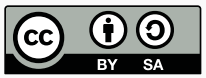
VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Fundamentals of the Analysis of Algorithm Efficiency

Jiří Dvorský, Ph.D.

Presentation status to date February 24, 2025

Department of Computer Science
VSB – Technical University of Ostrava



Lecture outline

Fundamentals of the Analysis of Algorithm Efficiency

- Basics of algorithm complexity analysis
- Worst, Best and Average Case
- Asymptotic Notation of Complexity
- Analysis of Non-Recursive Algorithms
- Analysis of Recursive Algorithms

Fundamentals of the Analysis of Algorithm Efficiency

Basics of algorithm complexity analysis

Algorithm analysis

What to analyze?

- correctness
- time complexity
- space complexity
- optimality

Possible approaches

- empirical and
- theoretical

Time and Space Complexity of an Algorithm

- **Time complexity** – how long the algorithm will run.
- **Space complexity** – how much extra memory the algorithm will need in addition to the storage required for the data itself.
- Previously, both resources were critical.
- Thanks to advances in computing technology, memory is relatively abundant.
- We will examine time complexity – significant progress can be made here.
- It turns out that space complexity can be studied using the same apparatus as time complexity.

Measuring Input Size

- Trivial observation – larger data usually takes an algorithm longer to process.
- We introduce the parameter n denoting the size of the input data, which represents for example:
  - searching in a list, array – length of the array
  - evaluating the polynomial $p(x) = a_n x^n + \cdots + a_1 x + a_0$ at point x – degree of the polynomial
  - multiplying matrices of type $n \times n$ – dimension of the matrix. The actual number of input numbers is n^2, but this still depends on n
  - spell checking – number of characters or number of words, depending on what the algorithm works with

Measuring Input Size (cont.)

- primality testing – the input is always a single number, the running time depends on the size of the number (compare testing 2^3 and 2^{64}), the input size will be the number of bits required to write the number

$$n = \lfloor \log_2 a \rfloor + 1 \tag{1}$$

- graph problems – number of vertices and/or number of edges – here we already have two parameters

Empirical Measurement of Complexity

- We provide suitable input data and measure the program's running time in standard units of time.
- Disadvantages:
  - Dependence on specific hardware, implementation method, and compiler.
  - We want to measure algorithm complexity – we do not have the means to capture the aforementioned influences.
  - Hardware development – does this mean algorithms are accelerating? No, they remain the same.
  - The number of operations performed by the program can be difficult to determine.
  - We want to avoid implementation – after all, we are examining algorithms.

Time complexity of the algorithm

Time complexity of the algorithm will be expressed (measured) by **the number of performed basic operations** with respect to (as a function of) **the size of the input n**:

$$T(n) \approx c_{op} C(n),$$

where

- n is the size of the input,
- $T(n)$ is the running time of the algorithm,
- c_{op} is the time to perform one basic operation and
- $C(n)$ is the number of basic operations.

Basic Operations

Typical operations for a given algorithm that significantly contribute to the overall “running time” of the algorithm.

Problem	Input size	Basic operation
Searching for an element in a list	Number of elements in the list	Comparing elements
Matrix multiplication	Matrix dimensions	Arithmetic operations (multiplication)
Primality testing	Number of bits of the number	Dividing numbers
Graph problems	Number of vertices and/or edges	Processing a vertex or traversing an edge

Order of Growth of Complexity

Regarding the relationship

$$T(n) \approx c_{op} C(n),$$

we must approach it with caution, because

1. $C(n)$ does not take into account the influence of operations other than basic ones and
2. c_{op} cannot be reliably determined.

We understand this relationship as a **reasonable estimate of the algorithm's running time**, except for extremely small n.

Order of Growth of Complexity (cont.)

Problem

How many times faster will my algorithm run on a computer that is **10×** faster than my current computer?

Solution

Of course, **10×**, c_{op} is one-tenth.

Order of Growth of Complexity (cont.)

Problem

How many times longer will my algorithm run for a twice-as-large input when $C(n) = \frac{1}{2}n(n-1)$?

Solution

We approximate from above the number of operations $C(n)$

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$$

However, in the context of order of growth, lower-order terms like $-\frac{1}{2}n$ are typically ignored. Thus,

$$C(n) = \frac{1}{2}n^2$$

Order of Growth of Complexity (cont.)

and

$$C(2n) = \frac{1}{2}(2n)^2 = 2n^2$$

Therefore,

$$\frac{C(2n)}{C(n)} = \frac{2n^2}{\frac{1}{2}n^2} = 4$$

Order of Growth of Complexity (cont.)

Remarks

- The base of the logarithm is not significant: $\log_a n = \log_a b \cdot \log_b n$.
- A computer with a speed of 10^{12} (one trillion) operations per second would take approximately 40 billion years to perform $2^{100} \approx 1.3 \cdot 10^{30}$ operations. The age of the Earth is approximately 4.4 billion years.
- We will not even consider performing **100!** operations...

Algorithms with exponential or factorial order of complexity are only usable for very small input sizes!

Order of Growth of Complexity (cont.)

Problem

How many times longer will my algorithm run for a twice-as-large input, for algorithms with different orders of growth?

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
$2n$	$+1$	$2\times$	$\approx 2\times$	$4\times$	$8\times$	$(\dots)^2$	n/a

because

$$\log_2(2n) = \log_2 2 + \log_2 n = 1 + \log_2 n$$

$$2^{2n} = (2^n)^2$$

Fundamentals of the Analysis of Algorithm Efficiency

Worst, Best and Average Case

Worst, Best and Average Case

- The number of basic operations is expressed as a function with one parameter n, the input size.
- Some algorithms may have different numbers of basic operations even for the same n, such as the linear search algorithm.

Input : Array $A[0 \dots n-1]$ and the target element x
Output: Index of the first occurrence of element x in array A, otherwise -1

```
1 for i ← 0 to n - 1 do
2 |   if A[i] = x then
3 |   |   return i;
4 |   end
5 end
6 return -1;
```

Worst, Best and Average Case (cont.)

Significant numbers of basic operations:

- $C_{worst}(n)$ – worst case, highest number of operations
- $C_{best}(n)$ – best case, lowest number of operations
- $C_{avg}(n)$ – average case, average number of operations.

Worst-case scenario $C_{worst}(n)$

- We analyze the algorithm and look for an input of size n that results in the maximum possible number of operations.
- The worst-case scenario provides an upper bound on complexity, all other cases are either the same or better.
- A low number of operations in the worst case – good news.

Example

Linear search: element x in array A is not found or is found at the end, thus $C_{worst}(n) = n$.

Best-case scenario $C_{best}(n)$

- Generally, we seek an input of size n for which the algorithm performs the smallest number of operations.
- The average best-case scenario is not as crucial as the worst-case scenario.
- Inputs are ”similar” and ”close” to the best case. Sorting nearly sorted sequences.
- A best-case scenario with a ”frightening” number of operations – generally bad news and ”final” for the algorithm. But for an encryption algorithm, a ”frightening” number of cryptanalysis operations is necessary even in the best case.

Example

Linear search: element x is the first element in array A, $C_{best}(n) = 1$.

Average case $C_{avg}(n)$

- Number of operations in the average, “typical”, “random” case (best and worst cases are extremes).
- It is not the average of the best and worst case!
- We must take into account the probabilities of individual possible inputs of size n.
- Analysis of the average case is thus more complicated than the previous two.
- There are algorithms where the worst and average number of operations differ significantly, for example QuickSort.

Average Case $C_{avg}(n)$ – Linear Search

Assumptions

1. probability of successful search p, where $0 \leq p \leq 1$
2. probability of finding at all positions in the array is the same and equals $\frac{p}{n}$

Average Case $C_{avg}(n)$ – Linear Search (cont.)

Successful Search

- finding at the first position – one comparison with probability $\frac{p}{n}$,
- finding at the second position – two comparisons with probability $\frac{p}{n}$, and so on, thus

$$1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}$$

Average Case $C_{avg}(n)$ – Linear Search (cont.)

