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 that assign to each input x from set of all inputs In:

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

Remark: This is not yes the time and space complexity of an algorithm executed by a given machine $\mathcal{M}$.

Complexity of algorithms — Turing Machines

For example, in the case of **Turing machines**, the following could be analyzed:

- the **number of steps** that the given machine performs in the computation over a given word

  — this value represents the running time of the computation

- the **number of cells of the tape** visited by the given machine during the computation oven a given word

  — this value represents the amount of the used memory

Complexity of algorithms — Turing Machines

Formally, these notions can be defined as follows
(we assume a given Turing machine $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$):

- Function

$$time_{\mathcal{M}} : \Sigma^* \to \mathbb{N} \cup \{\infty\}$$

For $w \in \Sigma^*$, the value $time_{\mathcal{M}}(w)$ specifies the number of steps the machine $\mathcal{M}$ performs in a computation over the word w.

I.e., if this computation is finite and looks as follows

$$\alpha_0 \longrightarrow \alpha_1 \longrightarrow \alpha_2 \longrightarrow \alpha_3 \longrightarrow \quad \cdots \quad \longrightarrow \alpha_{t-1} \longrightarrow \alpha_t$$

where α_t is a final configuration, then $time_{\mathcal{M}}(w) = t$.

In the case of an infinite computation

$$\alpha_0 \longrightarrow \alpha_1 \longrightarrow \alpha_2 \longrightarrow \alpha_3 \longrightarrow \quad \cdots$$

we have $time_{\mathcal{M}}(w) = \infty$.

Complexity of algorithms — Turing Machines

- Function

$$space_{\mathcal{M}} : \Sigma^* \rightarrow \mathbb{N} \cup \{\infty\}$$

For $w \in \Sigma^*$, the value $space_{\mathcal{M}}(w)$ specifies the number of cells that the machine $\mathcal{M}$ visits during a computation over an the input w.

Remark: It is obvious that in the case of a finite computation also the value $space_{\mathcal{M}}(w)$ is finite.

In the case of an infinite computation, the value $space_{\mathcal{M}}(w)$ can be finite or infinite.

Complexity of algorithms — RAMs

A running time of a RAM can be computed in two different ways:

- **uniform cost** — the number of executed instructions
- **logarithmic cost** — the sum of cost of individual instructions; the cost of one instruction depends on the number of bits of values used in the given instruction.

  For example:

  - The cost of execution of instructions for addition and subtruction is the sum of the number of bits of their operands.
  - The cost of execution of instructions for multiplication and division is the product of the number of bits of their operands.
  - The cost of instructions accessing memory (load, store) is the sum of the number of bits of an address and the number of bits of a number that is read or written.

  Remark: When counting the number of bits of a given number, it is assumed that value 0 has 1 bit.

Complexity of algorithms — RAMs

Also the amount of memory used during a computation by a RAM can be computed in two different ways:

- **uniform cost** — the number of memory cells used, i.e., the number of cells, which were read or written to during the computation.
- **logarithmic cost** — the maximal number of bits of memory that were used during the computation.

  The number of bits includes both the number of bits of used cells and the number of bits of their addresses.

Complexity of algorithms — RAMs

The uniform cost realistically represents the amount of a work done during the execution and the amount of used memory only in those cases where the values stored in memory cells are “small”, i.e., if in a real implemantation for “reasonably” big inputs it would be possible to represents them for example as 32-bit or 64-bit numbers.

Complexity of algorithms — RAMs

- For those machines that do not have an instruction for multiplication, it can be easily shown that each instruction can produce a number that has at most one bit more than the bigger (in absolute value) of its operands.

  For such machines, after t steps of computation, each cell contains a number that has at most $t + m + n$ bits where m is the number of bits of the biggist constant occurring in the program and n is the biggest number of bits of a number in the input.

- If the machine has an instruction for the multiplication then after t steps, some memory cell can contain a number that has approximately 2^t bits.

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 $0, 1, 2, 3, 4, \ldots$.

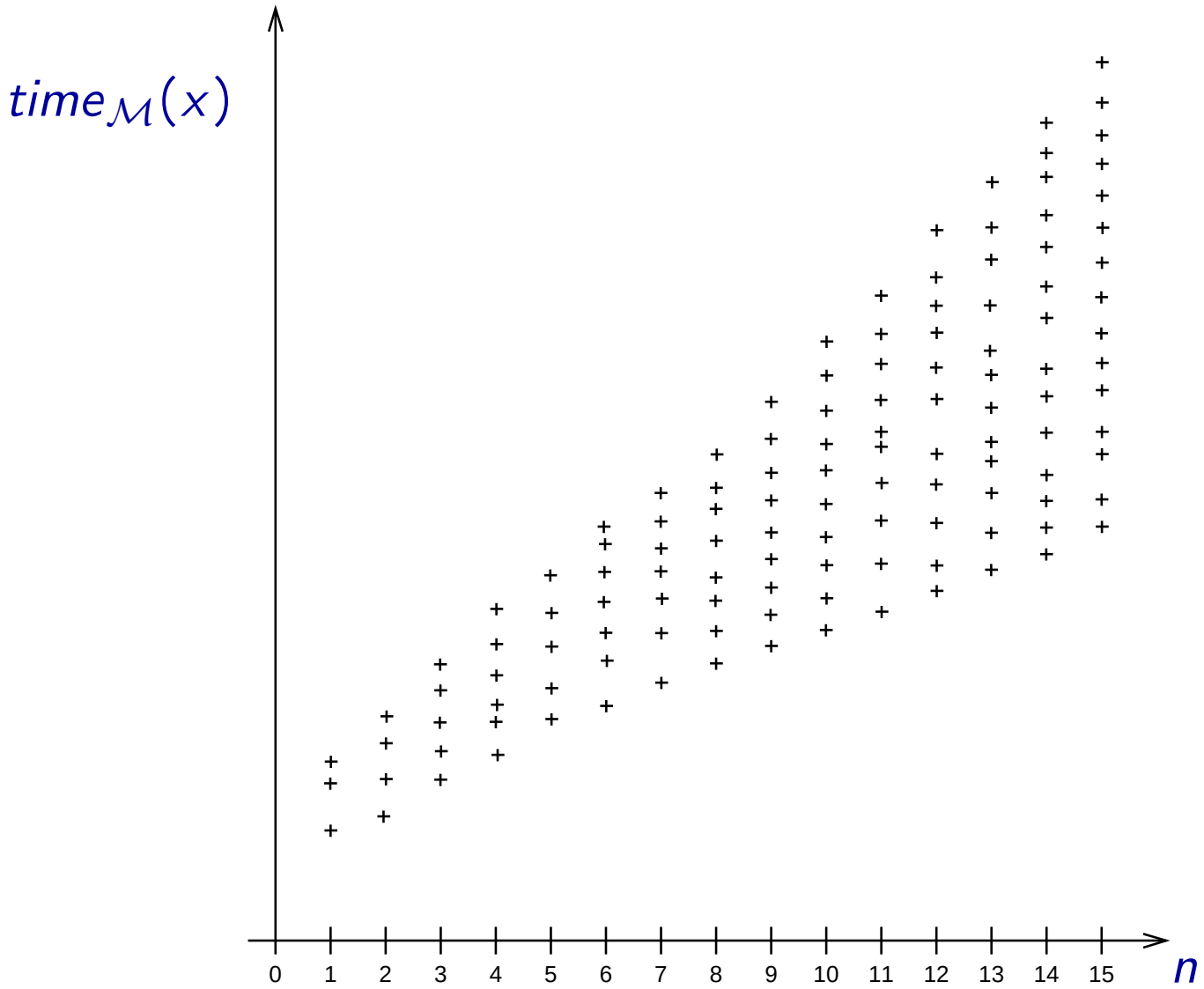
It is obvious that even for inputs of the same size the number of performed steps can be different.

