ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Algorithms complexity

Algorithms and data structures

- lectures -

АПС

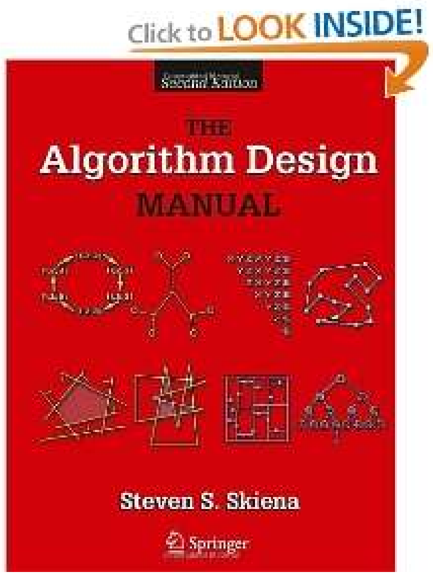
Algorithms and data structures

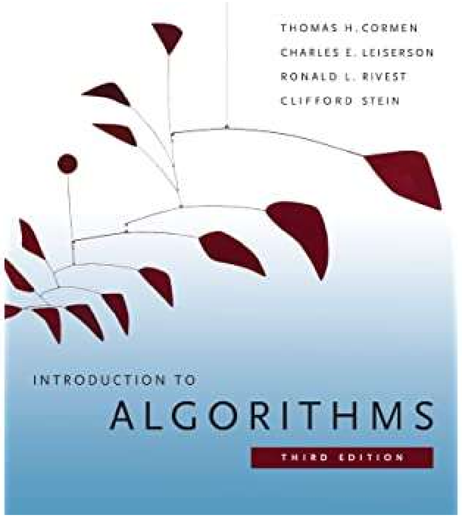
- Professor:
  - Dr. Ilinka Ivanoska
- Assistant:
  - Dr. Ilinka Ivanoska
- Labs:
  - http://code.finki.ukim.mk/

Outline

- Algorithms complexity
- Arrays and lists
- Algorithms techniques - introduction
- Algorithms techniques 2
- One-dimensional data structures (stacks, queues, priority lists)
- Sorting
- Hashing
- Trees (general, binary)
- Trees (AVL, rest)
- Graphs introduction
- Graphs algorithms

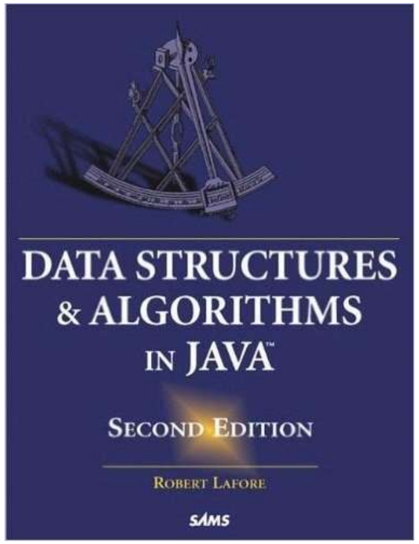
Literature



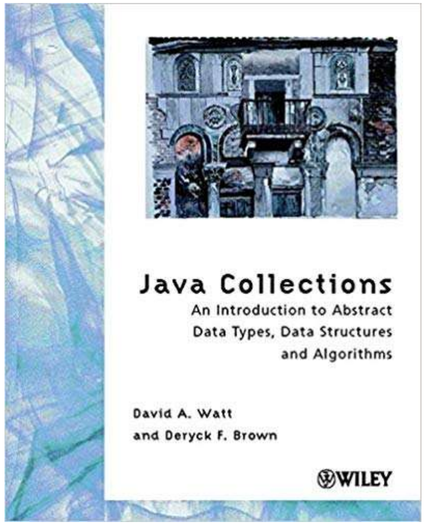


- The Algorithm Design Manual
  - Steven S. Skiena
  - 2008
  - http://www3.cs.stonybrook.edu/~skiena/373/videos/
- Introduction to Algorithms, 3rd Edition (The MIT Press)
  - Cormen, Leiserson, Rivest, Stein
  - 2009

Literature



- Data Structures & Algorithms in Java, 2nd edition
  - Robert Lafore
  - 2002



- *Java Collections*
  - David Watt and Deryck Brown
  - 2001

Activities

- Lectures
  - Major course topics
- Exercises
  - Examples, implementations, solved examples
- Labs
  - Individual work
  - Should be done within the deadline of 1 week after the publication of the assignment.