Unsuccessful Search

- probability of failure is $1 - p$ and we perform n comparisons, i.e., $n(1 - p)$

From this

$$\begin{aligned} C_{avg}(n) &= \left(1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}\right) + n(1 - p) \\ &= \frac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1 - p) \\ &= \frac{p}{n}\left[\frac{1}{2}n(n + 1)\right] + n(1 - p) \\ &= \frac{1}{2}p(n + 1) + n(1 - p) \end{aligned}$$

Average Case $C_{avg}(n)$ – Linear Search (cont.)

Analysis

- always successful search, $p = 1$ and thus $C_{avg}(n) = \frac{1}{2}(n + 1)$
- unsuccessful search, $p = 0$ and thus $C_{avg}(n) = n$

Amortized Complexity

- We do not examine a single, isolated run of the algorithm, but rather examine a “set” of runs with different inputs of the same size.
- We are interested in the total number of operations for the set.
- The number of operations for one input from the set may be high, but it is balanced, “amortized” by a significantly smaller number of operations for other inputs from the set.
- For example, one of the inputs causes a significant change in the data structure, making the processing of subsequent inputs easier.
- In industry, for example, the purchase of an expensive machine is amortized by cheaper production of products.

Sources for Independent Study

- Book [1], chapter 2.1, pages 42 – 51
- Book [2], chapter 2.2, pages 25 – 34 partially

Fundamentals of the Analysis of Algorithm Efficiency

Asymptotic Notation of Complexity

Big O Notation

Definition

Let us have functions $t(n)$ and $g(n)$, where $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. We say that function $t(n)$ belongs to $O(g(n))$, if there exists a positive non-zero real constant c and a natural number $n_0 \geq 0$ such that

$$t(n) \leq cg(n)$$

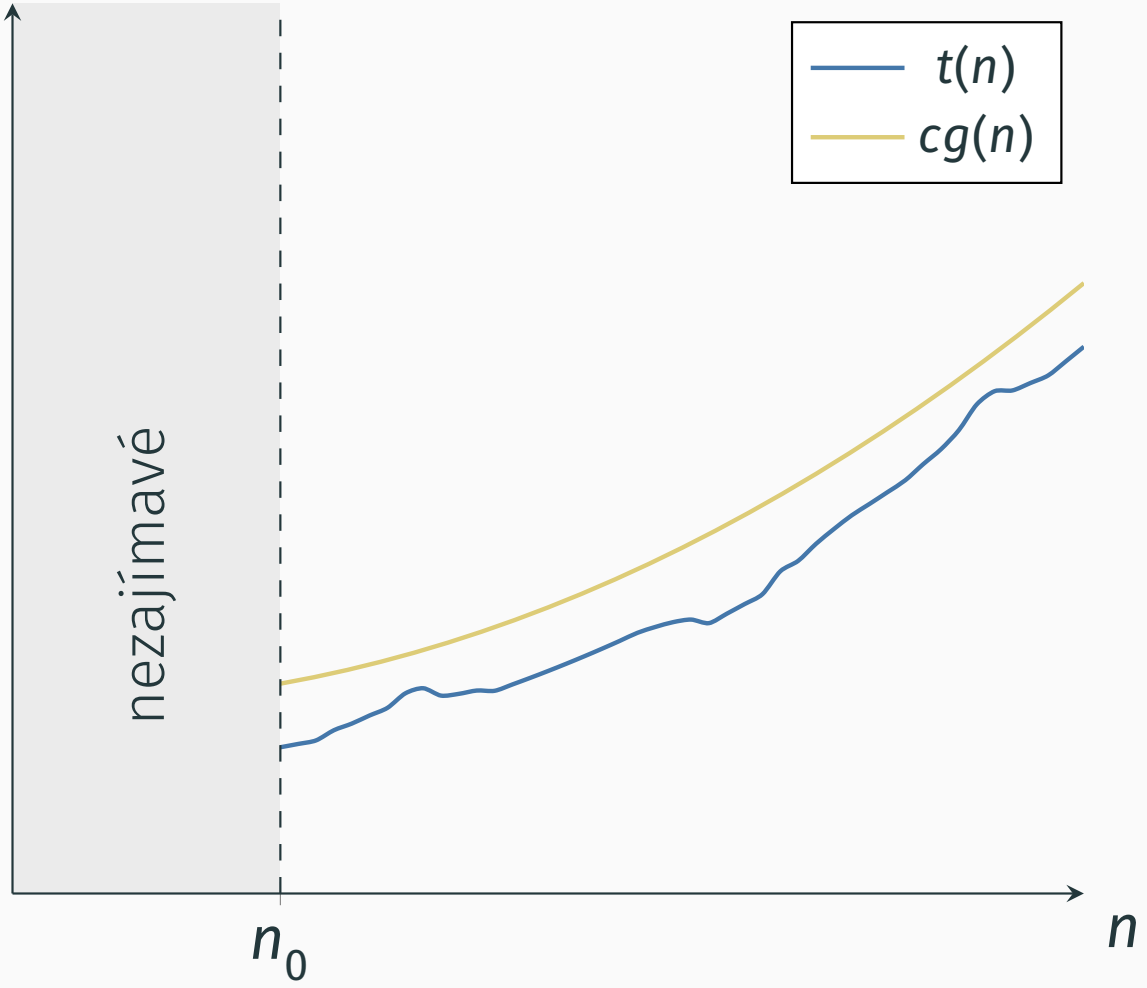
for all $n \geq n_0$.

Remark

Instead of saying "$t(n)$ belongs to $O(g(n))$", we can say that "$t(n)$ is of order $O(g(n))$".

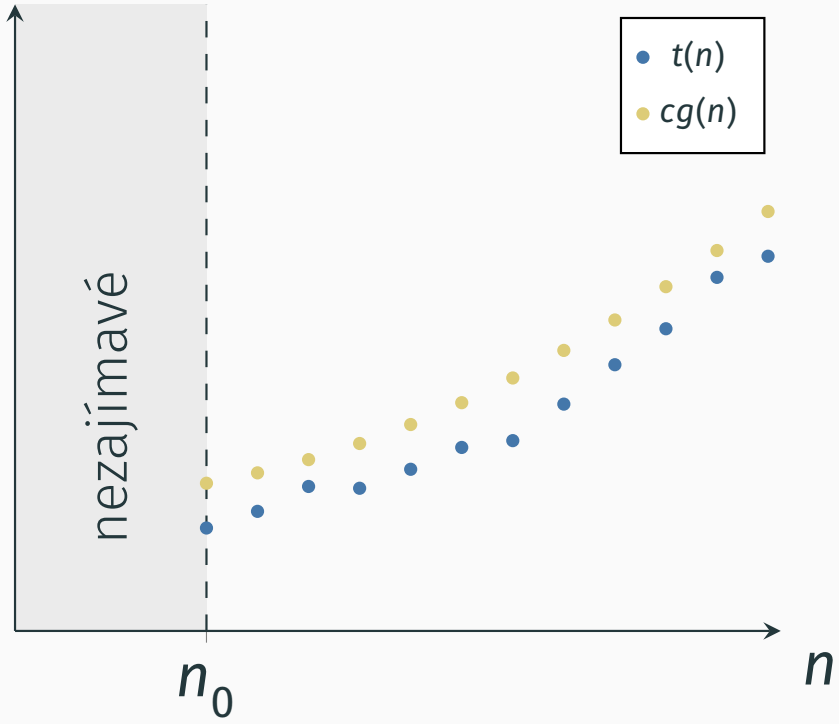
Big O notation graphically

$t(n) \in O(g(n))$



Big O notation – formally correct graph

Formally, the domain of definition and the range of values of functions $t(n)$ and $g(n)$ are natural numbers ⇒ the graph should consist only of points, not curves.



