# Computer Science Manual for Bachelor Graduation Examination 2020

# Mathematics - Computer Science Specialization

## General topics:

**Algorithms and Programming.**

1. Search (sequential and binary), sort (selection sort, bubble sort, quicksort). The "divide and conquer" method.
2. Algorithms and specifications. Writing an algorithm starting from a given specification. Given an algorithm, determine the result of its execution.
3. OOP concepts in programming languages (Python, C++, Java, C#): Classes and objects, Members of a class and access modifiers, Constructors and destructors.
4. Relationships between classes. Derived classes and inheritance. Method overriding. Polymorphism. Dynamic binding. Abstract classes and interfaces.
5. Proposed problems

# 1. Searching and sorting

## 1.1. Searching

The data are available in the internal memory, as a *sequence of records*. We will search a record having a certain value for one of its fields, called *search key*. If the search is successful, we will have the position of the record in the given sequence.

We denote by $k_1, k_2, ...., k_n$ the record keys and by $a$ the key value to be found. The problem is, thus, to find the position $p$ characterized by $a = k_p$.

It is a usual practice to store the keys in an increasing sequence. Consequently, in the following we will assume that

$k_1 < k_2 < .... < k_n$ .

Sometimes, when the keys are already sorted, we may not only be interested to find the record having the requested key, but, if such a record is not available, we may need to know the insertion place of a new record with this key, such that the sort order is preserved.

We thus have the following specification for the *searching problem*:

*Data a,n,($k_i$, i=1,n);*

*Precondition: $n \in N$, $n \geq 1$ and $k_1 < k_2 < .... < k_n$ ;*

*Results p;*

*Postcondition: ($p=1$ and $a \leq k_1$) or ($p=n+1$ and $a > k_n$) or ($1<p\leq n$) and ($k_{p-1} < a \leq k_p$).*

### 1.1.1. Sequential search

The first method is *sequential search*, where the keys are successively examined. We distinguish three cases: $a \leq k_1$, $a > k_n$, and $k_1 < a \leq k_n$, the last case leading to the actual search.

```
Subalgorithm SearchSeq(a, n, K, p) is:          {n∈N, n≥1 and k1 < k2 < .... < kn}
                                                {Search p such that: (p=1 and a ≤ k1) or}
                                                { (p=n+1 and a>kn) or (1<p≤n) and (kp-1 < a ≤ kp).
  Let p := 0;                                                    {Case "not yet found"}
  If a ≤ k1 then p := 1 else
     If a > kn then p := n + 1 else
          For i := 2; n do
               If (p = 0) and (a ≤ ki) then p := i endif
          endfor
     endif
  endif
end-SearchSeq
```

We remark that this method leads to *n-1* comparisons in the worst case, because the counter *i* will take all the values from *2* to *n*. The *n* keys divide the real axis in *n+1* intervals. The same number of comparisons will be made in *n-1* from the *n+1* intervals where the searched key can be, so the average complexity has the same order of magnitude at the worst-case complexity.

There are many situations when this algorithm does useless computations. When the key has already been identified, it is useless to continue the loop for the remaining values of *i*. In other words, it is desirable to replace the **for** loop with a **while** loop. We get the second subalgorithm, described as follows.

```
Subalgorithm SearchSucc(a, n, K, p) is:          {n∈N, n≥1 and k1 < k2 < .... < kn}
                                                 {Search p such that: p=1 and a ≤ k1) or }
                                                 {(p=n+1 and a>kn) or (1<p≤n) and (kp-1 < a ≤ kp).
   Let p:=1;
   If a>k1 then
      While p≤n and a>kp do p:=p+1 endwhile
   endif
end-SearchSucc
```

In the worst case this subalgorithm does the same number of operations as the subalgorithm *SearchSeq*. On the average, the number of operations is reduced to half of the operations executed by the subalgorithm *SearchSeq*, and, as such, the average running-time complexity order of *SearchSucc* is the same as with the *SearchSeq* subalgorithm. We note that this type of searching can also be applied in the case that the keys are not in an increasing sequence.