Consultations and office hours

- Preferred way:
  - Mail ilinka.ivanoska@finki.ukim.mk, KONSULTACII course
  - Greater transparency
  - Opportunity for the students to help each other
- Office hours
  - Virtual - on the KONSULTACII course

Learning management system

- moodle based solution
- http://courses.finki.ukim.mk/
- Learning material, useful links
- Lectures and exercises
- Discussion forums

Exams (under construction)

- The exams (colloquia or final) are executed on a PC.
- They are twofold.
  - The first part is theory of 8 questions from the lectures (in total 10 points).
  - The second part is solving 1 problem on the code system, and additionally 2 problems for a higher grade.
- In order to qualify for the second part, the students must win at least 5 points.
- Every problem is graded with 2 points. If all test cases pass, the solution is given 2 points. If a predefined minimal percentage (which differ per task) is reached, the solution is graded with 1 point.
- Note that the solutions are double checked after the exams.

Grading (final version will be posted separately on the courses page)

- Signature (the right to enter the exams)
  - Lab assignments, individually solved, with correct solutions.
  - Maximum exception is 2 non-submitted or incorrect assignments.
- Grading
  - 2 points from practical problems and 5, 6, 7, 8, 9 or 10 points from theoretical questions -> 6
  - 3 points from practical problems and 5, 6, 7, 8, 9 or 10 points from theoretical questions -> 7
  - 4 points from practical problems and 5, 6, 7, 8, 9 or 10 points from theoretical questions -> 8
  - 5 points from practical problems and 5, 6, 7, 8, 9 or 10 points from theoretical questions -> 9
  - 6 points from practical problems and 5, 6 or 7 points from theoretical questions -> 9
  - 6 points from practical problems and 8, 9 and 10 points from theoretical questions -> 9 with a possibility of 10
- Students with a grade 9 with a possibility of 10, according to these criteria can take an additional oral exam where they will answer for a grade of 10.

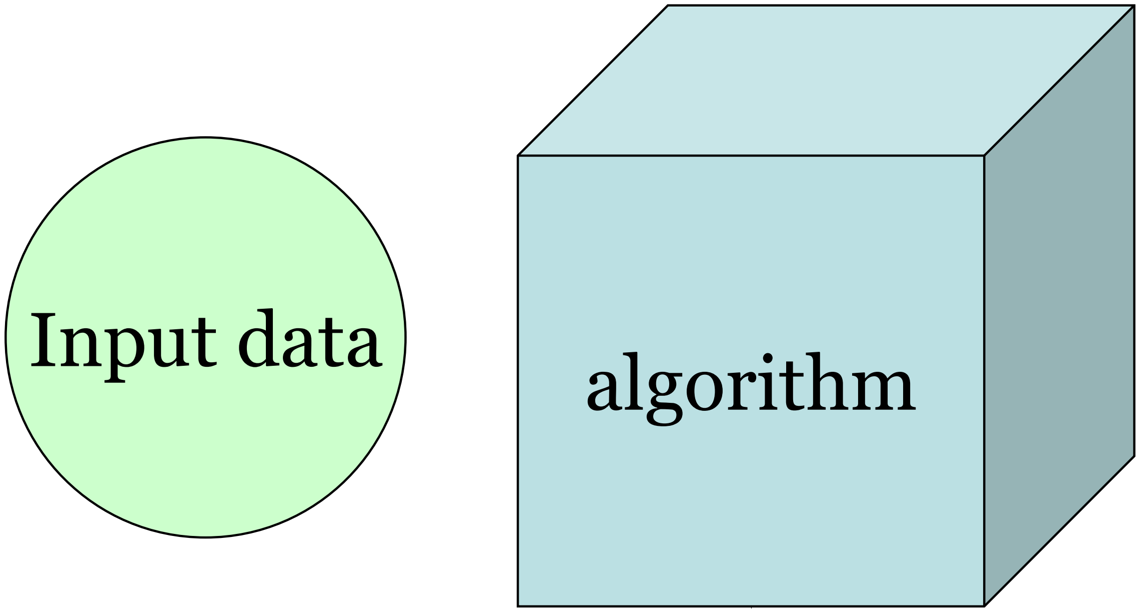
At the end (or the beginning) ...

