Computational Complexity of Algorithms

Complexity of Algorithms

- Computers work fast but not infinitely fast. Execution of each instruction takes some (very short) time.
- The same problem can be solved by several different algorithms. The time of a computation (determined mostly by the number of executed instructions) can be different for different algorithms.
- We would like to compare different algorithms and choose a better one.
- We can implement the algorithms and then measure the time of their computation. By this we find out how long the computation takes on particular data on which we test the algorithm.
- We would like to have a more precise idea how long the computation takes on all possible input data.

Complexity of Algorithms

Consider some particular machine executing an algorithm — e.g., RAM, Turing machine, ...

We will assume that for the given machine $\mathcal{M}$ we have defined the following two functions for each input x from set of all inputs In:

- $time_{\mathcal{M}} : In \to \mathbb{N}$ — represents the running time of the machine $\mathcal{M}$ on an input x
- $space_{\mathcal{M}} : In \to \mathbb{N}$ — represents an amount of memory used by the machine $\mathcal{M}$ in a computation on an input x

Remark: We assume that a computation of the machine $\mathcal{M}$ halts after a finite number of steps for each input x.

Size of Input

For different input data the program performs a different number of instructions.

If we want to analyze somehow the number of performed instructions, it is useful to introduce the notion of the **size of an input**.

Typically, the size of an input is a number specifying how "big" is the given instance (a bigger number means a bigger instance).

Remark: We can define the size of an input as we like depending on what is useful for our analysis.

The size of an input is not strictly determinable but there are usually some natural choices based on the nature of the problem.

Size of Input

Examples:

- For the problem "Sorting", where the input is a sequence of numbers $a_1, a_2, \ldots, a_n$ and the output the same sequence sorted, we can take n as the size of the input.

- For the problem "Primality" where the input is a natural number x and where the question is whether x is a prime, we can take the number of bits of the number x as the size of the input.

  (The other possibility is to take directly the value x as the size of the input.)

Size of Input

Sometimes it is useful to describe the size of an input with several numbers.

For example for problems where the input is a graph, we can define the size of the input as a pair of numbers n, m where:

- n – the number of nodes of the graph
- m – the number of edges of the graph

Remark: The other possibility is to define the size of the input as one number $n + m$.

Size of Input

In general, we can define the size of an input for an arbitrary problem as follows:

- When the input is a word over some alphabet Σ:
  the length of word w
- When the input as a sequence of bits (i.e., a word over $\{0, 1\}$):
  the number of bits in this sequence
- When the input is a natural number x:
  the number of bits in the binary representation of x

Time Complexity

We want to analyze a particular algorithm (its particular implementation).

We want to know how many steps the algorithm performs when it gets an input of size $0, 1, 2, 3, 4, \ldots$.

