Course 4: Analysis of the complexity of algorithms

1 The purpose of the analysis

Complexity analysis of an algorithm has the purpose of establishment of necessary *resources* for algorithm execution on a machine. By *resources* we mean:

- *Memory space* needed for storing the data that the algorithm is processing.
- *Time* that is required for execution of all processings specified in algorithm.

This analysis is useful for establishing whether an algorithm uses decent amount of resources for solving a problem. If not, the algorithm, even if it is correct, is not considered efficient and cannot be applied in practice. Complexity analysis, or Algorithm efficiency analysis, is used for comparating algorithms with purpose of choosing the most efficient one.

In most algorithms, the volume of necessary resources depends on *size* of the problem. The *size* is usually determined by the volume of input data required by the problem. If one numeric value *n* is processed (for example, we verify is *n* is a prime number), then the *size* is considered the number of bits used for representing *n*, meaning $[\boldsymbol{log_2 x}] + 1$. If data that must be processed are organised in an array then the size of the problem can be the number of components of the array(for example, for determining the minimal value from an array with *n* elements, or finding the value of a polynom of order *n*,then the size of the problem is considered *n*). There are situations when the volume of input data is specified by more values (for example, for processing a matrix with *m* rows and *n* columns). In those cases the size of the problem will their respective values (for example, (m,n)).

Sometimes, for establishing the size of the problem, we must keep in mind the processings that will be applied upon our data. For example, if proccesing of a text is made based on number of words, then the size of the problem will be determined by the number of words, while as it is made on characters then the size of the problem will be the number of characters.

Memory storage is influenced by how the data is represented. For example, a matrix with whole numbers with 100 rows and 100 column of which only 50 are not null(rare matrix), can be represented as: (i) bidimensional array 100×100 (10000 whole values); (ii) onedimensional array where are stored only those 50 notnull values and their coresponded indexes (150 whole values). Choosing a more efficient way of representing the data can influence upon processing complexity.

For example, algorithm of adding 2 rare matrixes becomes more complicated if the matrixes are represented as onedimensional arrays. In general, obtaining efficient algorithms, from the point of view of execution time, require increasing the memory storage of data.

2 Execution time

From now on, we will use T(*n*) for execution time of an algorithm designated for solving a problem with *n* size. To estimate the execution time, we need to establish a *calculation model* and measure unit. We will consider one calculation model(called calculation machine with random acces), characterised by:

- Processings are done in sequential mode.
- *Elementary* operations are done in constant time, no matter the value of operands.
- Time for accesing information doesn't depend on its position (there are no differences between processing the first element or the last element of an array).

Establishing a measure unit means establishing what are the elementary operation and consider their execution time as measure unit. Doing so, time of execution will be determined by the number of executed elementary operations. Elementary operations are: arithmetic operations (add, subtract, multiply, divide), comparations and logical operations(negation, conjunction, disjunction). Sometimes, it is enough to count only some type of elementary operations, called *basic operations* (for example, in case of a searching or sorting algorithm we can count only the operations of comparison) and consider that their time of execution is unitary.

Time of execution of an algorithm is obtained by adding times of execution of all processings.

Example 1. Consider the problem: $\sum_{i=1}^{n} i$. Size of that problem is *n*. Solving algorithm can be described as:

```
suma(n)
1:   $S \leftarrow 0$
2:   $i \leftarrow 1$
3:   WHILE $i <= n$ DO
4:       $\|S \leftarrow S + i$
5:       $\|i \leftarrow i + 1$
RETURN $S$
```

Execution time of the algorithm can be determined by cost table:

Operation	Cost	No. repetitions
1	c_1	1
2	c_2	1
3	c_3	$n+1$
4	c_4	n
5	c_5	n

Summing up the execution times of elementary processings we obtain $T(n) = n(c_3 + c_4 + c_5) + c_1 + c_2 + c_3 = k_1n + k_2$, meaning that execution time linearly depends on problem's size. Elementary operations's cost only influence constants that interfere in function T(n).

