



SAPIENZA
UNIVERSITÀ DI ROMA

Algorithms

Volume 4 — Analysis of algorithms

The analysis of algorithms: the input size, the running time in the best, worst and average case, and correctness proved with loop invariants.

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

11	Input size	2
	Introduction	2
	11.1 Input size	2
	11.2 Asymptotic notation: a recall	3
	11.3 Input instance consisting of one natural number	4
	11.4 Input instance consisting of two natural numbers	6
	11.5 Input instance consisting of an array of natural numbers	8
	11.6 Input size and number of steps	13
12	Running time	17
	12.1 Running time of an algorithm	17
	12.2 Find maximum	19
	12.3 Insertion sort	23
	12.4 Change of base	27
	12.5 Primality test	30
	12.6 Summary	33
13	Correctness	35
	13.1 Correctness of an algorithm	35
	13.2 Correctness of Find Maximum	37
	13.3 Correctness of Insertion Sort	39
	13.4 Correctness of Binary Search	43
	13.5 Correctness of the Euclidean algorithm	45
	13.6 Correctness of recursive algorithms	47



Input size

Introduction

To compare two algorithms that solve the same problem we count how many operations they execute. This number grows with the input: sorting an array of one million entries takes longer than sorting an array of ten entries, and testing whether a number with twenty digits is prime takes longer than testing a number with two digits. To make this statement precise we need a measure of how *large* an input is.

In this chapter we fix this measure: the **input size** is the number of bits needed to write the input in binary, as in the chapter «Representation of numbers». We proceed as follows:

- we define the input size and we recall the asymptotic notation used to describe its order of growth (chapter 9 «Asymptotic notation»);
- we compute the input size, and its order of growth, when the input is one natural number, two natural numbers, or an array of natural numbers;
- we show why the distinction matters: the same number of steps can be small or huge with respect to the input size, depending on whether the parameter n is the *value* of a number or the *number of entries* of an array.

11.1 Input size

Definition 11.1: input size

The **input size** $|I|$ of an algorithm is the **total number of bits needed to represent a given input instance I in binary**.

- The input size is a function that maps each input instance to a natural number corresponding to the total number of bits needed to represent the input instance in binary.
- We consider algorithms whose input instances consist of one or more natural numbers, or arrays of natural numbers.
- In the chapter «Representation of numbers» we have seen that the minimum number of digits needed to write a natural number $n \geq 1$ in base 2 is $\lfloor \log_2 n \rfloor + 1$. Equivalently, a

natural number n is written with exactly k bits if and only if

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

Example 11.1: the number of bits of some input instances

- One natural number, $n = 37$. Since $2^5 = 32 \leq 37 < 64 = 2^6$, we have $\lfloor \log_2 37 \rfloor = 5$, and

$$37 = (100101)_2, \quad |I| = \lfloor \log_2 37 \rfloor + 1 = 6.$$

- Two natural numbers, $n = 37$ and $m = 200$. Since $2^7 = 128 \leq 200 < 256 = 2^8$, the number $200 = (11001000)_2$ needs 8 bits, and

$$|I| = 6 + 8 = 14.$$

- The array $a = [6, 1, 3, 8]$. Its entries are $6 = (110)_2$, $1 = (1)_2$, $3 = (11)_2$ and $8 = (1000)_2$, and

$$|I| = 3 + 1 + 2 + 4 = 10.$$

The exact input size depends on the specific values of the instance. In the analysis of algorithms we are interested in how it *grows*, and for this we use the asymptotic notation.

11.2 Asymptotic notation: a recall

The asymptotic notation is defined and studied in chapter 9 «Asymptotic notation». Here we only recall the definitions and the two facts that are used in this chapter. As there, f and g are functions defined on the natural numbers $n \geq 1$ and with non-negative real values, and the notation compares their orders of growth for all sufficiently large n .

- $f(n) = O(g(n))$ (**upper bound**) if there exist $c \in \mathbb{R}_{>0}$ and $n_1 \in \mathbb{N}$ such that

$$0 \leq f(n) \leq c g(n), \quad \forall n \geq n_1;$$

- $f(n) = \Omega(g(n))$ (**lower bound**) if there exist $c \in \mathbb{R}_{>0}$ and $n_1 \in \mathbb{N}$ such that

$$0 \leq c g(n) \leq f(n), \quad \forall n \geq n_1;$$

- $f(n) = \Theta(g(n))$ (**both**) if there exist $c_1, c_2 \in \mathbb{R}_{>0}$ and $n_1 \in \mathbb{N}$ such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall n \geq n_1.$$

- To prove one of these statements we **determine** the constants (c , or c_1 and c_2) and the threshold n_1 .
- For functions of two parameters, such as $f(n, m)$, the definitions are the same, with the inequalities required for every $n \geq n_1$ and every $m \geq m_1$.
- The following two facts are proved in chapter 9 «Asymptotic notation».

Observation 11.2: Θ is the same as O and Ω

We have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Proposition 11.3: limit criterion

Let $g(n) > 0$ for every $n \geq 1$. If

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L \in \mathbb{R}_{>0},$$

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

11.3 Input instance consisting of one natural number

We start from the simplest input: a single natural number.

- If the input consists of a single natural number $n \geq 1$, then the input size is

$$|I| = \lfloor \log_2 n \rfloor + 1,$$

which is the minimum number of bits needed to represent n in binary.

Observation 11.4: input size of one natural number

We have

$$\lfloor \log_2 n \rfloor + 1 = \Theta(\log_2 n). \quad (11.1)$$

Proof. To prove the result, we determine $c_1, c_2 \in \mathbb{R}_{>0}$ and $n_1 \in \mathbb{N}$ such that

$$0 \leq c_1 \log_2 n \leq \lfloor \log_2 n \rfloor + 1 \leq c_2 \log_2 n, \quad \forall n \geq n_1.$$

Since $x < \lfloor x \rfloor + 1$ for every real number x , we have

$$\log_2 n \leq \lfloor \log_2 n \rfloor + 1, \quad \forall n \geq 1.$$

Moreover, since $\lfloor x \rfloor \leq x$ for every real number x , and $\log_2 n \geq 1$ for every $n \geq 2$, we have

$$\lfloor \log_2 n \rfloor + 1 \leq \log_2 n + 1 \leq 2 \log_2 n, \quad \forall n \geq 2.$$

Thus, by choosing $c_1 = 1$, $c_2 = 2$, and $n_1 = 2$, we obtain

$$0 \leq \log_2 n \leq \lfloor \log_2 n \rfloor + 1 \leq 2 \log_2 n, \quad \forall n \geq 2.$$

□

If the input of an algorithm consists of a single natural number $n \geq 1$, then the input size satisfies:

$$|I| = \Theta(\log_2 n).$$

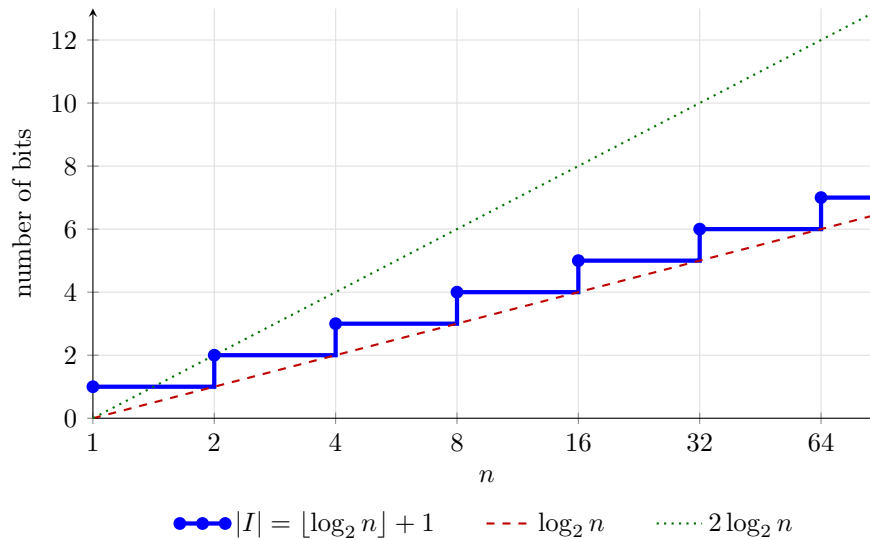


Figure 11.1: The input size $|I| = \lfloor \log_2 n \rfloor + 1$ of one natural number n (horizontal axis in logarithmic scale): it increases by one bit at every power of 2, and for $n \geq 2$ it lies between $\log_2 n$ and $2 \log_2 n$, as stated by Observation 11.4.

Example 11.2: the bounds for one natural number

For $n = 37$ we have $|I| = 6$ (Example 11.1) and $\log_2 37 \approx 5.21$. The inequalities of the proof of Observation 11.4 become

$$\log_2 37 \approx 5.21 \leq |I| = 6 \leq 2 \log_2 37 \approx 10.42.$$

For $n = 1$ we have $|I| = 1$ but $\log_2 1 = 0$: this is why the threshold of the proof is $n_1 = 2$.

11.4 Input instance consisting of two natural numbers

When the input consists of two numbers, the bits of both have to be written.

- If the input consists of two natural numbers $n, m \geq 1$, then the input size is

$$|I| = (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1),$$

which is the minimum number of bits needed to represent both n and m in binary.

Observation 11.5: input size of two natural numbers

We have

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) = \Theta(\log_2 n + \log_2 m). \quad (11.2)$$

Proof. To prove the result, we determine $c_1, c_2 \in \mathbb{R}_{>0}$ and $n_1, m_1 \in \mathbb{N}$ such that

$$0 \leq c_1(\log_2 n + \log_2 m) \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq c_2(\log_2 n + \log_2 m)$$

for every $n \geq n_1$ and $m \geq m_1$.

We have

$$\log_2 n \leq \lfloor \log_2 n \rfloor + 1, \quad \log_2 m \leq \lfloor \log_2 m \rfloor + 1, \quad \forall n, m \geq 1.$$

Therefore,

$$\log_2 n + \log_2 m \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1), \quad \forall n, m \geq 1.$$

Moreover,

$$\lfloor \log_2 n \rfloor + 1 \leq \log_2 n + 1, \quad \lfloor \log_2 m \rfloor + 1 \leq \log_2 m + 1.$$

Hence

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq (\log_2 n + 1) + (\log_2 m + 1).$$

If $n, m \geq 2$, then $\log_2 n \geq 1$ and $\log_2 m \geq 1$, so

$$\log_2 n + 1 \leq 2 \log_2 n, \quad \log_2 m + 1 \leq 2 \log_2 m.$$

Thus,

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq 2(\log_2 n + \log_2 m), \quad \forall n, m \geq 2.$$

Therefore, by choosing $c_1 = 1$, $c_2 = 2$, $n_1 = 2$, and $m_1 = 2$, we obtain

$$0 \leq \log_2 n + \log_2 m \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq 2(\log_2 n + \log_2 m), \quad \forall n, m \geq 2.$$

□

If the input of an algorithm consists of two natural numbers $n, m \geq 1$, then the input size

satisfies:

$$|I| = \Theta(\underbrace{\log_2 n + \log_2 m}_{\log_2(nm)}).$$

The sum of the two logarithms can be replaced by the logarithm of the larger of the two numbers: the larger number needs at least half of the bits of the instance.

