It initializes an array $d[1..nd]$ to store the digits and a counter k to track the current position.
- The main loop repeatedly divides n by b to extract the digits from right to left (from the least significant to the most significant one).
- At each iteration, the remainder $r = n \bmod b$ gives the next digit, which is stored in $d[k]$.
- The quotient $q = \lfloor n/b \rfloor$ becomes the new value of n for the next iteration.
- The algorithm terminates when $n = 0$, returning the array d and the total number of digits nd .
- The digits in d are stored in reverse order: the entry $d[k]$ is the digit d_{k-1} of weight b^{k-1} , so $d[1] = d_0$ is the least significant digit (rightmost) and $d[nd] = d_{nd-1}$ is the most significant digit (leftmost).

Example 5.1: conversion from base ten into base b

- Translation of 7 into base $b = 2$:

iteration k	n	n/b	q	r	digit
1	7	$7/2$	3	1	$d[1] = d_0$ (LSD)
2	3	$3/2$	1	1	$d[2] = d_1$
3	1	$1/2$	0	1	$d[3] = d_2$ (MSD)

Accordingly, 7 in base 2 is $(111)_2$, with $nd = \lfloor \log_2 7 \rfloor + 1 = 3$ digits.

- Translation of 8 into base $b = 2$:

iteration k	n	n/b	q	r	digit
1	8	$8/2$	4	0	$d[1] = d_0$ (LSD)
2	4	$4/2$	2	0	$d[2] = d_1$
3	2	$2/2$	1	0	$d[3] = d_2$
4	1	$1/2$	0	1	$d[4] = d_3$ (MSD)

Accordingly, 8 in base 2 is $(1000)_2$, with $nd = \lfloor \log_2 8 \rfloor + 1 = 4$ digits.

- Translation of 25 into base $b = 3$:

iteration k	n	n/b	q	r	digit
1	25	$25/3$	8	1	$d[1] = d_0$ (LSD)
2	8	$8/3$	2	2	$d[2] = d_1$
3	2	$2/3$	0	2	$d[3] = d_2$ (MSD)

Accordingly, 25 in base 3 is $(221)_3$, with $nd = \lfloor \log_3 25 \rfloor + 1 = 3$ digits.

- Translation of 100 into base $b = 8$:

iteration k	n	n/b	q	r	digit
1	100	$100/8$	12	4	$d[1] = d_0$ (LSD)
2	12	$12/8$	1	4	$d[2] = d_1$
3	1	$1/8$	0	1	$d[3] = d_2$ (MSD)

Accordingly, 100 in base 8 is $(144)_8$, with $nd = \lfloor \log_8 100 \rfloor + 1 = 3$ digits.

5.1.3 C++ implementation

- This is a C++ implementation of the conversion algorithm from base ten into base b :

```

1 // Convert a decimal number n to base b
2 // The function returns the number of digits nd in the result
3 // digits[i] stores the digit at position i (from right to left, starting at i=0)
4
5 int ChangeToBase(int n, int b, int *digits)
6 {
7     int nd = 0; // counter for the number of digits
8
9     while (n > 0)
10    {
11        int q = n / b; // quotient
12        int r = n % b; // remainder
13
14        digits[nd] = r; // store the digit at position nd
15        n = q; // update n to the quotient
16        nd++; // increment the counter
17    }
18
19    return nd; // return the number of digits
20 }
```

- For two natural numbers n and b with $b \geq 1$, the C++ operators n / b and $n \% b$ compute exactly the quotient $\lfloor n/b \rfloor$ and the remainder $n \bmod b$ of the division with remainder.
- The array `digits` is indexed from 0: the entry `digits[i]` is the digit d_i of weight b^i . The digit stored in the pseudo-code in $d[k]$ is stored in C++ in `digits[k-1]`.
- The function does not compute $\lfloor \log_b n \rfloor + 1$: it counts the digits in the variable `nd` while it extracts them. This also avoids a subtle problem of floating point arithmetic: in double precision, $\log(1000.0)/\log(10.0)$ evaluates to 2.9999999999999996, so the formula would give 3 digits instead of the 4 digits of $(1000)_{10}$.
- The array `digits` must be declared by the caller with enough entries; for instance, 31 entries are enough for every positive `int` (at most $2^{31} - 1$) and every base $b \geq 2$.

Example 5.2: execution of the C++ function

Each row shows the values computed in one iteration of the while loop and the content of the array `digits` after the iteration (the entry just written is in blue; “–” denotes an entry not yet written).