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

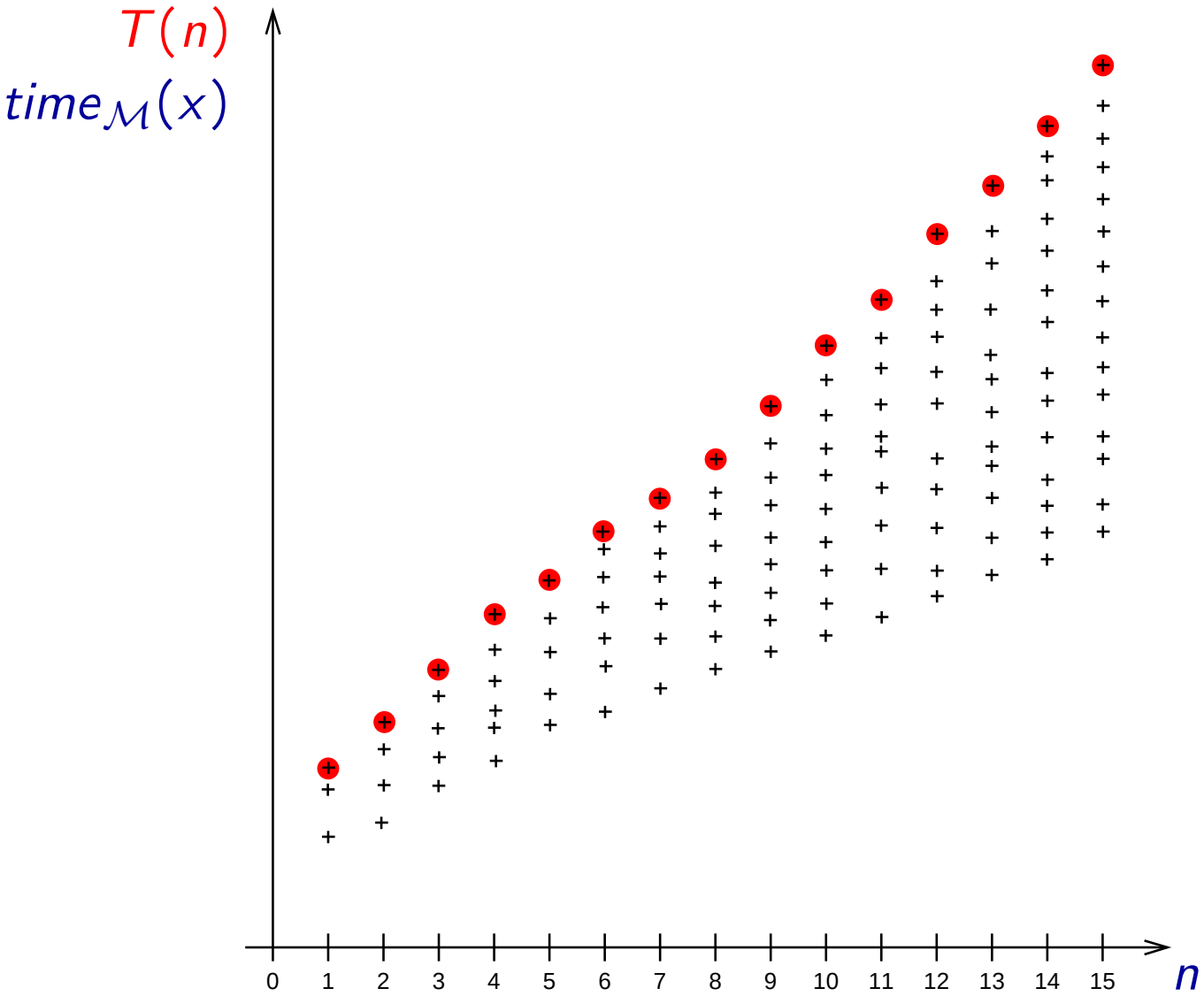
Now we define a function $T : \mathbb{N} \to \mathbb{N}$ such that for $n \in \mathbb{N}$ is

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

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**.

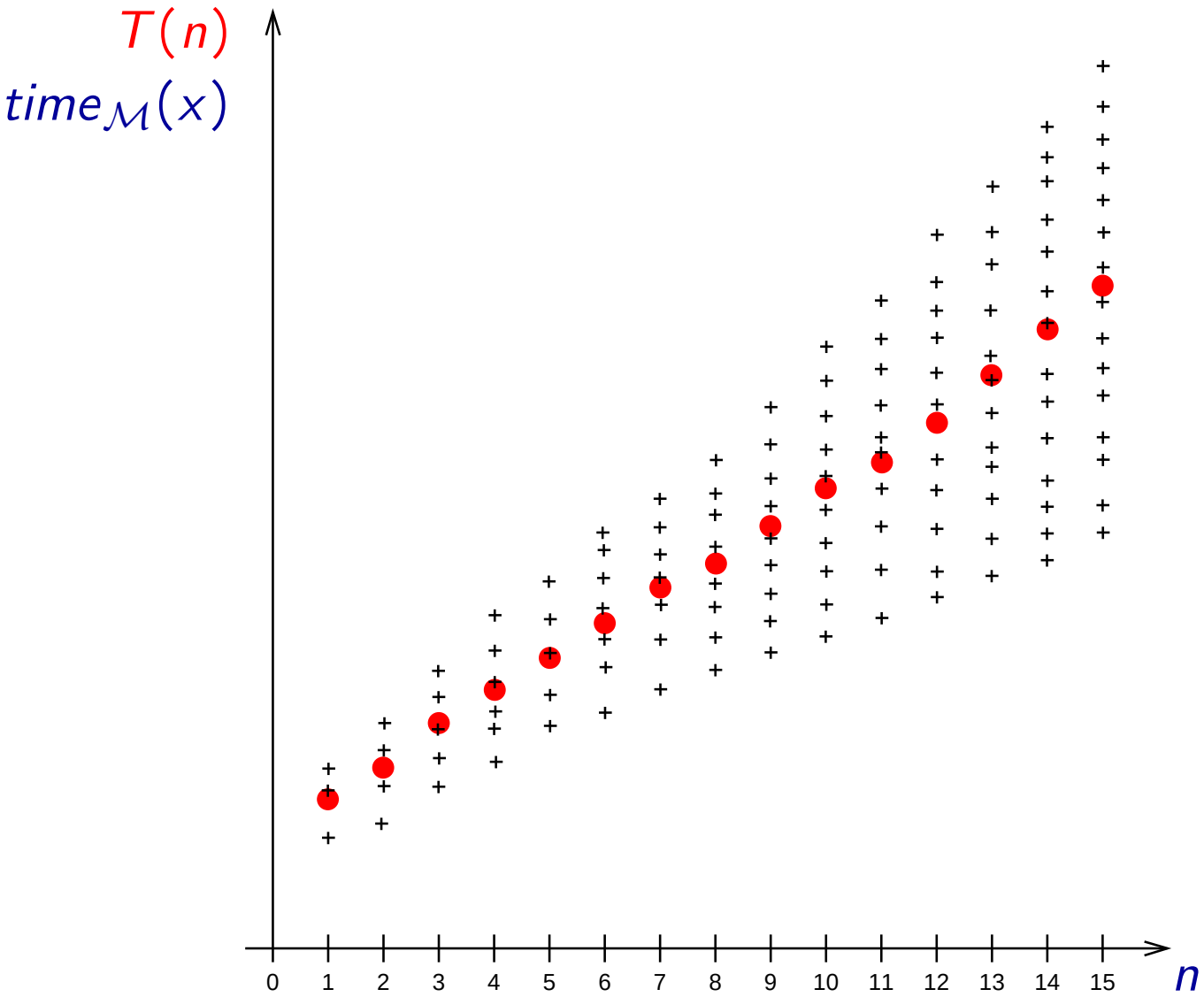
In this case, we do not define $T(n)$ as the maximum but as the arithmetic mean of the set

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

- 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 μs	40 μs	60 μs	80 μs	0.1 ms	0.2 ms	0.5 ms	1 ms
$n \log n$	86 μs	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

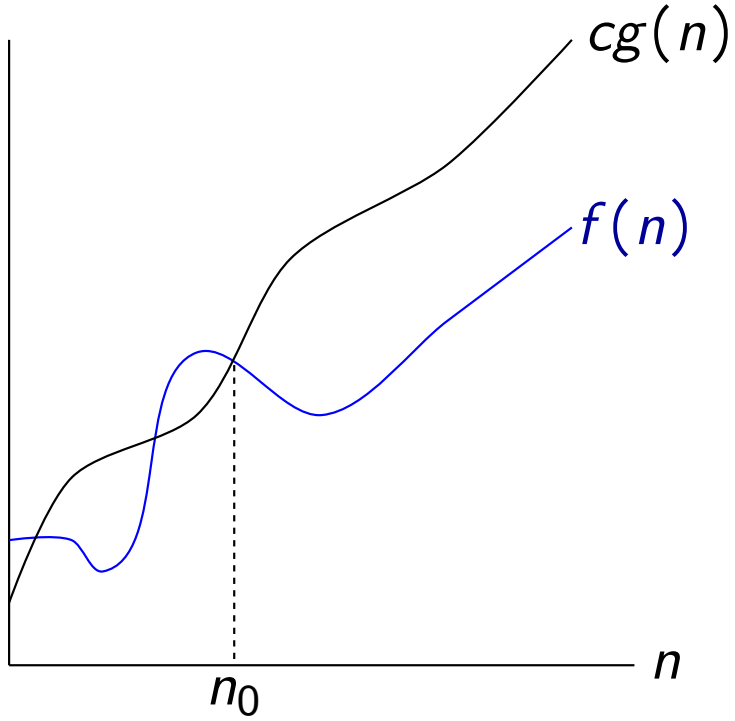
Let us take an arbitrary function $g : \mathbb{N} \to \mathbb{N}$. Expressions $\mathcal{O}(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$, and $\omega(g)$ denote **sets of functions** of the type $\mathbb{N} \to \mathbb{N}$, where:

- $\mathcal{O}(g)$ – 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.