Observation 11.6: input size of two natural numbers and their maximum

We have

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) = \Theta(\log_2 M), \quad (11.3)$$

where

$$M = \max\{n, m\}.$$

Proof. To prove the result, we determine $c_1, c_2 \in \mathbb{R}_{>0}$ and $n_1, m_1 \in \mathbb{N}$ such that

$$0 \leq c_1 \log_2 M \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq c_2 \log_2 M, \quad \forall n \geq n_1, m \geq m_1.$$

Since one of the two numbers is equal to M , we have

$$\lfloor \log_2 M \rfloor + 1 \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1).$$

Moreover,

$$\log_2 M \leq \lfloor \log_2 M \rfloor + 1, \quad \forall M \geq 1.$$

Therefore,

$$\log_2 M \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1), \quad \forall n, m \geq 1.$$

On the other hand, since $n \leq M$ and $m \leq M$, we have

$$\lfloor \log_2 n \rfloor + 1 \leq \lfloor \log_2 M \rfloor + 1, \quad \lfloor \log_2 m \rfloor + 1 \leq \lfloor \log_2 M \rfloor + 1.$$

Hence

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq 2(\lfloor \log_2 M \rfloor + 1).$$

If $M \geq 2$, then $\log_2 M \geq 1$, so

$$\lfloor \log_2 M \rfloor + 1 \leq \log_2 M + 1 \leq 2 \log_2 M.$$

Thus,

$$(\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq 4 \log_2 M, \quad \forall M \geq 2.$$

Therefore, by choosing $c_1 = 1$, $c_2 = 4$, $n_1 = 2$, and $m_1 = 2$ (which give $M \geq 2$), we obtain

$$0 \leq \log_2 M \leq (\lfloor \log_2 n \rfloor + 1) + (\lfloor \log_2 m \rfloor + 1) \leq 4 \log_2 M, \quad \forall n, m \geq 2.$$

□

If the input of an algorithm consists of two natural numbers $n, m \geq 1$, then the input size satisfies:

$$|I| = \Theta(\log_2 M),$$

where

$$M = \max\{n, m\}.$$

Example 11.3: the bounds for two natural numbers

For $n = 37$ and $m = 200$ we have $|I| = 14$ (Example 11.1), $M = 200$, and

$$\log_2 37 + \log_2 200 \approx 12.85, \quad \log_2 200 \approx 7.64.$$

The inequalities of the proofs of Observations 11.5 and 11.6 become

$$12.85 \leq 14 \leq 2 \cdot 12.85 = 25.71 \quad \text{and} \quad 7.64 \leq 14 \leq 4 \cdot 7.64 = 30.58,$$

where all the values are rounded to two decimals.

11.5 Input instance consisting of an array of natural numbers

Most of the algorithms of the course receive an array. Its input size depends both on the number n of entries and on their values.

- If the input instance consists of an array $a[1..n]$ of n natural numbers, with $a[i] \geq 1$ for every $i \in \{1, 2, \dots, n\}$, then the input size is

$$|I| = \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1),$$

which is the minimum number of bits needed to represent all the entries of the array in binary.

11.5.1 An upper bound

Observation 11.7: upper bound on the input size of an array

Given an array $a[1..n]$ of n natural numbers, with $a[i] \geq 1$ for every $i \in \{1, 2, \dots, n\}$, we have

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) = O(n(1 + \log_2 M)), \quad (11.4)$$

where

$$M = \max \{a[1], a[2], \dots, a[n]\}.$$

Proof. To prove the result, we determine $c \in \mathbb{R}_{>0}$ and $n_1 \in \mathbb{N}$ such that

$$0 \leq \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \leq cn(1 + \log_2 M), \quad \forall n \geq n_1.$$

Since $a[i] \leq M$ for every $i \in \{1, 2, \dots, n\}$, we have

$$\lfloor \log_2 a[i] \rfloor + 1 \leq \lfloor \log_2 M \rfloor + 1, \quad \forall i \in \{1, 2, \dots, n\}.$$

Therefore,

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \leq \sum_{i=1}^n (\lfloor \log_2 M \rfloor + 1) = n(\lfloor \log_2 M \rfloor + 1).$$

Moreover, we have

$$\lfloor \log_2 M \rfloor + 1 \leq \log_2 M + 1.$$

Hence,

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \leq n(1 + \log_2 M), \quad \forall n \geq 1.$$

Therefore, by choosing $c = 1$ and $n_1 = 1$, we obtain

$$0 \leq \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \leq n(1 + \log_2 M), \quad \forall n \geq 1.$$

□

If the input of an algorithm consists of an array $a[1..n]$ of n natural numbers, with $a[i] \geq 1$ for every $i \in \{1, 2, \dots, n\}$, then the input size satisfies

$$|I| = O(n(1 + \log_2 M)),$$

where

$$M = \max \{a[1], a[2], \dots, a[n]\}.$$

The upper bound $n(1 + \log_2 M)$ counts every entry as if it were as large as the maximum. When only a few entries are large, this is a strong overestimate, and the bound cannot be replaced by Θ .

Example 11.4: the upper bound is not tight

Given an array $a[1..n]$ of n natural numbers, with $a[i] \geq 1$ for every $i \in \{1, 2, \dots, n\}$, to prove that, in general,

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \neq \Theta(n(1 + \log_2 M)), \quad M = \max\{a[1], a[2], \dots, a[n]\},$$

it is enough to provide a counterexample.

- Consider the family of arrays defined by

$$a[1] = M, \quad a[2] = a[3] = \cdots = a[n] = 1,$$

with $M \geq 2$.

- For these arrays, the input size is

$$|I| = \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) = (\lfloor \log_2 M \rfloor + 1) + (n-1)(\lfloor \log_2 1 \rfloor + 1).$$

Since $\lfloor \log_2 1 \rfloor + 1 = 1$, we obtain

$$|I| = (\lfloor \log_2 M \rfloor + 1) + (n-1) = n + \lfloor \log_2 M \rfloor.$$

- Now choose $M = 2^n$, so that $\log_2 M = n$. Then

$$|I| = n + n = 2n, \quad \text{whereas} \quad n(1 + \log_2 M) = n(1 + n) = n^2 + n.$$

- If we had $2n = \Theta(n^2 + n)$, there would exist $c_1 \in \mathbb{R}_{>0}$ and $n_1 \in \mathbb{N}$ such that $c_1(n^2 + n) \leq 2n$ for every $n \geq n_1$, that is, $c_1(n+1) \leq 2$ for every $n \geq n_1$, which is false for n large enough.

Thus, for this family of arrays, $|I| \neq \Theta(n(1 + \log_2 M))$.

- If $M \geq 2$, then $\log_2 M \geq 1$, and therefore $1 + \log_2 M \leq 2 \log_2 M$. Hence

$$n(1 + \log_2 M) \leq 2n \log_2 M,$$

and the input size satisfies

$$|I| = O(n \log_2 M).$$

- If $M = 1$, then all the entries of the array are equal to 1, and therefore

$$|I| = \sum_{i=1}^n (\lfloor \log_2 1 \rfloor + 1) = n.$$

11.5.2 A lower bound

Every entry of the array needs at least one bit. This simple remark gives a lower bound that holds for every array.

Observation 11.8: lower bound on the input size of an array

Given an array $a[1..n]$ of n natural numbers, with $a[i] \geq 1$ for every $i \in \{1, 2, \dots, n\}$, we have

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) = \Omega(n). \quad (11.5)$$

Proof. To prove the result, we determine $c \in \mathbb{R}_{>0}$ and $n_1 \in \mathbb{N}$ such that

$$0 \leq cn \leq \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1), \quad \forall n \geq n_1.$$

Since $a[i] \geq 1$ for every $i \in \{1, 2, \dots, n\}$, we have

$$\lfloor \log_2 a[i] \rfloor + 1 \geq \lfloor \log_2 1 \rfloor + 1 = 1, \quad \forall i \in \{1, 2, \dots, n\}.$$

Therefore,

$$\sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \geq \sum_{i=1}^n 1 = n.$$

Thus, by choosing $c = 1$ and $n_1 = 1$, we obtain

$$0 \leq n \leq \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1), \quad \forall n \geq 1.$$

□

If the input of an algorithm consists of an array $a[1..n]$ of n natural numbers, with $a[i] \geq 1$ for every $i \in \{1, 2, \dots, n\}$, then the input size satisfies

$$|I| = \Omega(n).$$

Example 11.5: the bounds for an array

For the array $a = [6, 1, 3, 8]$ of Example 11.1 we have $n = 4$, $|I| = 10$ and $M = 8$. The lower bound of Observation 11.8 and the upper bound of Observation 11.7 give

$$n = 4 \leq |I| = 10 \leq n(1 + \log_2 M) = 4(1 + 3) = 16.$$

11.5.3 Entries of constant size

In many applications the entries of the array are stored with a fixed number of bits, independent of n . In this case the two bounds meet.

