Computational Complexity of Algorithms

Complexity of an Algorithm

- 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 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 an Algorithm

- A running time is affected by many factors, e.g.:
  - the algorithm that is used
  - the amount of input data
  - used hardware (e.g., the frequency at which a CPU is running can be important)
  - the used programming language — its implementation (compiler/interpreter)
  - ...
- If we need to solve problem for "small" input data, the running time is usually negligible.
- With increasing amount of input data (the size of input), the running time can grow, sometimes significantly.

Complexity of an Algorithm

- **Time complexity of an algorithm** — how the running time of the algorithm depends on the amount of input data
- **Space complexity of an algorithm** — how the amount of a memory used during a computation grows with respect to the size of input

Remark: The precise definitions will be given later.

Remark:

- There are also other types of computational complexity, which we will not discuss here (e.g., communication complexity).

Complexity of an Algorithm

To determine the precise running time or the precise amount of used memory just by an analysis of an algorithm can be extremely difficult.

Usually the analysis of complexity of an algorithm involves many simplifications:

- It is usually not analysed how the running time or the amount of used memory depends precisely on particular input data but how they depend on the **size of the input**.
- Functions expressing how the running time or the amount of used memory grows depending on the size of the input are not computed precisely — instead **estimations** of these functions are computed.
- Estimations of these functions are usually expressed using **asymptotic notation** — e.g., it can be said that the running time of MergeSort is $O(n \log n)$, and that the running time of BubbleSort is $O(n^2)$.

Size of Input

Size of the input — a value describing how "big" is an input instance

- In most cases, the size of an input is just one number — it is usually denoted n or N.
- Sometimes it is more appropriate to express the size of an input by pair (sometimes even with three, four, etc.) of parameters — in this case, they are ofted denoted n and m (or N and M).
- We can choose what should be considered as the size of an input.

Size of Input

Examples, what the size of an input can be:

- An input is a sequence of some values, an array of elements, etc. (e.g., in problems like sorting, searching in an array, finding the maximal element, etc.):

  n — the number of elements in this sequence or array

- An input is a string of characters (a word from some alphabet):

  n — the number of characters in this string

- An input consists of two strings, e.g., a (long) text that will be searched through, and a (shorter) searched pattern:

  n — the number of characters in the text
  m — the number of characters in the searched pattern

Size of Input

- An input is a set of strings:

  One possibility:

  n — the sum of lengths of all strings

  Other variant:

  n — the sum of lengths of all strings, m — the number of strings

- The input is a graph:

  n — the number of nodes, m — the number of edges

Size of Input

- The input is one number (e.g., in the primarity testing):

  One possibility:
  n — the number of bits of the number — e.g., the size of input 962261 is 20

  Other variant:
  n — the value of the number — the size of input 962261 is 962261

- The input is a sequence of numbers, and the running time is affected by the values of the numbers (e.g., in the problem where the goal is to compute the greatest common divisor of all numbers in a given sequence):

  n — the sum of numbers of bits of all numbers in the given sequence

Running Time

Let us say that we have:

- an algorithm Alg solving a problem P (resp. a particular implementation of algorithm Alg),
- a machine $\mathcal{M}$ executing the algorithm Alg,
- an input w from the set In, which is a set of all inputs of problem P

An example:

- a particular implementation of Quicksort in C++ solving the problem of sorting,
- a computer with some particular type of processor working on some particular frequency, with some particular amount of memory, operating system, etc.
- input: array $[6, 13, 1, 8, 4, 5, 8]$
  (remark: a more realistic example would be an array with one million elements)

Running Time

$t(w)$ — the running time of the algorithm Alg on input w on machine $\mathcal{M}$

What units should be used for expressing time? (As we will see, this is not important when asymptotic notation is used.)