- Bad programmers worry about the code. Good programmers worry about data structures and their relationships

Linus Torvalds

Algorithms

- Before starting to write the programming code of a problem being solved, the problem solution - **the algorithm** - should be devised



Algorithms

- Algorithm
  - A procedure for problem solution
  - It goes step-by-step
  - Finite number of steps
- Notations for algorithms description
  - Spoken language
  - Pseudo language
  - Programming language

Algorithms should be?

- Algorithms should be:
  - Correct
  - Efficient
  - Easy to implement

Data structures

- The information that should be processed by the program are stored in ***data structures***
  - The algorithms efficiency depends on the data structures choice

Example:
Phone number search for a known address and address search for a known phone number

- Data structures can be
  - Static
    - Change just in values
    - *Arrays or records*
  - Dynamic
    - Change in look, size
    - *Stacks, lists, trees, files*

Algorithms performance

- Actions performed on data structures:
  - Search and find
  - Count
  - Insert (add)
  - Sort
  - Delete
- In order to improve the algorithm, a good one should be written, preferably even the best one (**optimization**)

How to compare algorithms?

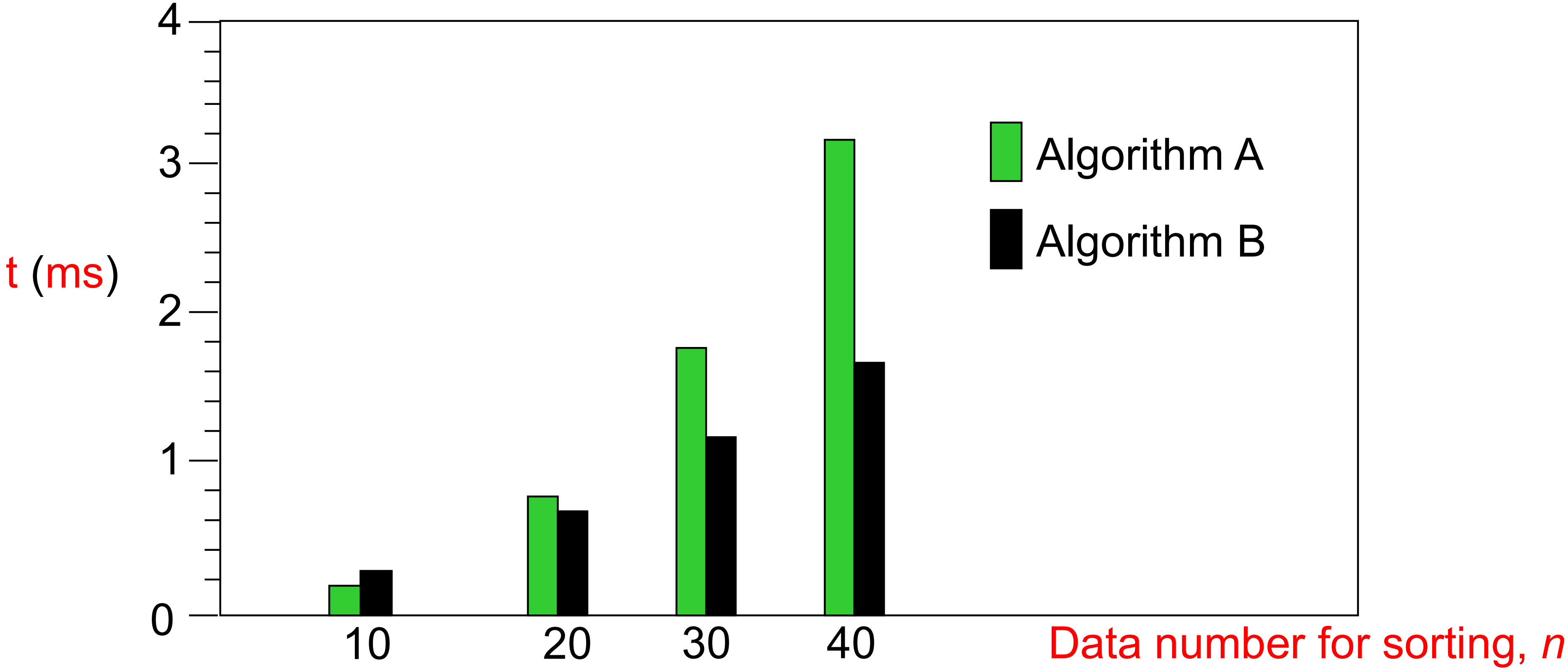
- One problem can be solved in several ways
  - Different algorithms

BUT,

- How to evaluate their efficiency?
- How to compare them?