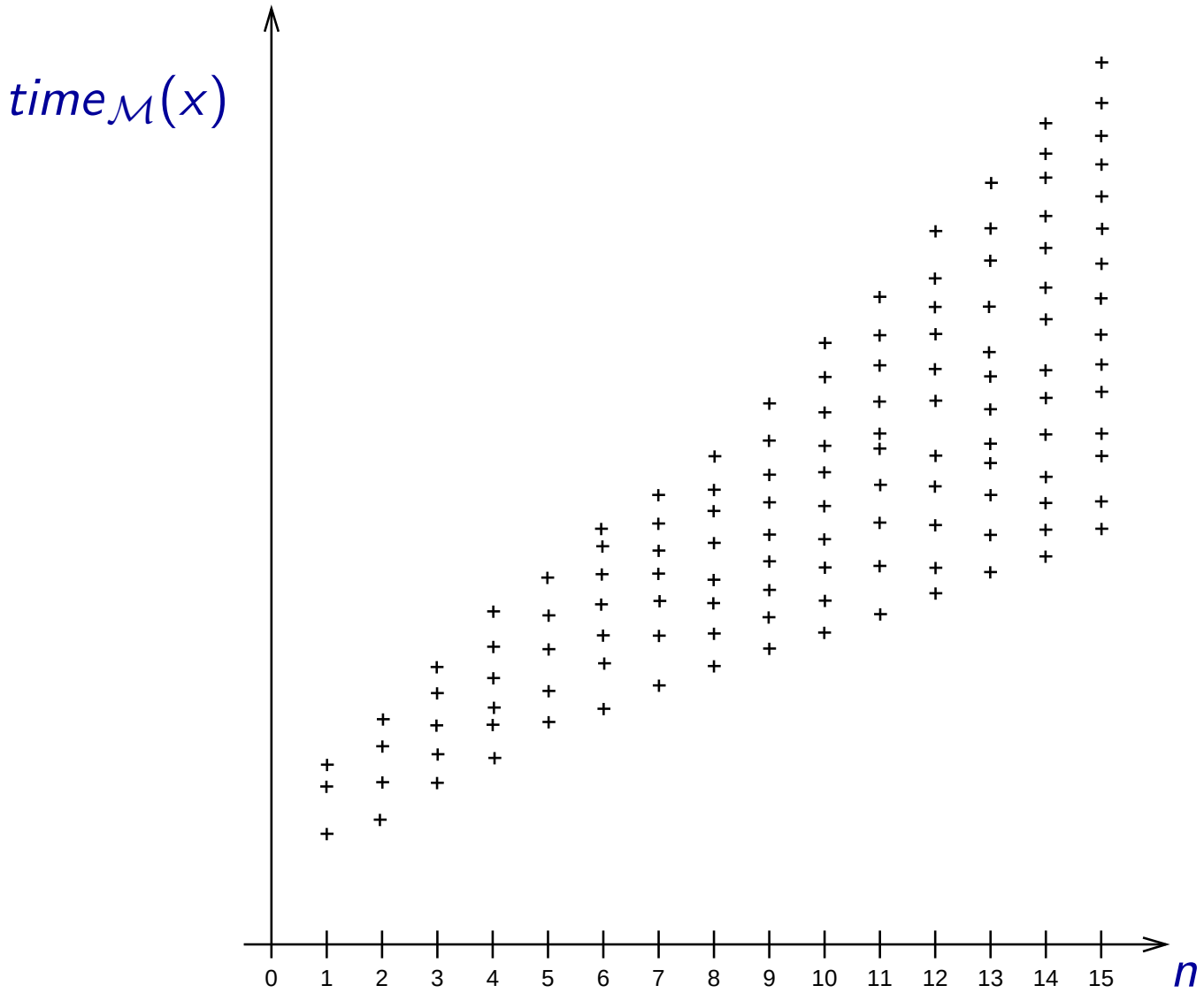
It is obvious that even for inputs of the same size the number of performed steps can be different.

Let us denote the size of input $x \in In$ as $size(x)$.