- **in seconds** — it depends on too many details of implementation, it is difficult to determine it in other way than by measurement

  (even on the same computer with the same data the running time can fluctuate)

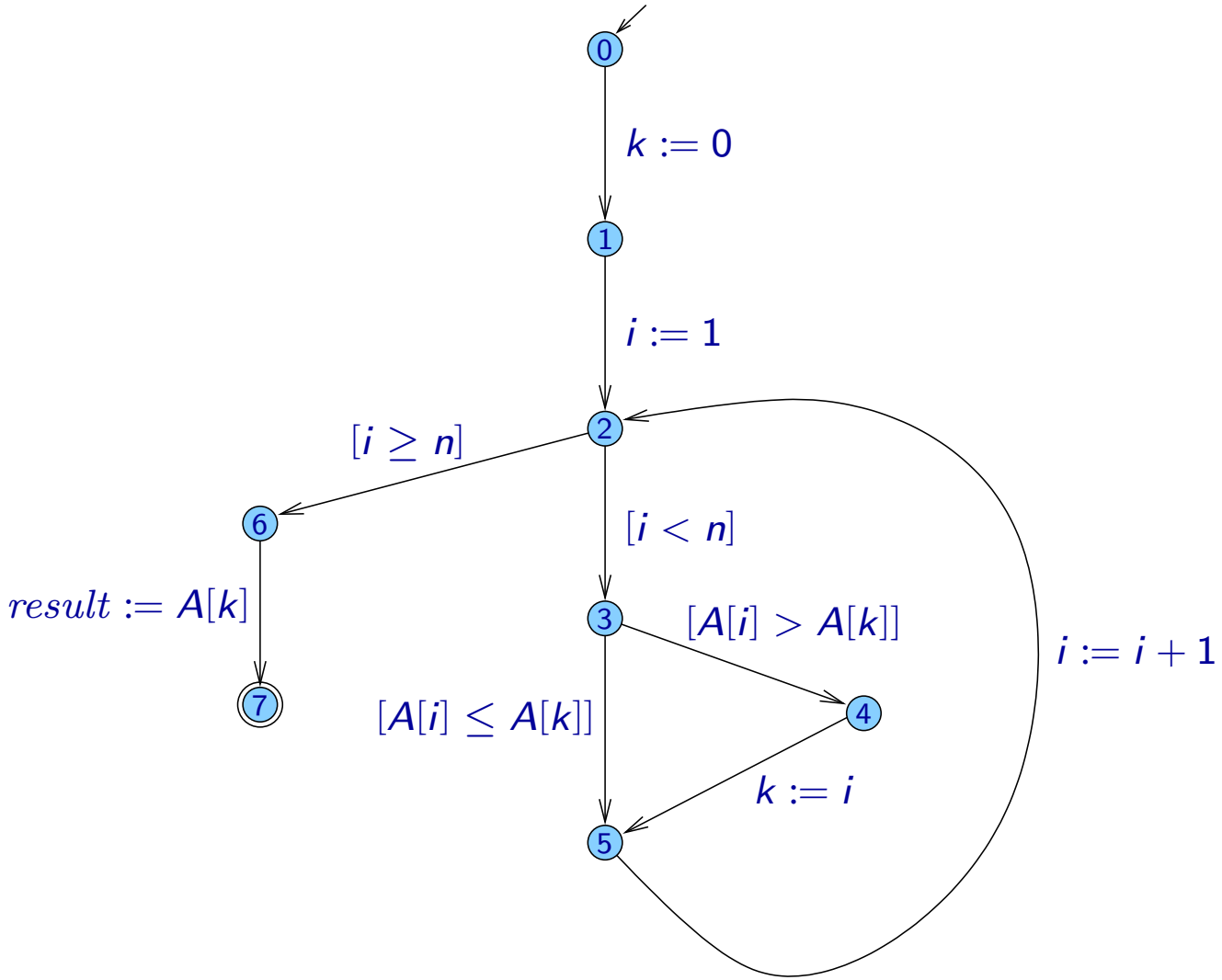
- **the number of steps** — it must be specified what is considered as one step, for example:
  - one statement of a high level programming language
  - one instruction of machine code or bytecode
  - one tick of a processor
  - one operation of some particular type — e.g., a comparison, an arithmetic operation, etc. (while all other operations are ignored)
  - ...