If we interpolate the points with a curve ⇒ we obtain continuous functions ⇒ we can use mathematical analysis (limits, derivatives, etc.) for calculations.

Big O notation – example 1

Problem statement

Prove that $3n + 7 \in O(n)$.

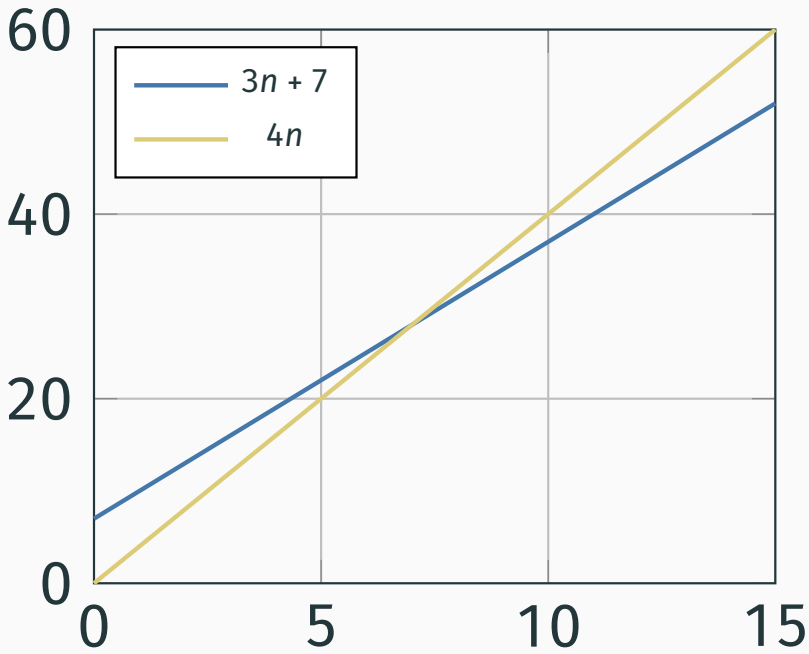
Solution

1. We seek constants c and n_0 such that

$$3n + 7 \leq cn$$

holds for all $n \geq n_0$.

2. It is clear that necessarily $c > 3$. If we choose, for example, $c = 4$, then $n_0 = 7$.



Big O notation – example 2

Problem statement

Prove that $3n + 7 \in O(n^2)$.

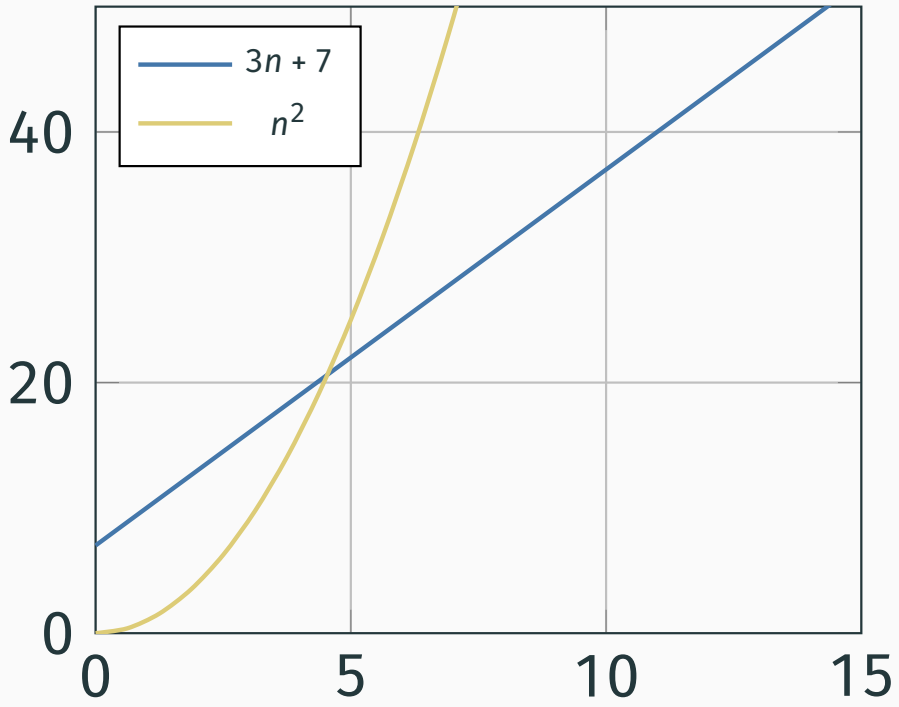
Solution

1. We are looking for constants c and n_0 such that

$$3n + 7 \leq cn^2$$

holds for all $n \geq n_0$.

2. If we choose $c = 1$, then $n_0 = 5$.



Big O notation – example 3

Problem statement

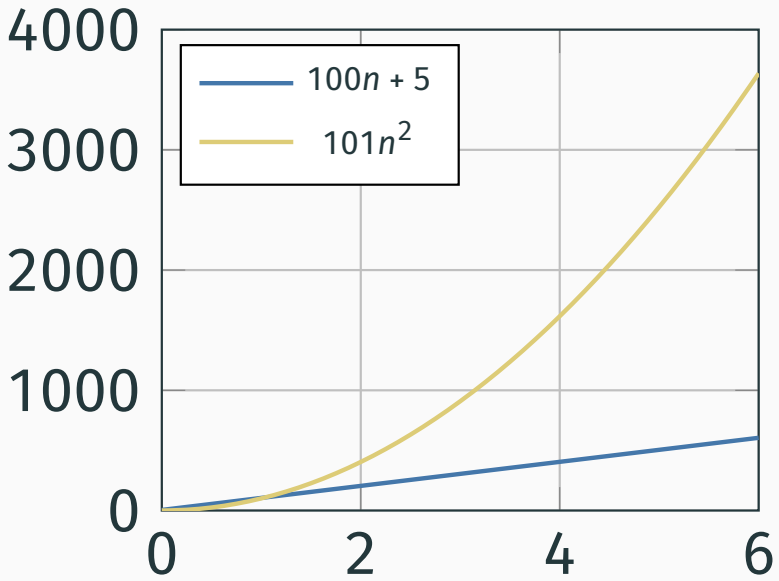
Prove that $100n + 5 \in O(n^2)$.

Solution

1. It holds that $100n + 5 \leq 100n + n$ for all $n \geq 5$.
2. Furthermore, it holds that $101n \leq 101n^2$.
3. From this

   $$100n + 5 \leq 101n \leq 101n^2$$

   and thus $c = 101$ and $n_0 = 5$.



Big O notation – example 3 (cont.)

The proof can also be conducted as follows:

$$100n + 5 \leq 100n + 5n = 105n$$

for all $n \geq 1$. This implies that

$$105n \leq 105n^2$$

and thus $c = 105$ and $n_0 = 0$.

The definition of Big O notation does not say anything about the uniqueness of the values c and n_0, it only requires their existence.