#### 1.1.2. Binary search

Another method, called **binary search**, more efficient than the previous two methods, uses the "divide and conquer" technique with respect to working with the data. We start by considering the relation of the search key to the key of the element in the middle of the collection. Based on this check we will continue our search in one of the two halves of the collection. We can thus successively halve the collection portion we use for our search. Since we modify the size of the collection, we need to consider the ends of the current collection as parameters for the search.

The binary search may effectively be realized with the function call `SearchBin(a, n, K, p)`. This function is described below.

```
Subalgorithm SearchBin(a, n, K, p) is:          {n∈N, n≥1 and k1 < k2 < .... < kn}
                                                {Search  p such that: (p=1 and a ≤ k1) or}
                                                {(p=n+1 and a>kn) or (1<p≤n) and (kp-1 < a ≤ kp)}
 If a ≤ k1 then p := 1 else
     If a > kn then p := n+1 else
         P := BinarySearch(a, n, K, 1, n)
     endif
 endif
end-SearchBin
Function BinarySearch(a, n, K, Left, Right) is:
  If Left ≥ Right - 1
     then BinarySearch:= Right
     else m := (Left+Right) Div 2;
       If a ≤ km
          then BinarySearch:= BinarySearch (a, n, K, Left, m)
          else BinarySearch:= BinarySearch (a, n, K, m, Right)
       endif
  endif
end-BinarySearch
```

The variables `Left` and `Right` in the *BinarySearch* function described above represent the ends of the search interval, and `m` represents the middle of the interval. Using this method, in a collection with `n` elements, the search result may be provided after at most $log_2 n$ comparisons. Thus, the worst case time complexity is proportional to $log_2 n$. Without going into details, let us note that the average running-time complexity is the same.

We remark that the function *BinarySearch* is a recursive function. We can easily remove the recursion, as shown in the following function:

```
Function BinarySearchN(a, n, K, Left, Right) is:
 While Right – Left > 1 do
         m := (Left+Right) Div 2;
         If a ≤ km then Right := m else Left := m endif
 endwhile
 BinarySearchN:= Right
end-BinarySearchN
```

## 1.2. Sorting

Internal sorting is the operation to reorganize the elements in a collection already available in the internal memory, in such a way that the record keys are sorted in increasing (or decreasing, if necessary) order.

From an algorithms complexity point of view, our problem is reduced to keys sorting. So, the specification of the **internal sorting** problem is the following:

*Data n,K;* *{K=($k_1$,$k_2$,...,$k_n$)}*

*Precondition:* $k_i \in R$, $i=1,n$

*Results K';*

*Postcondition: K' is a permutation of K, having the elements sorted in increasing order, that is* $k'_1 \leq k'_2 \leq ... \leq k'_n$.

### 1.2.1. Selection sort

The first technique, called *Selection Sort*, works by determining the element having the minimal (or maximal) key, and swapping it with the first element. Now, forget about the first element and resume the procedure for the remaining elements, until all elements have been considered.

```
Subalgorithm SelectionSort(n, K) is:                    {Do a permutation of}
                                                        {the n components of K}
                                                        {such that k1 ≤ k2 ≤ .... ≤ kn }
 For i := 1; n-1 do
   Let ind := i;
   For j := i + 1; n do
      If kj < kind then ind := j endif
   endfor
   If i < ind then t := ki; ki := kind; kind := t endif
 endfor
end-SelectionSort
```

We remark that the total number of comparisons is

$$(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2$$

independently of the input data. So, the average computational complexity, as well as the worstcase computational complexity, is $\boldsymbol{O}(n^2)$.

### 1.2.2. Bubble sort

The *BubbleSort* method compares two consecutive elements, which, if not in the expected relationship, will be swapped. The comparison process will end when all pairs of consecutive elements are in the expected order relationship.

```
Subalgorithm BubbleSort(n, K) is:
  Repeat
    Let kod := 0;                                {Hypothesis "is sorted"}
    For i := 2; n do
      If k_{i-1} > k_i then
        t := k_{i-1};
        k_{i-1} := k_i;
        k_i := t;
       kod := 1                                          {Not sorted yet!}
      endif
    endfor
  until kod = 0 endrepeat                                        {Sorted}
end-BubbleSort
```

This algorithm performs (n-1)+(n-2)+ ... +2+1 = n(n-1)/2 comparisons in the worst case, so the time complexity is $\boldsymbol{O(n^2)}$.