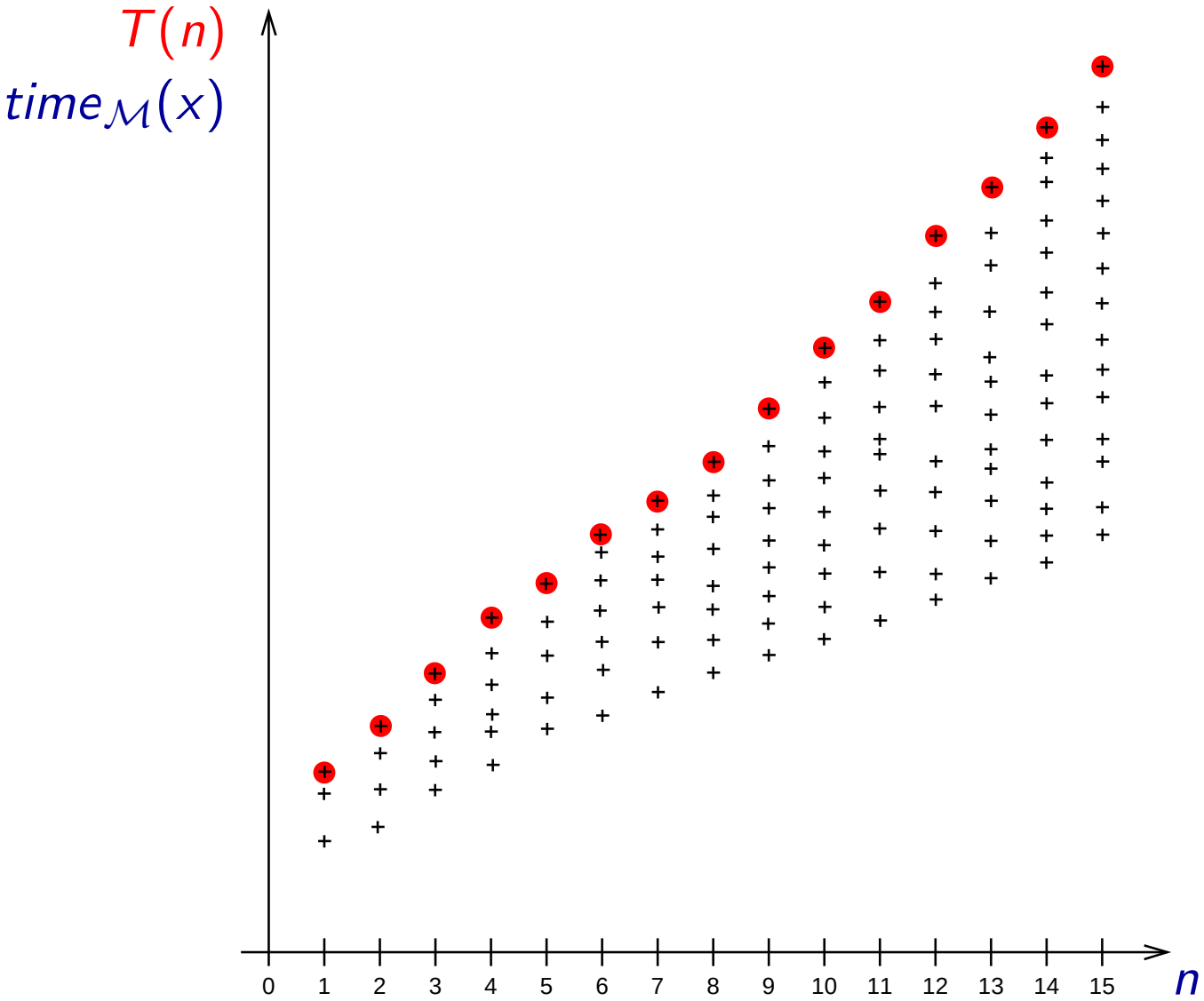
Now we define a function $T : \mathbb{N} \to \mathbb{N}$ such that for $n \in \mathbb{N}$ is

$$T(n) = \max\{\, time_{\mathcal{M}}(x) \mid x \in In,\, size(x) = n \,\}$$

Time Complexity in the Worst Case



Time Complexity in the Worst Case



Time Complexity in the Worst Case

Such function $T(n)$ (i.e., a function that for the given algorithm and the given definition of the size of an input assignes to every natural number n the maximal number of instructions performed by the algorithm if it obtains an input of size n) is called the **time complexity of the algorithm in the worst case**.

$$T(n) = \max\{\, time_{\mathcal{M}}(x) \mid x \in In,\, size(x) = n \,\}$$

Analogously, we can define **space complexity** of the algorithm in the worst case as a function $S(n)$ where:

$$S(n) = \max\{\, space_{\mathcal{M}}(x) \mid x \in In,\, size(x) = n \,\}$$

Time Complexity in an Average Case

Sometimes it make sense to analyze the time complexity **in an average case**.