Omega notation

Definition

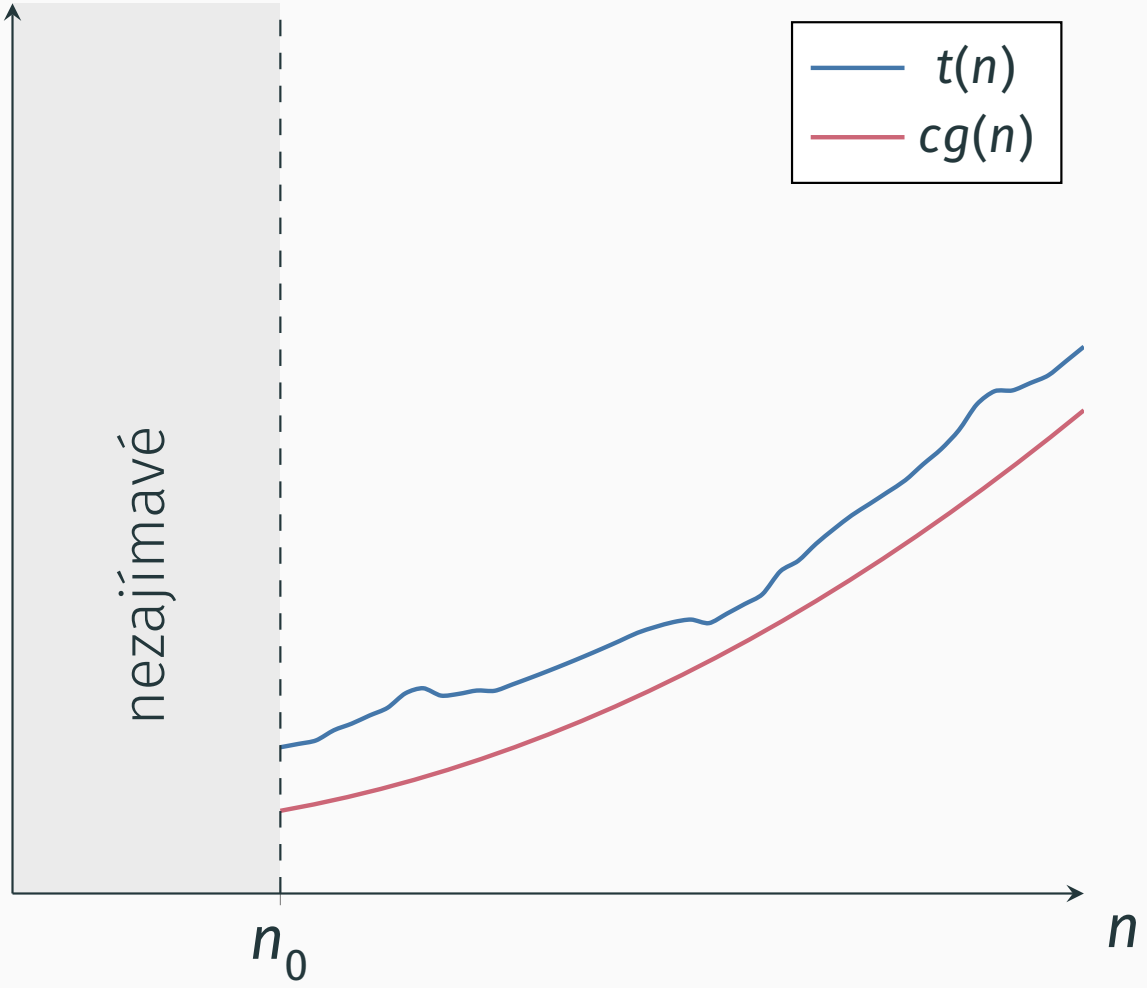
Given functions $t(n)$ and $g(n)$, where $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. We say that function $t(n)$ belongs to $\Omega(g(n))$, if there exists a positive non-zero real constant c and a natural number $n_0 \geq 0$ such that

$$t(n) \geq cg(n)$$

for all $n \geq n_0$.

Lower bound notation graphically

$$t(n) \in \Omega(g(n))$$



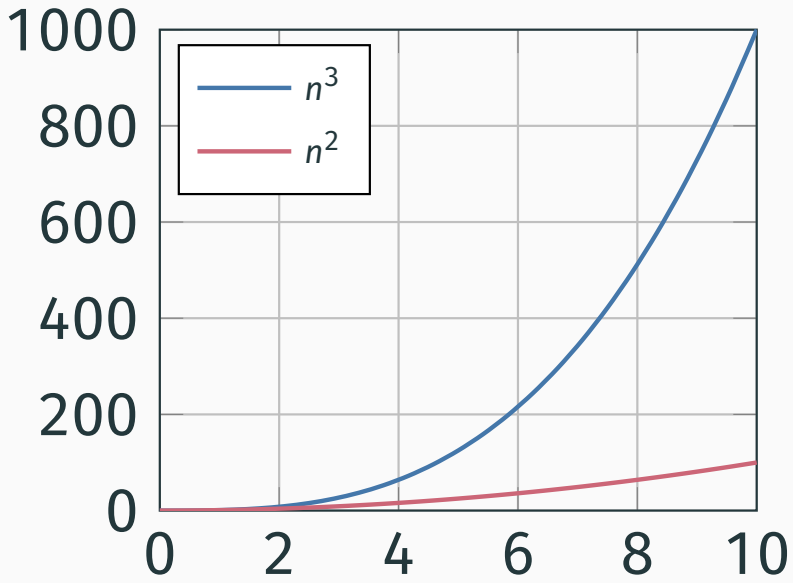
Ω-notation – example 1

Problem statement

Prove that $n^3 \in \Omega(n^2)$.

Solution

1. Clearly, it holds that $n^3 \geq n^2$ for all $n \geq 0$.
2. Thus we can choose $c = 1$ and $n_0 = 0$.



Omega notation – example 2

Problem statement

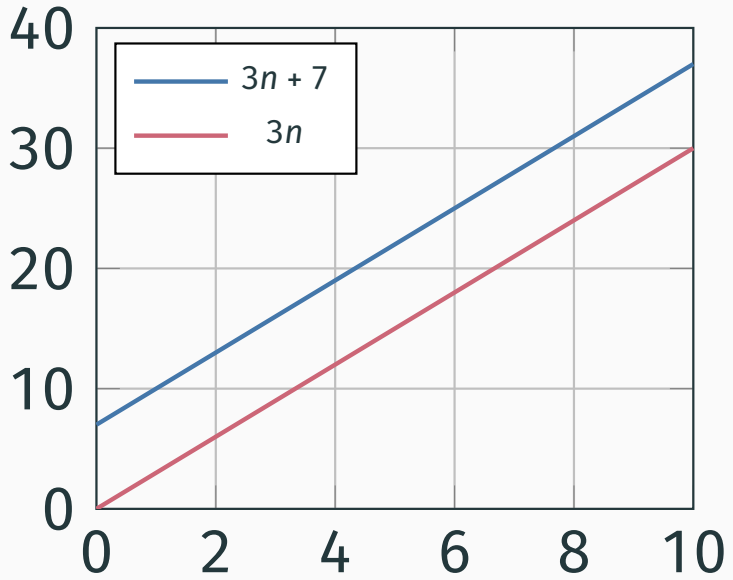
Prove that $3n + 7 \in \Omega(n)$.

Solution

1. We are looking for constants c and n_0 such that

$$3n + 7 \geq cn$$

   holds for all $n \geq n_0$.
2. The expression $3n + 7 \geq 3n$ is valid for all $n \geq 0$, so $c = 3$ and $n_0 = 0$.