Example

- Two sorting algorithms comparison:



- Algorithm B is faster than algorithm A.

Here is how you compare algorithms!

- Algorithms comparison is done by a model which is independent of hardware

- Hypothetical computer - Random Access Machine

Algorithms performance

- Random Access Machine
  - Every simple operation (+, *, -, =, if, call) **is executed in one time unit**
  - The cycles (loops), procedures and functions are executed in as many time units as there are iterations
  - There is unlimited memory

Counting steps

Sum(x, n)	Number of executed operations
{	
int sum = 0;	1
for (int i=0; i<=n; ++i)	1+(n+2)+(n+1)=2n+4
sum = sum*x+1;	(n+1)(1+1+1)=3n+3
return sum;	1
}	

Total: T(n) = 5n+9

$$\sum_{i=0}^{n} x^i$$

T(n) is a function that gives the number of executed operations in dependance of the number of input data

How about comparison?

- Lets consider 2 algorithms, *A* and *B*, that solve the same problem.
- We have done analysis of the necessary steps for each of the algorithms
  - $T_A(n)$ and $T_B(n)$
  - n is a measure for the problem size
    - n is the input data size
- Hence, we are left **only** to compare the two functions $T_A(n)$ and $T_B(n)$ and we determine the winner!

BUT!

- We can check for some finite value of n (n_0) which algorithm is faster (has a smaller number of operations)
- In general case we do not know how much n can be.
  - 10, 100, ..100 000…1000 000 ??
- It can be shown that **if** T_A (n) <= T_B (n) **for all** n>=0,
  - The algorithm A is faster, better!
- However,
  - We do not know n beforehand!!
- Therefore – we consider setting an upper asymptotic limit for high problems sizes
  - For big n

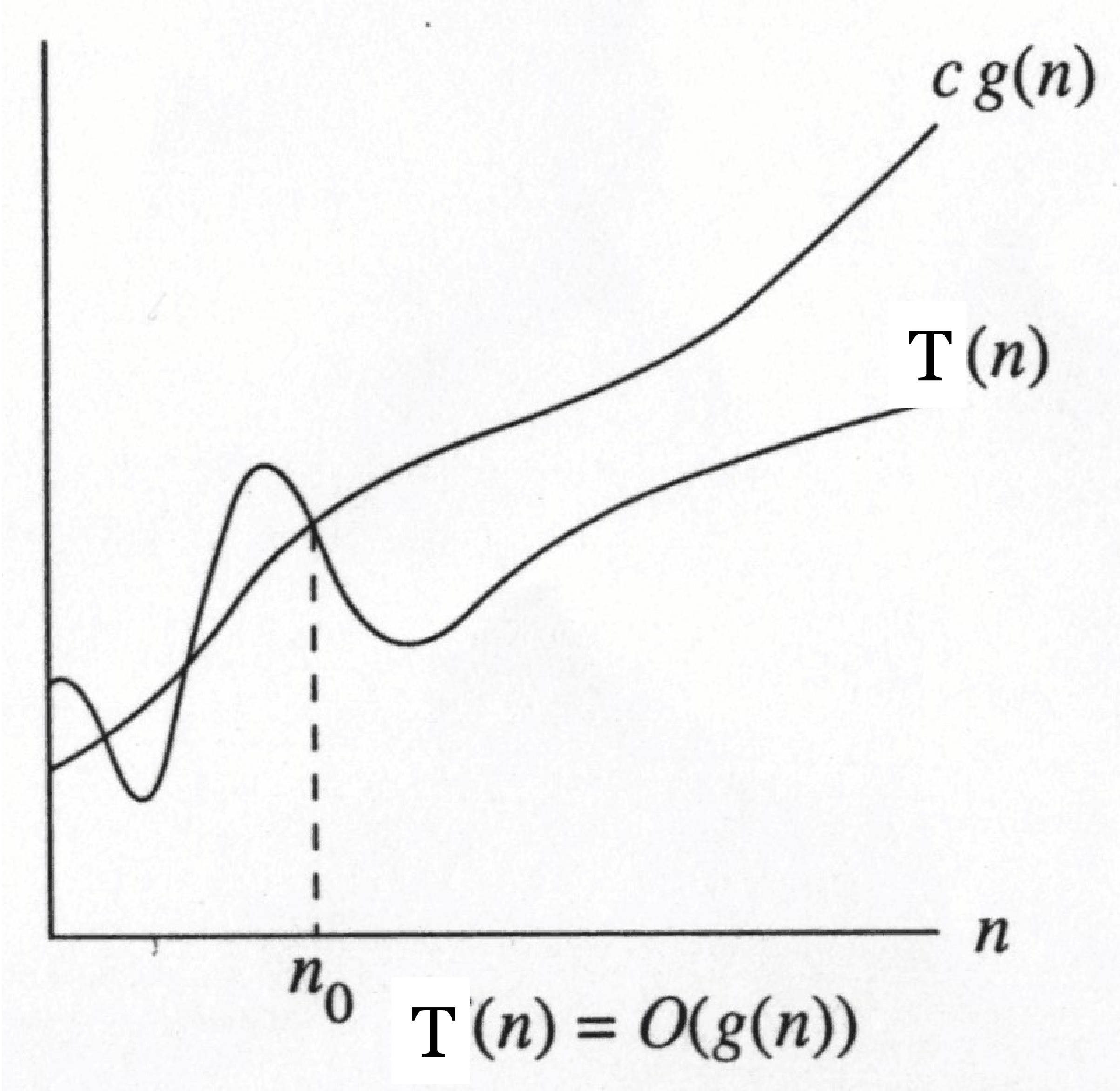
Also...