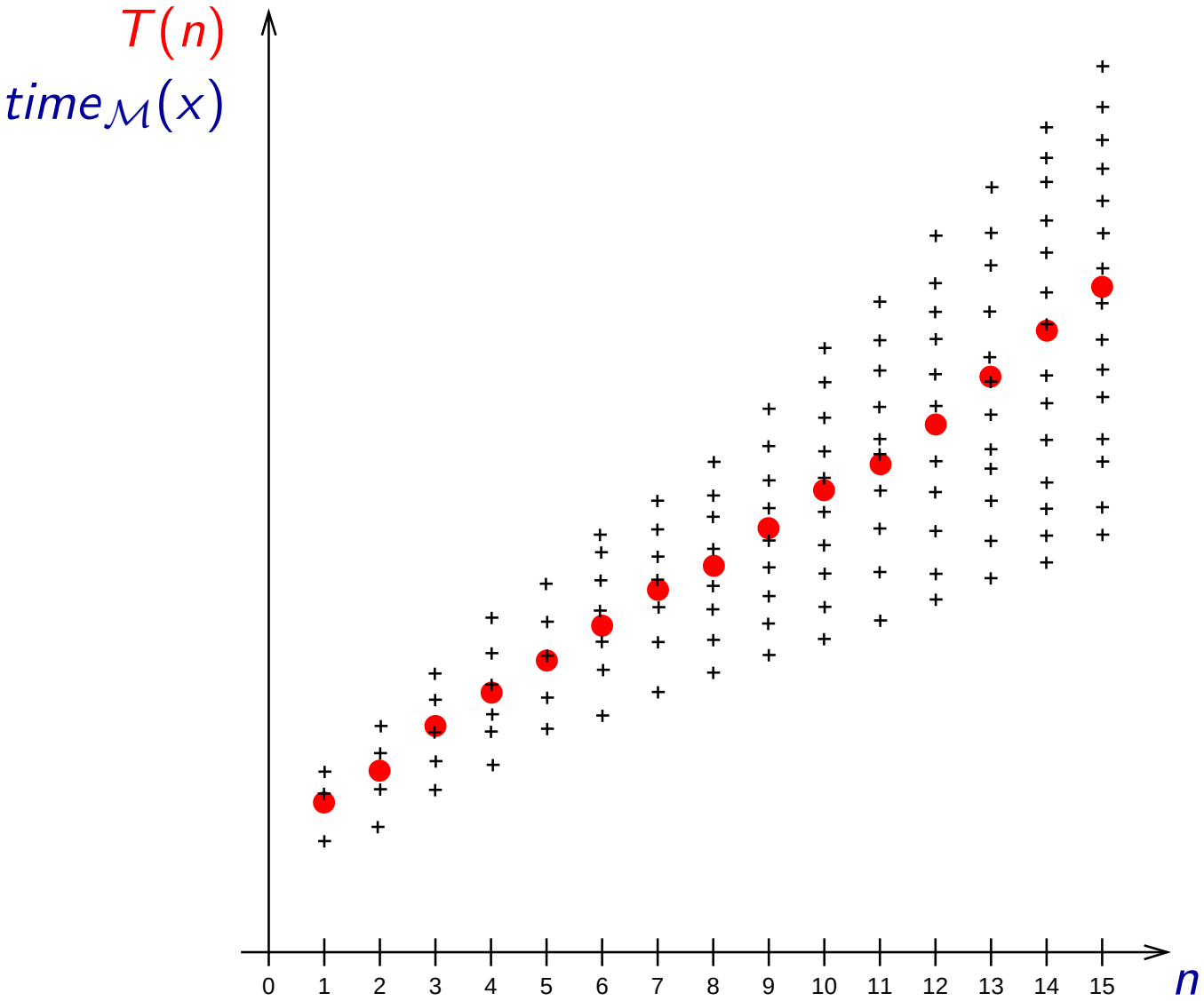
In this case, we do not define $T(n)$ as the maximum but as the arithmetic mean of the set

$$\{ time_{\mathcal{M}}(x) \mid x \in In, size(x) = n \}$$

- It is usually more difficult to determine the time complexity in an average case than to determine the time complexity in the worst case.
- Often, these two function are not very different but sometimes the difference is significant.

Remark: It usually makes no sense to analyze the time complexity in the best case.

Time Complexity in an Average Case



Growth of Functions

A program works on an input of size n.
Let us assume that for an input of size n, the program performs $T(n)$ operations and that an execution of one operation takes $1\,\mu s$ ($10^{-6}\,s$).

	n							
$T(n)$	20	40	60	80	100	200	500	1000
n	$20\,\mu s$	$40\,\mu s$	$60\,\mu s$	$80\,\mu s$	0.1 ms	0.2 ms	0.5 ms	1 ms
$n \log n$	$86\,\mu s$	0.213 ms	0.354 ms	0.506 ms	0.664 ms	1.528 ms	4.48 ms	9.96 ms
n^2	0.4 ms	1.6 ms	3.6 ms	6.4 ms	10 ms	40 ms	0.25 s	1 s
n^3	8 ms	64 ms	0.216 s	0.512 s	1 s	8 s	125 s	16.7 min.
n^4	0.16 s	2.56 s	12.96 s	42 s	100 s	26.6 min.	17.36 hours	11.57 days
2^n	1.05 s	12.75 days	36560 years	$38.3 \cdot 10^9$ years	$40.1 \cdot 10^{15}$ years	$50 \cdot 10^{45}$ years	$10.4 \cdot 10^{136}$ years	–
$n!$	77147 years	$2.59 \cdot 10^{34}$ years	$2.64 \cdot 10^{68}$ years	$2.27 \cdot 10^{105}$ years	$2.96 \cdot 10^{144}$ years	–	–	–