Running Time

Let us say that an algorithm is represented by a control-flow graph:

- To every instruction (i.e., to every edge) we assign a value specifying how long it takes to perform this instruction once.
- The execution time of different instructions can be different.
- For simplicity we assume that an execution of the same instruction takes always the same time — the value assigned to an instruction is a number from the set $\mathbb{R}^+$ (the set of nonnegative real numbers).

Running Time



Instr.	time
$k := 0$	c_0
$i := 1$	c_1
$[i < n]$	c_2
$[i \geq n]$	c_3
$[A[i] \leq A[k]]$	c_4
$[A[i] > A[k]]$	c_5
$k := i$	c_6
$i := i + 1$	c_7
$result := A[k]$	c_8

Running Time

Example: The execution times of individual instructions could be for example:

Instr.	symbol	time
$k := 0$	c_0	4
$i := 1$	c_1	4
$[i < n]$	c_2	10
$[i \geq n]$	c_3	12
$[A[i] \leq A[k]]$	c_4	14
$[A[i] > A[k]]$	c_5	12
$k := i$	c_6	5
$i := i + 1$	c_7	6
$result := A[k]$	c_8	5

For a particular input w, e.g., for $w = ([3, 8, 4, 5, 2], 5)$, we could simulate the computation and determine the precise running time $t(w)$.

Time Complexity of an Algorithm

Let us say that:

- For a given algorithm Alg and machine $\mathcal{M}$, and for every input w from the set of all inputs In, the running time $t(w)$ is precisely defined.
- To each input w from set In, a number $size(w)$ describing the size of the input w is assigned .

  (Formally, it is a function $size : In \to \mathbb{N}$.)

Definition

The **time complexity of algorithm Alg in the worst case** is the function $T : \mathbb{N} \to \mathbb{R}^+$that assigns to each natural number n the maximal running time of the algorithm Alg on an input of size n.
So for each $n \in \mathbb{N}$ we have:

- For each input $w \in In$ such that $size(w) = n$ is $t(w) \leq T(n)$.
- There exists an input $w \in In$ such that $size(w) = n$ and $t(w) = T(n)$.

Time Complexity of an Algorithm

It is obvious from this definition that the time complexity of an algorithm is a function whose precise values depend not only on the given algorithm Alg but also on the following things:

- on a machine $\mathcal{M}$, on which the algorithm Alg runs,
- on the precise definition of the running time $t(w)$ of algorithm Alg on machine $\mathcal{M}$ with input $w \in In$,
- on the precise definition of the size of an input (i.e., on the definition of function $size$).

Time Complexity of an Algorithm

Sometimes, the time complexity in the **average case** is also analyzed:

- Some particular **probabilistic distribution** on the set of inputs must be assumed.
- Instead of the maximal running time on inputs of size n, the expected value of the running times is considered.
- Usually, the analysis of the avarage case is much more complicated than the analysis of the worst case.
- Often, these two functions are not very different but sometimes the difference is significant.

Remark: It usually makes little sense to analyze the time complexity in the best case.

Time Complexity of an Algorithm

An example of an analysis of the time complexity of algorithm FIND-MAX **without** the use of asymptotic notation:

- Such precise analysis is almost never done in practice — it is too tedious and complicated.
- This illustrates what things are ignored in an analysis where asymptotic notation is used and how much the analysis is simplified by this.
- We will compute with constants $c_0, c_1, \ldots, c_8$, which specify the execution time of individual instructions — we won't compute with concrete numbers.

Time Complexity of an Algorithm

The inputs are of the form (A, n), where A is an array and n is the number of elements in this array (where $n \geq 1$).

We take n as the size of input (A, n).

Consider now some particular input $w = (A, n)$ of size n:

- The running time $t(w)$ on input w can be expressed as

$$t(w) = c_0 m_0 + c_1 m_1 + \cdots + c_8 m_8,$$

where $m_0, m_1, \ldots, m_8$ are numbers specifying how many times is each instruction performed in the computation on input w.