Theta notation

Definition

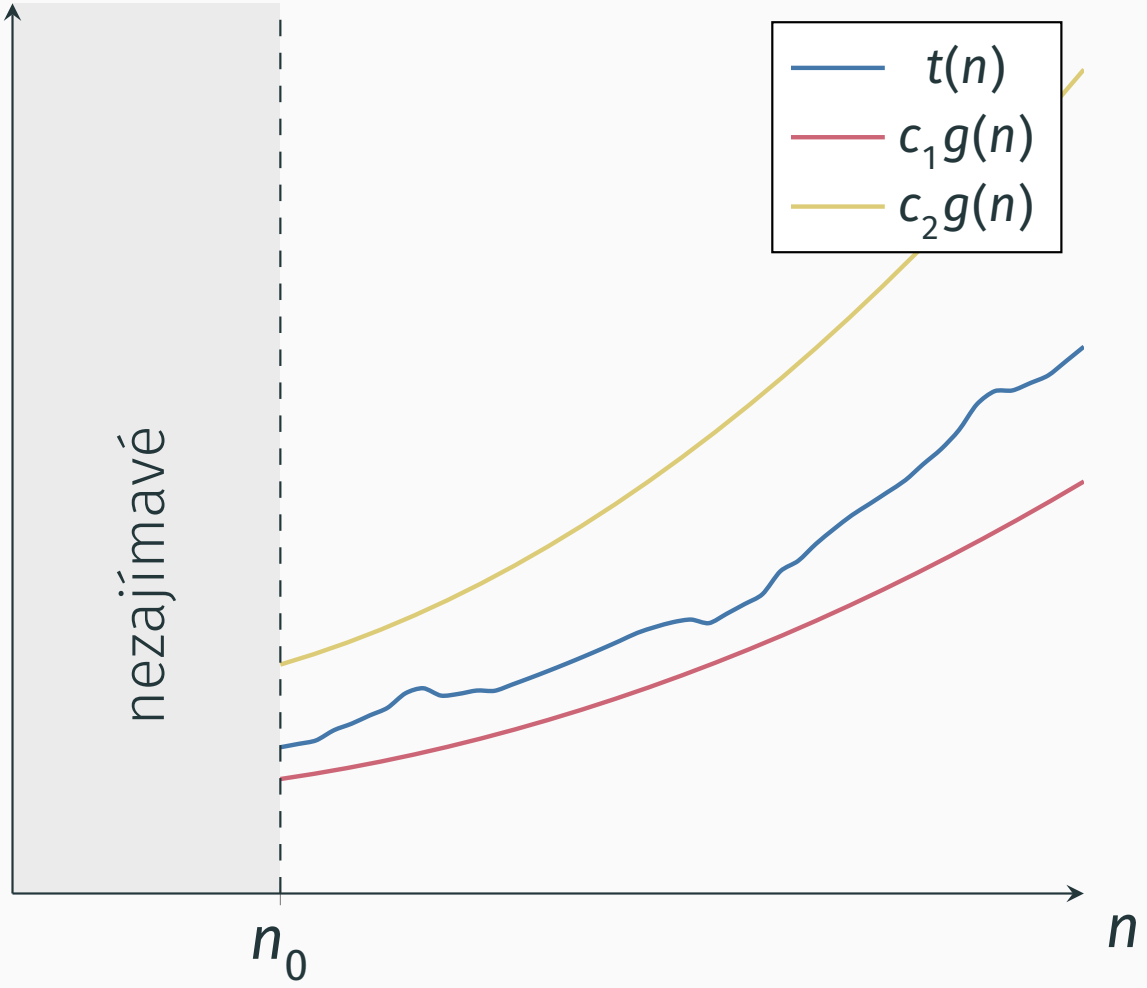
Given functions $t(n)$ and $g(n)$, where $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. We say that the function $t(n)$ belongs to $\Theta(g(n))$, if there exist positive nonzero real constants c_1, c_2 and a natural number $n_0 \geq 0$ such that

$$c_1 g(n) \leq t(n) \leq c_2 g(n)$$

for all $n \geq n_0$.

Theta notation graphically

$$t(n) \in \Theta(g(n))$$



Θ-notation – example

Problem Statement

Prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$.

Solution

1. First, we prove the right inequality $t(n) \leq c_2 g(n)$ (upper bound)

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$$

   for all $n \geq 0$.

Θ-notation – example (cont.)

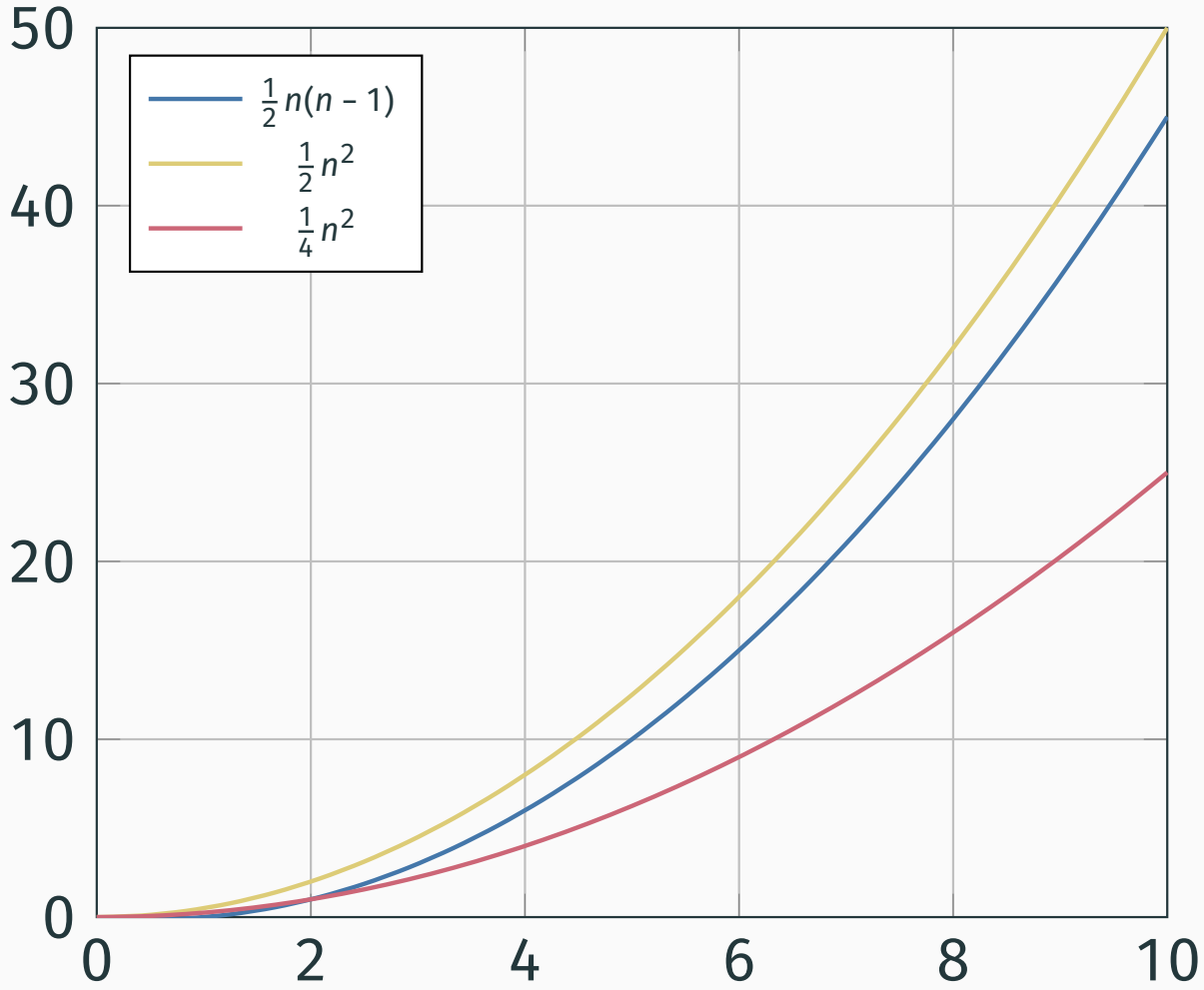
2. The left inequality $c_1 g(n) \leq t(n)$ (lower bound) can be proven as follows:

$$\begin{aligned} t(n) = \frac{1}{2}n(n-1) &= \frac{1}{2}n^2 - \frac{1}{2}n \\ &\geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \\ &\geq \frac{1}{2}n^2 - \frac{1}{4}n^2 \\ &\geq \frac{1}{4}n^2 \end{aligned}$$

   In summary, $\frac{1}{4}n^2 \leq \frac{1}{2}n(n-1)$ for all $n \geq 2$.

3. From the previous inequalities, it follows that $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$, and $n_0 = 2$.

Θ-notation – example (cont.)