- Conversion of $n = 7$ into base $b = 2$:

iteration	q	r	digits[3]	digits[2]	digits[1]	digits[0]	n	nd
start	–	–	–	–	–	–	7	0
1	3	1	–	–	–	1	3	1
2	1	1	–	–	1	1	1	2
3	0	1	–	1	1	1	0	3

The function returns $\text{nd} = 3$; reading **digits** from position 2 down to position 0 gives $(111)_2$.

- Conversion of $n = 8$ into base $b = 2$:

iteration	q	r	digits[4]	digits[3]	digits[2]	digits[1]	digits[0]	n	nd
start	–	–	–	–	–	–	–	8	0
1	4	0	–	–	–	–	0	4	1
2	2	0	–	–	–	0	0	2	2
3	1	0	–	–	0	0	0	1	3
4	0	1	–	1	0	0	0	0	4

The function returns $\text{nd} = 4$; reading **digits** from position 3 down to position 0 gives $(1000)_2$.

- Conversion of $n = 25$ into base $b = 3$:

iteration	q	r	digits[3]	digits[2]	digits[1]	digits[0]	n	nd
start	–	–	–	–	–	–	25	0
1	8	1	–	–	–	1	8	1
2	2	2	–	–	2	1	2	2
3	0	2	–	2	2	1	0	3

The function returns $\text{nd} = 3$; reading **digits** from position 2 down to position 0 gives $(221)_3$.

- Conversion of $n = 100$ into base $b = 8$:

iteration	q	r	digits[3]	digits[2]	digits[1]	digits[0]	n	nd
start	–	–	–	–	–	–	100	0
1	12	4	–	–	–	4	12	1
2	1	4	–	–	4	4	1	2
3	0	1	–	1	4	4	0	3

The function returns $\text{nd} = 3$; reading **digits** from position 2 down to position 0 gives $(144)_8$.

Key points.

- The remainder of the division of n by b is the least significant digit of n in base b ; the

quotient is the number written with the other digits.

- The algorithm performs one division for each digit: $\lfloor \log_b n \rfloor + 1$ iterations.
- The digits are produced from the least significant to the most significant one, so the result is read backwards: from $d[nd]$ to $d[1]$ in the pseudo-code, from `digits[nd-1]` to `digits[0]` in C++.

5.2 Primality test

The change of base uses the remainder $n \bmod b$ to read a digit. A remainder equal to zero has a meaning of its own: it says that the division is exact, that is, that b is a divisor of n . Counting divisors leads to the prime numbers.

Definition 5.3: divisor

A natural number $d \geq 1$ is a **divisor** of a natural number n if there exists a natural number k such that $n = k d$:

$$d \text{ is a divisor of } n \iff \exists k \in \mathbb{N}, n = k d$$

or equivalently, if $n \bmod d = 0$.

Definition 5.4: prime number

A natural number $n \geq 2$ is **prime** if its only divisors are 1 and n itself. A natural number $n \geq 2$ which is not prime is called **composite**: it has a divisor d with $2 \leq d \leq n - 1$. The numbers 0 and 1 are not prime.

Given a natural number n , we consider the problem of determining whether n is a prime number.

5.2.1 A first algorithm

The definition suggests a direct algorithm: test, one by one, the candidate divisors of n different from 1 and n . The following observation shows that the candidates larger than $n/2$ can be skipped.

Observation 5.5

For every natural number $n \geq 2$, if d is a divisor of n with $d < n$, then $d \leq n/2$.

Proof. Since d is a divisor of n , there exists a natural number k such that $n = k d$. Since $d < n$, we have $k \geq 2$, hence

$$d = \frac{n}{k} \leq \frac{n}{2}.$$

□

- It follows that it suffices to check whether any natural number d in $\{2, 3, \dots, \lfloor n/2 \rfloor\}$ is a divisor of n . If no such divisor exists, n is prime.
- Pseudo-code of the Basic Primality Test algorithm:

Algorithm: BasicPrimalityTest

Input : A natural number n

Output: true if n is prime, false otherwise

```

1 function BasicPrimalityTest(n)
2   if  $n < 2$  then
3     return false                                // 0 and 1 are not prime
4   end
5   for  $d := 2$  to  $\lfloor n/2 \rfloor$  do
6     if  $n \bmod d = 0$  then
7       return false                                //  $d$  divides  $n$ : not prime
8     end
9   end
10  return true                                    // No divisor found:  $n$  is prime
11 end

```

- Explanation of the algorithm:
 - Numbers less than 2 are not prime by definition.
 - The loop tests every candidate divisor d from 2 up to $\lfloor n/2 \rfloor$.
 - If d is a divisor of n (i.e., $n \bmod d = 0$), the function returns **false** immediately.
 - If the loop completes without finding a divisor, n is prime and the function returns **true**.
- For a prime number the loop is executed completely: for $n = 29$ the algorithm tests the 13 candidates $d = 2, 3, \dots, 14$. We now show that far fewer candidates are enough.