Example 2. Consider the problem of finding the product of 2 matrixes: matrix A of size $m \times n$ and matrix B of size $n \times p$. In this case, problem's size is determined by 3 values: (m ,n, p). Algorithm ca be described as:

```
produs($a[1..m, 1..n]$, $b[1..n, 1..p]$)
1:    FOR $i = \overline{1,m}$ DO
2:        ||FOR $j = \overline{1,p}$ DO
3:        ||    ||$c[i,j] \leftarrow 0$
4:        ||    ||FOR $k = \overline{1,n}$ DO
5:        ||    ||    ||$c[i,j] \leftarrow c[i,j] + a[i,k]b[k,j]$
RETURN $c[1..m, 1..p]$
```

The cost of processings on the lines 1, 2 and 4 represent the counter management cost and will be treated globally. Assuming that all arithmetic and comparation operations have unitary cost, the table of cost becomes:

Operation	Cost	No. repetitions
1	$2(m+1)$	1
2	$2(p+1)$	m
3	1	$m \cdot p$
4	$2(n+1)$	$m \cdot p$
5	2	$m \cdot p \cdot n$

Time of execution in this case is: $T(m, n, p) = 4mnp + 5mp + 4m + 2$.

On practice, it is not necessary for such detailed analysis. It is enough to find the base operation and estimate its number of repetitions. *Base operation* is considered the operation that has the most contribution on algorithm execution time and is the operation that appears in the innermost loop.

In the above example, base operation can be considered the multiplication operation. In that algorithm's execution cost will be: $T(m, n, p) = mnp$.

Example 3. Consider the problem of determining the minimal value from an array $x[1..n]$. The size of that problem is given by n, which is the number of elements in the array. Algorithm is:

```
minim(x[1..n])
1:    $m \leftarrow x[1]$
2:    FOR $i \leftarrow \overline{2,n}$ DO
3:        || IF $m > x[i]$ THEN
4:        ||     $m \leftarrow x[i]$
RETURN $m$
```

Table of processings cost:

Operation	Cost	No. repetitions
1	1	1
2	$2n$	1
3	1	$n-1$
4	1	$\tau(n)$

Unlike previous examples, time of execution cannot be calculated explicitly because number of repetitions of processing numbered as 4 depends on values that are stored in the array.

If the minimal value of the array is located on the first position, then processing 4 will not be executed a single time, and $\boldsymbol{\tau(n) = 0}$. This is considered the most *favorable* case.

If array's elements are sorted in descending order then processing 4 will be executed at every iteration, meaning $\boldsymbol{\tau(n) = n - 1}$. This is the most *unfavorable* case.

Time of execution can be put between 2 limits: *3n ≤ T(n) ≤ 4n-1.*

In this case, it is easy to observe that the base operation is comparison operation between variable *m* and array's elements. In this case, the cost of algorithm would be: *T(n) = n – 1.*

Example 4. Consider the problem of finding a value *v* in an array *x[1..n]*. Size of the problem is, again, given by *n*.

```
cautare(x[1..n], v)
1:    $gasit \leftarrow$ FALSE
2:    $i \leftarrow 1$
3:    WHILE ($gasit$ =FALSE) AND $i \le n$ DO
4:        || IF $v = x[i]$
5:        ||    THEN $gasit \leftarrow$ TRUE
6:        ||    ELSE $i \leftarrow i + 1$
RETURN $gasit$
```

Considering processing of unitary cost, we obtain:

Operation	Cost	No. repetitions
1	1	1
2	1	1
3	3	$\tau_1(n)+1$
4	1	$\tau_1(n)$
5	1	$\tau_2(n)$
6	1	$\tau_3(n)$