Properties of asymptotic notation

Basic properties:

1. $f(n) \in O(f(n))$
2. $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$
3. $f(n) \in O(g(n)) \land g(n) \in O(h(n)) \implies f(n) \in O(h(n))$
4. $\Theta(f(n)) = O(f(n)) \land \Omega(f(n))$

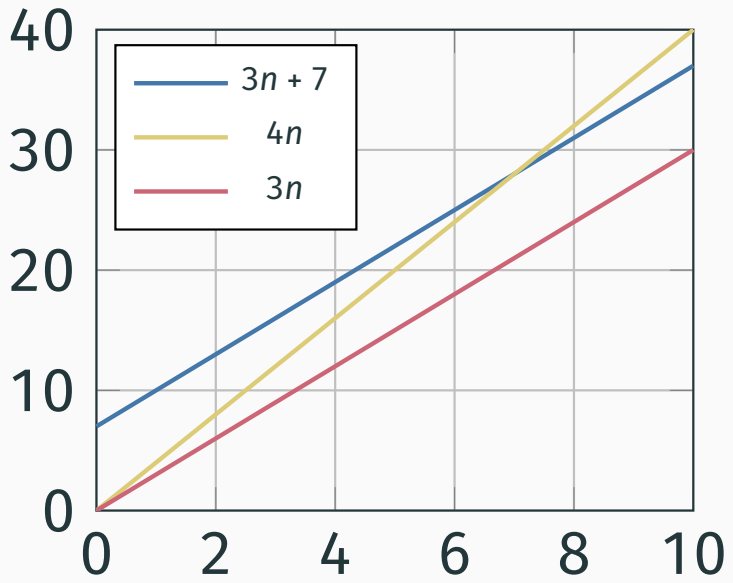
Properties of Asymptotic Notation – Application

Task

Prove that $3n + 7 \in \Theta(n)$.

Solution

1. From previous examples, we know that $3n + 7 \in O(n)$ and simultaneously $3n + 7 \in \Omega(n)$.
2. Therefore, it holds that $3n + 7 \in \Theta(n)$.
3. Specifically, $c_1 = 3$, $c_2 = 4$ and $n_0 = 7$.



Properties of Asymptotic Notation – Computing Complexity

- Algorithm A consists of parts A_1 and A_2.
- The parts of the algorithm are executed sequentially, i.e., after completing A_1, A_2 begins execution.
- The complexity of part A_1 is $t_1(n) \in O(g_1(n))$, the complexity of part A_2 is $t_2(n) \in O(g_2(n))$.
- The question is – what is the overall complexity of algorithm A?

Properties of asymptotic notation – auxiliary lemma

Lemma

Let us have arbitrary real numbers a_1, a_2, b_1, b_2. *Then the following holds:*

$$a_1 \leq b_1 \wedge a_2 \leq b_2 \implies a_1 + a_2 \leq 2 \max(b_1, b_2).$$

Properties of asymptotic notation – auxiliary lemma (cont.)

Proof.

From the assumption, we know that

$$\begin{array}{rcl} a_1 & \leq & b_1 \\ a_2 & \leq & b_2 \\ \hline a_1 + a_2 & \leq & b_1 + b_2. \end{array}$$

Furthermore, it holds that

$$b_1 + b_2 \leq 2 \max(b_1, b_2).$$

From this, we obtain

$$a_1 + a_2 \leq b_1 + b_2 \leq 2 \max(b_1, b_2).$$

□

Properties of asymptotic notation – computation of complexity

Theorem

If $t_1(n) \in O(g_1(n))$ *and simultaneously* $t_2(n) \in O(g_2(n))$*, then*

$$t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n))).$$

Remark

The same statement can be expressed for Ω and Θ notation.

Properties of asymptotic notation – computation of complexity (cont.)

Proof.

Since $t_1(n) \in O(g_1(n))$, there exists a positive non-zero constant c_1 and a non-negative constant n_1 such that

$$t_1(n) \leq c_1 g_1(n) \qquad \forall n \geq n_1.$$

Similarly,

$$t_2(n) \leq c_2 g_2(n) \qquad \forall n \geq n_2.$$

Properties of asymptotic notation – computation of complexity (cont.)

Proof.

Let $c_3 = \max(c_1, c_2)$ and $n_0 \geq \max(n_1, n_2)$. Then,

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\leq 2c_3 \max(g_1(n), g_2(n)). \end{aligned}$$

Thus, $t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$, since there exist constants $c = 2c_3 = 2\max(c_1, c_2)$ and $n_0 = \max(n_1, n_2)$. □

The overall complexity of an algorithm is determined by the part with the highest complexity.

Properties of asymptotic notation – complexity calculation, example

Problem statement

Test whether two identical values occur in the array.

Solution

1. Sorting the array requires no more than $\frac{1}{2}n(n-1)$ comparisons, i.e., a complexity of class $O(n^2)$.
2. Comparing all pairs of adjacent elements will require $n-1$ comparisons, i.e., a complexity of class $O(n)$.

The overall complexity of the algorithm is therefore $O(\max(n^2, n)) = O(n^2)$.

Utilization of limits for computations

The growth rate of functions can be more easily calculated using limits:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & t(n) \text{ grows slower than } g(n) \\ c & t(n) \text{ grows at the same rate as } g(n) \\ \infty & t(n) \text{ grows faster than } g(n) \end{cases}$$

It is clear that:

$t(n) \in O(g(n)) \quad \Leftrightarrow \quad t(n)$ grows slower or at the same rate as $g(n)$

$t(n) \in \Omega(g(n)) \quad \Leftrightarrow \quad t(n)$ grows at the same rate or faster than $g(n)$

$t(n) \in \Theta(g(n)) \quad \Leftrightarrow \quad t(n)$ grows at the same rate as $g(n)$

Utilization of limits for computations (cont.)

Some useful formulas

L'Hospital's rule

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Using limits for calculations – example I

Compare the growth rate of functions $\frac{1}{2}n(n-1)$ and n^2.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} &= \frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2} \\
&= \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right) \\
&= \frac{1}{2}\left(\lim_{n\to\infty}1-\lim_{n\to\infty}\frac{1}{n}\right) \\
&= \frac{1}{2}(1-0) = \frac{1}{2} > 0
\end{aligned}$$

The functions $\frac{1}{2}n(n-1)$ and n^2 grow at the same rate, so

$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$

.

Utilization of limits for computations – example II

Compare the growth rate of functions $\log_2 n$ and $\sqrt{n}$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} &= \lim_{n\to\infty} \frac{\left(\log_2 n\right)'}{\left(\sqrt{n}\right)'} \\
&= \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = (\log_2 e) \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} \\
&= \log_2 e \lim_{n\to\infty} \frac{2\sqrt{n}}{n} \\
&= 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0
\end{aligned}$$

The function $\log_2 n$ therefore **grows more slowly** than $\sqrt{n}$.

Using limits for computations – example III

Compare the growth rate of the functions $n!$ and 2^n.

$$\begin{aligned}\lim_{n\to\infty}\frac{n!}{2^n} &= \lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}\\ &= \sqrt{2\pi}\lim_{n\to\infty}\sqrt{n}\frac{n^n}{2^n e^n}\\ &= \sqrt{2\pi}\lim_{n\to\infty}\sqrt{n}\left(\frac{n}{2e}\right)^n = \infty\end{aligned}$$

Remarks

- The function $n!$ therefore grows faster than 2^n.
- The definition of Θ-notation does not exclude that $n! \in \Omega(2^n)$, but the limit calculation clearly states that $n!$ grows faster than 2^n

Basic Complexity Classes

Although theoretically there are infinitely many complexity classes, the complexity of most algorithms falls into a few classes.

Class	Name	Note
1	constant	complexity does not depend on the size of the input; very few algorithms
$\log n$	logarithmic	typically algorithms reducing the size of the input by a constant factor; interval halving search
n	linear	algorithms processing a list of n elements; e.g. sequential search
$n \log n$	linearithmic	divide and conquer algorithms; average complexity of QuickSort, MergeSort

Basic Complexity Classes (cont.)