- T(n) computation is exhaustive for algorithms with a lot of instructions and operations
- It is not necessary to know the exact number of operations
  - to determine the complexity of a given algorithm
  - to determine which algorithm is more efficient (faster)
- We need an assessor (function) that will be upper asymptotic limit of T(n)

“Big O” definition

- The most used method and notation for assessing algorithms complexity is *Big O*
- *Big O* is the asymptotic execution time (operations number) of a given algorithm
- We need an answer to the question: *How does the algorithm execution time increases as a function of the input data number?*
- *Big O* is an upper limit
- Mathematical tool
- There are a lot of unnecessary details in a simple assessment of a given algorithm

"Big O" definition



"Big O" formal definition

- T(n) is O(g(n)) if there are positive constants c and n_0 such that *$T(n) \leq c\, g(n)$* for $n \geq n_0$
  - n is the input data number
  - T(n) is a function that describes the real algorithm execution time (the executed operations number – one operation is executes in one time unit)
  - g(n) is a function that characterizes the upper limit of T(n) (asymptotic limit of T(n))
    - It is closest to T(n), and always above it, for any n, starting from some n_0
    - It is guarantied that T(n) can not be bigger than g(n) for every $n \geq n_0$

How to determine O(g(n))?

- The assessment of algorithms complexity can be made even easier
- O(g(n)) – ONLY the term with the highest power of the asymptotic function g(n) is taken

Rules for O(..) use

- All terms are removed, except the term with the highest degree
  - Example $$O(n^2 + n\log n + n) \to O(n^2)$$
  - g(n)= $n^2 + n\ logn + n$

- The constants are removed
  - Example $$O(3n^2) \to O(n^2)$$ $$O(1024) \to O(1)$$

Algorithms comparison

- According to RAM model we have:
  - The best execution time
  - The average execution time
  - The worst execution time
    - Big O
- Example: Write code in pseudo language for the calculation of the sum $\sum_{i=1}^{n} i^3$

Algorithms comparison

```
sum (n)
{
    partial_sum = 0;
    for (i=1; i<=n; i++)
        partial_sum += i*i*i;
    return partial_sum;
}
```

$T(n) = 6n + 4$

1

1 + n+1 + n = 2n+2

n+n+n+n = 4n

1

$\boldsymbol{O}(n)$

- Declarations do not affect execution time

It is called algorithms **COMPLEXITY**

Algorithms comparison rules

- Cycles:
  - Sum of times necessary to execute operations in the cycle multiplied by the number of operations

```
for (int i=0; i<n; i++)
{
    a += i;
    b *=i;
}
```

2n

$O(n)$

Algorithms comparison rules

- Nested cycles:
  - Sum of the times necessary to execute the cycle operations multiplied by the product of the cycle iterations number

```
for (i=0; i<n; i++)
{
    for (j=0; j<n; j++)
        b *=i;
}
```