In case where *v*'s value is in the array, we denote with *k* first position on which it is located. We obtain:

$$\tau_1(n) = \begin{cases} k & \text{value is present in the array} \\ n & \text{value is not present in the array} \end{cases} \qquad \text{so } \ 1 \le \tau_1(n) \le n.$$

$$\tau_2(n) = \begin{cases} 1 & \text{value is present in the array} \\ 0 & \text{value is not present in the array} \end{cases}.$$

$$\tau_3(n) = \begin{cases} k-1 & \text{value is present in the array} \\ n & \text{value is not present in the array} \end{cases} \qquad \text{so } \ 0 \le \tau_3(n) \le n.$$

The most favorable case is when the value is located on the first position of the array, in which case $\boldsymbol{T(n) = 3}(\tau1(n)+1)\boldsymbol{+\tau1(n) + \tau2(n) + \tau3(n) + 2 = 6 + 1 + 1 + 0 + 2 = 10}$

The most unfavorable case is when the value is not located in array: $\boldsymbol{T(n) = 3(n+1) + n + 0 + n + 2 = 5(n+1)}$.

Importance of the unfavorable case. For appreciation an algorithm we are interested in the unfavorable case because it shows the greatest time of execution relative to *any* input data of fixed size.

The most favorable case is useful for checking inefficient algorithms (for example, if an algorithm has a very large cost in its favorable case, then it cannot be considered acceptable solution).

Average time of execution. Sometimes, extreme cases (of most favorable or unfavorable) are met rarely. In these cases a useful measure for algorithm complexity is *average time of execution.* It represents an average value of execution times, calculated in relation to the probability distribution corresponding to the input data space. If *v(n)* represents of possible variants, P_k is the probability of apparition of case *k* and $T_k(n)$ is the thime of execution coressponding to case *k*, then average time is given by relation:

$$T_m(n) = \sum_{k=1}^{\nu(n)} T_k(n) P_k .$$

In case when are cases are equally possible, ($P_k = 1 / v(n)$) we obtain $\boldsymbol{T_m(n) = \sum_{k=1}^{\nu(n)} T_k(n)/\nu(n)}$.
Example. Consider again the problem of searching of a value *v* in an array *x[1..n]* (example 4).
To simplify the analysis, we will consider that all elements of the array are distinct. To calculate the

average time of execution we have to make assumptions about the distribution of the input data.

Let's assume that value of *v* can be located on any of the array's positions or outside of it with equal probability. The amount of total cases is *v(n) = n + 1* (*n* cases where the value is situated in the bounds of the array and one case when *v* is not in the array) results that the probability of each case is 1/(*n* + 1). Corresponding time to the case when *v* is on position *k* is $\boldsymbol{T_k = 5(k+1)}$ and for the case when the values is outside the array is $\boldsymbol{T_{n+1} = 5(n+1)}$ results that the average time of execution is:

$$T_m(n) = \frac{1}{n+1}(\sum_{k=1}^{n} 5(k+1) + 5(n+1)) = \frac{5(n^2+5n+2)}{2(n+1)}$$

The main difficulty in establishing the average time consists in establishing the distribution corresponding to the input data space. For example, we can assume:

P(v is in array) = P(v is not in the array) = 0.5 and in case if it is in the array, it can be found on any of the *n* positions with the same probability: *P(v is on position k) = 1/n.* In that case, average time is:

$$T_m(n) = \frac{1}{2}(\frac{1}{n}\sum_{k=1}^{n} 5(k+1) + 5(n+1)) = \frac{5(3n+5)}{4}$$

It is observed that the average execution time depends on the assumptions made on the distribution of the input data and that it is not a simple arithmetic mean of the times corresponding to the extreme cases (the most favorable and the most unfavorable).

Due to the difficulties that can intervene in the estimation of the average time and due to the fact that in many situations it differs from the time in the unfavorable case only by the values of the constants involved in the analysis rule, it refers to the estimation of the time in the unfavorable case. The average time has significance when, for the problem under study, the unfavorable case occurs rarely.