Corollary 11.9: entries of constant size

Given an array $a[1..n]$ of n natural numbers, with $a[i] \geq 1$ for every $i \in \{1, 2, \dots, n\}$, if all the entries of the array can be represented using at most a fixed constant number of bits, then

$$|I| = \Theta(n).$$

Proof. By hypothesis there exists a constant $c \in \mathbb{N}$ such that

$$\lfloor \log_2 a[i] \rfloor + 1 \leq c, \quad \forall i \in \{1, 2, \dots, n\}.$$

Therefore,

$$|I| = \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1) \leq \sum_{i=1}^n c = cn, \quad \forall n \geq 1,$$

and hence $|I| = O(n)$. By Observation 11.8 we also have $|I| = \Omega(n)$, and therefore, by Observation 11.2,

$$|I| = \Theta(n).$$

□

11.5.4 Two variants of the array input

- If the input consists of a natural number $n \geq 1$ and an array $a[1..n]$ of n natural numbers, with $a[i] \geq 1$ for every $i \in \{1, 2, \dots, n\}$, then the input size is

$$|I| = (\lfloor \log_2 n \rfloor + 1) + \sum_{i=1}^n (\lfloor \log_2 a[i] \rfloor + 1).$$

Let

$$M = \max\{n, a[1], a[2], \dots, a[n]\}.$$

Then we have

$$\lfloor \log_2 n \rfloor + 1 \leq 1 + \log_2 M,$$

and, for every $i \in \{1, 2, \dots, n\}$,

$$\lfloor \log_2 a[i] \rfloor + 1 \leq 1 + \log_2 M.$$

Therefore,

$$|I| \leq (1 + \log_2 M) + n(1 + \log_2 M) = (n + 1)(1 + \log_2 M).$$

Hence

$$|I| = O((n + 1)(1 + \log_2 M)).$$

Since $n + 1 \leq 2n$ for every $n \geq 1$, we also obtain

$$|I| = O(n(1 + \log_2 M)).$$

Moreover, if $M \geq 2$, then

$$1 + \log_2 M \leq 2 \log_2 M,$$

and therefore

$$|I| = O(n \log_2 M).$$

The lower bound $|I| = \Omega(n)$ still holds, since the bits of the array alone are at least n .

- If the entries of the array are allowed to be 0, then the input size is

$$|I| = \sum_{i=1}^n (\lfloor \log_2(\max\{1, a[i]\}) \rfloor + 1),$$

so that each entry equal to 0 contributes exactly 1 bit. In this case, if we let

$$M = \max\{1, a[1], a[2], \dots, a[n]\},$$

then the previous formulas become

$$|I| = O(n(1 + \log_2 M)),$$

and, if $M \geq 2$,

$$|I| = O(n \log_2 M).$$

Moreover, we still have

$$|I| = \Omega(n),$$

since every entry contributes at least 1 bit.

In particular, if all the entries of the array can be represented using at most a fixed constant number of bits, then

$$|I| = \Theta(n).$$

11.6 Input size and number of steps

We can now explain why the input size is measured in bits. The time needed by an algorithm is estimated by counting its **steps**: in pseudo-code, a step is the execution of one line, and the condition of a **while** loop counts as one step each time it is evaluated. The number of steps is first computed as a function $S(n)$ of a natural parameter n of the input, and then expressed as a function of the input size $|I|$.

Definition 11.10: worst-case running time

The **worst-case running time** $T(|I|)$ of an algorithm is the maximum number of steps executed by the algorithm on any input I of a given input size $|I|$.

Definition 11.11: polynomial and exponential algorithms

An algorithm is **polynomial** if its worst-case running time satisfies $T(|I|) = O(|I|^k)$ for some constant $k \in \mathbb{N}$.

An algorithm is **exponential** if its worst-case running time satisfies $T(|I|) = 2^{\Theta(|I|)}$, that is, if there exist $c_1, c_2 \in \mathbb{R}_{>0}$ and $k_1 \in \mathbb{N}$ such that

$$2^{c_1|I|} \leq T(|I|) \leq 2^{c_2|I|}, \quad \forall |I| \geq k_1.$$

The parameter n can play two very different roles. It can be the **value** of a number given as input, or the **number of entries** of an array given as input. The next observation shows how n is related to $|I|$ in the two cases.

Observation 11.12: value of a number and number of entries of an array

1. If the input consists of a single natural number $n \geq 1$, then

$$2^{|I|-1} \leq n \leq 2^{|I|} - 1 :$$

the value n is exponential in the input size.

2. If the input consists of an array $a[1..n]$ of n natural numbers (and possibly of the number n), then

$$n \leq |I| :$$

the number of entries n is at most linear in the input size.

Proof. 1. Since $|I| = \lfloor \log_2 n \rfloor + 1$, we have $\lfloor \log_2 n \rfloor = |I| - 1$, that is,

$$|I| - 1 \leq \log_2 n < |I|.$$

The function $x \mapsto 2^x$ is strictly increasing, so $2^{|I|-1} \leq n < 2^{|I|}$. Since n is an integer, $n \leq 2^{|I|} - 1$.

2. Every entry of the array contributes at least one bit to the input size (Observation 11.8, and the same holds for entries equal to 0), so $|I| \geq n$. □

When a number of steps is bounded by exponentials with constant factors, the constant factors can be absorbed in the exponent.

Observation 11.13: constant factors in the exponent

Let $a_1, a_2, b_1, b_2 \in \mathbb{R}_{>0}$. If a function T satisfies

$$a_1 2^{b_1 x} \leq T(x) \leq a_2 2^{b_2 x}$$

for every sufficiently large natural number x , then $T(x) = 2^{\Theta(x)}$.

Proof. We write the constant factors as powers of 2: $a_1 = 2^{\log_2 a_1}$ and $a_2 = 2^{\log_2 a_2}$. Then

$$a_2 2^{b_2 x} = 2^{b_2 x + \log_2 a_2} \leq 2^{2b_2 x} \quad \text{whenever} \quad b_2 x \geq \log_2 a_2,$$

and

$$a_1 2^{b_1 x} = 2^{b_1 x + \log_2 a_1} \geq 2^{\frac{b_1}{2} x} \quad \text{whenever} \quad \frac{b_1}{2} x \geq -\log_2 a_1.$$

Both conditions hold for every sufficiently large x . Therefore, with $c_1 = \frac{b_1}{2}$ and $c_2 = 2b_2$, we obtain $2^{c_1 x} \leq T(x) \leq 2^{c_2 x}$ for every sufficiently large x , that is, $T(x) = 2^{\Theta(x)}$. $\square$

Example 11.6: the sum of the entries of an array

Consider the following algorithm, which computes the sum of the entries of an array.

Algorithm: SumArray

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

Output: $a[1] + a[2] + \cdots + a[n]$

```

1 function SumArray( $a, n$ )
2    $s := 0$ 
3    $i := 1$ 
4   while  $i \leq n$  do
5      $s \leftarrow s + a[i]$ 
6      $i \leftarrow i + 1$ 
7   end
8   return  $s$ 
9 end
```

- **Step 1: number of steps.** Lines 2, 3 and 8 are executed 1 time each; the condition at line 4 is evaluated $n + 1$ times (it is false the last time); lines 5 and 6 are executed n times each. Therefore

$$S(n) = 3 + (n + 1) + 2n = 3n + 4,$$

for every array of n entries, whatever its values.

- **Step 2: from n to $|I|$.** By Observation 11.12, every input of size $|I|$ has $n \leq |I|$ entries, so it requires at most $3|I| + 4$ steps. The array of $|I|$ entries all equal to 1 has input size $|I|$ and requires exactly $3|I| + 4$ steps. Hence

$$T(|I|) = 3|I| + 4 = O(|I|).$$

- **Step 3: conclusion.** SumArray is polynomial (linear) in the input size.

If instead the same loop ran from 1 to the *value* of a number n given as input, the $\Theta(n)$ steps would be exponential in the input size, because n can be as large as $2^{|I|} - 1$ (Observation 11.12).

Key points of the chapter

- The input size $|I|$ is the number of bits needed to write the input instance in binary.
- One natural number n : $|I| = \lfloor \log_2 n \rfloor + 1 = \Theta(\log_2 n)$. Two natural numbers n, m : $|I| = \Theta(\log_2 n + \log_2 m) = \Theta(\log_2 \max\{n, m\})$.
- An array of n natural numbers: $|I| = \Omega(n)$ and $|I| = O(n(1 + \log_2 M))$, with M the maximum entry; the upper bound is not tight in general. If the entries have a constant number of bits, $|I| = \Theta(n)$.
- The same number of steps $S(n)$ can be polynomial or exponential in $|I|$: the value of a number is exponential in its input size, while the number of entries of an array is at most its input size.

Running time

In the chapter «Input size» we learned how to measure the size $|I|$ of an input instance I , that is, the number of bits needed to represent it. In this chapter we use this measure to answer the central question of the analysis of algorithms: **how does the time needed by an algorithm grow when the input grows?**

The plan of the chapter is the following.

- In Section 12.1 we define the running time of an algorithm, the primitive operations (**steps**) that it executes, and the **worst case**. We also fix the conventions used to count the steps of a pseudo-code line by line.
- In Sections 12.2–12.5 we apply the method to four algorithms already met in the previous chapters: **FindMaximum**, **InsertionSort**, **ChangeToBase2** and **PrimalityTest**. For each of them we first count the number of steps $S(n)$ as a function of a natural parameter n , and then we express the order of growth as a function of the input size $|I|$.
- The four analyses lead to four different behaviours (linear, quadratic, linear in $|I|$ although logarithmic in n , exponential in $|I|$ although only $\Theta(\sqrt{n})$ in n), which are compared in Section 12.6.

12.1 Running time of an algorithm

Definition 12.1: running time

The **running time** of an algorithm is an estimate of the time needed to execute it, as a function of the **input size** $|I|$.

- We are not interested in the exact time in seconds, which depends on the computer, on the compiler and on many other details, but in the **order of growth** of the running time, expressed with the asymptotic notation O , Ω and Θ (chapter 9 «Asymptotic notation», recalled in the chapter «Input size»).
- The running time of an algorithm depends on the number of **primitive operations** executed by the algorithm on an input of a given size.

12.1.1 Primitive operations

Definition 12.2: primitive operation

A **primitive operation** is a single operation that can be performed on a computer.

- The principal primitive operations are the following:
 1. arithmetic operations on fixed-size numbers;
 2. comparison of two fixed-size numbers;
 3. assignment of a value to a variable;
 4. accessing an element of an array by index;
 5. control-flow operations (for example, evaluating a condition in an `if` statement, or iterating in a loop).

We make the assumption that a **constant amount of time** is required to execute each primitive operation of an algorithm. In this way the analysis is as machine-independent as possible.

- For short, we call each primitive operation a **step**.

12.1.2 Counting the steps of a pseudo-code

- Every line of the pseudo-codes of these notes executes a **constant** number of primitive operations (a few comparisons, arithmetic operations and assignments). Since we are interested in the order of growth, we count **one step for each execution of a line**: this changes the number of steps only by a constant factor.
- To count the steps we determine, for every line, how many times it is executed, and we add up these numbers. The following conventions are used in all the chapter (and in its exercises).

Line-by-line counting.