$n*n$

$O(n^2)$

Algorithms comparison rules

- Consecutive times:
  - With consecutive executions the highest cardinality is taken from the blocks of consecutive codes

```
for (i=0; i<n; i++)
   a[i] = 0;
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      a[i] += a[j]+i+j;
```

n	$O(n)$	$O(n^2)$
n*n	$O(n^2)$	

  - Algorithms complexity is $O(n^2)$

Algorithms comparison rules

- IF / ELSE:
  - Sum of the duration of the condition and the longer time of the blocks S1 and S2 durations

```
if (condition)
    S1
else
    S2
```

Algorithms comparison rules

- Recursions:
  - The recursion complexity is considered on a case-by-case basis and there is no general template for its duration

```
factorial (n)
{
    if (n<=1)
        return 1;
    else
        return (n*factorial(n-1));
}
```

What is the complexity of this code?

Algorithms comparison rules

- Recursions:

```
fib (n)
{
    if (n<=1)
        return 1;
    else
        return (fib(n-1) + fib(n-2));
}
```

T(n) – fib(n) execution time

T(n) = T(n-1) + T(n-2) + 2 $\Rightarrow$ $\boldsymbol{O}(2^n)$

Comparison principles

In the following example, solved with 3 different algorithms **the practical way** of their complexity assessment will be shown

- Example:
  - Find the biggest positive subsum in an array of n integers
  - Input: -2, $\boxed{11, -4, 13}$, -5, -2

20

The example solved with 3 nested cycles

```
max_subsequence_sum( a[],  n )
{

    max_sum = 0; best_i = best_j = -1;
    for( i=0; i<n; i++ )                    n
       for( j=i; j<n; j++ )                 n-i+1
       {
          this_sum=0;
          for( k = i; k<=j; k++ )           j-i+1
             this_sum += a[k];
          if( this_sum > max_sum )
          {
             /* update max_sum, best_i, best_j */
             max_sum = this_sum;
             best_i = i;
             best_j = j;
          }
       }
       return max_sum;
}
```

The algorithm complexity is $\boldsymbol{O}(n^3)$

The example solved with 2 nested cycles

```
max_subsequence_sum( a[],  n )
{

    max_sum = 0; best_i = best_j = -1;
    for( i=0; i<n; i++ )
    {
        this_sum=0;
        for( j = i; j<=n; j++ )
        {
                this_sum += a[j];
                if( this_sum > max_sum )
                {
                    /* update max_sum, best_i, best_j */
                    max_sum = this_sum;
                    best_i = i;
                    best_j = j;
                }
        }
        return max_sum;
}
```

The algorithm complexity is $O(n^2)$

The example solved with one cycle

```
max_subsequence_sum( a[],  n )
{
    i = this_sum = max_sum = 0; best_i = best_j = -1;
    for( j=0; j<n; j++ )
    {
        this_sum += a[j];
        if( this_sum > max_sum )
        {
            /* update max_sum, best_i, best_j */
            max_sum = this_sum;
            best_i = i;
            best_j = j;
         }
        else
         if( this_sum < 0 )
         {
            i = j + 1;
            this_sum = 0;
         }
     }
     return max_sum;
}
```

The algorithm complexity is O(n)

And what if...

- ...the input data is always divisible by 2?

The complexity is described by a **logarithmic function**!

```
foo (n) {
        // n > 0
        total = 0;
        while (n > 1) {
                n = n / 2;
            total++;
        }
        return total;
    }
```

The algorithms complexity is $O(\log n)$

$\log_2 8 = ?$

Answer: 3

Back to the example

```
int  max_sub_seq_sum( int a[], unsigned int n )
{
        return max_sub_sum( a, 0, n-1 );
}
```

```
max_sub_sum( a[], left, right )
{
    if ( left == right ) /* base case */
        if( a[left] > 0 ) return a[left];
        else return 0;
    center = (left + right )/2;
    max_left_sum = max_sub_sum( a, left, center );
    max_right_sum = max_sub_sum( a, center+1, right );
    max_left_border_sum = 0; left_border_sum = 0;
```