3 Order of growth

In order to appreciate the efficiency of an algorithm, knowledge of the detailed expression of the execution time is not required. Rather, it interests how the execution time increases with the increase in the size of the problem. A useful measure in this regard is the *order of growth*. It is determined by the *dominant term* in the expression of execution time. When the size of the problem is large the value of the dominant term significantly exceeds the values of the other terms so that the latter can be neglected.

Examples. If $T(n) = an + b$ $(a > 0)$ when the size of the problem increases by k times and the dominant term in the execution time increases by the same number of times so that $T(kn) = (ka)n + b$. In this case it is a linear order of increase. If $T(n) = an^2 + bn + c$ $(a > 0)$ then $T(kn) = (k^2a)n^2 + (kb)n + c$, so the dominant term increases k^2 times, which is why we say it is a quadratic order of growth.

If $T(n) = algn$ then $T(kn) = algn + algk$, that is, the dominant term does not change, the execution time increasing by a constant (with respect to n). In previous relationships and in all that will follow by lg we denote the logarithm in base 2 (since the transition from one base to another is equivalent to multiplication by a constant that depends only on the bases involved and in establishing the order of increase the constants are ignored in the analysis of efficiency the base of the logarithms is not relevant). A small logarithic order of growth represents good behavior. If instead $T(n) = a2^n$ then $T(kn) = a(2^n)^k$ that is, the order of growth is exponential.

Since the problem of efficiency becomes critical for large-scale problems, the complexity analysis is done for the case when n is large (theoretically it is considered that $n \to \infty$), in this way considering only the behavior of the dominant term . This type of analysis is called *asymptotic analysis*. In asymptotic analysis one algorithm is considered to be more efficient than another if the order of increase of the execution time of the first is less than that of the second. The relationship between growth orders is only meaningful for problems of large size. If we consider the times $T_1(n) = 10n + 10$ and $T_2(n) = n^2$ then it is easily observed that $T_1(n) > T_2(n)$for $n \leq 10$, although the order of increase of T_1 is obviously lower than that of T_2. Therefore, an asymptotic algorithm more efficient than another is the best option only for large-scale problems.

4 Asymptotic notations

To facilitate the asymptotic analysis and to allow the grouping of algorithms in classes according to the order of increasing execution time, some classes of functions and associated notations have been introduced.

Notation Θ. For a function g : $N \to R_+, \Theta(g(n))$ represents the set of functions:

$$\Theta(g(n)) = \{f : N \to R; \exists c_1, c_2 \in R_+^*, n_0 \in N \quad \text{so that} \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\} \tag{1}$$

About execution time of some algorithm, *T(n)*, we say that it is *Θ(g(n))* if *T(n) ϵ Θ(g(n))*. Through notation abuse in algorithmics it is used to write *T(n) = Θ(g(n))*. From intuitive point of view, *f(n) ϵ Θ(g(n))* means that *f(n)* and *g(n)* are equivalent asymptotic. In another words, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, *k* being a value strictly positive.

In figure 1 is ilustrated relationship between function *f* and *g* (for $f(n) = n^2 + 5nlgn + 10$ and $g(n)=n^2$) when *f(n) ϵ Θ(g(n))* for various domains of *n* ($n \epsilon \{1, \ldots ,5\}$, $n \epsilon \{1, \ldots ,50\}, n \epsilon \{1, \ldots ,500\}$). For inferior margin was used $c_1 = 1$ and for superior $c_2 = 4$.