5.2.2 Testing the divisors up to $\sqrt{n}$

The divisors of a natural number come in pairs: d and n/d . In each pair, one of the two numbers is at most $\sqrt{n}$.

Observation 5.6

For every natural number $n \geq 2$, if d is a divisor of n then n/d is a divisor of n .

Proof. If d is a divisor of n , then there exists a natural number k such that

$$n = k d.$$

Since $k = n/d$, it follows that n/d is a natural number. Accordingly,

$$n = d \left(\frac{n}{d} \right),$$

so n/d is also a divisor of n . □

Observation 5.7

For every natural number $n \geq 2$, if $d > \sqrt{n}$ then $n/d < \sqrt{n}$.

Proof. If $d > \sqrt{n}$, dividing n by both sides of this inequality, we obtain

$$\frac{n}{d} < \frac{n}{\sqrt{n}} = \sqrt{n}.$$

Therefore, n/d is smaller than $\sqrt{n}$. □

Corollary 5.8

Every composite natural number $n \geq 2$ has a divisor d with $2 \leq d \leq \lfloor \sqrt{n} \rfloor$. Equivalently, a natural number $n \geq 2$ is prime if and only if no natural number in $\{2, 3, \dots, \lfloor \sqrt{n} \rfloor\}$ is a divisor of n .

Proof. Since n is composite, it has a divisor d with $2 \leq d \leq n - 1$. If $d \leq \sqrt{n}$, then $d \leq \lfloor \sqrt{n} \rfloor$, because d is a natural number, and we are done. Otherwise $d > \sqrt{n}$: by the two previous observations, n/d is a divisor of n with $n/d < \sqrt{n}$, and $n/d \geq 2$ because $d < n$. Hence n/d is a divisor of n with $2 \leq n/d \leq \lfloor \sqrt{n} \rfloor$. □

From the previous observations, we know that for every natural number $n \geq 2$, if $d > \sqrt{n}$ is a divisor of n , then n/d is also a divisor of n and $n/d < \sqrt{n}$. It follows that every divisor of n greater than $\sqrt{n}$ is paired with a divisor of n smaller than $\sqrt{n}$. Therefore, if n has any

divisor at all (other than 1 and n), at least one of them must be $\leq \sqrt{n}$.

Example 5.3: illustration of the property

Let us consider $n = 36$: $\sqrt{36} = 6$. The divisors of 36 other than 1 and 36 are 2, 3, 4, 6, 9, 12, 18.

divisor $d > \sqrt{36}$	paired divisor $36/d < \sqrt{36}$
9	$36/9 = 4$
12	$36/12 = 3$
18	$36/18 = 2$

Every divisor > 6 is paired with a divisor < 6 ; the divisor $6 = \sqrt{36}$ is paired with itself.

- By the corollary, the loop of the basic algorithm can stop at $\lfloor \sqrt{n} \rfloor$ instead of $\lfloor n/2 \rfloor$.
- Pseudo-code of the Primality Test algorithm:

Algorithm: PrimalityTest

Input : A natural number n

Output: **true** if n is prime, **false** otherwise

```

1 function PrimalityTest( $n$ )
2   if  $n < 2$  then
3     return false                                // 0 and 1 are not prime
4   end
5    $d := 2$ 
6   while  $d \leq \lfloor \sqrt{n} \rfloor$  do
7     if  $n \bmod d = 0$  then
8       return false                                //  $d$  divides  $n$ : not prime
9     end
10     $d \leftarrow d + 1$ 
11  end
12  return true                                    // No divisor found:  $n$  is prime
13 end

```

- Explanation of the algorithm:
 - Numbers less than 2 are not prime by definition.
 - The loop tests every candidate divisor d from 2 up to $\lfloor \sqrt{n} \rfloor$.
 - If d divides n (i.e., $n \bmod d = 0$), the function returns **false** immediately.
 - If the loop completes without finding a divisor, then, by the corollary, n is prime and the function returns **true**.
 - For $n = 2$ and $n = 3$ we have $\lfloor \sqrt{n} \rfloor = 1 < 2$: the loop is not executed and the function returns **true**.