An optimized variant of *BubbleSort* is:

```
Subalgorithm BubbleSort(n, K) is:
  Let  s := 0
  Repeat
    Fie kod := 0;                                {Hypothesis "is sorted"}
    For i := 2; n-s do
      If k_{i-1} > k_i then
        t := k_{i-1};
        k_{i-1} := k_i;
        k_i := t;
       kod := 1                                          {Not sorted yet!}
      endif
    endfor
    s := s + 1
  until kod = 0 endrepeat                                        {Sorted}
sf-BubbleSort
```

### 1.2.3. Quicksort

Another more efficient sorting method is described hereby. The method, called *QuickSort*, is based on the "divide and conquer" technique. The subsequence to be sorted is given through two input parameters, the inferior and superior limits of the substring elements indices. The procedure call to sort the whole sequence is: `QuickSort(n, K)`, where `n` is the number of records of the given collection. So,

```
Subalgorithm QuickSort(n, K) is:
  Call QuickSortRec(n, K, 1, n)
sf-QuickSort
```

The procedure `QuickSortRec(n, K, Left, Right)` will sort the subsequence $k_{Left}$, $k_{Left+1}$, ..., $k_{Right}$. Before performing the actual sort, the subsequence will be rearranged in such a way that the value of the element $k_{Left}$ (called pivot) occupies its final position (when the sequence is sorted). If `i` is this position, the subsequence will be rearranged such that the following condition is fulfilled:

$k_j \le k_i \le k_l$, for $Left \le j < i < l \le Right$ (*****)

Once the partitioning is achieved, we will only need to sort the $k_{Left}$, $k_{Left+1}$, ... , $k_{i-1}$ using a recursive call to `QuickSortRec(n, K, Left, i-1)` and then the subsequence $k_{i+1}$, ..., $k_{Right}$ using a recursive call to `QuickSort(n, K, i+1, Right)`. Of course, we will need to sort these subsequences (by the recursive call of the procedure) only if they have at least two elements.

The procedure *QuickSortRec* is described below.

```
Subalgorithm QuickSortRec (n, K, Left, Right) is:
   Let i := Left; j := Right; a := ki;
   Repeat
     While kj ≥ a and (i < j) do j := j - 1 endwhile
     ki := kj;
     While ki ≤ a and (i < j) do i := i + 1 endwhile
     kj := ki ;
  until i = j endrepeat
  Let ki := a;
  If Left < i-1 then Call QuickSortRec(n, K, Left, i - 1) endif
  If i+1 < Right then Call QuickSortRec(n, K, i + 1, Right) endif
end-QuickSortRec
```

The time complexity of the described algorithm is $\Theta(n^2)$ in the worst case, but the average time complexity is $\boldsymbol{\Theta}(nLog_2n)$.

## 1.3. The "divide and conquer" method

The programming strategy "divide and conquer" ("Divide et Impera") means:

- Dividing the data ("divide and conquer");
- Breaking the problem in subproblems ("top-down").

The method can be applied to problems that can be divided into independent subproblems similar to the initial problem, which are of lower size and can be solved easily.

Note that:

- The division is done until we get a problem that can be immediately solved.
- The technique can make use of a recursive implementation.

**Formalization**

```
Sublalgorithm AlgName(D) is:
    If dim(D) ≤ a then
         @solve the problem
    else
         @ Divide D in d1, d2,..., dk
         Call AlgName (d1)
         Call AlgName (d2)
         .
         .
         Call AlgName (dk)
         @ build the final result using partial results of the above calls
    endif
end-AlgName
```

# 2. Algorithms and specifications

Algorithms and specifications. Writing an algorithm starting from a given specification. Given an algorithm, determine the result of its execution.