- Every execution of a numbered line is **one step**, including the lines `return`.
- The condition of a `while` loop whose body is executed k times is evaluated $k + 1$ times: the last evaluation is the one that finds the condition false and ends the loop.
- The condition of an `if` is evaluated every time the line is reached; the lines of the branch that is not taken are not executed.
- When the number of executions of a line depends on the specific input (and not only on its size), we describe it with an auxiliary function, usually denoted by t , and then we look for the input that makes the total number of steps as large as possible.

12.1.3 Worst-case running time

- Even for inputs of a given size, the running time of an algorithm may depend on which specific input of that size is given.
- Typically, in the analysis of algorithms, the worst possible case is considered, since it determines the longest running time for any input of a given size.

Definition 12.3: worst-case running time

The **worst-case running time** $T(|I|)$ of an algorithm is the maximum number of steps executed by the algorithm on any input I of a given input size $|I|$.

- It is usually easier to count the steps as a function of a **natural parameter** of the input, such as the number n of entries of an array, or the value n of the input number. The input size $|I|$, instead, is the number of bits of the input. This leads to the method followed in all the examples of the chapter.

The two steps of the analysis.

1. Count the number of steps $S(n)$ executed in the worst case, as a function of the natural parameter n , and determine its order of growth (for example $S(n) = \Theta(n^2)$).
2. Determine the relation between n and the input size $|I|$ (for example $|I| = \Theta(n)$, or $|I| = \Theta(\log_2 n)$), and use it to express the order of growth of the worst-case running time $T(|I|)$.

- The second step is not a formality: as we shall see, the same number of steps $\Theta(n)$ corresponds to a running time that is linear in $|I|$ if n is the number of entries of an array, and exponential in $|I|$ if n is the value of the input number.

12.2 Find maximum

Given an array $a[1..n]$ of $n \geq 1$ integer numbers, consider the problem of determining the maximum value in the array.

- This is the pseudo-code of a simple algorithm, called **FindMaximum**, that solves this problem.

Algorithm: FindMaximum**Input** : An array $a[1..n]$ of $n \geq 1$ integer numbers**Output**: The maximum value in array a

```

1 function FindMaximum( $a, n$ )
2    $max := a[1]$ 
3    $i := 2$ 
4   while  $i \leq n$  do
5     if  $a[i] > max$  then
6        $max \leftarrow a[i]$ 
7     end
8      $i \leftarrow i + 1$ 
9   end
10  return  $max$ 
11 end

```

12.2.1 Number of steps

- Let $S(n)$ denote the number of steps executed by FindMaximum in the worst case, where n is the size of the array.
- The body of the loop is executed for $i = 2, 3, \dots, n$, that is, $n - 1$ times, so the condition of line 4 is evaluated n times. The execution of line 6, instead, depends on the iteration. We define a function

$$t : \{2, 3, \dots, n\} \rightarrow \{0, 1\},$$

where $t(i) = 1$ if line 6 is executed at iteration i , and $t(i) = 0$ otherwise.

- The number of times each line of code is executed is:

line	number of executions
2	1
3	1
4	n
5	$n - 1$
6	$\sum_{i=2}^n t(i)$
8	$n - 1$
10	1

- Therefore, the total number of steps is

$$1 + 1 + n + (n - 1) + \sum_{i=2}^n t(i) + (n - 1) + 1 = 3n + 1 + \sum_{i=2}^n t(i).$$

- The value of $t(i)$ depends on the specific input array, so we analyse the worst possible case. The worst possible case is when the elements of the array are in increasing order, that is,

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

Then $t(i) = 1$ for every $i \in \{2, 3, \dots, n\}$: we enter the **if** at every iteration, and we update the maximum value each time. The total number of steps in the worst case is

$$3n + 1 + \sum_{i=2}^n 1 = 3n + 1 + (n - 1) = 4n.$$

- Therefore,

$$S(n) = 4n = \Theta(n).$$

Thus, $S(n)$ grows linearly in n . Indeed,

$$\lim_{n \rightarrow \infty} \frac{4n}{n} = 4 \in \mathbb{R}_{>0}.$$

Example 12.1: counting the steps of FindMaximum

Consider the array $a[1..4] = (3, 1, 4, 5)$.

- Iteration $i = 2$: $a[2] = 1 > 3$ is false, so $t(2) = 0$.
- Iteration $i = 3$: $a[3] = 4 > 3$ is true, max becomes 4, so $t(3) = 1$.
- Iteration $i = 4$: $a[4] = 5 > 4$ is true, max becomes 5, so $t(4) = 1$.

The algorithm returns 5 after

$$3 \cdot 4 + 1 + (t(2) + t(3) + t(4)) = 13 + 2 = 15$$

steps. On the increasing array $(1, 2, 3, 4)$, instead, $t(2) = t(3) = t(4) = 1$ and the algorithm executes $S(4) = 4 \cdot 4 = 16$ steps, the maximum for $n = 4$.

12.2.2 Running time as a function of the input size

- We now determine the worst-case running time of the algorithm by expressing the order of growth of the number of steps as a function of the input size $|I|$.

General case. Since the input of the algorithm consists of an array $a[1..n]$ of n integer numbers, the input size satisfies

$$|I| = \Omega(n),$$

because the input must contain at least the n array entries. This means that, for some constant $c > 0$ and for all sufficiently large n ,

$$|I| \geq cn.$$

Equivalently,

$$n \leq \frac{1}{c}|I|,$$

and hence $n = O(|I|)$. Since $S(n) = \Theta(n)$, there exists a constant $a > 0$ such that

$$S(n) \leq an$$

for all sufficiently large n . Since $n \leq \frac{1}{c}|I|$, we obtain

$$S(n) \leq \frac{a}{c}|I|.$$

Thus, the number of steps is bounded above by a constant times the input size.

The worst-case running time of the **FindMaximum** algorithm, as a function of the input size $|I|$, is

$$T(|I|) = O(|I|).$$

Thus, in the general case, the worst-case running time is at most linear in the input size.

Entries with a fixed number of bits. We often make the hypothesis that all input values can be stored using a fixed constant number of bits, for example 32 bits. Under this hypothesis, the input size satisfies

$$|I| = \Theta(n),$$

because the input consists of n array entries and each entry requires a constant number of bits. This means that there exist constants $c_1, c_2 > 0$ such that, for all sufficiently large n ,

$$c_1 n \leq |I| \leq c_2 n.$$

Equivalently,

$$\frac{1}{c_2}|I| \leq n \leq \frac{1}{c_1}|I|,$$

that is, $n = \Theta(|I|)$. Since $S(n) = \Theta(n)$, there exist constants $a_1, a_2 > 0$ such that, for all sufficiently large n ,

$$a_1 n \leq S(n) \leq a_2 n.$$

Using the bounds on n in terms of $|I|$, we obtain

$$\frac{a_1}{c_2}|I| \leq S(n) \leq \frac{a_2}{c_1}|I|,$$

that is, $S(n) = \Theta(|I|)$.

If all input values use a fixed constant number of bits, then the worst-case running time of the **FindMaximum** algorithm is

$$T(|I|) = \Theta(|I|).$$

Thus, under this hypothesis, the algorithm is linear in the input size.

12.3 Insertion sort

- In `FindMaximum` the loop body executes a constant number of steps. In `InsertionSort` the body of the outer loop contains another loop, whose number of iterations depends on the input: this produces a quadratic number of steps.

Given an array $a[1..n]$ of n natural numbers, consider the problem of sorting the array in non-decreasing order:

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

- This is the pseudo-code of the `InsertionSort` algorithm.