- $\mathcal{O}$ – big "O"
- Ω – uppercase Greek letter "omega"
- Θ – uppercase Greek letter "theta"
- o – small "o"
- ω – small "omega"

Asymptotic Notation – Symbol $\mathcal{O}$

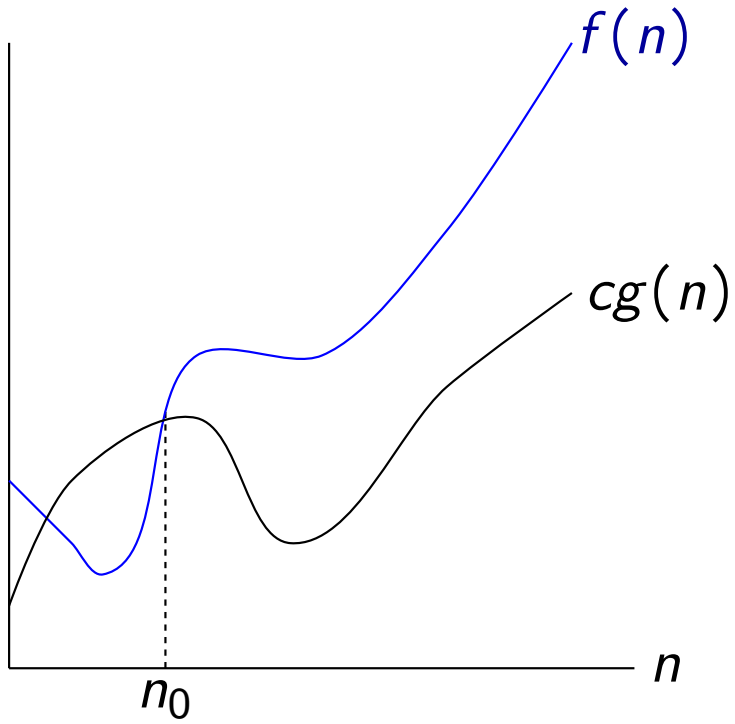


Definition

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

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

Asymptotic Notation – Symbol Ω

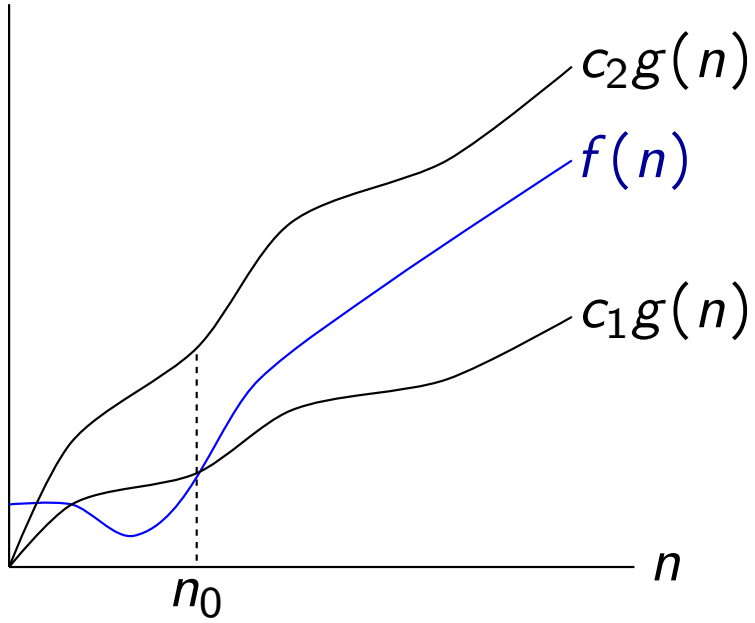


Definition

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

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

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

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

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

- $\mathcal{O}(1)$ 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 $f(n) \in \mathcal{O}(n^k)$ for some $k > 0$
  exponential, if $f(n) \in \mathcal{O}(c^{n^k})$ for some $c > 1$ and $k > 0$

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

Asymptotic Notation

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

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

- an expression $\mathcal{O}(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$ 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: $3n^3 + 5n^2 - 11n + 2 = 3n^3 + \mathcal{O}(n^2)$

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

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 $T \in \mathcal{O}(t_1 + t_2 + \cdots + t_k)$.

Complexity of Algorithms

- Let us recall that if $f \in \mathcal{O}(g)$ then $f + g \in \mathcal{O}(g)$.
- If there is a function t_i such that for all t_j, where $j \neq i$, we have $t_j \in \mathcal{O}(t_i)$, then
$$T \in \mathcal{O}(t_i).$$
- 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: 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
        A[i + 1] := x
```

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

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

- 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 $T \in \mathcal{O}(n^2)$.

- 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 $\mathcal{O}(f(n))$ then the space complexity of the algorithm is also in $\mathcal{O}(f(n))$.

Complexity of Algorithms

Some typical values of the size of an input n, for which an algorithm with the given time complexity usually computes the output on a "common PC" within a fraction of a second or at most in seconds.

(Of course, this depends on particular details. Moreover, it is assumed here that no big constants are hidden in the asymptotic notation)

$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$
$1\,000\,000 - 100\,000\,000$	$100\,000 - 1\,000\,000$	$1000 - 10\,000$	$100 - 1000$

$2^{\mathcal{O}(n)}$	$\mathcal{O}(n!)$
$20 - 30$	$10 - 15$

Polynomial Algorithms

Polynomial — an expression of the form

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

where $a_0, a_1, \ldots, a_k$ are constants.

Examples of polynomials:

$$4n^3 - 2n^2 + 8n + 13 \qquad 2n + 1 \qquad n^{100}$$

Polynomial Algorithms

Function f is called **polynomial** if it is bounded from above by some polynomial, i.e., if there exists a constant k such that $f \in \mathcal{O}(n^k)$.

For example, the functions belonging to the following classes are polynomial:

$$\mathcal{O}(n) \qquad \mathcal{O}(n \log n) \qquad \mathcal{O}(n^2) \qquad \mathcal{O}(n^5) \qquad \mathcal{O}(\sqrt{n}) \qquad \mathcal{O}(n^{100})$$

Function such as 2^n or $n!$ are not polynomial — for arbitrarily big constant k we have

$$2^n \in \omega(n^k) \qquad\qquad n! \in \omega(n^k)$$

Polynomial Algorithms

Polynomial algorithm — an algorithm whose time complexity is polynomial — i.e., bounded from above by some polynomial (so it is in $\mathcal{O}(n^k)$ where k is a constant).

The notion of "polynomial algorithm" can be viewed as a certain approximation of what algorithms are generally viewed as "efficient" and useable in practive for quite long inputs.

Roughly we can say that:

- Polynomial algorithms are effiecient algorithms that can be used in practice for inputs of considerable size.
- Algorithms, which are not polynomial (i.e., that have a greater time complexity than polynomial, e.g., exponential), are generally not viewed as efficient.

  Such algorithms can be usually used only for "small" inputs.

Polynomial Algorithms

However, we must be aware of the following:

- An algorithm whose time complexity is for example in $\Theta(n^{100})$ surely can not be viewed as effiecient from a practical point of view.
- It can be shown that for each k it is possible to construct an artificial example of an algorithmic problem that can be solved using an algorithm with time complexity in $\mathcal{O}(n^{k+1})$ but there with no algorithm with time complexity in $\mathcal{O}(n^k)$.
- For "naturally" defined problems that are solved in practice it is not the case that for there would be some polynomial algorithm with a big degree of a polynomial and would not be some algorithm with a small degree of polynomial.

  Usually, we have one of two possibilities:

  - A polynomial algorithm is known and the degree of the polynomial is quite small, e.g., at most 5.
  - There is no known algorithm for the given problem.

Polynomial Algorithms

- From the practical point of view, sometimes even an algorithm with time complexity for example $\Theta(n^2)$ can be viewed as inefficient for some purpose — e.g., for extremely bigs inputs or if there are some very strict timing constraints.
- On the other hand, for some purposes even an algorithm with exponential time complexity can sometimes useful in practice.
- There are examples of algorithms that have an an exponential time complexity in the worst case but for many inputs they actually work efficiently and they can be used to process quite big inputs.

Complexity of Algorithms

For most of common algorithmic problems, one of the following three possibilities happens:

- A polynomial algorithm with time complexity $\mathcal{O}(n^k)$ is known, where k is some very small number (e.g., 5 or more often 3 or less).

- No polynomial algorithm is known and the best known algorithms have complexities such as $2^{\Theta(n)}$, $\Theta(n!)$, or some even bigger.

  In some cases, a proof is known that there does not exist a polynomial algorithm for the given problem (it cannot be constructed).

- No algorithm solving the given problem is known (and it is possibly proved that there does not exist such algorithm)

Complexity of Algorithms

A typical example of polynomial algorithm — matrix multiplication with time complexity $\Theta(n^3)$ and space complexity $\Theta(n^2)$:

Algorithm: Matrix multiplication