**Problem 1**

Write a function that satisfies the following specification:

*Data nr;*

*Precondition:* $nr \in N$, $nr \geq 1$

*Results* $l_1, l_2, ..., l_n$;

*Postcondtion:* $n \in N^*$, $\left[\frac{nr}{l_i}\right] \cdot l_i = nr \ \forall 1 \leq i \leq n, l_i \neq l_j \ \forall i \neq j, 1 \leq i, j \leq n$, $n$ is maximum

**Problem 2**

Write a function that satisfies the following specification:

*Data* $n, L=(l_1, l_2, ..., l_n)$;

*Precondition:* $l_i \in R$, $i=1,n$

*Results* $R=(r_1, r_2, ..., r_n)$;

*Postcondtion: R is a permutation of L,* $r_1 \geq r_2 \geq ... \geq r_n$.

**Problem 3**

Write an algorithm/program to solve the following problem: When Ana goes shopping, she always prepares a shopping list: name, quantity, department (food, clothing, shoes, consumables), estimated price. The requirement is to display Ana's shopping list alphabetically ordered according to the department, the list sorted based on quantity in decreasing order, and Ana's list for a certain department. It is also required to compute an estimated price of Ana's expenses.

**Problem 4**

Write an algorithm/program to solve the following problem: Write a program the reads alist of integer numbers different than zero. Reading the list ends when the value zero is entered. The program should remove from the list the sequences of strictly positive consecutive elements having length greater than 3 (if they exist) and then print the obtained list.

**Problem 5**

What is the effect of the following C++ program?

```
#include <iostream>
using namespace std;

bool Prime(int p){
	int d = 2;
	while ((d * d <= p) && (p % d > 0)){
		d++;
	}
	return d * d > p;
}

bool Dec(int n, int &p1, int &p2){
	p1 = 2;
	while ((p1 <= n / 2) && (!Prime(p1) || !Prime(n - p1))){
		p1++;
	}
	p2 = n - p1;
	return p1 <= n / 2;
}

int main(){
	int a;
	int b = 0;
	int c = 0;

	do{
		cout << "Give a: ";
		cin >> a;
		if (a > 0){
			if (Dec(a, b, c))
				cout << a << ", " << b << ", " << c << endl;
			else
				cout << "There is no dec.";
		}
	} while (a > 0);
	return 0;
}
```

**Problem 6**

Consider the following C++ program and indicate:

a) what does the program do;

b) what does each subprogram do;

c) what results are printed for 12 1233 1132 2338 8533 10000 21500 0 ?

```cpp
#include <iostream>
#include <fstream>
#include <set>

using namespace std;

void function1(int x[], int &n){
	ifstream fin("data.in");
	n = 0;
	int v = 0;
	do{

		fin >> v;
		if (v > 0){
			x[n++] = v;
		}
	} while (v > 0);
	fin.close();
}

void function2(int x[], int n){
	ofstream fout("data.out");
	for (int i = 0; i < n; i++){
		fout << x[i] << " ";
	}
	fout << endl;
	fout.close();
}

void function3(int a, std::set<int> &s){
	s.clear();
	do{
		s.insert(a % 10);
		a /= 10;
	} while (a > 0);
}

bool function4(int a, int b){
	std::set<int> Ma;
	std::set<int> Mb;
	function3(a, Ma);
	function3(b, Mb);
	return Ma == Mb;
}

void function5(int &p){
	int v = 0;
	do{
		v = v * 10 + p % 10;
		p /= 10;
	} while (p > 0);
	p = v;
}
```

```
void function6(int x[], int n){
	for (int i = 0; i < n - 1; i++){
		if (function4(x[i], x[i + 1]))
			function5(x[i]);
	}
}

int main(){
	int x[100];
	int n = 0;
	function1(x, n);
	function6(x, n);
	function2(x, n);
	return 0;
}
```

Answer c) 12 3321 1132 2338 8533 10000 21500

**Problem 7**

Specify what is the following program doing, and then write the C++ program for the inverse function.

```
#include <cctype>
#include <string>
#include <iostream>

using namespace std;

char UrmL(char l){
	switch (l){
	case 'Z':
		return 'A';
	case 'z':
		return 'a';
	default:
		return l + 1;
	}
}

char ModC(char c){
	if (std::isalpha(c))
		return UrmL(c);
	else
		return c;
}

std::string Modif(std::string s){
	for (int i = 0; i < s.length(); i++){
		s[i] = ModC(s[i]);
	}
	return s;
}
```

```
int main(){
	std::string s;
	do{
		cin >> s;
		cout << Modif(s) << endl;
	} while (s != "stop");
	return 0;
}
```

Note. Assuming that this program creates a text *encoding*, write the program that generates the *decoding*!