Example 5.4: primality test

- Check $n = 36$: $\lfloor \sqrt{36} \rfloor = 6$

d	$36 \bmod d$	action
2	0	return false
36 is not prime ($36 = 2 \cdot 18$).		

- Check $n = 29$: $\lfloor \sqrt{29} \rfloor = 5$

d	$29 \bmod d$	action
2	$1 \neq 0$	continue
3	$2 \neq 0$	continue
4	$1 \neq 0$	continue
5	$4 \neq 0$	continue
Loop ends. Return true: 29 is prime.		

- Check $n = 25$: $\lfloor \sqrt{25} \rfloor = 5$

d	$25 \bmod d$	action
2	$1 \neq 0$	continue
3	$1 \neq 0$	continue
4	$1 \neq 0$	continue
5	0	return false
25 is not prime ($25 = 5 \cdot 5$).		

For the prime number 29 the algorithm tests 4 candidates, while the basic algorithm tests 13.

5.2.3 C++ implementation

- This is a C++ implementation of the Primality Test algorithm:

```

1 // Returns true if n is prime, false otherwise
2
3 bool PrimalityTest(int n)
4 {
5     if (n < 2)
6     {
7         return false; // 0 and 1 are not prime
8     }
9
10    for (int d = 2; d <= (int)sqrt((double)n); d++)
11    {
12        if (n % d == 0)
13        {
14            return false; // d divides n: not prime

```

```

15     }
16 }
17
18 return true; // No divisor found: n is prime
19 }

```

- The function `sqrt` of the header `cmath` works on floating point numbers: the cast `(double)` converts `n` to a floating point number, and the cast `(int)` truncates the square root, so the test compares `d` with $\lfloor \sqrt{n} \rfloor$.
- The while loop of the pseudo-code becomes a for loop: the initialization `d = 2`, the test on `d` and the increment `d++` are written in its header.

Example 5.5: execution of the C++ function

The function executes, on the three numbers of the previous example, the same tests:

<code>n</code>	<code>(int)sqrt((double)n)</code>	values of <code>d</code> tested	returned value
36	6	2 (with <code>36 % 2 == 0</code>)	false : 36 is not prime
29	5	2, 3, 4, 5 (no remainder is 0)	true : 29 is prime
25	5	2, 3, 4, 5 (with <code>25 % 5 == 0</code>)	false : 25 is not prime

For $n = 29$, after the iteration with `d = 5` the increment gives `d = 6`, the test $6 \leq 5$ is false and the loop ends.

Key points.

- The divisors of n come in pairs d and n/d , and in each pair one of the two numbers is at most $\sqrt{n}$.
- Hence $n \geq 2$ is prime if and only if no natural number in $\{2, 3, \dots, \lfloor \sqrt{n} \rfloor\}$ divides it.
- The Primality Test executes at most $\lfloor \sqrt{n} \rfloor - 1$ iterations, instead of the $\lfloor n/2 \rfloor - 1$ iterations of the basic algorithm.

5.3 Sieve of Eratosthenes

The Primality Test answers the question for one number. When we need all the prime numbers up to n , calling it on $2, 3, \dots, n$ repeats many divisions; the sieve works the other way around: instead of searching for the divisors of each number, it crosses out the multiples of each prime.

Given a natural number n , we consider the problem of determining all prime numbers smaller than or equal to n .

- The **Sieve of Eratosthenes** is an ancient algorithm (attributed to the Greek mathematician Eratosthenes of Cyrene, c. 276–194 BC) that computes all prime numbers up to a natural number n .
- The key idea is to start with all natural numbers from 2 to n marked as potentially prime, and then iteratively mark the multiples of each prime as composite.

Definition 5.9: multiple

A natural number m is a **multiple** of a natural number $p \geq 1$ if there exists a natural number k such that $m = k p$:

$$m \text{ is a multiple of } p \iff \exists k \in \mathbb{N}, m = k p$$

or equivalently, if $m \bmod p = 0$.

The first observation tells where the marking of the multiples of p can start.