```
MATRIX-MULT (A, B, C, n):
    for i := 1 to n do
        for j := 1 to n do
            x := 0
            for k := 1 to n do
                x := x + A[i][k] * B[k][j]
            C[i][j] := x
```

Complexity of Algorithms

- For a rough estimation of complexity, it is often sufficient to count the number of nested loops — this number then gives the degree of the polynomial

  Example: Three nested loops in the matrix multiplication — the time complexity of the algorithm is $\mathcal{O}(n^3)$.

- If it is not the case that all the loops go from 0 to n but the number of iterations of inner loops are different for different iterations of an outer loops, a more precise analysis can be more complicated.

  It is often the case, that the sum of some sequence (e.g., the sum of arithmetic or geometric progression) is then computed in the analysis.

  The results of such more detailed analysis often does not differ from the results of a rough analysis but in many cases the time complexity resulting from a more detailed analysis can be considerably smaller than the time complexity following from the rough analysis.

Arithmetic Progression

Arithmetic progression — a sequence of numbers $a_0, a_1, \ldots, a_{n-1}$, where

$$a_i = a_0 + i \cdot d,$$

where d is some constant independent on i.

So in an arithmetic progression, we have $a_{i+1} = a_i + d$ for each i.

Example: The arithmetic progression where $a_0 = 1$, $d = 1$, and $n = 100$:

$$1, 2, 3, 4, 5, 6, \ldots, 96, 97, 98, 99, 100$$

The sum of an arithmetic progression:

$$\sum_{i=0}^{n-1} a_i = a_0 + a_1 + \cdots + a_{n-1} = \frac{1}{2} n (a_0 + a_{n-1})$$

Arithmetic Progression

Example:

$$1 + 2 + \cdots + n \;=\; \frac{1}{2}n(n+1) \;=\; \frac{1}{2}n^2 + \frac{1}{2}n \;=\; \Theta(n^2)$$

For example, for $n = 100$ we have

$$1 + 2 + \cdots + 100 \;=\; 50 \cdot 101 \;=\; 5050.$$

Arithmetic Progression

Proof: Let us denote

$$s = \sum_{i=0}^{n-1} a_i = a_0 + a_1 + \cdots + a_{n-1}$$

$$\begin{aligned}
2s &= s + s \\
&= (a_0 + a_1 + \cdots + a_{n-1}) + (a_0 + a_1 + \cdots + a_{n-1}) \\
&= (a_0 + a_1 + \cdots + a_{n-1}) + (a_{n-1} + a_{n-2} + \cdots + a_0) \\
&= (a_0 + a_{n-1}) + (a_1 + a_{n-2}) + \cdots + (a_{n-1} + a_0) \\
&= ((a_0 + 0 \cdot d) + (a_0 + (n-1) \cdot d)) + ((a_0 + 1 \cdot d) + (a_0 + (n-2) \cdot d)) + \\
&\quad \cdots + ((a_0 + (n-1) \cdot d) + (a_0 + 0 \cdot d)) \\
&= n \cdot (a_0 + a_0 + (n-1) \cdot d) \\
&= n \cdot (a_0 + a_{n-1})
\end{aligned}$$

Arithmetic Progression

Example: $s = 1 + 2 + 3 + \cdots + 99 + 100$

$$\begin{aligned} 2s &= s + s \\ &= (1 + 2 + \cdots + 100) + (1 + 2 + \cdots + 100) \\ &= (1 + 2 + \cdots + 100) + (100 + 99 + \cdots + 1) \\ &= (1 + 100) + (2 + 99) + (3 + 98) + \cdots + (99 + 2) + (100 + 1) \\ &= 100 \cdot (1 + 100) = 10100 \end{aligned}$$

So

$$s = \frac{1}{2} \cdot 10100 = 5050$$

Geometric Progression

Geometric progression — a sequence of numbers $a_0, a_1, \ldots, a_n$, where

$$a_i = a_0 \cdot q^i,$$

where q is some constant independent on i.

So in a geometric progression we have $a_{i+1} = a_i \cdot q$ for each i.

Example: The geometric progression where $a_0 = 1$, $q = 2$, and $n = 14$:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384

The sum of a geometic progression (where $q \neq 1$):

$$\sum_{i=0}^{n} a_i = a_0 + a_1 + \cdots + a_n = a_0 \frac{q^{n+1} - 1}{q - 1}$$

Geometric Progression

Example:

$$1 + q + q^2 + \cdots + q^n = \frac{q^{n+1} - 1}{q - 1}$$

In particular, for $q = 2$:

$$1 + 2^1 + 2^2 + 2^3 + \cdots + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2 \cdot 2^n - 1 = \Theta(2^n)$$