# 3. OOP concepts in programming languages

OOP concepts in programming languages (Python, C++, Java, C#): Classes and objects, Members of a class and access modifiers, Constructors and destructors.

## 3.1. Data protection in modular programming

In procedural programming, developing programs means using functions and procedures for writing these programs. In the C/C++ programming language instead of functions and procedures we have functions that return a value and functions that do not return a value. But in case of large applications it is desirable to have some kind of data protection. This means that only some functions have access to problem data, specifically those functions referring to that data. In modular programming, data protection may be achieved by using static memory allocation. If in a file a datum outside any function is declared static then it can be used from where it was declared to the end of the file, but not outside it.

Let us consider the following example dealing with integer vector processing. Write a module for integer vector processing that contains functions corresponding to vector initialization, disposing occupied memory, raising to the power two and printing vector elements. A possible implementation of this module is presented in the file **vector1.cpp**:

```
#include <iostream>
using namespace std;

static int* e;              // vector elements
static int d;               // vector size

void init(int* e1, int d1){  // initialization
     d = d1;
     e = new int[d];
     for (int i = 0; i < d; i++)
          e[i] = e1[i];
}

void destroy(){               // disposing occupied memory
     delete[] e;
}

void squared(){             // raising to the power two
     for (int i = 0; i < d; i++)
          e[i] *= e[i];
}

void print(){            // printing
     for (int i = 0; i < d; i++)
          cout << e[i] << ' ';
```

```
	cout << endl;
}
```

The module is individually compiled and an object file is produced. A main program example is presented in the file **vector2.cpp**:

```
#include "functions.h"
extern void init(int*, int);  //extern may be omitted
extern void distroy();
extern void squared();
extern void print();
//extern int* e;

int main() {
	int x[5] = { 1, 2, 3, 4, 5 };
	init(x, 5);
	squared();
	print();
	destroy();
	int y[] = { 1, 2, 3, 4, 5, 6 };
	init(y, 6);
	//e[1]=10;                    error, data are protected
	squared();
	print();
	destroy();
	return 0;
}
```

Note that even though the main program uses two vectors, we cannot use them together, so for example the module **vector1.cpp** cannot be extended to implement vector addition. In order to overcome this drawback, abstract data types have been introduced.

## 3.2. Abstract data types

Abstract data types enable a tighter bound between the problem data and operations (functions) referring to these data. An abstract data type declaration is similar to a structure declaration, which apart of the data also declares or defines functions referring to these data.

For example in the integer vector case we can declare the abstract data type:

```
struct vect {
	int* e;
	int d;
	void init(int* e1, int d1);
	void destroy() { delete[] e; }
	void squared();
	void print();
};
```

The functions declared or defined within the structure will be called *methods* and the data will be called *attributes*. If a method is defined within the struct (like the *destroy* method from the above example) then it is considered an *inline* method. If a method is defined outside the struct then the function name will be replaced by the abstract data type name followed by the scope resolution operator (::) and the method name. Thus the *init*, *squared* and *print* methods will be defined as follows:

```
void vect::init(int *e1, int d1){
      d = d1;
      e = new int[d];
      for (int i = 0; i < d; i++)
            e[i] = e1[i];
}

void vect::squared(){
      for (int i = 0; i < d; i++)
            e[i] *= e[i];
}

void vect::print(){
      for (int i = 0; i < d; i++)
            cout << e[i] << ' ';
      cout << endl;
}
```

Even though by the above approach a tighter bound between problem data and functions referring to these data has been accomplished, data are not protected, so they can be accessed by any user defined function, not only by the methods. This drawback may be overcome by using classes.