Time Complexity of an Algorithm

Instr.	time	occurences	value of m_i
$k := 0$	c_0	m_0	1
$i := 1$	c_1	m_1	1
$[i < n]$	c_2	m_2	$n-1$
$[i \geq n]$	c_3	m_3	1
$[A[i] \leq A[k]]$	c_4	m_4	$n-1-\ell$
$[A[i] > A[k]]$	c_5	m_5	ℓ
$k := i$	c_6	m_6	ℓ
$i := i+1$	c_7	m_7	$n-1$
$result := A[k]$	c_8	m_8	1

ℓ — the number of iterations of the cycle where $A[i] > A[k]$
(obviously $0 \leq \ell < n$)

Time Complexity of an Algorithm

By assigning values to

$$t(w) \;=\; c_0 m_0 + c_1 m_1 + \;\cdots\; + c_8 m_8,$$

we obtain

$$t(w) \;=\; d_1 \;+\; d_2 \cdot (n-1) \;+\; d_3 \cdot (n-1-\ell) \;+\; d_4 \cdot \ell,$$

where

$$d_1 = c_0 + c_1 + c_3 + c_8 \qquad d_3 = c_4$$
$$d_2 = c_2 + c_7 \qquad d_4 = c_5 + c_6$$

After simplification we have

$$t(w) \;=\; (d_2 + d_3) \cdot n \;+\; (d_4 - d_3) \cdot \ell \;+\; (d_1 - d_2 - d_3)$$

Remark: $t(w)$ is not the time complexity but the running time for a particular input w

Time Complexity of an Algorithm

For example, if the execution times of instructions will be:

Instr.	symb.	time
$k := 0$	c_0	4
$i := 1$	c_1	4
$[i < n]$	c_2	10
$[i \geq n]$	c_3	12
$[A[i] \leq A[k]]$	c_4	14
$[A[i] > A[k]]$	c_5	12
$k := i$	c_6	5
$i := i + 1$	c_7	6
$result := A[k]$	c_8	5

then $d_1 = 25$, $d_2 = 16$, $d_3 = 14$, and $d_4 = 17$.

In this case is $t(w) = 30n + 3\ell - 5$.

For the input $w = ([3, 8, 4, 5, 2], 5)$ is $n = 5$ and $\ell = 1$, therefore $t(w) = 30 \cdot 5 + 3 \cdot 1 - 5 = 148$.

Time Complexity of an Algorithm

It can depend on details of implementation and on the precise values of constants, for which inputs of size n the compution takes the longest time (i.e., which are the worst cases):

The running time of algorithm FIND-MAX for an input $w = (A, n)$ of size n:

$$t(w) \;=\; (d_2 + d_3) \cdot n \;+\; (d_4 - d_3) \cdot \ell \;+\; (d_1 - d_2 - d_3)$$

- If $d_3 \geq d_4$ — the worst cases are those where ℓ has the smallest value $\ell = 0$ — for example inputs of the form $[0, 0, \ldots, 0]$ or of the form $[n,\ n-1,\ n-2,\ \ldots,\ 2,\ 1]$
- If $d_3 \leq d_4$ — the worst are those cases where ℓ has the greatest value $\ell = n - 1$ — for example inputs of the form $[0, 1, \ldots, n-1]$

Time Complexity of an Algorithm

The time complexity $T(n)$ of algorithm FIND-MAX in the worst case is given as follows:

- If $d_3 \geq d_4$:

$$T(n) = (d_2 + d_3) \cdot n + (d_1 - d_2 - d_3)$$

- If $d_3 \leq d_4$:

$$\begin{aligned} T(n) &= (d_2 + d_3) \cdot n + (d_4 - d_3) \cdot (n-1) + (d_1 - d_2 - d_3) \\ &= (d_2 + d_4) \cdot n + (d_1 - d_2 - d_4) \end{aligned}$$