Growth of Functions

Let us consider 3 algorithms with complexities $T_1(n) = n$, $T_2(n) = n^3$, $T_3(n) = 2^n$. Our computer can do in a reasonable time (the time we are willing to wait) 10^{12} steps.

Complexity	Input size
$T_1(n) = n$	10^{12}
$T_2(n) = n^3$	10^4
$T_3(n) = 2^n$	40

Growth of Functions

Let us consider 3 algorithms with complexities $T_1(n) = n$, $T_2(n) = n^3$, $T_3(n) = 2^n$. Our computer can do in a reasonable time (the time we are willing to wait) 10^{12} steps.

Complexity	Input size
$T_1(n) = n$	10^{12}
$T_2(n) = n^3$	10^4
$T_3(n) = 2^n$	40

Now we speed up our computer 1000 times, meaning it can do 10^{15} steps.

Complexity	Input size	Growth
$T_1(n) = n$	10^{15}	$1000\times$
$T_2(n) = n^3$	10^5	$10\times$
$T_3(n) = 2^n$	50	$+10$

Asymptotic Notation

- It is usually quite difficult to express the complexity exactly.
- The exact complexity depends on the used model of computation and on the particular implementation (on details of this implementation).
- We are interested in the complexity for big inputs. For small inputs usually even nonefficient algorithms work fast.
- We usually do not need to know the exact number of performed instructions and we will be satisfied with some estimation of how fast this number grows when the size of an input grows.
- So we use the so called **asymptotic notation**, which allows us to ignore unimportant details and to estimate approximately how fast the given function grows. This simplifies the analysis considerably.

Asymptotic Notation

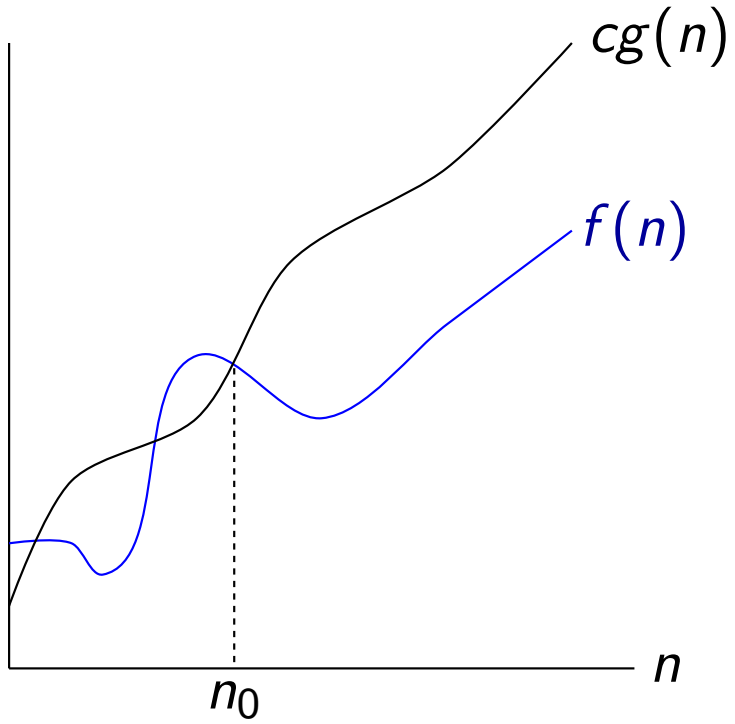
Let us take an arbitrary function $g : \mathbb{N} \to \mathbb{N}$. Expressions $O(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$, and $\omega(g)$ denote **sets of functions** of the type $\mathbb{N} \to \mathbb{N}$, where:

- $O(g)$ – the set of all functions that grow at most as fast as g
- $\Omega(g)$ – the set of all functions that grow at least as fast as g
- $\Theta(g)$ – the set of all functions that grow as fast as g
- $o(g)$ – the set of all fuctions that grow slower than function g
- $\omega(g)$ – the set of all functions that grow faster than function g

Remark: These are not definitions! The definitions will follow on the next slides.