Geometric Progression

Proof: Let us denote

$$s = \sum_{i=0}^{n} a_i = a_0 + a_1 + \cdots + a_n$$

$$\begin{aligned}
s &= a_0 \cdot q^0 + a_0 \cdot q^1 + \cdots + a_0 \cdot q^n \\
s \cdot q &= (a_0 \cdot q^0 + a_0 \cdot q^1 + \cdots + a_0 \cdot q^n) \cdot q \\
&= a_0 \cdot q^1 + a_0 \cdot q^2 + \cdots + a_0 \cdot q^{n+1} \\
s \cdot q - s &= a_0 \cdot q^{n+1} - a_0 \cdot q^0 \\
s \cdot (q-1) &= a_0 \cdot (q^{n+1} - 1) \\
s &= a_0 \cdot \frac{q^{n+1} - 1}{q - 1}
\end{aligned}$$

Complexity of Algorithms

An **exponential** function: a function of the form c^n, where c is a constant — e.g., function 2^n

Logarithm — the inverse function to an exponential function: for a given n,

$$\log_c n$$

is the value x such that $c^x = n$.

Complexity of Algorithms

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	131072
18	262144
19	524288
20	1048576

n	$\lceil \log_2 n \rceil$
0	—
1	0
2	1
3	2
4	2
5	3
6	3
7	3
8	3
9	4
10	4
11	4
12	4
13	4
14	4
15	4
16	4
17	5
18	5
19	5
20	5

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20

Complexity of Algorithms

Examples where exponential functions and logarithms can appear in an analysis of algorithms:

- Some value is repeatedly decreased to one half or is repeatedly doubled.

  For example, in the **binary search**, the size of an interval halves in every iteration of the loop.

  Let us assume that an array has size n.

  What is the minimal size of an array n, for which the algorithm performs at least k iterations?

  The answer: 2^k

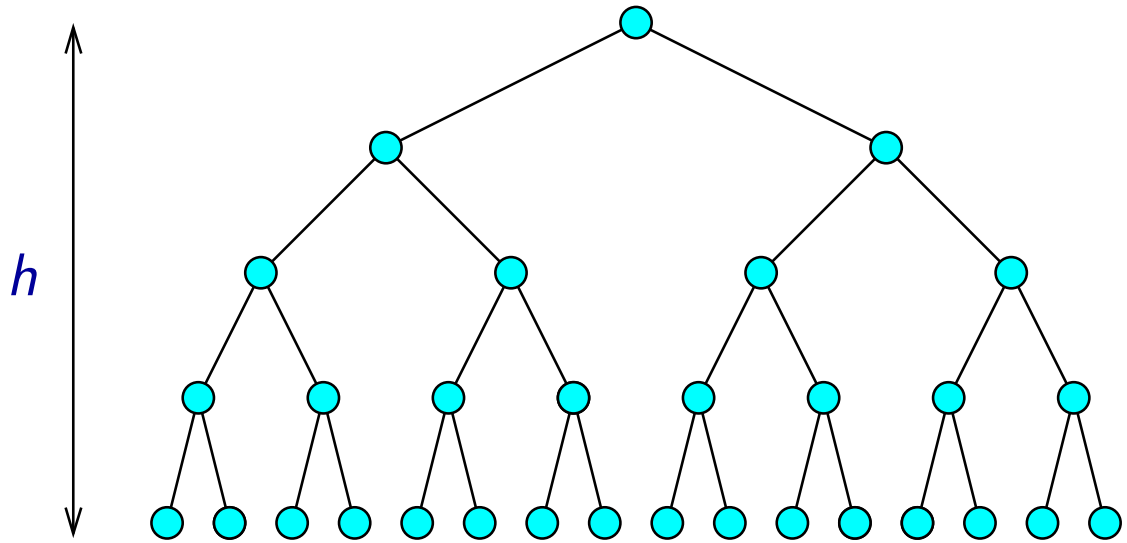
  So we have $k = \log_2(n)$. The time complexity of the algorithm is then $\Theta(\log n)$.

Complexity of Algorithms

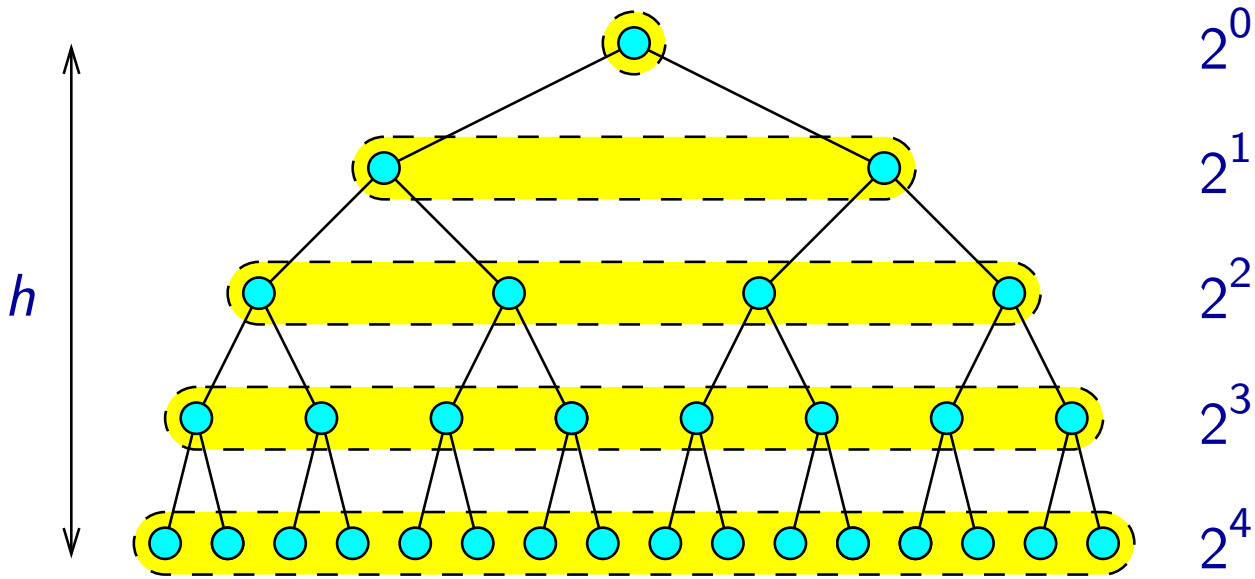
- Using n bits we can represent numbers from 0 to $2^n - 1$.
- The minimal numbers of bits, which are sufficient for representing a natural number x in binary is
$$\lceil \log_2(x+1) \rceil.$$
- A perfectly balanced tree of height h has $2^{h+1} - 1$ nodes, and 2^h of these nodes are leaves.
- The height of a perfectly balanced binary tree with n nodes is $\log_2 n$.

  An illustrating example: If we would draw a balanced tree with $n = 1\,000\,000$ nodes in such a way that the distance between neighbouring nodes would be 1 cm and the height of each layer of nodes would be also 1 cm, the width of the tree would be 10 km and its height would be approximately 20 cm.

A perfectly balanced binary tree of height h:

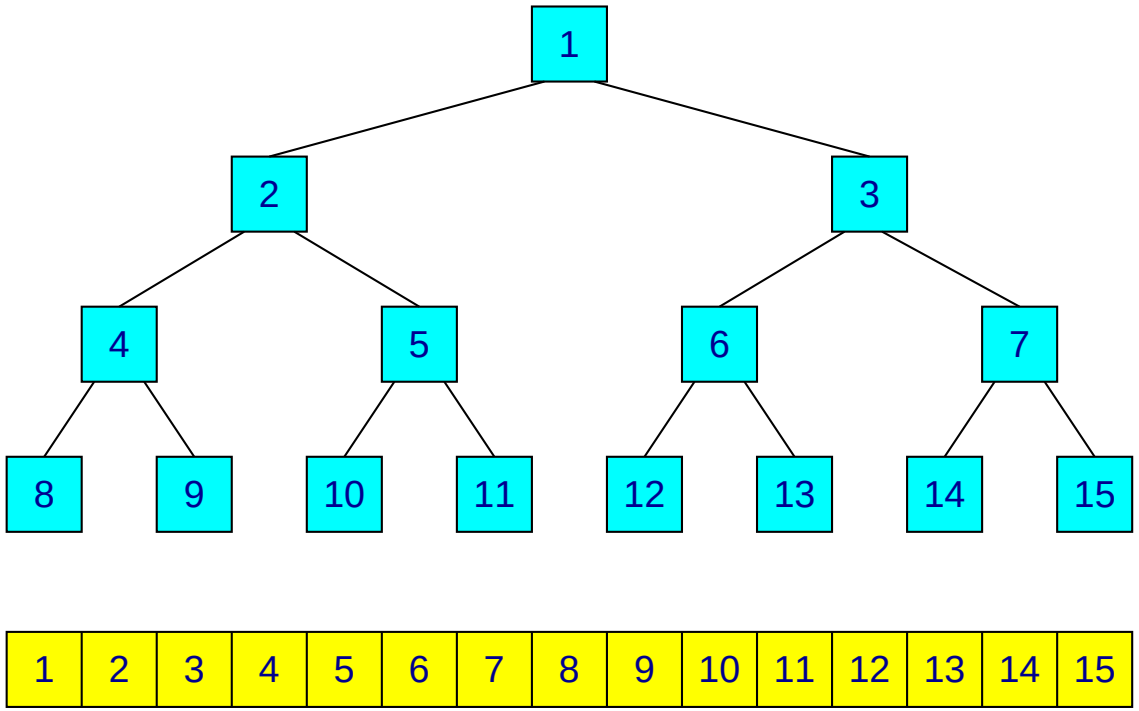


A perfectly balanced binary tree of height h:



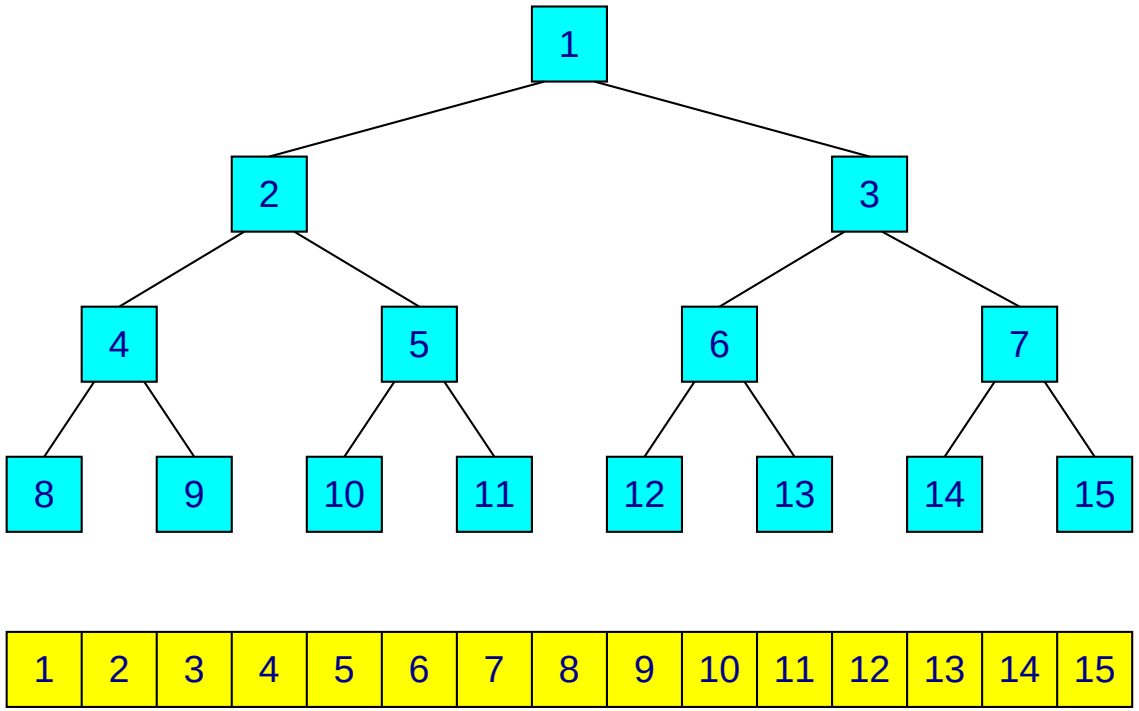
Complexity of Algorithms

An efficient way to store a complete binary tree in an array:



Complexity of Algorithms

An efficient way to store a complete binary tree in an array:



Children of a node with index i have indexes $2i$ and $2i + 1$.
The parent of a node with index i has index $\lfloor i/2 \rfloor$.

Complexity of Algorithms

Heap — a complete binary tree stored in an array A in way described on the previous slide, where moreover the following invariant holds for each $i = 1, 2, \ldots, n$:

- if $2i \leq n$ then $A[i] \leq A[2i]$
- if $2i + 1 \leq n$ then $A[i] \leq A[2i + 1]$

Examples of a usage of a heap:

- sorting algorithm **HeapSort**
- an efficient implementation of a **priority queue** — this allows to perform most operations on this queue with time complexity in $\mathcal{O}(\log n)$ where n is the number of elements currently in the queue

Complexity of Algorithms

Algorithm: Construction of a heap from an unsorted array