Example: For $d_1 = 25$, $d_2 = 16$, $d_3 = 14$, $d_4 = 17$ is

$$\begin{aligned} T(n) &= (16 + 17) \cdot n + (25 - 16 - 17) \\ &= 33n - 8 \end{aligned}$$

Time Complexity of an Algorithm

In both cases (when $d_3 \geq d_4$ or when $d_3 \leq d_4$), the time complexity of the algorithm FIND-MAX is a function

$$T(n) = an + b$$

where a and b are some constants whose precise values depend on the execution time of individual instructions.

Remark: These constants could be expressed as

$$a = d_2 + \max\{d_3, d_4\} \qquad b = d_1 - d_2 - \max\{d_3, d_4\}$$

For example

$$T(n) = 33n - 8$$

Time Complexity of an Algorithm

If it would be sufficient to find out that the time complexity of the algorithm FIND-MAX is some function of the form

$$T(n) = an + b,$$

where the precise values of constants a and b would not be important for us, the whole analysis could be considerably simpler.

- In fact, we usually do not want to know precisely how function $T(n)$ look (in general, it can be a very complicated function), and it would be sufficient to know that values of the function $T(n)$ "approximately" correspond to values of a function $S(n) = an + b$, where a and b are some constants.

Time Complexity of an Algorithm

For a given function $T(n)$ expressing the time or space complexity, it is usually sufficient to express it approximately — to have an **estimation** where

- we ignore the less important parts

  (e.g., in function $T(n) = 15n^2 + 40n - 5$ we can ignore $40n$ and -5, and to consider function $T(n) = 15n^2$ instead of the original function),

- we ignore multiplication constants

  (e.g., instead of function $T(n) = 15n^2$ we will consider function $T(n) = n^2$)

- we won't ignore constants in exponents — for example there is a big difference between functions $T_1(n) = n^2$ and $T_2(n) = n^3$.

- we will be interested how function $T(n)$ behaves for "big" values of n, we can ignore its behaviour on small values

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

In the following, we will consider functions of the form $f : \mathbb{N} \to \mathbb{R}$, where:

- The values of $f(n)$ need not to be defined for all values of $n \in \mathbb{N}$ but there must exist some constant n_0 such that the value of $f(n)$ is defined for all $n \in \mathbb{N}$ such that $n \geq n_0$.

  Example: Function $f(n) = \log_2(n)$ is not defined for $n = 0$ but it is defined for all $n \geq 1$.

- There must exist a constant n_0 such that for all $n \in \mathbb{N}$, where $n \geq n_0$, is $f(n) \geq 0$.

  Example: It holds for function $f(n) = n^2 - 25$ that $f(n) \geq 0$ for all $n \geq 5$.

Asymptotic Notation

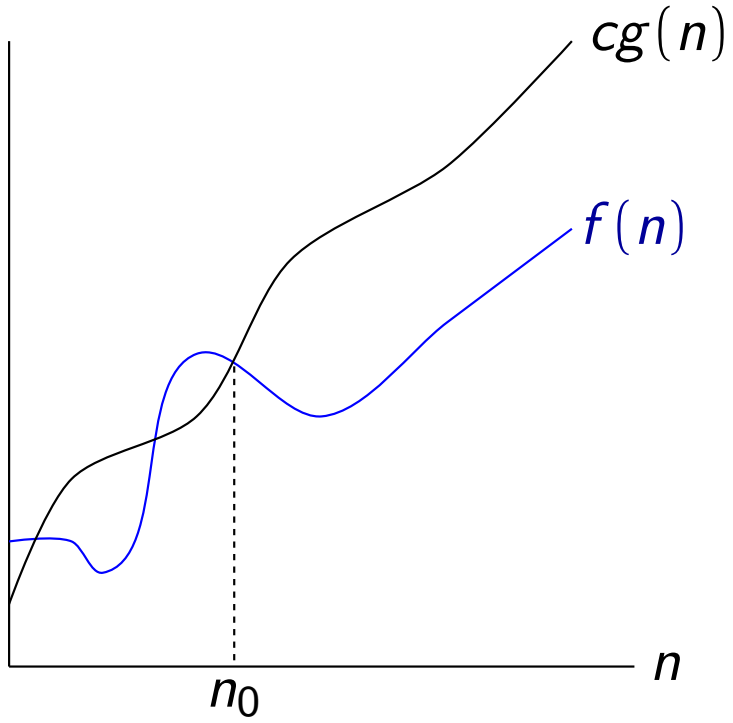
Let us take an arbitrary function $f : \mathbb{N} \to \mathbb{R}$. Expressions $O(f)$, $\Omega(f)$, and $\Theta(f)$ denote **sets of functions** of the type $\mathbb{N} \to \mathbb{R}$, where:

- $O(f)$ – the set of all functions that grow at most as fast as f
- $\Omega(f)$ – the set of all functions that grow at least as fast as f
- $\Theta(f)$ – the set of all functions that grow as fast as f

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"

Asymptotic Notation – Symbol O



Definition

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

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

Asymptotic Notation – Symbol O

Remarks:

- c is a posive real number (i.e., $c \in \mathbb{R}$ and $c > 0$)
- n_0 and n are natural numbers (i.e., $n_0 \in \mathbb{N}$ and $n \in \mathbb{N}$)

Asymptotic Notation – Symbol O

Example: Let us consider functions $f(n) = 2n^2 + 3n + 7$ and $g(n) = n^2$.

We want to show that $f \in O(g)$, i.e., $f \in O(n^2)$:

- Approach 1:

  Let us take for example $c = 3$.

  $$cg(n) = 3n^2 = 2n^2 + \frac{1}{2}n^2 + \frac{1}{2}n^2$$

  We need to find some n_0 such that for all $n \geq n_0$ it holds that

  $$2n^2 \geq 2n^2 \qquad \frac{1}{2}n^2 \geq 3n \qquad \frac{1}{2}n^2 \geq 7$$

  We can easily check that for example $n_0 = 6$ satisfies this.

  For each $n \geq 6$ we have $cg(n) \geq f(n)$:

  $$cg(n) = 3n^2 = 2n^2 + \frac{1}{2}n^2 + \frac{1}{2}n^2 \geq 2n^2 + 3n + 7 = f(n)$$

Asymptotic Notation – Symbol O

The example where $f(n) = 2n^2 + 3n + 7$ and $g(n) = n^2$:

- Approach 2:

  Let us take $c = 12$.

  $$cg(n) = 12n^2 = 2n^2 + 3n^2 + 7n^2$$

  We need to find some n_0 such that for all $n \geq n_0$ we have

  $$2n^2 \geq 2n^2 \qquad 3n^2 \geq 3n \qquad 7n^2 \geq 7$$

  These inequalities obviously hold for $n_0 = 1$, and so for each $n \geq 1$ we have $cg(n) \geq f(n)$:

  $$cg(n) = 12n^2 = 2n^2 + 3n^2 + 7n^2 \geq 2n^2 + 3n + 7 = f(n)$$

Asymptotic Notation – Symbol O

Proposition

Let us assume that a and b are constants such that $a > 0$ and $b > 0$, and k and ℓ are some arbitrary constants where $k \geq 0$, $\ell \geq 0$ and $k \leq \ell$.