- O – big "O"
- Ω – uppercase Greek letter "omega"
- Θ – uppercase Greek letter "theta"
- o – small "o"
- ω – small "omega"

Asymptotic Notation – Symbol O

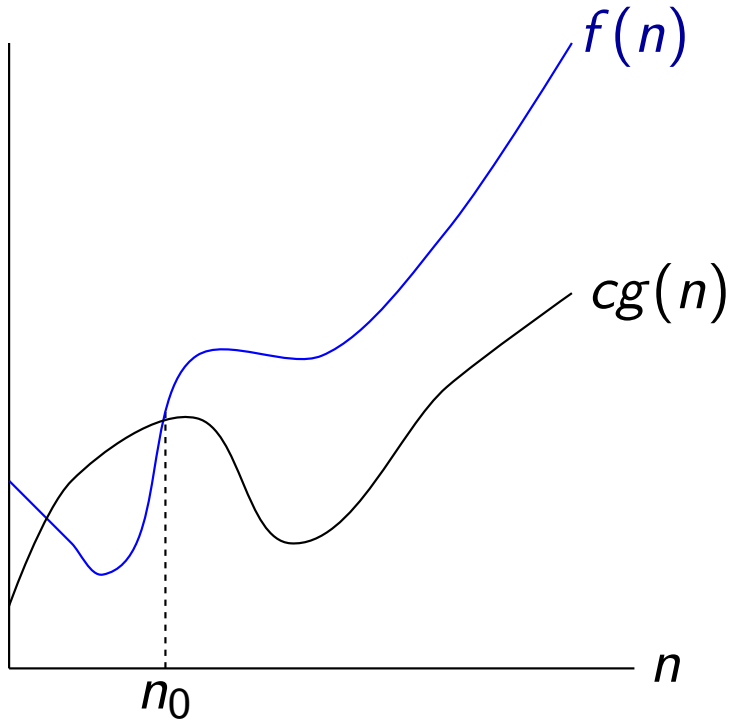


Definition

Let us consider an arbitrary function $g : \mathbb{N} \to \mathbb{N}$. For a function $f : \mathbb{N} \to \mathbb{N}$ we have $f \in O(g)$ iff

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0) : f(n) \leq c\, g(n).$$

Asymptotic Notation – Symbol Ω

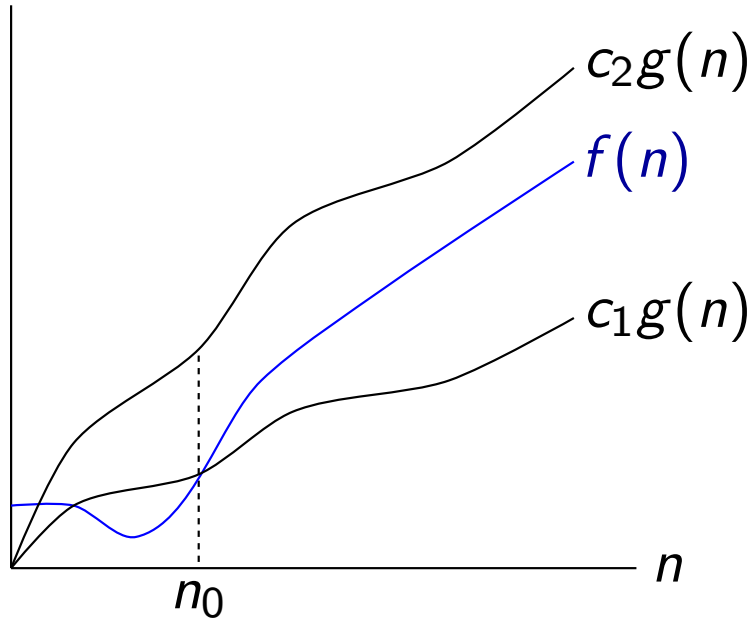


Definition

Let us consider an arbitrary function $g : \mathbb{N} \to \mathbb{N}$. For a function $f : \mathbb{N} \to \mathbb{N}$ we have $f \in \Omega(g)$ iff