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      $key := a[i]$ 
5      $j := i - 1$ 
6     while  $j \geq 1$  and  $a[j] > key$  do
7        $a[j + 1] \leftarrow a[j]$ 
8        $j \leftarrow j - 1$ 
9     end
10     $a[j + 1] \leftarrow key$ 
11     $i \leftarrow i + 1$ 
12  end
13 end

```

12.3.1 Number of steps

- Let $S(n)$ denote the number of steps executed by `InsertionSort` in the worst case, where n is the size of the array.
- The execution of lines 7–8 depends on the iteration. We define a function

$$t : \{2, 3, \dots, n\} \rightarrow \{0, 1, \dots, n - 1\},$$

where $t(i)$ denotes the number of times the body of the inner loop (lines 7–8) is executed during the iteration i of the outer loop. The condition of the inner loop (line 6) is then evaluated $t(i) + 1$ times during the same iteration.

- The number of times each line of code is executed is:

line	number of executions
2	1
3	n
4	$n - 1$
5	$n - 1$
6	$(n - 1) + \sum_{i=2}^n t(i)$
7	$\sum_{i=2}^n t(i)$
8	$\sum_{i=2}^n t(i)$
10	$n - 1$
11	$n - 1$

- Therefore, the total number of steps is

$$\begin{aligned}
& 1 + n + (n - 1) + (n - 1) + \left((n - 1) + \sum_{i=2}^n t(i) \right) + \sum_{i=2}^n t(i) + \sum_{i=2}^n t(i) + (n - 1) + (n - 1) \\
&= 6n - 4 + 3 \sum_{i=2}^n t(i).
\end{aligned}$$

- The value of $t(i)$ depends on the specific input array, so we analyse the worst possible case. Since j starts from $i - 1$ and decreases down to 0 at most, we always have $t(i) \leq i - 1$. The worst possible case is when the elements of the array are in strictly decreasing order, that is,

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

In this case, at every outer iteration i , the element $key = a[i]$ is smaller than all elements before it. Hence the inner while loop runs to completion:

$$t(i) = i - 1 \quad \forall i \in \{2, 3, \dots, n\}.$$

Therefore,

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

- The total number of steps in the worst case is

$$6n - 4 + 3 \cdot \frac{n(n - 1)}{2} = \frac{12n - 8 + 3n^2 - 3n}{2} = \frac{3n^2 + 9n - 8}{2}.$$

Therefore,

$$S(n) = \frac{3n^2 + 9n - 8}{2} = \Theta(n^2).$$

Thus, $S(n)$ grows quadratically in n . Indeed,

$$\lim_{n \rightarrow \infty} \frac{3n^2 + 9n - 8}{2n^2} = \lim_{n \rightarrow \infty} \frac{1}{2} \left(3 + \frac{9}{n} - \frac{8}{n^2} \right) = \frac{3}{2} \in \mathbb{R}_{>0}.$$

Example 12.2: the function t of InsertionSort

Consider the array $a[1..6] = (5, 2, 4, 6, 1, 3)$. At each iteration i , $t(i)$ is the number of elements of $a[1..i-1]$ that are larger than $key = a[i]$, i.e., the number of elements shifted one position to the right.

i	key	$t(i)$	array after the iteration
2	2	1	(2, 5, 4, 6, 1, 3)
3	4	1	(2, 4, 5, 6, 1, 3)
4	6	0	(2, 4, 5, 6, 1, 3)
5	1	4	(1, 2, 4, 5, 6, 3)
6	3	3	(1, 2, 3, 4, 5, 6)

We have $\sum_{i=2}^6 t(i) = 9$, hence the algorithm executes

$$6 \cdot 6 - 4 + 3 \cdot 9 = 59$$

steps. On a decreasing array of 6 elements, instead, $\sum_{i=2}^6 t(i) = 1 + 2 + 3 + 4 + 5 = 15$ and the algorithm executes

$$S(6) = \frac{3 \cdot 36 + 9 \cdot 6 - 8}{2} = 77$$

steps.

12.3.2 Running time as a function of the input size

- We now determine the worst-case running time of the algorithm by expressing the order of growth of the number of steps as a function of the input size $|I|$.

General case. Since the input of the algorithm consists of an array $a[1..n]$ of n natural numbers, the input size satisfies

$$|I| = \Omega(n),$$

because the input must contain at least the n array entries. This means that, for some constant $c > 0$ and for all sufficiently large n ,

$$|I| \geq cn.$$

Equivalently,

$$n \leq \frac{1}{c}|I|,$$

and hence $n = O(|I|)$. Since $S(n) = \Theta(n^2)$, there exists a constant $a > 0$ such that

$$S(n) \leq an^2$$

for all sufficiently large n . Since $n \leq \frac{1}{c}|I|$, we obtain

$$S(n) \leq a \left(\frac{1}{c}|I| \right)^2 = \frac{a}{c^2}|I|^2.$$

Thus, the number of steps is bounded above by a constant times $|I|^2$.

The worst-case running time of the **InsertionSort** algorithm, as a function of the input size $|I|$, is

$$T(|I|) = O(|I|^2).$$

Thus, in the general case, the worst-case running time is at most quadratic in the input size.

Entries with a fixed number of bits. We often make the hypothesis that all input values can be stored using a fixed constant number of bits, for example 32 bits. Under this hypothesis, the input size satisfies

$$|I| = \Theta(n),$$

because the input consists of n array entries and each entry requires a constant number of bits. This means that there exist constants $c_1, c_2 > 0$ such that, for all sufficiently large n ,

$$c_1 n \leq |I| \leq c_2 n.$$

Equivalently,

$$\frac{1}{c_2} |I| \leq n \leq \frac{1}{c_1} |I|.$$

Since $S(n) = \Theta(n^2)$, there exist constants $a_1, a_2 > 0$ such that, for all sufficiently large n ,

$$a_1 n^2 \leq S(n) \leq a_2 n^2.$$

Using the bounds on n in terms of $|I|$, we obtain

$$a_1 \left(\frac{1}{c_2} |I| \right)^2 \leq S(n) \leq a_2 \left(\frac{1}{c_1} |I| \right)^2, \quad \text{that is,} \quad \frac{a_1}{c_2^2} |I|^2 \leq S(n) \leq \frac{a_2}{c_1^2} |I|^2.$$

Thus, $S(n) = \Theta(|I|^2)$.

If all input values use a fixed constant number of bits, then the worst-case running time of the **InsertionSort** algorithm is

$$T(|I|) = \Theta(|I|^2).$$

Thus, under this hypothesis, the algorithm is quadratic in the input size.

12.4 Change of base

- In the first two examples the input is an array and the natural parameter n is the number of its entries. We now consider an algorithm whose input is a **single number** n : the parameter n is now a value, and the relation between n and $|I|$ changes completely.

Given a natural number $n \geq 1$ represented in base 10, consider the problem of converting n to base 2.

- This is the pseudo-code of the **ChangeToBase2** algorithm. It is the algorithm **ChangeOfBase** of the chapter «Numerical algorithms», with base $b = 2$. The binary digits are stored in an array $d[0..nd - 1]$ indexed from 0, so that $d[k]$ is the digit of weight 2^k .

Algorithm: ChangeToBase2

Input : A natural number $n \geq 1$ in base 10

Output: An array d of binary digits and the number of digits nd

```

1 function ChangeToBase2( $n$ )
2    $nd := \lfloor \log_2 n \rfloor + 1$ 
3    $d := \text{new array}[0..nd - 1]$ 
4    $k := 0$ 
5   while  $n > 0$  do
6      $q := \lfloor n/2 \rfloor$ 
7      $r := n \bmod 2$ 
8      $d[k] \leftarrow r$ 
9      $n \leftarrow q$ 
10     $k \leftarrow k + 1$ 
11  end
12  return  $d, nd$ 
13 end
```

12.4.1 Number of steps

- Let $S(n)$ denote the number of steps executed by **ChangeToBase2**, where n is the input value to be converted. There is only one input for each value of n , so there is no distinction between the worst case and the other cases.
- The number of digits needed to represent n in base 2 is

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

At every iteration of the while loop, the current value of n is replaced by $\lfloor n/2 \rfloor$. After s iterations, the current value is

$$\left\lfloor \frac{n}{2^s} \right\rfloor.$$

The loop stops when this value becomes 0, that is, when

$$\left\lfloor \frac{n}{2^s} \right\rfloor = 0.$$

This happens exactly when $2^s > n$. The smallest integer s such that $2^s > n$ is

$$s = \lfloor \log_2 n \rfloor + 1.$$

Therefore, the body of the while loop is executed exactly $\lfloor \log_2 n \rfloor + 1$ times (one iteration for each binary digit), and its condition is evaluated $\lfloor \log_2 n \rfloor + 2$ times.

- The number of times each line of code is executed is:

line	number of executions
2, 3, 4	1 each
5	$\lfloor \log_2 n \rfloor + 2$
6, 7, 8, 9, 10	$\lfloor \log_2 n \rfloor + 1$ each
12	1

- Therefore, the total number of steps is

$$1 + 1 + 1 + (\lfloor \log_2 n \rfloor + 2) + 5(\lfloor \log_2 n \rfloor + 1) + 1 = 6(\lfloor \log_2 n \rfloor + 1) + 5 = 6\lfloor \log_2 n \rfloor + 11.$$

Therefore,

$$S(n) = 6\lfloor \log_2 n \rfloor + 11 = \Theta(\log_2 n).$$

In words, the number of steps grows logarithmically in the numerical value n .

- Indeed, for $n > 1$,

$$\log_2 n - 1 < \lfloor \log_2 n \rfloor \leq \log_2 n.$$

Dividing by $\log_2 n > 0$, we obtain

$$1 - \frac{1}{\log_2 n} < \frac{\lfloor \log_2 n \rfloor}{\log_2 n} \leq 1.$$

Since both $1 - \frac{1}{\log_2 n}$ and 1 converge to 1, by the squeeze theorem,

$$\lim_{n \rightarrow \infty} \frac{\lfloor \log_2 n \rfloor}{\log_2 n} = 1.$$

Therefore,

$$\lim_{n \rightarrow \infty} \frac{6\lfloor \log_2 n \rfloor + 11}{\log_2 n} = \lim_{n \rightarrow \infty} \left(6 \frac{\lfloor \log_2 n \rfloor}{\log_2 n} + \frac{11}{\log_2 n} \right) = 6 \cdot 1 + 0 = 6 \in \mathbb{R}_{>0}.$$

Example 12.3: ChangeToBase2 on $n = 13$

We have $\lfloor \log_2 13 \rfloor + 1 = 4$, so the array $d[0..3]$ is created and the loop executes 4 iterations:

iteration	n	$q = \lfloor n/2 \rfloor$	$r = n \bmod 2$	assignment
1	13	6	1	$d[0] = 1$
2	6	3	0	$d[1] = 0$
3	3	1	1	$d[2] = 1$
4	1	0	1	$d[3] = 1$

After the fourth iteration $n = 0$: the condition of line 5 is evaluated for the fifth time, it is false, and the algorithm returns the digits of $13 = (1101)_2$. The number of steps is

$$S(13) = 3 + 5 + 5 \cdot 4 + 1 = 29 = 6 \cdot 3 + 11.$$

12.4.2 Running time as a function of the input size

- We now express the running time as a function of the input size $|I|$. This is the delicate change of variable.
- The input is not the abstract value n , but a representation of n . Since n is represented in base 10, the number of digits needed to write n is

$$\lfloor \log_{10} n \rfloor + 1.$$

Each decimal digit requires a constant number of bits. Hence the input size satisfies

$$|I| = \Theta(\lfloor \log_{10} n \rfloor + 1) = \Theta(\log_{10} n).$$

Since the base of a logarithm changes it only by a constant factor ($\log_{10} n = \log_2 n / \log_2 10$), we get

$$|I| = \Theta(\log_2 n),$$

the same order of growth obtained in the chapter «Input size» for a number written in binary. This means that there exist constants $c_1, c_2 > 0$ such that, for all sufficiently large n ,

$$c_1 \log_2 n \leq |I| \leq c_2 \log_2 n.$$

Equivalently,

$$\frac{1}{c_2} |I| \leq \log_2 n \leq \frac{1}{c_1} |I|.$$

- Since $S(n) = \Theta(\log_2 n)$, there exist constants $a_1, a_2 > 0$ such that, for all sufficiently large n ,

$$a_1 \log_2 n \leq S(n) \leq a_2 \log_2 n.$$

Using the bounds on $\log_2 n$ in terms of $|I|$, we obtain

$$\frac{a_1}{c_2} |I| \leq S(n) \leq \frac{a_2}{c_1} |I|,$$

that is, $S(n) = \Theta(|I|)$.

The running time of the **ChangeToBase2** algorithm is

$$T(|I|) = \Theta(|I|).$$

Thus, the algorithm is logarithmic in the numerical value n , but linear in the input size $|I|$, because the input size is itself proportional to $\log_2 n$.

12.5 Primality test

- The input of the last example is again a single number n , but now the number of steps grows as $\sqrt{n}$, much faster than $\log_2 n$. We shall see that this makes the algorithm exponential in the input size.

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

- This is the pseudo-code of the **PrimalityTest** algorithm of the chapter «Numerical algorithms».

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
4   end
5    $d := 2$ 
6   while  $d \leq \lfloor \sqrt{n} \rfloor$  do
7     if  $n \bmod d = 0$  then
8       return false
9     end
10     $d \leftarrow d + 1$ 
11  end
12  return true
13 end
```

12.5.1 Number of steps

- Let $S(n)$ denote the number of steps executed by **PrimalityTest** in the worst case, where n is the input value. We assume $n \geq 2$, since for $n < 2$ the algorithm stops after 2 steps.
- **The worst case is when n is prime.** In this case the algorithm never returns early at line 8, so the while loop runs to completion. The variable d takes the values

$$2, 3, \dots, \lfloor \sqrt{n} \rfloor,$$

hence the body of the while loop is executed $\lfloor \sqrt{n} \rfloor - 1$ times. The condition of the while loop is evaluated once more, when $d = \lfloor \sqrt{n} \rfloor + 1$, so line 6 is executed $\lfloor \sqrt{n} \rfloor$ times.