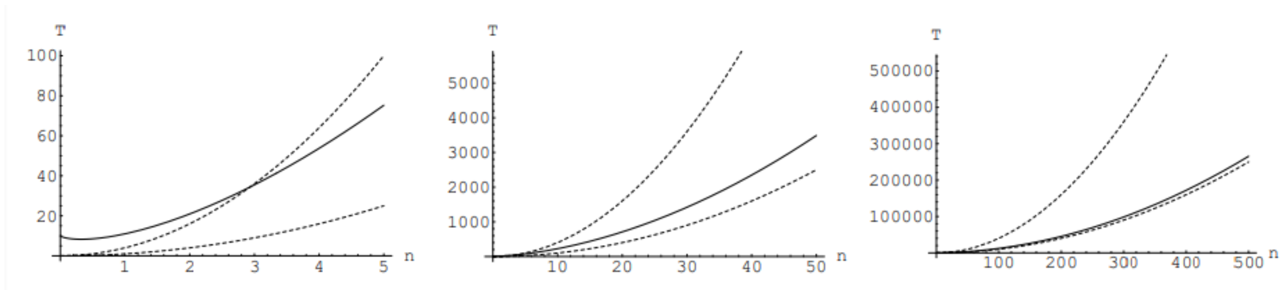
Figure 1. Function graphics, *f* (continuos line*), c₁g c₂g (dotted line)*

Example. For example 1 of section 2 (calculate sum) we obtained $T(n) = k_1 n + k_2$ $(k_1 > 0,$ $k_2 > 0)$ thus for $c_1 = k_1,\ c_2 = k_1 + 1$ and $n_0 > k_2$ we obtain $c_1 n \le T(n) \le c_2 n \; for \; n \ge n_0$, meaning $T(n) = \Theta(n)$.

For example 3 (determine minimal) we obtained $3n \le T(n) \le 4n - 1$ thus $T(n) = \Theta(n)$. In general, we can demonstrate that if $T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0, a_k > 0$ then $T(n) = \Theta(n^k)$. From $\lim_{n\to\infty} T(n)/n^k = a_k$ result that there exists $\varepsilon > 0$ and $n_0(\varepsilon)$with property that $\left|\frac{T(n)}{n^k} - a_k\right| \le \varepsilon$ for any $n \ge n_0(\varepsilon)$. So

$$a_k - \varepsilon \le \frac{T(n)}{n^k} \le a_k + \varepsilon, \quad \forall n \ge n_0(\varepsilon)$$

meaning that for $c_1 = a_k - \varepsilon, c_2 = a_k + \varepsilon$ and $n_0 = n_0(\varepsilon)$ we obtain inequalities from (1).

Notation O. For a function g : $N \to R_+, O(g(n))$ represents the set of functions:

$$O(g(n)) = \{f : N \to R; \exists c \in R_+^*, n_0 \in N \text{ astfel încât } 0 \le f(n) \le cg(n), \forall n \ge n_0\} \qquad (2)$$

This class of function allows describing of an algorithm's behaviour in case of the most unfavorable case without reffering to other situations. Because, usually, we are interest in algorithm's behaviour for random data input, it is enough to specify a superior margin for the time of execution. Intuitive, *f(n) ϵ O(g(n))* means that *f(n)* grows asimptotically as much as fast as *g(n).* In other words, $\lim_{n\to\infty} \frac{f(n)}{g(n)} \le k$, *k* being a value strictly positive.

From definitions of Θ and O results that $\Theta(g(n)) \subset O(g(n))$. Thus if *T(n) = Θ(g(n))* then *T(n) = O(g(n)).*

Using the definition of *O* we can easily verify if $g_1(n) < g_2(n)$ for $n \ge n_0$ and $f(n) \,\epsilon\, O(g_1(n))$ then $f(n) \,\epsilon\, O(g_2(n))$. Thus if $T(n) = O(n)$ then $T(n) = O(n^d)$ for any $d \ge 1$

In general, if $T(n) \le a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0,\ \ a_k > 0$ then $T(n) = O(d)$ for any $d \ge k$.

Notation Ω. For a function g : $N \to R_+,\ \Omega(g(n))$ represents the set of functions:

$$\Omega(g(n)) = \{f : N \to R; \exists c \in R_+^*, n_0 \in N \text{ astfel încât } 0 \le cg(n) \le f(n), \forall n \ge n_0\} \qquad (3)$$

Is a measurement for growth order of execution time in the most favorable case. Intuitive,

$f(n) \epsilon \Omega(\mathrm{g(n)})$ means that *f(n)* grows asymptotically at least as fast as *g(n)*, meaning $\lim\limits_{n\to\infty} f(n)/g(n) \geq k$, *k* being a value strictly positive.