... continue

The algorithms complexity is $O(n\log_2 n)$

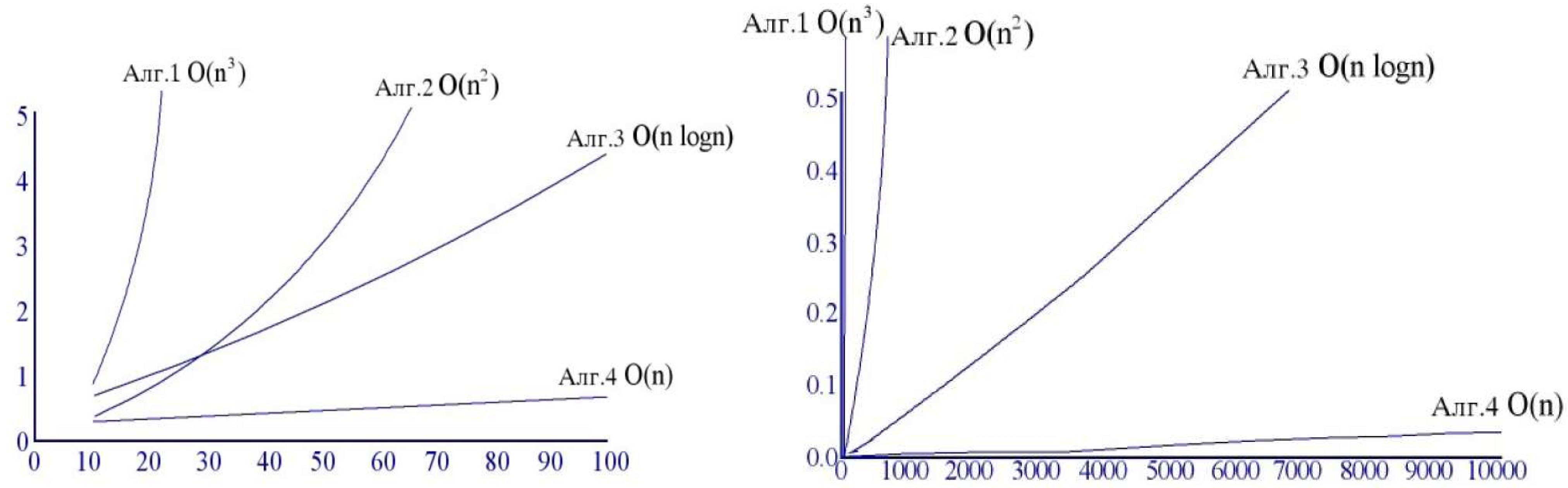
```
for( i=center; i>=left; i-- )
{
    left_border_sum += a[i];
    if( left_border_sum > max_left_border_sum )
        max_left_border_sum = left_border_sum;
}
max_right_border_sum = 0; right_border_sum = 0;
for( i=center+1; i<=right; i++ )
{
    right_border_sum += a[i];
    if( right_border_sum > max_right_border_sum )
        max_right_border_sum = right_border_sum;
}
return max3( max_left_sum, max_right_sum,
  max_left_border_sum + max_right_border_sum );
}
```

Algorithms comparison

Алгоритам		1	2	3	4
кардиналност		$O(n^3)$	$O(n^2)$	$O(n \log n)$	$O(n)$
влез	$n = 10$	0.00103	0.00045	0.00066	0.00034
	$n = 100$	0.47015	0.01112	0.00486	0.00063
	$n = 1{,}000$	448.77	1.1233	0.05843	0.00333
	$n = 10{,}000$	NA	111.13	0.68631	0.03042
	$n = 100{,}000$	NA	NA	8.0113	0.29832