- The number of times each line of code is executed in the worst case is:

line	number of executions (n prime)
2	1
3	0 (since $n \geq 2$)
5	1
6	$\lfloor \sqrt{n} \rfloor$
7	$\lfloor \sqrt{n} \rfloor - 1$
8	0 (since n is prime)
10	$\lfloor \sqrt{n} \rfloor - 1$
12	1

- Therefore, the total number of steps in the worst case is

$$1 + 1 + \lfloor \sqrt{n} \rfloor + (\lfloor \sqrt{n} \rfloor - 1) + (\lfloor \sqrt{n} \rfloor - 1) + 1 = 3\lfloor \sqrt{n} \rfloor + 1.$$

- This is indeed the worst case. If $n \geq 2$ is not prime, let p be its smallest divisor larger than 1: then $p \leq \lfloor \sqrt{n} \rfloor$, and the algorithm returns **false** at line 8 when $d = p$. Line 6 is executed $p - 1$ times, line 7 $p - 1$ times, line 10 $p - 2$ times, and lines 2, 5, 8 once each, for a total of

$$3p - 1 \leq 3\lfloor \sqrt{n} \rfloor - 1 < 3\lfloor \sqrt{n} \rfloor + 1$$

steps.

- Therefore,

$$S(n) = 3\lfloor \sqrt{n} \rfloor + 1 = \Theta(\sqrt{n}).$$

Thus, $S(n)$ grows as the square root of n . Indeed,

$$\sqrt{n} - 1 < \lfloor \sqrt{n} \rfloor \leq \sqrt{n}.$$

Dividing by $\sqrt{n} > 0$, we obtain

$$1 - \frac{1}{\sqrt{n}} < \frac{\lfloor \sqrt{n} \rfloor}{\sqrt{n}} \leq 1.$$

Since both $1 - \frac{1}{\sqrt{n}}$ and 1 converge to 1, by the squeeze theorem,

$$\lim_{n \rightarrow \infty} \frac{\lfloor \sqrt{n} \rfloor}{\sqrt{n}} = 1.$$

Therefore,

$$\lim_{n \rightarrow \infty} \frac{3\lfloor \sqrt{n} \rfloor + 1}{\sqrt{n}} = \lim_{n \rightarrow \infty} \left(3 \frac{\lfloor \sqrt{n} \rfloor}{\sqrt{n}} + \frac{1}{\sqrt{n}} \right) = 3 \cdot 1 + 0 = 3 \in \mathbb{R}_{>0}.$$

Example 12.4: PrimalityTest on a prime and on a composite number

- $n = 97$ is prime and $\lfloor \sqrt{97} \rfloor = 9$. The algorithm tries the divisors $d = 2, 3, \dots, 9$ (8 iterations), finds no divisor, evaluates the condition of line 6 a ninth time with $d = 10$, and returns **true** after

$$S(97) = 3 \cdot 9 + 1 = 28$$

steps.

- $n = 91 = 7 \cdot 13$ is not prime, and its smallest divisor larger than 1 is $p = 7$. The algorithm tries $d = 2, 3, \dots, 6$ without success, finds $91 \bmod 7 = 0$, and returns **false** after

$$3 \cdot 7 - 1 = 20$$

steps, although $\lfloor \sqrt{91} \rfloor = 9$ as for 97.

12.5.2 Running time as a function of the input size

- We now determine the worst-case running time by expressing the order of growth of the number of steps as a function of the input size $|I|$.
- The input is a single natural number n . To represent n in binary, we need $\lfloor \log_2 n \rfloor + 1$ bits, so the input size satisfies

$$|I| = \Theta(\log_2 n).$$

This means that there exist constants $c_1, c_2 > 0$ such that, for all sufficiently large n ,

$$c_1 \log_2 n \leq |I| \leq c_2 \log_2 n.$$

Equivalently,

$$\frac{1}{c_2} |I| \leq \log_2 n \leq \frac{1}{c_1} |I|.$$

The function $x \mapsto 2^x$ is strictly increasing. Hence, applying 2^x to all terms of the inequality preserves the order:

$$2^{\frac{1}{c_2} |I|} \leq 2^{\log_2 n} \leq 2^{\frac{1}{c_1} |I|}.$$

Since $2^{\log_2 n} = n$, we obtain

$$2^{\frac{1}{c_2} |I|} \leq n \leq 2^{\frac{1}{c_1} |I|}.$$

Moreover, $\sqrt{n} = n^{1/2}$. Taking the square root of all terms, which again preserves the order because all terms are positive, and using $(2^x)^{1/2} = 2^{x/2}$, we get

$$2^{\frac{1}{2c_2} |I|} \leq \sqrt{n} \leq 2^{\frac{1}{2c_1} |I|}.$$

- Since $S(n) = \Theta(\sqrt{n})$, there exist constants $a_1, a_2 > 0$ such that, for all sufficiently large n ,

$$a_1\sqrt{n} \leq S(n) \leq a_2\sqrt{n}.$$

Using the bounds on $\sqrt{n}$ in terms of $|I|$, we obtain

$$a_1 2^{\frac{1}{2c_2}|I|} \leq S(n) \leq a_2 2^{\frac{1}{2c_1}|I|}.$$

Thus,

$$S(n) = 2^{\Theta(|I|)}.$$

- The worst case is attained on prime numbers, and there are prime numbers of every size: for every $k \geq 2$ there is a prime number with exactly k binary digits, i.e., between 2^{k-1} and 2^k (Bertrand's postulate). Hence, for every input size, the maximum number of steps is attained by a prime n and it satisfies the bounds above.

The worst-case running time of the **PrimalityTest** algorithm, as a function of the input size $|I|$, is

$$T(|I|) = 2^{\Theta(|I|)}.$$

Accordingly, the order of growth of the worst-case running time of the **PrimalityTest** algorithm is **exponential** in the input size.

12.6 Summary

- The following table collects the results of the chapter. For the two algorithms on arrays, the running time is given under the hypothesis that all entries use a fixed constant number of bits.

algorithm	parameter n	worst-case steps $S(n)$	input size $ I $	running time $T(I)$
FindMaximum	entries of the array	$4n = \Theta(n)$	$\Theta(n)$	$\Theta(I)$
InsertionSort	entries of the array	$\frac{3n^2+9n-8}{2} = \Theta(n^2)$	$\Theta(n)$	$\Theta(I ^2)$
ChangeToBase2	value of the number	$6\lfloor \log_2 n \rfloor + 11 = \Theta(\log_2 n)$	$\Theta(\log_2 n)$	$\Theta(I)$
PrimalityTest	value of the number	$3\lfloor \sqrt{n} \rfloor + 1 = \Theta(\sqrt{n})$	$\Theta(\log_2 n)$	$2^{\Theta(I)}$

Key points.

- The running time is measured in **steps** (primitive operations, each executed in constant time); we count one step for each execution of a line of pseudo-code.
- We consider the **worst case**: the maximum number of steps over all inputs of a given size. When the number of executions of a line depends on the input, an auxiliary function t describes it, and we look for the input that maximizes the total.
- The analysis has **two steps**: first the number of steps $S(n)$ as a function of a natural parameter n , then the running time $T(|I|)$ as a function of the input size.
- If n is the number of entries of an array (with entries of fixed size), then $|I| = \Theta(n)$ and the order of growth does not change: $S(n) = \Theta(n^k)$ gives $T(|I|) = \Theta(|I|^k)$.

- If n is the **value** of the input number, then $|I| = \Theta(\log_2 n)$, i.e., $n = 2^{\Theta(|I|)}$: a number of steps $\Theta(\log_2 n)$ is linear in $|I|$, while any number of steps $\Theta(\sqrt{n})$ or $\Theta(n)$ is **exponential** in $|I|$.

Correctness

13.1 Correctness of an algorithm

- The previous chapters measured how much an algorithm costs: the size of its input and its running time. Before asking how fast an algorithm is, however, we must be sure that it does what it promises. This chapter presents the standard tool to prove it, the **loop invariant**, and applies it to algorithms already met in the chapters «Sorting», «Linear and binary search» and «Numerical algorithms»: Find Maximum, Insertion Sort, Binary Search and the Euclidean algorithm.
- The first part of the analysis of an algorithm consists in determining its **correctness**, i.e., proving that it computes the correct output for every possible input instance.
- Testing an algorithm on some instances can show that it is wrong (one counterexample is enough), but it can never show that it is correct: there are infinitely many input instances. Correctness must be **proved**.

Definition 13.1: correct algorithm

An algorithm is **correct** if, for every input instance I of the problem, it **terminates** after a finite number of steps and it **returns the correct output** for I .

- The algorithms we consider consist of sequences of instructions and of loops. Sequences of instructions are easy to analyse, while a loop executes the same instructions a number of times that depends on the input.

The correctness of an algorithm is based on the correctness of its loops. The correctness of a loop is proved by means of a **loop invariant**.

13.1.1 Loop invariants

Definition 13.2: loop invariant

A **loop invariant** is a property of the variables of an algorithm that is true every time the condition of a loop is evaluated, i.e., before the first iteration of the loop and after each iteration of the loop.

- To prove the correctness of an algorithm by means of a loop invariant, we must show three things:
 1. **Initialization:** the loop invariant is true before the first iteration of the loop.
 2. **Maintenance:** if the loop invariant is true before an iteration of the loop, then it remains true at the end of the iteration, or equivalently, it is true before the next iteration.
 3. **Termination:** the loop terminates after a finite number of iterations. When the loop terminates, the loop invariant, together with the fact that the condition of the loop is false, gives a useful property that shows that the algorithm is correct.
- Note the similarity with **mathematical induction**, where to prove that a property holds it is necessary to prove a base case and an inductive step:
 - showing that the invariant holds before the first iteration corresponds to the **base case**;
 - showing that the invariant holds from iteration to iteration corresponds to the **inductive step**.

The termination property differs from the usual use of mathematical induction, in which the inductive step is applied infinitely many times: here, we stop the “induction” when the loop terminates.

- To prove that a loop terminates, we usually identify a natural number (for example, $n - i + 1$ or the length of a sub-array) that strictly decreases at every iteration and that must be positive for the loop to continue. Since a natural number cannot decrease forever, the loop terminates.