Observation 5.10

For every natural number $p \geq 2$, if a natural number m is a multiple of p with $p < m < p^2$, then m has a divisor d with $2 \leq d < p$.

Proof. Since m is a multiple of p , there exists a natural number d such that $m = d p$. Then d is a divisor of m .

Moreover, since $m > p$ and $m = d p$, we have $d \geq 2$. Since $m < p^2$ and $m = d p$, we have $d < p$. Hence $2 \leq d < p$. □

From the previous observation, for every prime p and every multiple m of p with $p < m < p^2$, the number m has a divisor d with $2 \leq d < p$. Hence m also has a prime divisor q with $2 \leq q < p$, and was therefore already marked as composite when q was processed. Thus, when marking the multiples of p , one can start with p^2 .

Example 5.6: illustration of the property

Let $p = 5$. Its multiples between p and $p^2 = 25$ are $m = 10, 15, 20$.

multiple m	$m = d \cdot p$	divisor $d < p$	already marked when
10	$10 = 2 \cdot 5$	$d = 2$	$p = 2$ (from $j = 4$, step 2)
15	$15 = 3 \cdot 5$	$d = 3$	$p = 3$ (from $j = 9$, step 3)
20	$20 = 4 \cdot 5$	$d = 4$	$p = 2$ (from $j = 4$, step 2)

The number 20 has the divisor $d = 4$, which is not prime, and also the prime divisor 2, which marked it. The next multiple is $25 = 5 \cdot 5 = p^2$, which has not been marked by any previous prime number. Therefore, for $p = 5$, one can start marking numbers from 25.

The second observation tells where the outer loop over p can stop.

Observation 5.11

For every natural number $n \geq 2$, every composite natural number $m \leq n$ has a prime divisor q with $q \leq \lfloor \sqrt{n} \rfloor$.

Proof. By the corollary of the section «Primality test», m has a divisor d with $2 \leq d \leq \lfloor \sqrt{m} \rfloor \leq \lfloor \sqrt{n} \rfloor$. Let q be the smallest divisor of m with $q \geq 2$; then $q \leq d \leq \lfloor \sqrt{n} \rfloor$. The number q is prime: if q had a divisor c with $2 \leq c < q$, then c would also be a divisor of m (from $q = k_1 c$ and $m = k_2 q$ we get $m = k_2 k_1 c$), against the choice of q as the smallest one. $\square$

- Hence, after the multiples of all the primes $p \leq \lfloor \sqrt{n} \rfloor$ have been marked, every composite number up to n is marked, and the outer loop can stop.
- Pseudo-code of the Sieve of Eratosthenes:

Algorithm: SieveOfEratosthenes**Input** : A natural number $n \geq 1$ **Output**: A boolean array $prime[1..n]$ where $prime[i] = 1$ if and only if i is prime

```

1 function SieveOfEratosthenes( $n$ )
2    $prime := \text{new array}[1..n]$                                 // declare boolean array
3    $prime \leftarrow 1$                                            // initialize all entries to 1
4    $prime[1] \leftarrow 0$                                          // 1 is not prime
5    $p := 2$ 
6   while  $p \leq \lfloor \sqrt{n} \rfloor$  do
7     if  $prime[p] = 1$  then
8        $j := p^2$ 
9       while  $j \leq n$  do
10         $prime[j] \leftarrow 0$                                 //  $j$  not prime
11         $j \leftarrow j + p$ 
12      end
13    end
14     $p \leftarrow p + 1$ 
15  end
16  return  $prime$ 
17 end

```

- Explanation of the algorithm:

- The array $prime$ is declared and all entries initialized to 1; the entry $prime[1]$ is immediately set to 0, since 1 is not prime.
- The outer loop iterates over each candidate prime p from 2 up to $\lfloor \sqrt{n} \rfloor$.
- If $prime[p] = 1$, then p is prime (a composite p would have been marked by its smallest prime divisor), and all its multiples starting from p^2 are set to 0.
- The inner loop starts at p^2 because all smaller multiples $2p, 3p, \dots, (p-1)p$ have already been set to 0 when processing the smaller primes $2, 3, 5, 7, \dots$
- After the execution of the algorithm, $prime[i] = 1$ if and only if i is prime.