From definitions we can see that $\Theta(g(n)) \subset O(g(n))$. Relation between the three mentioned classes is: $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Asymptotic analysis of fundamental structures. We consider the problem of determining the order of complexity in the most unfavorable case for algorithmic structures: sequential, alternative and repetitive.

We assume that the sequential structure is constituted by the processes $A_1, \ldots, A_k$ and each of these has the order of complexity $O(g_i(n))$. Then the structure will have the order of complexity $O(\max\{g_1(n), \ldots g_k(n)\})$.

If the condition of an alternative structure has constant cost and the processing corresponding to the two variants have the orders of complexity $O(g_1(n))$ respectively $O(g_2(n))$ then the cost of the alternative structure will be $O(\max\{g_1(n), g_2(n)\})$. In the case of a repetitive structure to determine the order of complexity in the worst case is considered the maximum number of iterations. If this is n and if in the body of the cycle processing is of constant cost then the order O(n) is obtained.

In the case of a double cycle, if for both the inner and outer cycle the limits vary between 1 and n then a quadratic complexity, O(n^2), is usually obtained. However, if the boundaries of the inner cycle change it is possible to obtain another order. Let's consider the following processing:

```
$m \leftarrow 1$
FOR $i \leftarrow 1, n$ DO
    $m \leftarrow 3 * m$
    FOR $j \leftarrow 1, m$ DO
        prelucrare $\Theta(1)$
```

As for each value of *i* we obtain $m = 3^i$ results that time of execution is of form:

$$T(n) = 1 + \sum_{i=1}^{n} (3^i + 1) \, \epsilon \, \Theta(3^n)$$

Complexity classes. In algorithm hierarchy we keep in mind following properties of function that interfere most often:

$$\lim_{n\to\infty} \frac{(\lg n)^b}{n^k} = 0, \quad \lim_{n\to\infty} \frac{n^k}{a^n} = 0, \quad \lim_{n\to\infty} \frac{a^n}{n^n} = 0, \quad \lim_{n\to\infty} \frac{a^n}{n!} = 0 \qquad (a > 1) \tag{4}$$

Complexity class	Order (most defavorble case)	Examples
Logarithmic	$O(lgn)$	Binary seach
Linear	$O(n)$	Sequential search
	$O(nlgn)$	Merge sort
Quadratic	$O(n^2)$	Insertion sort
Cubic	$O(n^3)$	Product of 2 square matrixes of order *n*
Exponential	$O(2^n)$	Processing all subsets of a set with *n* elements
Factorial	$O(n!)$	Processing all permutations of a set with *n* elements

5 Phases of complexity analysis

In complexity analysis we go through following phases:

1. We establish the size of the problem.
2. We find the base operation.
3. We verify if number of executions of base operation depends only on problem's size. If yes, that number is to be determined. If no, we analyze the most favorable case, most unfavorable case and (if it is possible) average case.
4. We establish the complexity class the algorithm belongs to.

6 Empirical analysis of the complexity of algorithms

Motivation. Mathematical analysis of the complexity of algorithms can be difficult in the case of algorithms that are not simple, especially if it is an average case analysis. An alternative to analysis the mathematics of complexity is *empirical analysis.*

It can be useful for: (i) to obtain preliminary information on the complexity class of an algorithm; (ii) to compare the efficiency of two (or more) algorithms intended to solve the same problem; (iii) to compare the efficiency of several implementations of the same algorithm; (iv) to obtain information on the effectiveness of implementing an algorithm on a given computer.

Stages of empirical analysis. In the empirical analysis of an algorithm, the following are usually followed stages:

1. We determine the purpose of the analysis.
2. We choose efficiency metric that we will use (number of executions of one/some operations or execution time of the entire algorithm or just a portition of the algorithm)
3. We establish input data properties with respect to data we will analyse (data size or specific properties)/

4. We implement the algorithm in a programming language.
5. We generate more data input sets.
6. We execute the program for each input set.
7. We analyse the obtained data.