Scheme of a correctness proof with a loop invariant

1. State the invariant: a property of the variables, true every time the condition of the loop is evaluated.
2. **Initialization:** check it with the values assigned before the loop.
3. **Maintenance:** assume it and the condition of the loop, execute the body once, and check it with the new values.
4. **Termination:** find a natural number that strictly decreases at each iteration; then combine the invariant with the negated condition of the loop to obtain the correct output.

13.2 Correctness of Find Maximum

Given an array $a[1..n]$ of n integer numbers, with $n \geq 1$, consider the problem of determining the maximum value in the array.

- The first example is the simplest loop: a single scan of the array. This is the pseudo-code of the `FindMaximum` algorithm.

Algorithm: FindMaximum

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

Output: The maximum value in array a

```

1 function FindMaximum( $a, n$ )
2    $max := a[1]$ 
3    $i := 2$ 
4   while  $i \leq n$  do
5     if  $a[i] > max$  then
6        $max \leftarrow a[i]$ 
7     end
8      $i \leftarrow i + 1$ 
9   end
10  return  $max$ 
11 end

```

- The **loop invariant** of the while loop of `FindMaximum` is:

Every time the condition at line 4 is evaluated, the variable max is equal to the maximum value of the sub-array $a[1..i-1]$:

$$max = \max\{a[1], a[2], \dots, a[i-1]\}.$$

Example 13.1: Loop invariant of FindMaximum

Consider the array $a = [9, 12, 3, 16, 19, 11, 12]$ with $n = 7$. The table shows the values of i and max every time the condition at line 4 is evaluated:

i	max	$a[1..i-1]$	$\max\{a[1], \dots, a[i-1]\}$
2	9	[9]	9
3	12	[9, 12]	12
4	12	[9, 12, 3]	12
5	16	[9, 12, 3, 16]	16
6	19	[9, 12, 3, 16, 19]	19
7	19	[9, 12, 3, 16, 19, 11]	19
8	19	[9, 12, 3, 16, 19, 11, 12]	19 (loop ends)

At every evaluation the invariant holds. When the loop ends $i = n + 1 = 8$, and the algorithm returns 19, the maximum of the whole array.

- Using the loop invariant we can prove the correctness of the **FindMaximum** algorithm, which is stated by the following observation.

Observation 13.3

Given an array $a[1..n]$ of $n \geq 1$ integer numbers, the **FindMaximum** algorithm terminates and returns

$$\max\{a[1], a[2], \dots, a[n]\}.$$

Proof. We prove the three properties of the loop invariant.

- **Initialization.** Before the first iteration, lines 2 and 3 set $max = a[1]$ and $i = 2$. The sub-array $a[1..i - 1] = a[1..1]$ consists of the single element $a[1]$, whose maximum is $a[1] = max$. Hence the invariant holds.
- **Maintenance.** Assume that, before an iteration, $max = \max\{a[1], \dots, a[i - 1]\}$. The body of the loop compares $a[i]$ with max (line 5):
 - if $a[i] > max$, then line 6 sets $max \leftarrow a[i]$, and $a[i]$ is greater than all the elements of $a[1..i - 1]$;
 - if $a[i] \leq max$, then max is not changed and it is greater than or equal to $a[i]$.

In both cases, at the end of the body

$$max = \max\{a[1], \dots, a[i - 1], a[i]\}.$$

Line 8 increases i by 1, so that, with the new value of i , we have again $max = \max\{a[1], \dots, a[i - 1]\}$. Hence the invariant holds before the next iteration.

- **Termination.** The variable i starts from $2 \leq n + 1$ and increases by 1 at every iteration, therefore the loop terminates after exactly $n - 1$ iterations, when for the first time $i = n + 1$. Substituting $i = n + 1$ in the loop invariant we obtain

$$max = \max\{a[1], a[2], \dots, a[n]\},$$

which is the value returned at line 10. Hence, the algorithm is correct. □

13.3 Correctness of Insertion Sort

Given an array $a[1..n]$ of n natural numbers, with $n \geq 1$, consider the problem of rearranging the elements of the array in non-decreasing order:

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

- With two nested loops the proof needs two invariants, one for each loop. This is the pseudo-code of the **InsertionSort** algorithm.

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 correct position
11     $i \leftarrow i + 1$ 
12  end
13 end

```

- The condition at line 6 is evaluated from left to right, and the second part is evaluated only if the first one is true: when $j = 0$ the condition is false and the non-existent element $a[0]$ is never accessed. The C++ condition `i > -1 and a[i] > key` behaves in the same way.
- The algorithm contains two nested loops. We first prove what a single execution of the **inner** while loop (lines 6–8) does, and then we use this result to prove the correctness of the **outer** while loop (lines 3–11).

13.3.1 The inner loop

- Consider one iteration of the outer loop, with a given value of $i \in \{2, 3, \dots, n\}$. We denote by

$$b[1..i]$$

the content of the sub-array $a[1..i]$ at the beginning of this iteration (i.e., when the condition at line 3 is true), and we assume that $b[1..i - 1]$ is sorted in non-decreasing order. After line 4 we have $key = b[i]$.

- The **loop invariant** of the inner while loop is:

Every time the condition at line 6 is evaluated:

- (I1) $a[1..j] = b[1..j]$ (the first j elements have not been moved);
- (I2) $a[j + 2..i] = b[j + 1..i - 1]$ (the elements $b[j + 1], \dots, b[i - 1]$ have been shifted one position to the right);
- (I3) every element of $a[j + 2..i]$ is strictly greater than key .

The entries $a[i + 1..n]$ are never modified by the inner loop.

- The position $j + 1$ is the “hole” in which key will be inserted: its current content is a copy that will be overwritten. A sub-array $a[l..r]$ with $l > r$ is **empty**, and any property of all its elements is trivially true.

Example 13.2: Loop invariant of the inner loop

Consider the array $a = [5, 2, 4, 1, 3]$ at the beginning of the iteration $i = 5$ of the outer loop: $b[1..5] = [1, 2, 4, 5, 3]$ and $key = 3$. The table shows the array every time the condition at line 6 is evaluated.

j	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[1..j]$	$a[j + 2..5]$	condition
4	1	2	4	5	3	[1, 2, 4, 5]	[]	$a[4] = 5 > 3$: true
3	1	2	4	5	5	[1, 2, 4]	[5]	$a[3] = 4 > 3$: true
2	1	2	4	4	5	[1, 2]	[4, 5]	$a[2] = 2 \leq 3$: false

At every evaluation $a[1..j] = b[1..j]$, $a[j + 2..5] = b[j + 1..4]$ and all the elements of $a[j + 2..5]$ are greater than 3. When the loop ends $j = 2$, and line 10 sets $a[3] \leftarrow 3$, giving $a[1..5] = [1, 2, 3, 4, 5]$.

Lemma 13.4: inner loop of Insertion Sort

Consider an iteration of the outer loop of **InsertionSort** with $i \in \{2, \dots, n\}$, and let $b[1..i]$ be the content of $a[1..i]$ at the beginning of the iteration. If $b[1..i - 1]$ is sorted in non-decreasing order, then the inner while loop terminates and, after line 10, the sub-array $a[1..i]$ contains the elements $b[1], \dots, b[i]$ sorted in non-decreasing order, while

$a[i + 1..n]$ is unchanged.

Proof. We prove the three properties of the loop invariant (I1)–(I3).

- **Initialization.** Before the first evaluation of the condition, $j = i - 1$ and nothing has been modified. Hence $a[1..i - 1] = b[1..i - 1]$, i.e. (I1) holds. The sub-arrays $a[j + 2..i] = a[i + 1..i]$ and $b[j + 1..i - 1] = b[i..i - 1]$ are both empty, so (I2) and (I3) hold trivially.
- **Maintenance.** Assume that (I1)–(I3) hold and that the condition at line 6 is true, i.e., $j \geq 1$ and $a[j] > key$. By (I1), $a[j] = b[j]$, so $b[j] > key$. Line 7 sets $a[j + 1] \leftarrow a[j] = b[j]$; therefore, by (I2),

$$a[j + 1..i] = b[j..i - 1],$$

and all these elements are greater than key (by (I3) and $b[j] > key$). Positions $1, \dots, j$ are not modified. Line 8 sets $j \leftarrow j - 1$. Denoting by $j' = j - 1$ the new value, we have $a[1..j'] = b[1..j']$, $a[j' + 2..i] = a[j + 1..i] = b[j..i - 1] = b[j' + 1..i - 1]$, and all the elements of $a[j' + 2..i]$ are greater than key . Hence (I1)–(I3) hold before the next evaluation of the condition.

- **Termination.** The variable j starts from $i - 1$ and decreases by 1 at every iteration; since the loop stops at the latest when $j = 0$, the inner loop executes at most $i - 1$ iterations and terminates. When it terminates, either $j = 0$ or $a[j] = b[j] \leq key$. By (I1)–(I3):

- $a[1..j] = b[1..j]$ is sorted and, if $j \geq 1$, its largest element $b[j]$ is $\leq key$;
- $a[j + 2..i] = b[j + 1..i - 1]$ is sorted and all its elements are $> key$.

Line 10 sets $a[j + 1] \leftarrow key = b[j]$. Therefore

$$a[1..i] = [b[1], \dots, b[j], key, b[j + 1], \dots, b[i - 1]],$$

which contains exactly the elements $b[1], \dots, b[i]$ and is sorted in non-decreasing order, because $b[1..j]$ is sorted, $b[j] \leq key < b[j + 1]$, and $b[j + 1..i - 1]$ is sorted. Positions $i + 1, \dots, n$ are never accessed.

□

13.3.2 The outer loop

- The **loop invariant** of the outer while loop is:

Every time the condition at line 3 is evaluated, the sub-array $a[1..i - 1]$ consists of the elements originally in $a[1..i - 1]$, but sorted in non-decreasing order, and the sub-array $a[i..n]$ is equal to the original sub-array $a[i..n]$.

Example 13.3: Loop invariant of the outer loop

Consider the array $a = [5, 2, 4, 1, 3]$ with $n = 5$. The table shows the array every time the condition at line 3 is evaluated; the sorted sub-array $a[1..i - 1]$ is highlighted in blue.

i	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	original $a[1..i - 1]$
2	5	2	4	1	3	[5]
3	2	5	4	1	3	[5, 2]
4	2	4	5	1	3	[5, 2, 4]
5	1	2	4	5	3	[5, 2, 4, 1]
6	1	2	3	4	5	[5, 2, 4, 1, 3] (loop ends)

- Using the loop invariant and Lemma 13.4 we can prove the correctness of the InsertionSort algorithm.

Observation 13.5

Given an array $a[1..n]$ of $n \geq 1$ natural numbers, the InsertionSort algorithm terminates and, at the end, the array $a[1..n]$ contains the original elements rearranged in non-decreasing order:

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

Proof. We prove the three properties of the loop invariant of the outer loop.

- **Initialization.** Before the first iteration, $i = 2$ (line 2) and the array has not been modified. The sub-array $a[1..i - 1] = a[1..1]$ consists of the single original element $a[1]$, and it is trivially sorted. The sub-array $a[2..n]$ is the original one. Hence the invariant holds.
- **Maintenance.** Assume that the invariant holds and that $i \leq n$. Then $a[1..i - 1]$ contains the original elements of $a[1..i - 1]$ sorted, and $a[i]$ is the original element $a[i]$. By Lemma 13.4, after line 10 the sub-array $a[1..i]$ contains these i elements, i.e., the original elements of $a[1..i]$, sorted in non-decreasing order, and $a[i + 1..n]$ is unchanged, hence still equal to the original one. Line 11 sets $i \leftarrow i + 1$, so the invariant holds with the new value of i .
- **Termination.** The variable i starts from $2 \leq n + 1$ and increases by 1 at every iteration, and each inner loop terminates (Lemma 13.4). Therefore the outer

loop terminates after exactly $n - 1$ iterations, when $i = n + 1$ for the first time. Substituting $i = n + 1$ in the loop invariant, the sub-array $a[1..n]$, i.e., the entire array, consists of the original elements sorted in non-decreasing order. Hence, the algorithm is correct. □

13.4 Correctness of Binary Search

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

- In the previous two algorithms the loop always runs until its condition becomes false. Binary search can also stop early, with a **return** inside the loop, and the proof must take both exits into account. This is the pseudo-code of the (iterative) **BinarySearch** algorithm.

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