Class	Name	Note
n^2	quadratic	generally algorithms with two nested loops; elementary sorting methods, summing $n \times n$ matrices
n^3	cubic	generally algorithms with three nested loops; multiplying $n \times n$ matrices
2^n	exponential	typically generating all subsets of an n-element set
$n!$	factorial	typically generating all permutations of an n-element set

Influence of the Multiplicative Constant

- The complexity class is given up to a multiplicative constant, which is usually not precisely specified.
- Could an algorithm with a higher complexity class therefore run faster than an algorithm from a better class for some reasonable n? For example:

Algorithm	Running Time
A	n^3
B	$10^6 n^2$

A will be better than B for $n < 10^6$.

- Multiplicative constants usually take on similar, relatively small values.
- **It can be expected that algorithms with lower complexity will be better than those with higher complexity already for moderately large inputs.**

Sources for Independent Study

- Book [1], chapter 2.2, pages 52 – 61
- Book [2], chapters 3.1 and 3.2, pages 49 – 63

Fundamentals of the Analysis of Algorithm Efficiency

Analysis of Non-Recursive Algorithms

Finding the Largest Element in an Array of n Numbers

```
   Input  : Array A[0 ... n - 1] of integers
   Output: Largest element of array A
1  max ← A[0];
2  for i ← 1 to n - 1 do
3  |   if A[i] > max then
4  |   |   max ← A[i];
5  |   end
6  end
7  return max;
```

Finding the Largest Element in an Array of n Numbers (cont.)

Working Procedure

1. Input size – size of array n
2. Basic operation:
   - most frequently performed operations are inside the loop – comparison $A[i] > max$ and assignment $max \leftarrow A[i]$
   - basic operation will be **comparison**, because it
     - is performed in each iteration of the loop,
     - is the key operation for the algorithm, "How many pairs of elements must I compare to find the maximum?"
3. Number of comparisons is the same for all inputs of size n, it is not necessary to distinguish between the best, average, and worst case

Finding the Largest Element in an Array of n Numbers (cont.)

4. Number of basic operations, comparisons, $C(n)$ will be equal to

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

5. **Conclusion**: Finding the largest element in an array of n numbers is a **linear algorithm**.

Finding the largest element in an array of n numbers, all operations

Number of operations	Description
1	assignment $max \leftarrow A[0]$
1	assignment $i \leftarrow 1$
$n - 1$	comparison $i \leq n - 1$
$n - 1$	increment i by 1
$n - 1$	comparison $A[i] > max$
$n - 1$	assignment $max \leftarrow A[i]$
1	return result **return** max
$4(n - 1) + 3 = 4n - 1 \in \Theta(n)$	

Conclusion: Finding the largest element in an array of n numbers is a **linear algorithm**.

General procedure for determining the time complexity of non-recursive algorithms

1. Selection of a parameter, or parameters, representing the size of the input n.
2. Identification of the basic operations of the algorithm (these are the ones in the most nested loop!).
3. Does the number of basic operations depend only on the size of the input? If it depends on something else as well, we must examine the worst, best, and average cases separately.
4. Establishment of a relationship, or relationships, (i.e., "formulas") expressing the number, or numbers, of executions of the basic operations.
5. Simplification of the established relationships and, at least, determination of the order of growth.

Useful Summation Formulas

$$\sum (a_i \pm b_i) = \sum a_i \pm \sum b_i \tag{2}$$

$$\sum c a_i = c \sum a_i \tag{3}$$

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{m} a_i + \sum_{i=m+1}^{n} a_i \tag{4}$$

$$\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1 \tag{5}$$

Specifically

$$\sum_{i=1}^{n} 1 = n \in \Theta(n) \tag{6}$$

Useful Summation Formulas (cont.)

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{1}{2}n(n+1) \approx \frac{1}{2}n^2 \in \Theta(n^2) \tag{7}$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{1}{6}n(n+1)(2n+1) \approx \frac{1}{3}n^3 \in \Theta(n^3) \tag{8}$$

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}, \text{ for } a \neq 1 \tag{9}$$

Specifically

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n) \tag{10}$$

Uniqueness of elements in an array

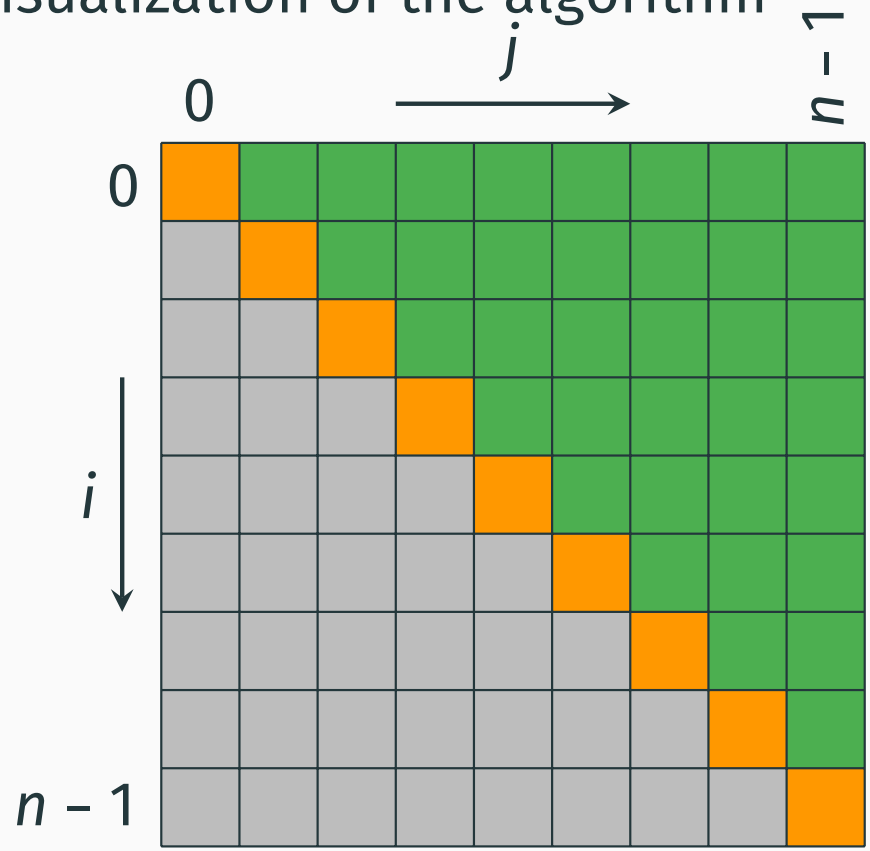
Given is an array of n elements. Our task is to analyze the algorithm that determines whether all elements in the array are mutually distinct, i.e., unique.

Input : Array $A[0 \ldots n-1]$
Output: Returns true if all elements are unique, otherwise returns false

```
1 for i ← 0 to n − 2 do
2 |   for j ← i + 1 to n − 1 do
3 |   |   if A[i] = A[j] then
4 |   |   |   return false;
5 |   |   end
6 |   end
7 end
8 return true;
```

Uniqueness of elements in an array (cont.)

Visualization of the algorithm



Legend

pairs that **must** be tested

an element with itself does not need to be tested

pairs already tested in previous iterations of the cycle

Uniqueness of elements in an array (cont.)

Procedure

1. Input size – size of the array n
2. Basic operation – the most nested cycle contains a single operation, comparison $A[i] = A[j]$
3. Dependence only on n? No, the number of basic operations depends also on whether a duplicate element appears in the array. Thus, we perform analysis of the **worst**, best, and average case.
4. Establishing relationships. For the worst case, it is clear from the inner cycle that premature termination of the cycle must not occur, either:
   4.1 because all elements are unique or

Uniqueness of elements in an array (cont.)

4.2 a duplicate appears only in the last pair.