```
CREATE-HEAP (A, n):
    i := ⌊n/2⌋
    while i ≥ 1 do
        j := i
        x := A[j]
        while 2 * j ≤ n do
            k := 2 * j
            if k + 1 ≤ n and A[k + 1] < A[k] then
                k := k + 1
            if x ≤ A[k] then break
            A[j] := A[k]
            j := k
        A[j] := x
        i := i − 1
```

Complexity of Algorithms

Time complexity of CREATE-HEAP:

- By a quick and rough analysis, we can easily determine that this complexity is in $\mathcal{O}(n \log n)$ and in $\Omega(n)$:
  - The outer cycle is executed always $\lfloor n/2 \rfloor$ times — so the number of its iterations is in $\Theta(n)$.
  - The number of iterations of the inner cycle (in one iteration of the outer cycle) is obviously in $\mathcal{O}(\log n)$.
- It is much less obvious that the total number of iterations of the inner cycle (i.e., over all iterations of the outer cycle) is in fact in $\mathcal{O}(n)$.

So together we obtain:

The time complexity of CREATE-HEAP is in $\Theta(n)$.

Complexity of Algorithms

Justification that the total number of iterations of the inner cycle is in $\mathcal{O}(n)$:

Let us assume for simplicity that all branches of the tree are of the same length and that their length is h — so we have $n = 2^{h+1} - 1$.

Let C_i, where $0 \leq i < h$, be the total number of iterations of the inner cycle where at the beginning of the cycle the node with index j is in i-th layer of the tree (the layers are numbered top to bottom as $0, 1, 2, \ldots$).

It is obvious that the total number of iterations s is

$$s = C_{h-1} + C_{h-2} + \cdots + C_0 = \sum_{i=0}^{h-1} C_i$$

The value of C_i can be computed as the total number of nodes in the layers $0, 1, \ldots, i$:

$$C_i = 2^0 + 2^1 + \cdots + 2^i = \sum_{k=0}^{i} 2^k = \frac{2^{i+1} - 1}{2 - 1} = 2^{i+1} - 1$$

The total sum then can be computed as follows:

$$s = \sum_{i=0}^{h-1} C_i = \sum_{i=0}^{h-1} (2^{i+1} - 1) = 2 \cdot \left(\sum_{i=0}^{h-1} 2^i \right) - \left(\sum_{i=0}^{h-1} 1 \right)$$

$$= 2 \cdot \frac{2^h - 1}{2 - 1} - h = 2^{h+1} - 2 - h = n - 1 - h = \mathcal{O}(n)$$

Analysis of Recursive Algorithms

Recursive Algorithms

A **recursive algorithm** is an algorithm that transforms solving of an original problem to solving of several similar problems for smaller instances.

A general form of recursive algorithms:

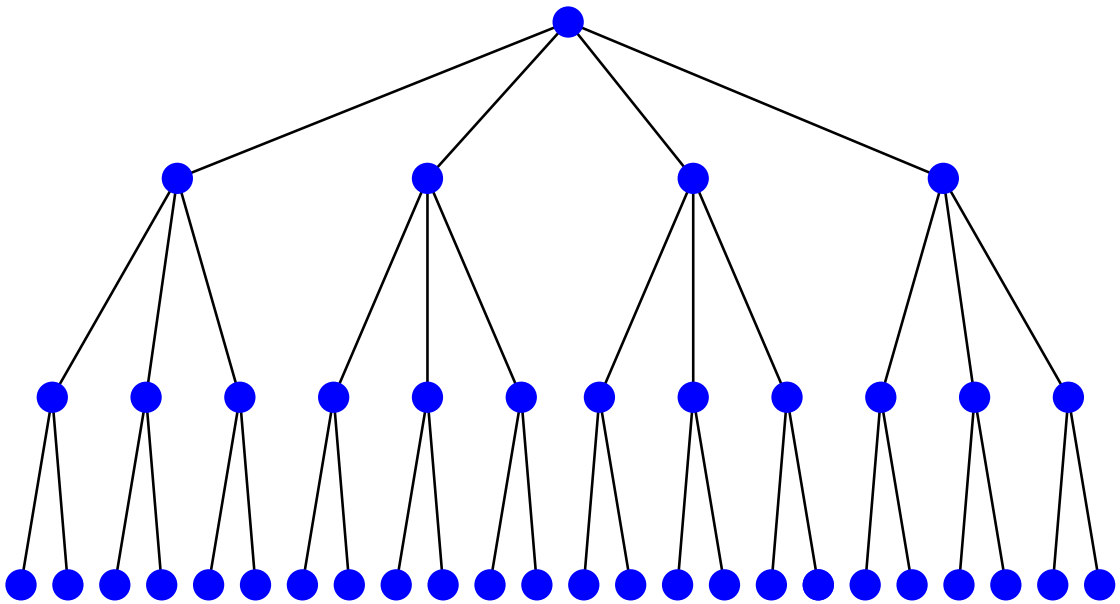
- If it is an elementary case, solve it directly and return the result.
- In other cases, create instances of subproblems.
- Call itself for each of the instances.
- Compute the solution of the original problem from solutions of individual subproblems and return it as the result.

Remark: Instances of the subproblems must be always in some sense smaller than the instance of the original problem. Very often (but not always), the size of an instance is decreased.

Recursive Algorithms

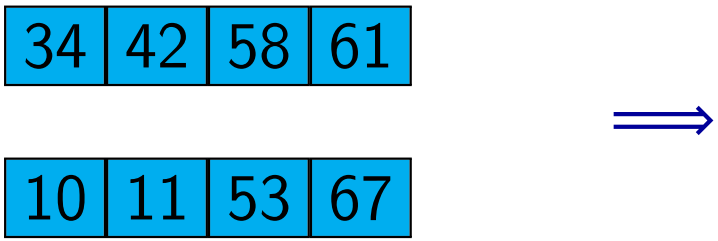
A computation of a recursive algorithm can be represented as a tree:

- nodes of the tree correspond to individual subproblems
- the root is the original problem
- children of a node correspond to subproblems of the given problem



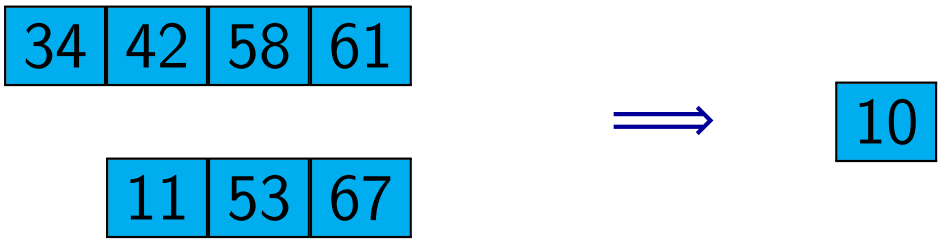
Example: Algorithm MERGE-SORT.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Example: Algorithm MERGE-SORT.

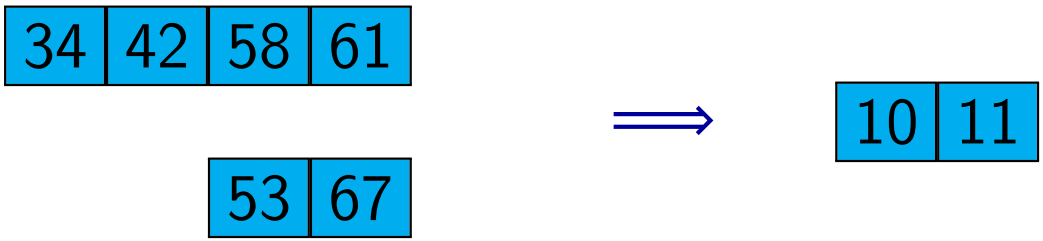
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Recursive Algorithms — Merge-Sort

Example: Algorithm MERGE-SORT.

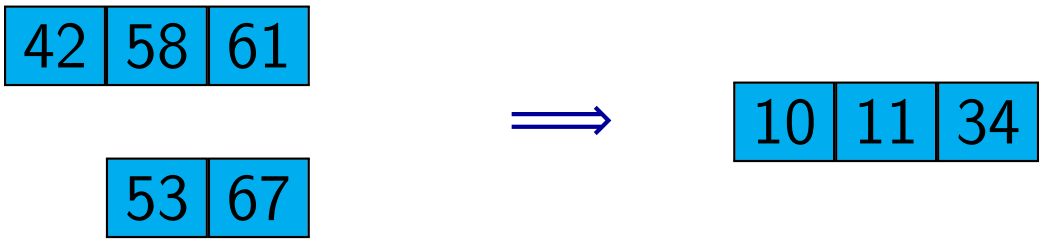
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Recursive Algorithms — Merge-Sort

Example: Algorithm MERGE-SORT.

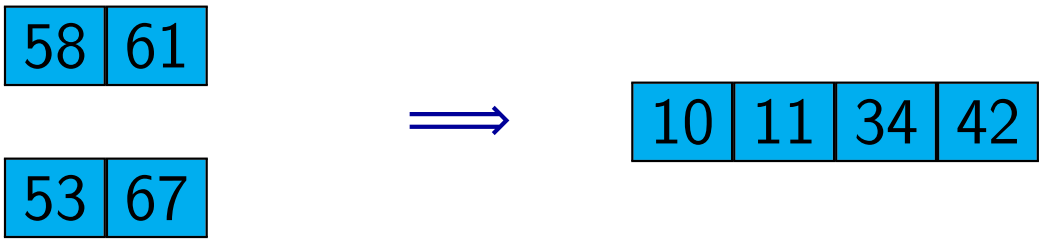
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Recursive Algorithms — Merge-Sort

Example: Algorithm MERGE-SORT.

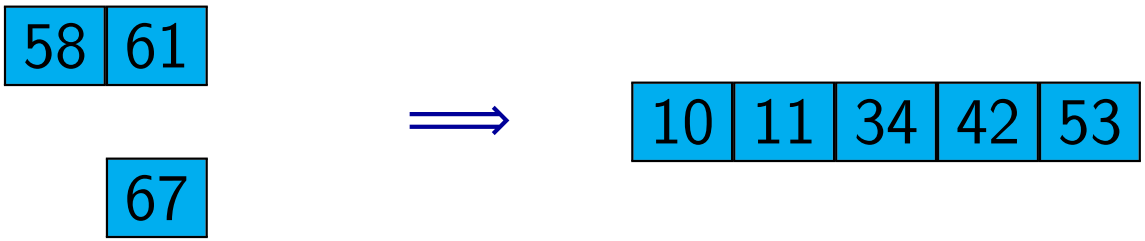
The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Recursive Algorithms — Merge-Sort

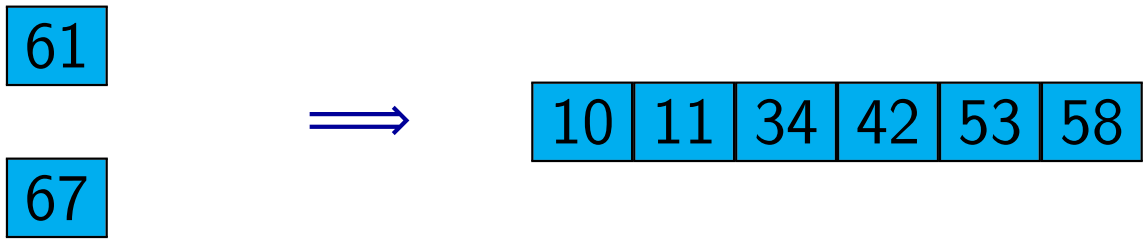
Example: Algorithm MERGE-SORT.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



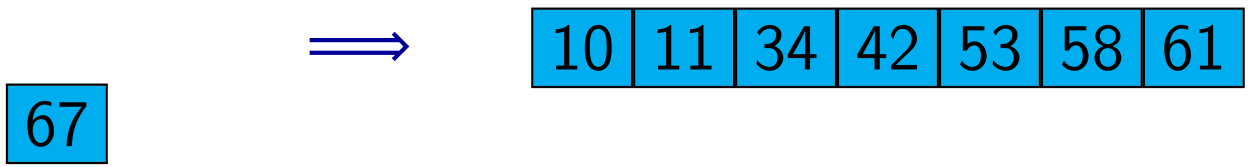
Example: Algorithm MERGE-SORT.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Example: Algorithm MERGE-SORT.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.



Recursive Algorithms — Merge-Sort

Example: Algorithm MERGE-SORT.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.
If both sequences have together n elements then this operation can be done in n steps.

Recursive Algorithms — Merge-Sort

Algorithm: Merge sort

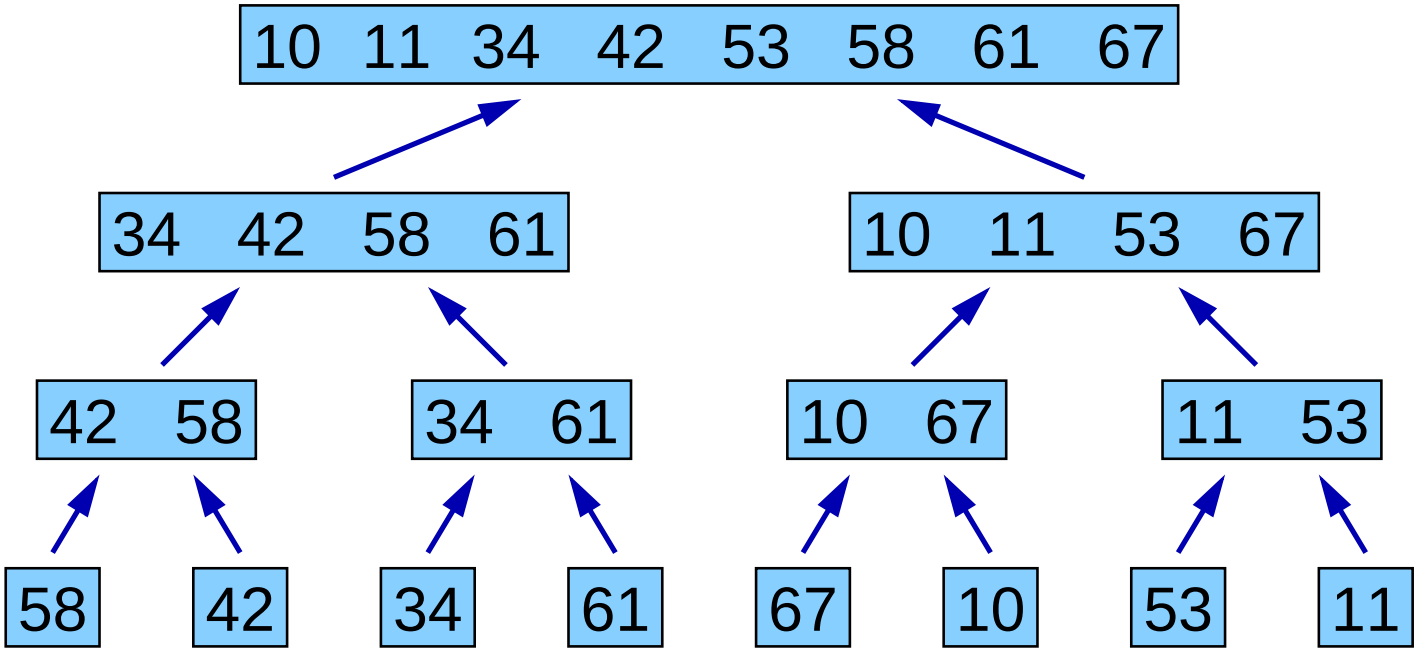
```
MERGE-SORT(A, p, r):
    if r − p > 1 then
        q := ⌊(p + r) / 2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q, r)
        MERGE(A, p, q, r)
```

To sort an array A containing elements $A[0], A[1], \cdots, A[n-1]$ we call MERGE-SORT$(A, 0, n)$.

Remark: Procedure MERGE(A, p, q, r) merges sorted sequences stored in $A[p \,..\, q-1]$ and $A[q \,..\, r-1]$ into one sequence stored in $A[p \,..\, r-1]$.

Recursive Algorithms — Merge-Sort

Input: 58, 42, 34, 61, 67, 10, 53, 11



The tree of recursive calls has $\Theta(\log n)$ layers. On each layer, $\Theta(n)$ operations are performed. The time complexity of MERGE-SORT is $\Theta(n \log n)$.

The master theorem

Master theorem

Let us assume that $a \geq 1$ and $b > 1$ are constants $f(n)$ is a function, and that function $T(n)$ is defined by a recursive equation

$$T(n) = a \cdot T(n/b) + f(n)$$