### 3.3. Class declaration

A class abstract data type is declared like a structure, but the keyword struct is replaced with *class*. Like in the struct case, in order to refer to a class data type one uses the name following the keyword class (the class name). Data protection is achieved with the access modifiers: *private*, *protected* and *public*. The access modifier is followed by the character ':'. The *private* and *protected* access modifiers represent protected data while the *public* access modifier represents unprotected data. An access modifier is valid until the next access modifier occurs within a class, the default access modifier being *private*. Note that struct also allow the use of access modifiers, but in this case the default access modifier is *public*.

For example, the vector class may be declared as follows:

```
class vector {
  int* e;  // vector elements
  int d;   // vector size
public:
  vector(int* e1, int d1);
  ~vector() { delete [] e; }
  void squared();
  void print();
};
```

Note that the attributes *e* and *d* have been declared *private* (restricted access), while methods have been declared *public* (unrestricted access). Of course that some attributes may be declared public and some methods may be declared private if the problem specifics require so. In general, private attributes can only be accessed by the methods from that class and by *friend functions*.

Another important remark regarding the above example is that attribute initialization and occupied memory disposal is done via some special methods.

Data declared as some class data type are called the classes' objects or simply *objects*. They are declared as follows:

```
class_name list_of_objects;
```

For example, a vector object is declared as follows:

```
vector v;
```

Object initialization is done with a special method called *constructor*. Objects are disposed by an automatic call of another special method called *destructor*. In the above example,

```
vector(int* e1, int d1);
```

is a constructor and

```
~vector() { delete [] e; }
```

is a destructor.

Abstract data types of type *struct* may also be seen as classes where all elements have unrestricted access. The above constructor is declared inside the class, but it is not defined, while the destructor is defined inside the class. So the destructor is an inline function. In order to define methods outside a class, the scope resolution operator is used (like in the struct case).

### 3.3.1. Class members. The this pointer

In order to refer to class attributes or methods the dot (.) or arrow (→) operator is used (like in the struct case). For example, if the following declarations are considered:

```
vector v;
vector* p;
```

then printing the vector v and the vector referred by the p pointer is done as follows:

```
v.print();
p->print();
```

However, inside methods, in order to refer to attributes or (other) methods only their name needs to be used, the dot (.) or arrow (→) operators being optional. In fact, the compiler automatically generates a special pointer, the *this* pointer, at each method call and it uses the generated pointer to identify attributes and methods.

The *this* pointer will be declared automatically as a pointer to the current object. In the above example, the *this* pointer is the address of the vector *v* and the address referred by the *p* pointer respectively.

For example, if inside the *print* method an attribute *d* is used then it is interpreted as *this->d*.

The *this* pointer may also be used explicitly by the programmer, if the problem specifics require so.

### 3.3.2. The constructor

Object initialization is done with a special method called constructor. The constructor name has to be the same with the class name. A class may have multiple constructors. In this case, these methods will have the same name and this is possible due to function overloading. Of course that the number and/or formal parameter types have to be different otherwise the compiler cannot choose the correct constructor.

Constructors do not return any value. In this situation the use of the keyword *void* is forbidden.

In the following we show an example of a class having as attributes a person's last name and first name and a method for returning the person's whole name.

File **person.h**:

```
class Person {
      char* lastname;
      char* firstname;
public:
      Person();                       //default constructor
      Person(char* ln, char* fn);     //constructor
      Person(const Person& p1);       //copy constructor
      ~Person();                      //destructor
      char* toString();
};
```