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0) : c\, g(n) \leq f(n).$$

Asymptotic Notation – Symbol Θ



Definition

Let us consider an arbitrary function $g : \mathbb{N} \to \mathbb{N}$. For a function $f : \mathbb{N} \to \mathbb{N}$ we have $f \in \Theta(g)$ iff

$$f \in O(g) \quad \text{and} \quad f \in \Omega(g).$$

Asymptotic Notation – Symbols o and ω

Definition

Let us consider an arbitrary function $g : \mathbb{N} \to \mathbb{N}$. For a function $f : \mathbb{N} \to \mathbb{N}$ we have $f \in o(g)$ iff

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

Definition

Let us consider an arbitrary function $g : \mathbb{N} \to \mathbb{N}$. For a function $f : \mathbb{N} \to \mathbb{N}$ we have $f \in \omega(g)$ iff

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

Asymptotic Notation

For simplicity, we consider only functions of type $\mathbb{N} \to \mathbb{N}$ in the previous definitions.

In fact, these definitions could be extended to all **asymptotically nonnegative** functions of type $\mathbb{R}_+ \to \mathbb{R}$, which moreover can be undefined on some finite subinterval of its domain.

Function $f : \mathbb{R}_+ \to \mathbb{R}$ is **asymptotically nonnegative** if it satisfies:

$$(\exists n_0 \geq 0)(\forall n \geq n_0)(f(n) \geq 0)$$

Remark: For $n < n_0$, the value of $f(n)$ can be undefined.

$\mathbb{R}_+ = \{x \in \mathbb{R} \mid x \geq 0\}$

Asymptotic Notation

- There are pairs of functions $f, g : \mathbb{N} \to \mathbb{N}$ such that

$$f \notin O(g) \qquad \text{and} \qquad g \notin O(f),$$

for example

$$f(n) = n \qquad g(n) = \begin{cases} n^2 & \text{if } n \bmod 2 = 0 \\ \lceil \log_2 n \rceil & \text{otherwise} \end{cases}.$$

Asymptotic Notation

- $O(1)$ denotes the set of all **bounded** functions, i.e., functions whose function values can be bounded from above by a constant.

- A function f is called:

  logarithmic, if $f(n) \in \Theta(\log n)$
  linear, if $f(n) \in \Theta(n)$
  quadratic, if $f(n) \in \Theta(n^2)$
  cubic, if $f(n) \in \Theta(n^3)$
  polynomial, if $f(n) \in O(n^k)$ for some $k > 0$
  exponential, if $f(n) \in O(c^{n^k})$ for some $c > 1$ and $k > 0$

- Exponential functions are often written in the form $2^{O(n^k)}$ when the asymptotic notation is used, since then we do not need to consider different bases.

Asymptotic Notation

As mentioned before, expressions $O(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$, and $\omega(g)$ denote certain sets of functions.

In some texts, these expressions are sometimes used with a slightly different meaning:

- an expression $O(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$ or $\omega(g)$ does not represent the corresponding set of functions but **some** function from this set.

This convention is often used in equations and inequations.

Example: $3n^3 + 5n^2 - 11n + 2 = 3n^3 + O(n^2)$

When using this convention, we can for example write $f = O(g)$ instead of $f \in O(g)$.

Complexity of Algorithms

Let us say we would like to analyze the time complexity $T(n)$ of some algorithm consisting of instructions $I_1, I_2, \ldots, I_k$:

- If $m_1, m_2, \ldots, m_k$ are the number of executions of individual instructions for some input x (i.e., the instruction I_i is performed m_i times for the input x), then the total number of executed instructions for input x is

$$m_1 + m_2 + \cdots + m_k.$$

- Let us consider functions $t_1, t_2, \ldots, t_k$, where $t_i : \mathbb{N} \to \mathbb{N}$, and where $t_i(n)$ is the maximum of numbers of executions of instruction I_i for all inputs of size n.
- Obviously, $T \in \Omega(t_i)$ for any function t_i.
- It is also obvious that $T \in O(t_1 + t_2 + \cdots + t_k)$.