(where n/b can be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$). Then it holds that:

a. If $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

b. If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

c. If $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all big enough n, then $T(n) = \Theta(f(n))$.

The master theorem

The master theorem can be used to analyze complexity of those recursive algorithms where:

- Solving of one subproblem of size n, where $n > 1$, is transformed to solving a subproblems where every one of them is of size n/b.
- The time, which is spent by solving of one subproblem of size n, when we do not count time spent in recursive invokations, is bounded from above by a function $f(n)$.

Example: Algorithm MERGE-SORT: $a = 2$, $b = 2$, $f(n) \in \Theta(n)$

(in one call — two subproblems, each of them of size $n/2$, merging two sorted sequences in time $\Theta(n)$)

It holds $f(n) \in \Theta(n^{\log_b a}) = \Theta(n)$, and so

$$T(n) \in \Theta(n^{\log_b a} \log n) = \Theta(n \log n).$$

The master theorem — multiplication of big numbers

Example: Let us say that we want to work with natural numbers that are so big that they do not fit to integer datatypes we have available.

For example, we may want to work with number that have 4096 bits but operations with numbers that are available in a given programming language, in which we program, allow to work directly only with numbers that have 32 or 64-bits.

A simple way how to do it, is to store each such "big" number into an array of a corresponding size where every element of this array will be a "small" number of a size, with which we can work directly.

The master theorem — multiplication of big numbers

So, a "big" number u can be stored in array U with elements $U[0], \ldots, U[n-1]$ where each element will be one "small" number in range $0, \ldots, q-1$ where q is some suitable base chosen so that we can work directly with numbers in this range.

It will hold that

$$u = \sum_{i=0}^{n-1} U[i] \cdot q^i$$

A number u stored in this way can be viewed as written a number representation with base q where elements of array U represent individual "digits" of this representation.

The size of number u will be the number of these "digits" necessary for its representation (i.e., the number n).

The master theorem — multiplication of big numbers

Two numbers of size n can easily added in time $\mathcal{O}(n)$ using a standard "elementary school" algorithm for addition.

Similarly we can start to think about the problem of **multiplication** of two natural numbers:

Multiplication of two natural numbers

Input: Number u stored in array U of size n and number v stored in array V of size n.

Output: Number w stored in array W of size $2n$ such that $u \cdot v = w$.

The standard "elementary school" algorithm for multiplication is of time complexity $\Theta(n^2)$.

The master theorem — multiplication of big numbers

Instead of this standard algorithm we consider a **recursive** algorithm based on the following idea:

- If the numbers are of size 1 (i.e., $n = 1$), then we multiply them directly.
- If they are bigger (i.e., when $n > 1$), both numbers can be decomposed into pairs of numbers whose size is approximately one half of the original size, i.e.,

$$u = U_1 \cdot Q + U_0$$
$$v = V_1 \cdot Q + V_0$$

where $Q = q^{\lceil n/2 \rceil}$.

The product $w = u \cdot v$ then can be computed as

$$w = W_2 \cdot Q^2 + W_1 \cdot Q + W_0$$

where

$$W_2 = U_1 \cdot V_1$$
$$W_1 = U_0 \cdot V_1 + U_1 \cdot V_0$$
$$W_0 = U_0 \cdot V_0$$

The master theorem — multiplication of big numbers

This way, the problem of multiplication of two numbers of size n is transformed to 4 problems of multiplication of numbers of size $n/2$.

To multiply these smaller numbers, we can use **recursion** — the function calls itself with these smaller numbers as parameters.

(Multiplication by powers of Q can be implemented using shifts by a corresponding number of positions.)

The master theorem — multiplication of big numbers

The total running time of the algorithm then can be expressed by the following recursive formula:

$$T(n) = 4 \cdot T(n/2) + \Theta(n)$$

To use the master theorem we have $a = 4$, $b = 2$ and $f(n) \in \Theta(n)$:

- It is $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$, since $n \in \mathcal{O}(n^{\log_2 4 - \epsilon}) = \mathcal{O}(n^{2-\epsilon})$ holds for example for $\epsilon = 1$.
- It follows by the master theorem that
$$T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2).$$

So the time complexity of the recursive algorithm is $\Theta(n^2)$, which is the same as the time complexity of the simple standard algorithm.

The master theorem — multiplication of big numbers

However, in the recursive algorithm, the number of multiplications of numbers of the half size can be decreased from 4 to 3:

- At first, the values W_2 and W_0 are computed the same way as before:

$$W_2 = U_1 \cdot V_1$$
$$W_0 = U_0 \cdot V_0$$

- The value W_1 then can be computed as follows:

$$W_1 = (U_0 + U_1) \cdot (V_0 + V_1) - W_0 - W_2$$

Verifying the correctness:

$$\begin{aligned} W_1 &= (U_0 + U_1) \cdot (V_0 + V_1) - W_0 - W_2 \\ &= U_0 V_0 + U_0 V_1 + U_1 V_0 + U_1 V_1 - U_0 V_0 - U_1 V_1 \\ &= U_0 V_1 + U_1 V_0 \end{aligned}$$

The master theorem — multiplication of big numbers

This recursive algorithm for multiplication of big numbers is called **Karatsuba multiplication**.