Example 5.7: Sieve of Eratosthenes ($n = 10$)

With $n = 10$ ($\lfloor \sqrt{10} \rfloor = 3$), each row shows the state of the array $prime[1..10]$ after the assignment $prime[j] \leftarrow 0$. Blue cells indicate entries just set to 0; red cells indicate the current value of p .

p	j	$prime[1..10]$										action
		1	2	3	4	5	6	7	8	9	10	
–	–	0	1	1	1	1	1	1	1	1	1	initialization
2	4	0	1	1	0	1	1	1	1	1	1	$prime[4] \leftarrow 0$
	6	0	1	1	0	1	0	1	1	1	1	$prime[6] \leftarrow 0$
	8	0	1	1	0	1	0	1	0	1	1	$prime[8] \leftarrow 0$
	10	0	1	1	0	1	0	1	0	1	0	$prime[10] \leftarrow 0$
	12											$j = 12 > 10$: stop
3	9	0	1	1	0	1	0	1	0	0	0	$prime[9] \leftarrow 0$
	12											$j = 12 > 10$: stop

Outer loop ends: $p = 4 > \lfloor \sqrt{10} \rfloor = 3$.

Result: the prime numbers ≤ 10 are 2, 3, 5, 7.

Example 5.8: Sieve of Eratosthenes ($n = 20$)

With $n = 20$ ($\lfloor \sqrt{20} \rfloor = 4$), each row shows the state of the array $prime[1..20]$ after the assignment $prime[j] \leftarrow 0$. Blue cells: entry just set to 0; red cells: current value of p . When the entry is already 0, the assignment does not change the array.

p	j	$prime[1..20]$																				action
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
—	—	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	initialization
2	4	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$prime[4] \leftarrow 0$
	6	0	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$prime[6] \leftarrow 0$
	8	0	1	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	$prime[8] \leftarrow 0$
	10	0	1	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	$prime[10] \leftarrow 0$
	12	0	1	1	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	$prime[12] \leftarrow 0$
	14	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1	$prime[14] \leftarrow 0$
	16	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	1	1	$prime[16] \leftarrow 0$
	18	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	$prime[18] \leftarrow 0$
	20	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	$prime[20] \leftarrow 0$
	22																					$j = 22 > 20$: stop
3	9	0	1	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	0	$prime[9] \leftarrow 0$
	12	0	1	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	$prime[12] \leftarrow 0$ (already 0)
	15	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	1	0	1	0	0	$prime[15] \leftarrow 0$
	18	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	$prime[18] \leftarrow 0$ (already 0)
	21																					$j = 21 > 20$: stop
4																						$prime[4] = 0$: skip

Outer loop ends: $p = 5 > \lfloor \sqrt{20} \rfloor = 4$.

Result: the prime numbers ≤ 20 are 2, 3, 5, 7, 11, 13, 17, 19.

5.3.1 C++ implementation

- This is a C++ implementation of the Sieve of Eratosthenes:

```

1 // the input array is supposed to be already declared with n+1 entries
2 // and initialized to 1, and n is at least 2
3 int SieveOfEratosthenes(int n, int *prime)
4 {
5     prime[0] = 0;
6     prime[1] = 0;
7
8     for (int p = 2; p <= (int)sqrt((double)n); p++)
9     {
10         if (prime[p] == 1)
11         {
12             // Mark multiples of p starting from p*p
13             for (int j = p * p; j <= n; j += p)
14             {
15                 prime[j] = 0; // j is not prime
16             }
17         }
18     }
19
20     return 1;
21 }

```

- In C++ the array is indexed from 0. To keep the entry `prime[i]` associated with the number i , the array has $n + 1$ entries, `prime[0..n]`; the extra entry `prime[0]` refers to the number 0, which is not prime, and is set to 0.
- The two while loops of the pseudo-code become two for loops; the inner one advances with `j += p`.

Example 5.9: execution of the C++ function

Each row shows the array `prime[0..n]` at the end of an iteration of the outer loop, with the values of `j` of its inner loop; the entries changed in that iteration are in blue.

- Execution for $n = 10$ (`(int)sqrt((double)10) = 3`):

p	prime[0..10]										
	0	1	2	3	4	5	6	7	8	9	10
start	0	0	1	1	1	1	1	1	1	1	1
2	0	0	1	1	0	1	0	1	0	1	0
3	0	0	1	1	0	1	0	1	0	0	0

Values of `j` in the inner loop: for `p = 2`: `j = 4, 6, 8, 10`; for `p = 3`: `j = 9`.