```

- The **loop invariant** of the while loop of BinarySearch is:

Every time the condition at line 4 is evaluated:

(I1) $1 \leq \text{left}$ and $\text{right} \leq n$;

(I2) if m is in a , then m is in $a[\text{left}..\text{right}]$, that is,

$$m \in \{a[1], \dots, a[n]\} \implies m \in \{a[\text{left}], \dots, a[\text{right}]\}.$$

- In words: the algorithm never discards a portion of the array that may contain m . The sub-array $a[\text{left}..\text{right}]$ is called the **search interval** and it contains $\text{right} - \text{left} + 1$ elements (it is empty if $\text{left} > \text{right}$).

Example 13.4: Loop invariant of BinarySearch

Search for $m = 10$ in $a = [3, 9, 11, 12, 12, 16, 19]$ ($n = 7$). The table shows the search interval every time the condition at line 4 is evaluated.

<i>left</i>	<i>right</i>	$a[\text{left}..\text{right}]$	action
1	7	[3, 9, 11, 12, 12, 16, 19]	$\text{mid} = 4, a[4] = 12 > 10 \Rightarrow \text{right} = 3$
1	3	[3, 9, 11]	$\text{mid} = 2, a[2] = 9 < 10 \Rightarrow \text{left} = 3$
3	3	[11]	$\text{mid} = 3, a[3] = 11 > 10 \Rightarrow \text{right} = 2$
3	2	[]	$\text{left} > \text{right} \Rightarrow \text{return } -1$

Since 10 is not in a , the invariant (I2) is trivially true; when the interval becomes empty the algorithm correctly returns -1 .

Observation 13.6

Given an array $a[1..n]$ of n natural numbers sorted in non-decreasing order and a natural number m , the **BinarySearch** algorithm terminates; it returns an index k such that $a[k] = m$ if m is in a , and it returns -1 otherwise.

Proof. We prove the three properties of the loop invariant.

- **Initialization.** Lines 2 and 3 set $\text{left} = 1$ and $\text{right} = n$. Then (I1) holds, and $a[\text{left}..\text{right}] = a[1..n]$ is the whole array, so (I2) holds.
- **Maintenance.** Assume that (I1)–(I2) hold and that $\text{left} \leq \text{right}$. Line 5 computes $\text{mid} = \text{left} + \lfloor (\text{right} - \text{left})/2 \rfloor$; since $0 \leq \lfloor (\text{right} - \text{left})/2 \rfloor \leq \text{right} - \text{left}$, we have

$$1 \leq \text{left} \leq \text{mid} \leq \text{right} \leq n,$$

so $a[\text{mid}]$ is a valid entry of the array. Three cases are possible.

- $a[\text{mid}] = m$: the algorithm returns mid at line 7, which is a correct output.

- $a[mid] < m$: since the array is sorted, for every $k \leq mid$ we have $a[k] \leq a[mid] < m$. Hence m is not in $a[left..mid]$, and if m is in $a[left..right]$ then it is in $a[mid + 1..right]$. Line 10 sets $left \leftarrow mid + 1 \geq 1$, so (I1) and (I2) still hold.
- $a[mid] > m$: since the array is sorted, for every $k \geq mid$ we have $a[k] \geq a[mid] > m$. Hence, if m is in $a[left..right]$, then it is in $a[left..mid - 1]$. Line 13 sets $right \leftarrow mid - 1 \leq n$, so (I1) and (I2) still hold.
- **Termination.** Consider the number of elements of the search interval, $right - left + 1$. In the second case the new number is $right - mid \leq right - left$, in the third case it is $mid - left \leq right - left$: in both cases it decreases by at least 1. Since the loop continues only while $right - left + 1 \geq 1$, the loop terminates after at most n iterations. The algorithm ends in one of two ways:
 - it returns mid with $a[mid] = m$, which is correct;
 - the condition $left \leq right$ is false, i.e., the search interval $a[left..right]$ is empty. Since an empty sub-array does not contain m , by (I2) m is not in a , and the algorithm correctly returns -1 at line 16.

Hence, the algorithm is correct.

□

13.5 Correctness of the Euclidean algorithm

Given two natural numbers $n, m \geq 1$, consider the problem of computing their greatest common divisor $\text{gcd}(n, m)$.

- In the last example the invariant is not a property of a portion of an array, but an equality between two numbers that never changes. This is the pseudo-code of the Euclidean algorithm.

Algorithm: EuclideanAlgorithm

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

Output: $\text{gcd}(n, m)$

```

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

```

- The proof is based on the following property of the greatest common divisor, proved in the chapter «Numerical algorithms»:

$$\gcd(n, m) = \gcd(m, n \bmod m) \quad \text{for every } n \geq 1, m \geq 1.$$

- We also use the convention $\gcd(n, 0) = n$ for $n \geq 1$: every natural number divides 0, so the common divisors of n and 0 are the divisors of n , and the greatest of them is n .
- We denote by n_0 and m_0 the values of n and m given in input. The **loop invariant** of the while loop is:

Every time the condition at line 2 is evaluated, $n \geq 1$, $m \geq 0$, and

$$\gcd(n, m) = \gcd(n_0, m_0).$$

Example 13.5: Loop invariant of the Euclidean algorithm

Consider $n_0 = 252$ and $m_0 = 105$. The table shows the values of n and m every time the condition at line 2 is evaluated:

n	m	$\gcd(n, m)$
252	105	21
105	42	21
42	21	21
21	0	21 (loop ends)

The value $\gcd(n, m) = 21$ never changes, and when $m = 0$ the algorithm returns $n = 21 = \gcd(252, 105)$.

Observation 13.7

Given two natural numbers $n_0, m_0 \geq 1$, the **EuclideanAlgorithm** terminates and returns $\gcd(n_0, m_0)$.

Proof. We prove the three properties of the loop invariant.

- **Initialization.** Before the first iteration $n = n_0 \geq 1$ and $m = m_0 \geq 1$, so trivially $\gcd(n, m) = \gcd(n_0, m_0)$.
- **Maintenance.** Assume that the invariant holds and that $m \neq 0$, i.e., $m \geq 1$. Line 3 computes $r = n \bmod m$, with $0 \leq r < m$. Lines 4–5 replace the pair (n, m) with (m, r) . The new value of n is $m \geq 1$, the new value of m is $r \geq 0$, and, since $n, m \geq 1$,

$$\gcd(m, r) = \gcd(m, n \bmod m) = \gcd(n, m) = \gcd(n_0, m_0).$$

Hence the invariant holds before the next iteration.

- **Termination.** At every iteration the value of m is replaced by $r < m$, so m is a natural number that strictly decreases. Therefore the loop terminates after at most m_0 iterations, when $m = 0$. Substituting $m = 0$ in the loop invariant we obtain

$$\gcd(n_0, m_0) = \gcd(n, 0) = n,$$

which is the value returned at line 7. Hence, the algorithm is correct. □

- The proof does not use the assumption $n_0 \geq m_0$: if $n_0 < m_0$, then $n_0 \bmod m_0 = n_0$ and the first iteration simply swaps the two numbers.
- The bound “at most m_0 iterations” is enough to prove termination, but it is very pessimistic: the number of iterations is actually $O(\log_2 m_0)$ (in the example, 3 iterations for $m_0 = 105$).

13.6 Correctness of recursive algorithms

- Loop invariants are the tool to prove the correctness of **iterative** algorithms, i.e., algorithms based on loops.
- A **recursive** algorithm solves an instance of a problem by calling itself on one or more smaller instances. Its correctness is proved directly by **mathematical induction** on the size of the instance:
 - **base case:** the algorithm returns the correct output on the smallest instances, which are solved without recursive calls;
 - **inductive step:** under the inductive hypothesis that the recursive calls return the correct output on the smaller instances, the algorithm combines their results into the correct output for the current instance.
- Termination follows from the fact that every recursive call is made on a strictly smaller instance, so the base case is eventually reached.

The correctness of recursive algorithms is proved by induction on the size of the input. This technique is applied together with the study of recursive algorithms.

Key points of the chapter

- An algorithm is correct if, for every input instance, it terminates and returns the correct output; testing can only show that an algorithm is wrong.
- The correctness of a loop is proved with a loop invariant: initialization, maintenance and termination.
- The invariant, together with the negated condition of the loop, gives the output: $i = n + 1$ for Find Maximum and Insertion Sort, an empty search interval for Binary

Search, $m = 0$ for the Euclidean algorithm.

- Nested loops need one invariant for each loop (Insertion Sort); a **return** inside the loop is a second exit that must be proved correct (Binary Search).
- Recursive algorithms are proved correct by induction on the size of the input.