Similarly as in the previous case, we can express the time complexity of this algorithm as a recursive formula

$$T(n) = 3 \cdot T(n/2) + \Theta(n)$$

Then, using the master theorem (for $a = 3$, $b = 2$ and $f(n) \in \Theta(n)$), we can derive

$$T(n) \in \Theta(n^{\log_2 3})$$

$\log_2 3$ is approximately 1.5849625

So $T(n) \in \mathcal{O}(n^{1.59})$, which is better than $\Theta(n^2)$.

The master theorem — multiplication of big numbers

Remark:

- There exist a whole range of even more efficient algorithms for multiplication of big numbers. This algorithms are based, similarly as Karatsuba multiplication, on the **recursive approach**:
  - for example several variants of Toom-Cook algorithm
- The most efficient algorithms for multiplication of natural numbers are based on **Fast Fourier transform** (**FFT**):
  - Schönhage–Strassen

  The time complexity of Schönhage-Strassen is $\mathcal{O}(n \cdot \log n \cdot \log\log n)$.

  In 2019, an algorithm with the time complexity $\mathcal{O}(n \cdot \log n)$ was discovered (D. Harvey, J. van der Hoeven).

The master theorem — matrix multiplication

Example: The multiplication of square matrices A and B of dimension $n \times n$ using a recursive approach:

- For $n = 1$, the result is computed directly.
- For $n > 1$, every of the matrices A and B is decomposed into four submatrices of size $(n/2) \times (n/2)$.
- The result is composed from these eigth submatrices using addition and multiplication. To multiply these smaller matrices, the function is called recursively.
- A straightforward way requires 8 multiplications of matrices of size $(n/2) \times (n/2)$.

  So we have $a = 8$, $b = 2$, $f(n) \in \Theta(n^2)$.

  Then $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$ because $n^2 \in \mathcal{O}(n^{\log_2 8 - \epsilon}) = \mathcal{O}(n^{3-\epsilon})$ hold for example for $\epsilon = 1$.

  So $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 8}) = \Theta(n^3)$.

This means that this approach is not better than standard simple algorithm for matrix multiplication.

The master theorem — matrix multiplication

But there is a clever way how the computation above can be done in a more complicated way where in one recursive call, it is sufficient to call the function 7-times
(using more additions and multiplications).

This is called **Strassen algorithm**.

Here we have $a = 7$, $b = 2$, and $f(n) \in \Theta(n^2)$.

Again we have $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$ because $n^2 \in \mathcal{O}(n^{\log_2 7 - \epsilon})$ holds for example for $\epsilon = 0.5$.

($\log_2 7$ is approximately 2.80735)

So $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$ and $T(n) \in \mathcal{O}(n^{2.81})$.

The master theorem — proof

Proof of the master theorem:

For simplicity, we will concentrate just to the cases when $f(n) = n^c$ for some constant $c > 0$.

We also assume for simplicity that n is a power of b, so we need not to deal with rounding.

Let us imagine a tree of recursive calls for an instance of size n:

- The height of the tree is $\log_b n$.
- The numbers of nodes on individual layers are $a^0, a^1, \ldots, a^{\log_b n}$
- The time spent in one node of layer i is

$$f\left(\frac{n}{b^i}\right) = \left(\frac{n}{b^i}\right)^c$$

The master theorem — proof

So we have

$$T(n) = \sum_{i=0}^{\log_b n} a^i \cdot f\left(\frac{n}{b^i}\right) = \sum_{i=0}^{\log_b n} a^i \cdot \left(\frac{n}{b^i}\right)^c = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i$$

Let us denote $q = a/b^c$. Three cases must be distinguished:

- $q > 1$ — i.e., when $a > b^c$, which holds iff $c < \log_b a$
- $q = 1$ — i.e., when $a = b^c$, which holds iff $c = \log_b a$
- $q < 1$ — i.e., when $a < b^c$, which holds iff $c > \log_b a$

The master theorem — proof

The case $q > 1$ — i.e., when $a > b^c$, which means $c < \log_b a$:

$$T(n) = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i = n^c \cdot \frac{q^{\log_b n + 1} - 1}{q - 1} \in \Theta(n^c \cdot q^{\log_b n})$$

It holds

$$n^c \cdot q^{\log_b n} = n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b n} = n^c \cdot n^{\log_b\left(\frac{a}{b^c}\right)}$$

$$= n^c \cdot n^{\log_b a - \log_b(b^c)} = n^{c + \log_b a - c} = n^{\log_b a}$$

And so $T(n) \in \Theta(n^{\log_b a})$.

Remark: The number of the leaves in the tree is $a^{\log_b n} = n^{\log_b a}$.

So the most of the time is spent by solving these elementary cases.

The master theorem — proof

The case $q = 1$ — i.e., when $a = b^c$, which means $c = \log_b a$:

$$T(n) = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i = n^c \cdot \sum_{i=0}^{\log_b n} 1 = n^c \cdot (\log_b n + 1) \in \Theta(n^{\log_b a} \log n)$$

Remark: Approximately the same time $\Theta(n^{\log_b a})$ is spent in each layer of the tree.

There are $\Theta(\log n)$ layers.

The master theorem — proof

The case $q < 1$ — i.e., when $a < b^c$, which means $c > \log_b a$:

$$T(n) = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i < n^c \cdot \sum_{i=0}^{\infty} \left(\frac{a}{b^c}\right)^i = n^c \cdot \frac{1}{1-q} \in \mathcal{O}(n^c)$$

because for q, where $0 < q < 1$, it holds that

$$\sum_{i=0}^{\infty} q^i = \lim_{z\to\infty} \sum_{i=0}^{z} q^i = \lim_{z\to\infty} \frac{q^{z+1}-1}{q-1} = \frac{1}{1-q}$$

Obviously $T(n) \in \Omega(n^c)$ (because the time spent just in the root of the tree is $\Theta(n^c)$), and therefore we have $T(n) \in \Theta(n^c)$.

Remark: In this case, the most of the time is spent in the root of the tree.