Let us consider functions

$$f(n) = a \cdot n^k \qquad\qquad g(n) = b \cdot n^\ell$$

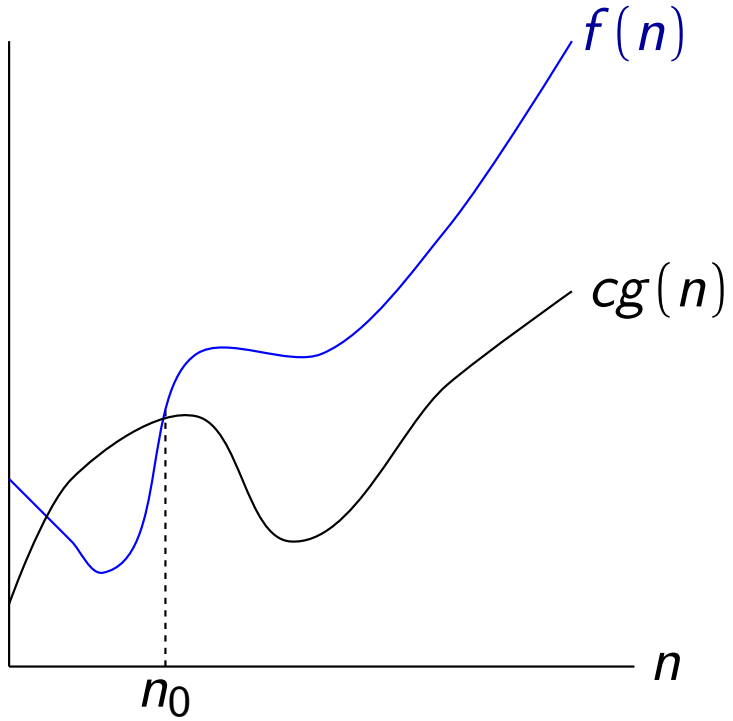
For each such functions f and g it holds that $f \in O(g)$:

Proof: Let us take $c = \frac{a}{b}$.

Because for $n \geq 1$ we obviously have $n^k \leq n^\ell$ (since $k \leq \ell$), for $n \geq 1$ we have

$$c \cdot g(n) = \frac{a}{b} \cdot g(n) = \frac{a}{b} \cdot b \cdot n^\ell = a \cdot n^\ell \geq a \cdot n^k = f(n)$$

Asymptotic Notation – Symbol Ω



Definition

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

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

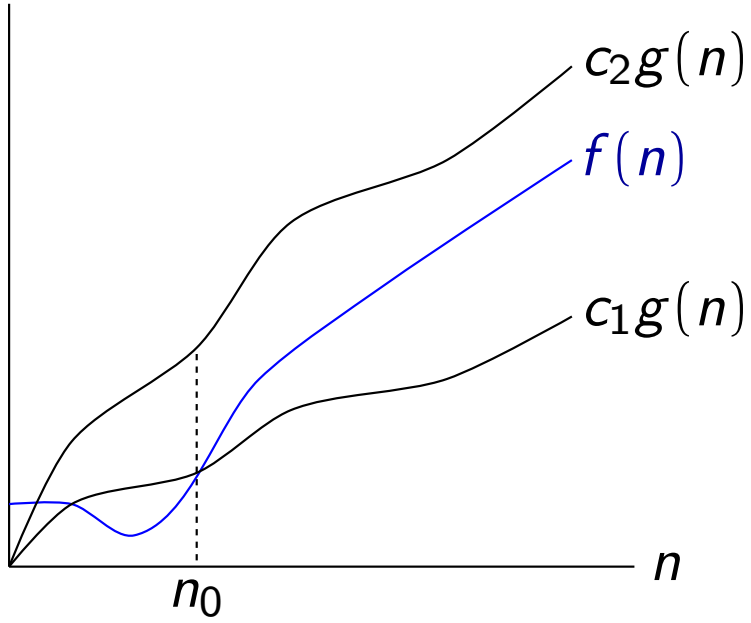
Asymptotic Notation – Symbol Ω

It is not difficult to prove the following proposition:

For arbitrary functions f and g we have:

$$f \in O(g) \qquad \text{iff} \qquad g \in \Omega(f)$$

Asymptotic Notation – Symbol Θ



Definition

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

$$(\exists c_1 > 0)(\exists c_2 > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\big(c_1\, g(n) \leq f(n) \leq c_2\, g(n)\big).$$

Asymptotic Notation – Symbol Θ

For arbitrary functions f and g we have:

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

$$f \in \Theta(g) \quad \text{iff} \quad f \in O(g) \text{ and } g \in O(f)$$

$$f \in \Theta(g) \quad \text{iff} \quad g \in \Theta(f)$$

Asymptotic Notation

For arbitrary functions f, g, and h we have:

- if $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$
- if $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$
- if $f \in \Theta(g)$ and $g \in \Theta(h)$ then $f \in \Theta(h)$

Asymptotic Notation

Examples:

$n \in O(n^2)$
$1000n \in O(n)$
$2^{\log_2 n} \in \Theta(n)$
$n^3 \notin O(n^2)$
$n^2 \notin O(n)$
$n^3 + 2^n \notin O(n^2)$

$n^3 \in O(n^4)$
$0.00001n^2 - 10^{10}n \in \Theta(10^{10}n^2)$
$n^3 - n^2 \log_2^3 n + 1000n - 10^{100} \in \Theta(n^3)$
$n^3 + 1000n - 10^{100} \in O(n^3)$
$n^3 + n^2 \notin \Theta(n^2)$
$n! \notin O(2^n)$

Asymptotic Notation

- There are pairs of functions f and g such that

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

for example

$$f(n) = n \qquad\qquad g(n) = n^{1+\sin(n)}.$$

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

Asymptotic Notation

- For any pair of functions f, g we have:
  - $\max(f, g) \in \Theta(f + g)$
  - if $f \in O(g)$ then $f + g \in \Theta(g)$
- For any functions f_1, f_2, g_1, g_2 we have:
  - if $f_1 \in O(f_2)$ and $g_1 \in O(g_2)$ then $f_1 + g_1 \in O(f_2 + g_2)$ and $f_1 \cdot g_1 \in O(f_2 \cdot g_2)$
  - if $f_1 \in \Theta(f_2)$ and $g_1 \in \Theta(g_2)$ then $f_1 + g_1 \in \Theta(f_2 + g_2)$ and $f_1 \cdot g_1 \in \Theta(f_2 \cdot g_2)$

Asymptotic Notation

- 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 $O(g)$, $\Omega(g)$, and $\Theta(g)$ denote certain sets of functions.

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

- an expression $O(g)$, $\Omega(g)$ or $\Theta(g)$ 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 numbers of executions of individual instructions for some input w (i.e., the instruction I_i is performed m_i times for the input w), then the total number of executed instructions for input w is
$$T(n) = c_1 m_1 + c_2 m_2 + \cdots + c_k m_k.$$
- Let us consider functions $f_1, f_2, \ldots, f_k$, where $f_i : \mathbb{N} \to \mathbb{R}$, and where $f_i(n)$ is the maximum of numbers of executions of instruction I_i for all inputs of size n.
- Obviously, $T \in \Omega(f_i)$ for any function f_i.
- It is also obvious that $T \in O(f_1 + f_2 + \cdots + f_k)$.

Complexity of Algorithms

- Let us recall that if $f \in O(g)$ then $f + g \in O(g)$.
- If there is a function f_i such that for all f_j, where $j \neq i$, we have $f_j \in O(f_i)$, then
$$T \in O(f_i).$$
- 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 $f_i(n)$ times for an input of size n), since we have
$$T \in \Theta(f_i).$$

Complexity of Algorithms

Example: In the analysis of the complexity of the searching of a number in a sequence we obtained

$$f(n) = an + b\,.$$

If we would not like to do such a detailed analysis, we could deduce that the time complexity of the algorithm is $\Theta(n)$, because:

- The algorithm contains only one cycle, which is performed $(n-1)$ times for an input of size n, the number of iterations of the cycle is in $\Theta(n)$.
- Several instructions are performed in one iteration of the cycle. The number of these instructions is bounded from both above and below by some constant independent on the size of the input.
- Other instructions are performed at most once, and so they contribute to the total running time by adding a constant.

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
        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 $(j-1)$ 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 have computed the total number of executions of the cycle **while** accurately.

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