Complexity of Algorithms

- Let us recall that if $f \in O(g)$ then $f + g \in O(g)$.
- If there is a function t_i such that for all t_j, where $j \neq i$, we have $t_j \in O(t_i)$, then
$$T \in O(t_i).$$
- This means that in an analysis of the time complexity $T(n)$, we can restrict our attention to the number of executions of the instruction that is performed most frequently (and which is performed at most $t_i(n)$ times for an input of size n), since we have
$$T \in \Theta(t_i).$$

Complexity of Algorithms

Let us try to analyze the time complexity of the following algorithm:

Algorithm 1: Insertion sort

```
INSERTION-SORT (A, n):
    for j := 1 to n − 1 do
        x := A[j]
        i := j − 1
        while i ≥ 0 and A[i] > x do
            A[i + 1] := A[i]
            i := i − 1
        end
        A[i + 1] := x
    end
```

I.e., we want to find a function $T(n)$ such that the time complexity of the algorithm INSERTION-SORT in the worst case is in $\Theta(T(n))$.

Complexity of Algorithms

Let us consider inputs of size n:

- The outer cycle **for** is performed at most $n-1$ times.
- The inner cycle **while** is performed at most j times for a given value j.
- There are inputs such that the cycle **while** is performed exactly j times for each value j from 1 to $(n-1)$.
- So in the worst case, the cycle **while** is performed exactly m times, where

$$m = 1 + 2 + \cdots + (n-1) = (1 + (n-1)) \cdot \tfrac{n-1}{2} = \tfrac{1}{2}n^2 - \tfrac{1}{2}n$$

- This means that the total running time of the algorithm INSERTION-SORT in the worst case is $\Theta(n^2)$.

Complexity of Algorithms

In the previous case, we accurately computed the total number of executions of the cycle **while**.

This is not always possible in general, or it can be quite complicated. It is also not necessary, if we only want an asymptotic estimation.

Complexity of Algorithms

For example, if we were not able to compute the sum of the arithmetic progression, we could proceed as follows:

- The outer cycle **for** is not performed more than n times and the inner cycle **while** is performed at most n times in each iteration of the outer cycle.

  So we have $T \in O(n^2)$.

- For some inputs, the cycle **while** is performed at least $\lceil n/2 \rceil$ times in the last $\lfloor n/2 \rfloor$ iterations of the cycle **for**.

  So the cycle **while** is performed at least $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ times for some inputs.

  $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \geq (n/2 - 1) \cdot (n/2) = \frac{1}{4}n^2 - \frac{1}{2}n$

  This implies $T \in \Omega(n^2)$.

Complexity of Algorithms

When we use asymptotic estimations of the complexity of algorithms, we should be aware of some issues:

- Asymptotic estimations describe only how the running time grows with the growing size of input instance.
- They do not say anything about exact running time. Some big constants can be hidden in the asymptotic notation.
- An algorithm with better asymptotic complexity than some other algorithm can be in reality faster only for very big inputs.
- We usually analyze the time complexity in the worst case. For some algorithms, the running time in the worst case can be much higher than the running time on "typical" instances.

Complexity of Algorithms

- This can be illustrated on algorithms for sorting.

Algorithm	Worst-case	Average-case
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$

- Quicksort has a worse asymptotic complexity in the worst case than Heapsort and the same asymptotic complexity in an average case but it is usually faster in practice.

Space Complexity of Algorithms

- So far we have considered only the time necessary for a computation
- Sometimes the size of the memory necessary for the computation is more critical.

Let us recall that for a machine $\mathcal{M}$, the function $space_{\mathcal{M}}(x)$ gives a value repsenting a amount of memory used by the machine $\mathcal{M}$ in a computation on input x.

Definition

For a given machine $\mathcal{M}$, the **space complexity** of the machine $\mathcal{M}$ is the function $S : \mathbb{N} \to \mathbb{N}$ defined as

$$S(n) = \max\{\, space_{\mathcal{M}}(x) \mid x \in In, size(x) = n \,\}$$

Space Complexity of Algorithms

- There can be two algorithms for a particular problem such that one of them has a smaller time complexity and the other a smaller space complexity.
- If the time comlexity of a given algorithm is in $O(f(n))$ then the space complexity of the algorithm is also in $O(f(n))$.