- Execution for $n = 20$ (`(int)sqrt((double)20) = 4`):

	prime[0..20]																				
p	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
start	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
3	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0
4	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0

Values of j in the inner loop: for $p = 2$: $j = 4, 6, 8, 10, 12, 14, 16, 18, 20$; for $p = 3$: $j = 9, 12, 15, 18$; for $p = 4$: no inner loop, since `prime[4]` is 0.

The entries equal to 1 at the end are those of the prime numbers: 2, 3, 5, 7 for $n = 10$, and 2, 3, 5, 7, 11, 13, 17, 19 for $n = 20$.

Key points.

- The sieve does not divide: it marks as composite the multiples of each prime p .
- The marking of the multiples of p starts at p^2 : the smaller multiples have a smaller prime divisor and are already marked.
- The outer loop stops at $\lfloor \sqrt{n} \rfloor$: every composite number up to n has a prime divisor not larger than $\lfloor \sqrt{n} \rfloor$.

5.4 Greatest common divisor

The last algorithm of the chapter deals with two numbers at once: we look for the divisors they have in common.

Definition 5.12: common divisor and greatest common divisor

Let n, m be two natural numbers with $n, m \geq 1$.

- A natural number $d \geq 1$ is a **common divisor** of n and m if d is a divisor of both n and m , that is, if there exist natural numbers k_1, k_2 such that

$$n = k_1 d \quad \text{and} \quad m = k_2 d.$$

Equivalently, $n \bmod d = 0$ and $m \bmod d = 0$.

- The **greatest common divisor** of n and m , denoted $\gcd(n, m)$, is the largest common divisor of n and m .

- The greatest common divisor always exists: 1 is a common divisor, and every common divisor is at most $\min\{n, m\}$.
- For instance, the common divisors of 48 and 18 are 1, 2, 3, 6, hence $\gcd(48, 18) = 6$.

Given two natural numbers n, m with $n, m \geq 1$, we consider the problem of computing their greatest common divisor.

The following observation replaces the pair (n, m) with a pair of smaller numbers which has the same greatest common divisor.

Observation 5.13

For every pair of natural numbers $n \geq 1$ and $m \geq 1$,

$$\gcd(n, m) = \gcd(m, n \bmod m).$$

Proof. Let $q = \lfloor n/m \rfloor$, so that $n = qm + (n \bmod m)$.

Let d_1 be a common divisor of n and m . Then there exist natural numbers k_1, k_2 such that

$$n = k_1 d_1 \quad \text{and} \quad m = k_2 d_1.$$

Since $n \bmod m = n - qm$, we obtain

$$n \bmod m = (k_1 - qk_2) d_1,$$

where $k_1 - qk_2 \geq 0$ because $n \bmod m \geq 0$. So d_1 is also a divisor of $n \bmod m$. Hence d_1 is a common divisor of m and $n \bmod m$.

Conversely, let d_2 be a common divisor of m and $n \bmod m$. Then there exist natural numbers k_1, k_2 such that

$$m = k_1 d_2 \quad \text{and} \quad n \bmod m = k_2 d_2.$$

Since $n = qm + (n \bmod m)$, we obtain

$$n = (qk_1 + k_2) d_2,$$

so d_2 is also a divisor of n . Hence d_2 is a common divisor of n and m .

Therefore, n and m have exactly the same common divisors as m and $n \bmod m$, and hence

$$\gcd(n, m) = \gcd(m, n \bmod m).$$

In the right-hand side, $\gcd(m, 0)$ is understood as the largest divisor of m (every $d \geq 1$ divides $0 = 0 \cdot d$), that is, m itself. $\square$

5.4.1 Euclidean algorithm

- The **Euclidean algorithm** (attributed to the ancient Greek mathematician Euclid, c. 300 BC) computes $\gcd(n, m)$ efficiently by repeatedly replacing the pair (n, m) with $(m, n \bmod m)$.
- The algorithm does not need $n \geq m$: if $n < m$, then $n \bmod m = n$, and the first replacement simply swaps the two numbers, turning (n, m) into (m, n) .