Execution time of each of the previous algorithms

Algorithms comparison with Big O



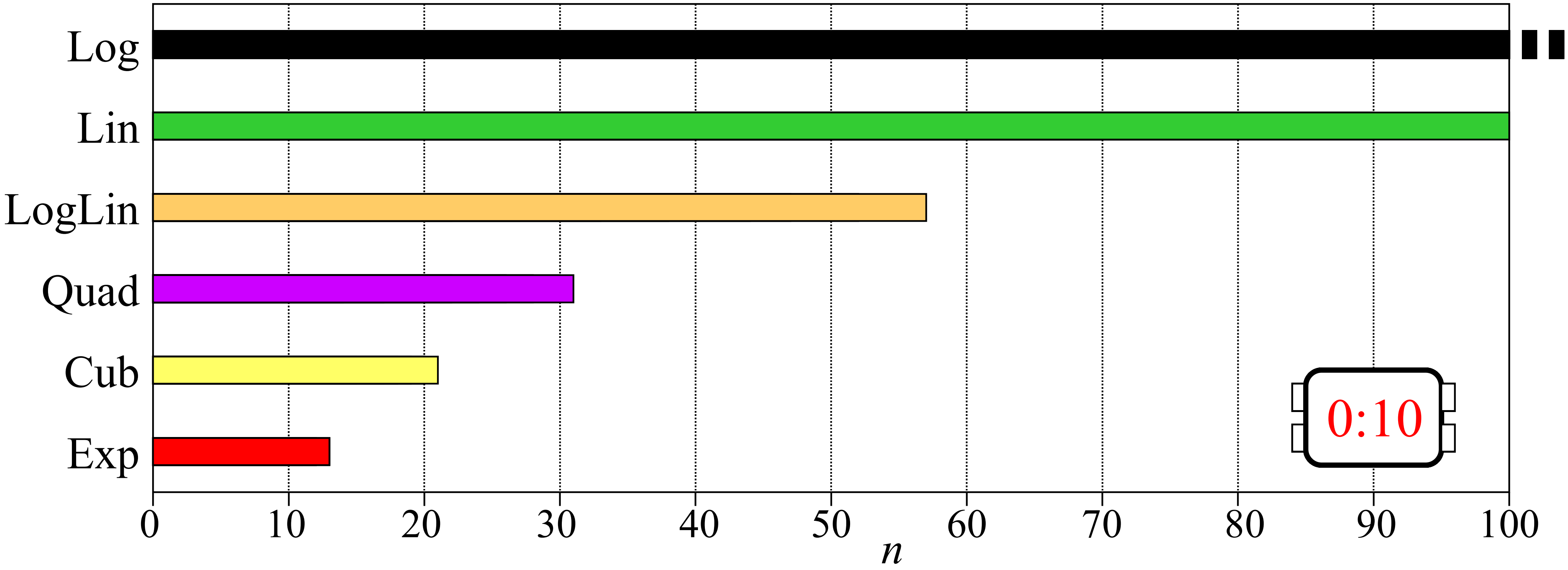
Execution time of each of the previous algorithms

The most used complexities

функција	Име
c	Константа
$\log n$	Логаритамска
$\log^2 n$	Квадратно-логаритамска
n	Линеарна
$n\log n$	
n^2	Квадратна
n^3	Кубна
2^n	Експоненцијална

Example

- It is compared how much data (*n*) can be processed by each of the algorithms for 1, 2, …, 10 sec:

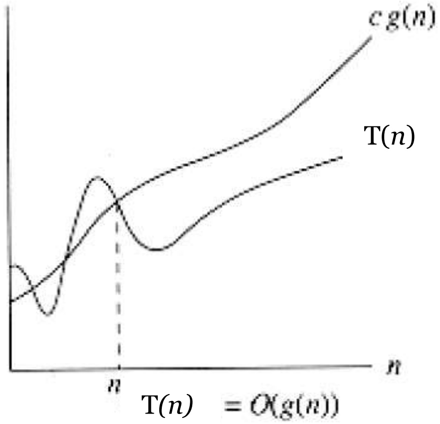


Growth rates of different complexities

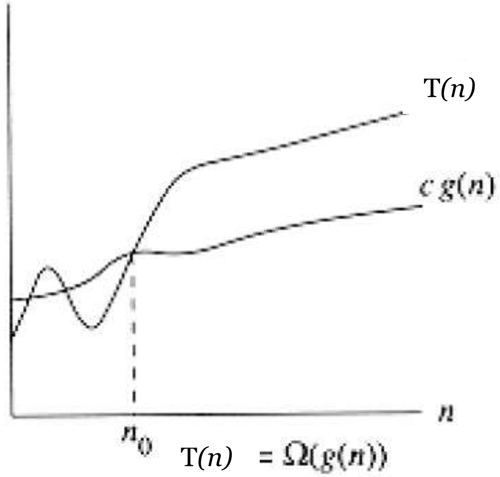
n	1	2	4	16	256	4096	65536
ln n	0	1	2	4	8	12	16
n · ln	0	2	8	64	2048	49152	1048576
n^2	1	2	16	256	65536	16777216	4.295×10^9
n^3	1	8	64	4096	16777216	6.872×10^{10}	2.815×10^{14}
2^n	2	4	16	65536	1.16×10^{73}	$> 10^{1232}$	Огромно

Notations

- O-notation
  - Upper limit on the algorithm execution time. It measures complexity in the **worst** case.



- Ω-notation
  - Lower limit on the algorithm execution time. It measures complexity in the **best** case (which is not very useful).



- θ-notation
  - Lower and upper limit on the algorithm execution time.
  - It is used in algorithms analysis research.