Thus, we perform:

- one comparison for each iteration of the inner cycle, i.e., $j = i + 1, \ldots, n - 1$
- the outer cycle iterates $n - 1$ times

Establishing relationships. For the worst case, it is clear from the inner cycle that premature termination of the cycle must not occur. Thus, we will perform:

- one comparison for each iteration of the inner cycle, i.e., $j = i + 1, \ldots, n - 1$
- the outer cycle iterates $n - 1$ times

Thus, we perform:

Uniqueness of elements in an array (cont.)

- one comparison for each iteration of the inner cycle, i.e., $j = i + 1, \ldots, n - 1$
- outer cycle runs $n - 1$ times

Uniqueness of elements in an array (cont.)

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \qquad \text{by (2)}$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \qquad \text{by (3) and (7)}$$

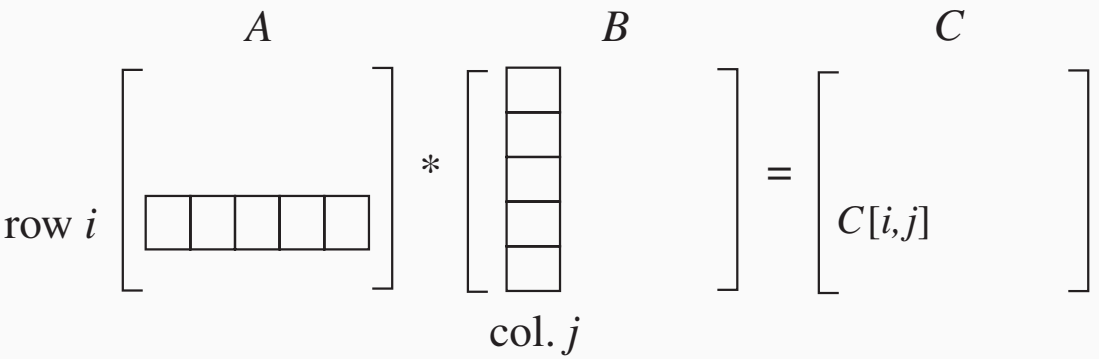
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} \qquad \text{by (5)}$$

$$= \frac{1}{2} n(n-1) \approx \frac{1}{2} n^2 \in \Theta(n^2)$$

Multiplication of Square Matrices

Our task is to perform an analysis of the algorithm for computing the product $C = AB$ of two square matrices A and B of order n.

By definition, the elements of the matrix are equal to the scalar products of the rows of matrix A with the columns of matrix B.



$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

for all

$0 \leq i, j \leq n - 1$

Multiplication of Square Matrices (cont.)

$$C[i,j] = A[i,0] \times B[0,j] + \cdots + A[i,k] \times B[k,j] + \cdots + A[i,n-1] \times B[n-1,j]$$

Input : Two square matrices A and B of order n
Output: Square matrix C of order n

```
1 for each element c_{i,j} of matrix C do
2 |   c_{i,j} = 0;
3 |   for k from 0 to n – 1 do
4 |   |   c_{i,j} = c_{i,j} + a_{i,k} × b_{k,j};
5 |   end
6 end
```

1. The algorithm must compute $n \times n$ elements of matrix C

Multiplication of Square Matrices (cont.)

2. Each element of matrix C is computed as the scalar product of the i-th row of matrix A and the j-th column of matrix B
3. The rows and columns have n elements that must be multiplied
4. Therefore, there are a total of $n^2 \times n = n^3$ multiplications

Multiplication of Square Matrices (cont.)

Informal Procedure

1. The algorithm must compute $n \times n$ elements of matrix C
2. Each element of matrix C is computed as the scalar product of the i-th row of matrix A and the j-th column of matrix B
3. The rows and columns have n elements that must be multiplied
4. Therefore, there are a total of $n^2 \times n = n^3$ multiplications

Multiplication of Square Matrices (cont.)

The running time of the algorithm on a specific computer

$$T(n) \approx c_m M(n) = c_m n^3$$

if we also count additions

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3,$$

where c_m and c_a are the times required for multiplication and addition, respectively, and $A(n)$ is the number of additions, which satisfies $A(n) = M(n)$.

Multiplication of Square Matrices (cont.)

Summary

The running time of the algorithm may vary depending on the specific computer, but **the order of complexity of the algorithm (n^3) remains the same**.

Number of bits in the binary representation of a number

Our task is to analyze the algorithm that for a given natural number n calculates the number of bits necessary for writing the number n in binary.

Input : Natural number n
Output: Number of bits in the binary representation of the number n

```
1 count ← 1;
2 while n > 1 do
3 |   count ← count + 1;
4 |   n ← ⌊n/2⌋;
5 end
6 return count;
```

Number of bits in the binary representation of a number (cont.)

- Input size – one number?
- Basic operation – addition, division, comparison with 1?
- Most importantly, in this case, we need to determine the number of loop iterations. The number of comparisons is one more than the number of loop iterations.
- The value of the number n decreases by half with each loop iteration, leading to the relation

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

  and which corresponds to the relation (1).
- To derive this, we will need to be able to solve recursive equations...

Sources for Independent Study

- Book [1], chapter 2.3, pages 61 – 70

Fundamentals of the Analysis of Algorithm Efficiency

Analysis of Recursive Algorithms

Calculation of Factorial

Our task is to analyze the recursive algorithm that calculates the factorial $n!$ for a given natural number n.

$$n! = \begin{cases} 1 & \text{for } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

```
1 Function F(n)
  |   Input: Natural number n
  |   Result: Result
2 |   if n = 0 then
3 |   |   return 1;
4 |   end
5 |   else
6 |   |   return n · F(n − 1);
7 |   end
8 end
```

Calculation of Factorial (cont.)

- The size of the input is n.
- We need to find a function $M(n)$ that represents the number of multiplications performed by the algorithm.
- The algorithm has a recursive structure, so we can write a recurrence relation for $M(n)$.

Remark

To solve the recurrence relation, we need to find an explicit expression for $M(n)$. We will use the method of backward substitution to solve the recurrence relation.

- The recurrence relation is $M(n) = M(n-1) + 1$ for $n > 0$.

Calculation of Factorial (cont.)

- We need to find an initial condition to make the recurrence relation unique.
- From the algorithm, we can see that when $n = 0$, no multiplications are performed, so $M(0) = 0$.
- Therefore, the complete recurrence relation is

$$\begin{aligned} M(n) &= M(n-1) + 1 \text{ for } n > 0 \\ M(0) &= 0 \end{aligned}$$

Calculation of Factorial (cont.)

- We will solve the recurrence relation using backward substitution. Substituting $M(n-1) = M(n-2) + 1$ into $M(n) = M(n-1) + 1$, we get

$$M(n) = [M(n-2) + 1] + 1 = M(n-2) + 2$$

Substituting $M(n-2) = M(n-3) + 1$ into the previous equation, we get

$$M(n) = [M(n-3) + 1] + 2 = M(n-3) + 3.$$

Calculation of Factorial (cont.)

We can see a pattern emerging: $M(n) = M(n - i) + i$. Using this formula, we can find an explicit expression for $M(n)$ by setting $i = n$, which gives

$$M(n) = M(0) + n = 0 + n = \boxed{n}.$$

Calculation of Factorial (cont.)

Summary

1. The result $M(n) = n$ was more or less expected.
2. An iterative algorithm performs the same number of multiplications as a recursive algorithm, without the overhead of function calls.
3. However, the approach used to solve the recurrence relation is important and can be applied to other problems.

General procedure for determining the time complexity of recursive algorithms

1. Selection of a parameter, or parameters, representing the size of the input n.
2. Identification of the basic operations of the algorithm.
3. Does the number of basic operations depend only on the size of the input? If it depends on something else as well, we must examine the worst, best, and average cases separately.
4. Construction of a recursive relation and suitable initial conditions, expressing the number of executions of basic operations.

General procedure for determining the time complexity of recursive algorithms (cont.)

5. Simplification of the constructed relations and, at least, determination of the order of growth.

Resources for Independent Study

- Book [1], chapter 2.4, pages 70 – 79

Thanks for your attention