- From the previous observation, replacing (n, m) with $(m, n \bmod m)$ does not change the greatest common divisor.
- Since $n \bmod m < m$, the second argument strictly decreases at each replacement.
- Therefore, the algorithm terminates when the remainder becomes 0; at that point, the greatest common divisor is the other argument:

$$\gcd(n, 0) = n.$$

- Pseudo-code of the Euclidean algorithm:

Algorithm: EuclideanAlgorithm

Input : Two natural numbers $n, m \geq 1$

Output: $\gcd(n, m)$

```

1 function EuclideanAlgorithm( $n, m$ )
2   while  $m \neq 0$  do
3      $r := n \bmod m$ 
4      $n \leftarrow m$ 
5      $m \leftarrow r$ 
6   end
7   return  $n$                                 //  $m = 0$ :  $n$  is the GCD
8 end

```

- Explanation of the algorithm:
 - At each step, (n, m) is replaced by $(m, n \bmod m)$.
 - By the previous observation, this replacement preserves the value of $\gcd(n, m)$: denoting by n_0 and m_0 the input values, at every iteration $\gcd(n, m) = \gcd(n_0, m_0)$.
 - Since $n \bmod m < m$, the value of m strictly decreases at every iteration, so the algorithm terminates.
 - When $m = 0$, the equality $\gcd(n, m) = \gcd(n_0, m_0)$ together with $\gcd(n, 0) = n$ gives the result.

Example 5.10: GCD computation

- Compute $\gcd(48, 18)$:

step	n	m	$r = n \bmod m$
1	48	18	$48 \bmod 18 = 12$
2	18	12	$18 \bmod 12 = 6$
3	12	6	$12 \bmod 6 = 0$
4	6	0	— (loop ends)

Result: $\gcd(48, 18) = 6$.

- Compute $\gcd(252, 105)$:

step	n	m	$r = n \bmod m$
1	252	105	$252 \bmod 105 = 42$
2	105	42	$105 \bmod 42 = 21$
3	42	21	$42 \bmod 21 = 0$
4	21	0	— (loop ends)

Result: $\gcd(252, 105) = 21$.

5.4.2 Coprimality**Definition 5.14: coprime numbers**

Two natural numbers $n, m \geq 1$ are said to be **coprime** (or **relatively prime**) if they share no common divisor other than 1:

$$n \text{ and } m \text{ are coprime} \iff \gcd(n, m) = 1.$$

- Since the Euclidean algorithm computes $\gcd(n, m)$, to test coprimality it suffices to call the algorithm and check whether the result is equal to 1.

Example 5.11: coprimality test

- Compute $\gcd(35, 12)$: the Euclidean algorithm returns 1. Therefore, 35 and 12 are **coprime**.
- Compute $\gcd(48, 18)$: the Euclidean algorithm returns 6. Therefore, 48 and 18 are **not coprime**.

5.4.3 C++ implementation

- This is a C++ implementation of the Euclidean algorithm:

```

1 int EuclideanAlgorithm(int n, int m)
2 {
3     while (m != 0)
4     {
5         int r = n % m;
6         n = m;
7         m = r;
8     }
9     return n; // m == 0: n is the GCD
10 }

```

- The parameters `n` and `m` are passed by value: the function modifies its own copies, and the variables of the caller keep their values.

Example 5.12: execution of the C++ function

Each row shows the test of the while loop, the remainder computed and the values of `n` and `m` at the end of the iteration (in blue).

- Execution for `n = 48, m = 18`:

iteration	test <code>m != 0</code>	<code>r = n % m</code>	<code>n</code>	<code>m</code>
start			48	18
1	$18 \neq 0$	$48 \bmod 18 = 12$	18	12
2	$12 \neq 0$	$18 \bmod 12 = 6$	12	6
3	$6 \neq 0$	$12 \bmod 6 = 0$	6	0
Test <code>m != 0</code> false: the loop ends and the function returns <code>n = 6</code> .				

- Execution for `n = 35, m = 12`:

iteration	test <code>m != 0</code>	<code>r = n % m</code>	<code>n</code>	<code>m</code>
start			35	12
1	$12 \neq 0$	$35 \bmod 12 = 11$	12	11
2	$11 \neq 0$	$12 \bmod 11 = 1$	11	1
3	$1 \neq 0$	$11 \bmod 1 = 0$	1	0
Test <code>m != 0</code> false: the loop ends and the function returns <code>n = 1</code> .				

Hence $\gcd(48, 18) = 6$ and $\gcd(35, 12) = 1$.

Key points.

- The pairs (n, m) and $(m, n \bmod m)$ have the same common divisors, hence the same greatest common divisor.
- The Euclidean algorithm repeats the replacement until the second number is 0, and returns the first one, since $\gcd(n, 0) = n$.
- Two numbers are coprime if and only if the Euclidean algorithm returns 1.