The choice of efficiency measure depends on the purpose of the analysis. If, for example, it is intended to obtain information on the complexity class or even to verify the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. If, however, the purpose is to evaluate the behavior of the implementation of an algorithm then the execution time is appropriate.

To perform an empirical analysis, not a single set of input data is sufficient, but several, which highlight the different characteristics of the algorithm. It is generally good to choose dates of different sizes so that the beach of all sizes that will appear in practice is covered. On the other hand it also has important to analyze the different values or configurations of the input data. If an algorithm is analyzed that checks whether a number is prime or not and testing is done only for numbers that are not prime or only for numbers that are prime then no relevant result will be obtained. The same can happen for an algorithm whose behavior depends on the degree of sorting of an array (if either only the array almost sorted by the desired criterion is chosen or arrays ordered in reverse the analysis will not be relevant).

For Empirical Analysis when implementing the algorithm in a programming language, sequences whose purpose is to monitor the execution will have to be introduced. If the efficiency metric is the number of executions of an operation then a counter is used that increments after each execution of that operation. If the metric is the execution time then the moment of entry into the analyzed sequence and the moment of exit must be recorded. Most languages of programming provides functions for measuring the elapsed time between two moments. It is important, especially if several tasks are active on the computer, to count only the time affected by the execution of the analyzed program. Especially if it is a question of time measurement it is advisable to run the test program several times and calculate the average value of times.

When generating input data sets the aim is to obtain data typical of the usual runs (not just exceptions). For this purpose, data is often generated in a random manner. In reality it is a pseudo-randomness because it is simulated by deterministic techniques.

After the execution of the program for the test data, the results are recorded and for the purpose of the analysis either synthetic quantities (mean, standard deviation, etc.) are calculated or pairs of shape points (problem size, efficiency measure) are graphically represented.

7 Exercises.

1. Establish complexity order for the following algorithm that processes *n* data volume:

```
FOR $i \leftarrow 1, n$ DO
    ||... ($\Theta(1)$)
    ||FOR $j \leftarrow 1, 2i$ DO
        ||... ($\Theta(1)$)
        ||$k \leftarrow j$ ($\Theta(1)$)
        ||WHILE $k \geq 0$ DO
            ||... ($\Theta(1)$)
            ||$k \leftarrow k - 1$
```

2. Establish complexity order for the following algorithm that processes *n* data volume:

```
$h \leftarrow 1$
WHILE $h \leq n$ DO
    ||...($\Theta(1)$)
    ||$h \leftarrow 2 * h$
```

3. Establish complexity order for the following algorithm that processes *n* data volume:

```
calcul (x[0..n-1])
$k \leftarrow 0$
FOR $i \leftarrow 0, n - 1$ DO
    ||FOR $j \leftarrow 0, n - 1$ DO
        ||$y[k] \leftarrow x[i] * x[j]$
        ||$k \leftarrow k + 1$
RETURN $y[1..k]$
```

4. Consider the next two algorithms for finding the power of a number:

```
putere1(x,n)
$p \leftarrow 1$
FOR $i \leftarrow 1, n$ DO
    ||$np \leftarrow 0$
    ||FOR $j \leftarrow 1, x$ DO
        ||$np \leftarrow np + p$
    ||$p \leftarrow np$
RETURN(p)
```

```
putere2(x,n)
$p \leftarrow 1$
FOR $i \leftarrow 1, n$ DO
    ||$p \leftarrow p * x$
RETURN(p)
```

Establish complexity order considering that all arithmetic operations have the same cost.

5. Show that $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$.
6. What relation there is between $\Theta(\mathbf{log}_a\, n)$ *and* $\Theta(\mathbf{log}_b\, n)$.
7. Suggest an algorithm of complexity $\Theta(n^2)$ and one of complexity $\Theta(n)$ for evaluating a polynom of order *n*.