File **person.cpp**:

```
#include <iostream>
#include <cstring>
#include "person.h"

using namespace std;

Person::Person(){
	lastname = new char[1];
	*lastname = 0;

	firstname = new char[1];
	*firstname = 0;

	cout << "Calling default constructor." << endl;
}

Person::Person(char* ln, char* fn){
	lastname = new char[strlen(ln) + 1];
	strcpy_s(lastname, strlen(ln) + 1, ln);

	firstname = new char[strlen(fn) + 1];
	strcpy_s(firstname, strlen(fn) + 1, fn);

	cout << "Calling constructor (lastname, firstname).\n";
}

Person::Person(const Person& p1){
	lastname = new char[strlen(p1.lastname) + 1];
	strcpy_s(lastname, strlen(p1.lastname) + 1, p1.lastname);

	firstname = new char[strlen(p1.firstname) + 1];
	strcpy_s(firstname, strlen(p1.firstname) + 1, p1.firstname);

	cout << "Calling copy constructor." << endl;
}

Person::~Person(){
	delete[] lastname;
	delete[] firstname;
}

char* Person::toString(){
	int l = strlen(firstname) + 1 + strlen(lastname) + 1;
	char* s = new char[l];

	strcpy_s(s, l, firstname);
	strcat_s(s, l, "-");

	strcat_s(s, l, lastname);
	strcat_s(s, l, "\0");

	return s;
}
```

File **personTest.cpp**:

```
#include "person.h"
#include <iostream>

using namespace std;

int main() {
	Person A;			//calling default constructor
	char* s = A.toString();
	cout << s << endl;
	delete[] s;
	Person B("Stroustrup", "Bjarne");
	s = B.toString();
	cout << s << endl;
	delete[] s;
	Person* C = new Person("Kernighan", "Brian");
	s = C->toString();
	cout << s << endl;
	delete[] s;
	delete C;
	Person D(B);			//equivalent to Person D = B;
					//calling copy constructor
	s = D.toString();
	cout << s << endl;
	delete[] s;
	return 0;
}
```

We may notice the presence of two special types of constructors: *the default constructor* and *the copy constructor*. If a class has a constructor without any parameters then this is called *default constructor*. The *copy constructor* is used for object initialization given an object of the same type (in the above example a person having the same last and first name). The copy constructor is declared as follows:

```
class_name(const class_name& object);
```

The *const* keyword expresses the fact that the copy constructor's argument is not changed. A class may contain attributes of other class type. Declaring the class as:

```
class class_name {
  class_name_1 ob_1;
  class_name_2 ob_2;
  ...
  class_name_n ob_n;
  ...
};
```

the header of the constructor for class *class_name* will have the following form:

```
class_name(argument_list):
        ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)
```

where *argument_list* and *l_arg_i* respectively represent the list of formal parameters from the *class_name*'s constructor and object *ob_i* respectively.

From the list *ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)* one my choose not to include the objects that do not have user defined constructors, or objects that are initialized by the default constructor, or by a constructor having only implicit parameters.

If a class contains attributes of another class type then first these attributes' constructors are called followed by the statements from this classes' constructor.

File **`pair.cpp`**:

```
#include <iostream>
#include "person.h"

using namespace std;

class Pair {
      Person husband;
      Person wife;
public:
      Pair(){               //default constructor definition
                            //call to the default constructors
      }                     //for objects husband and wife
      Pair(Person& ahusband, Person& awife);
      Pair(char* lname_husband, char* fname_husband,
            char* lname_wife, char* fname_wife) :
            husband(lname_husband, fname_husband),
            wife(lname_wife, fname_wife)     {
      }
      char* toString();
};

inline Pair::Pair(Person& ahusband, Person& awife) :
husband(ahusband), wife(awife){
}

char* Pair::toString(){
      char* s_husband = husband.toString();
      char* s_wife = wife.toString();
      int l = 9 + strlen(s_husband) + 2 + 6 + strlen(s_wife) + 1;
      char* s = new char[l];

      strcpy_s(s, l, "Husband: ");
      strcat_s(s, l, s_husband);

      strcat_s(s, l, "; Wife: ");
      strcat_s(s, l, s_ wife);

      strcat_s(s, l, "\0");
      return s;
}
```

```
int main() {
	Person A("Smith", "John");
	Person B("Smith", "Emma");
	Pair AB(A, B);
	char* s = AB.toString();
	cout << s << endl;
	delete[] s;
	Pair CD("C", "C", "D", "D");
	s = CD.toString();
	cout << s << endl;
	delete[] s;
	Pair EF;
	s = EF.toString();
	cout << s << endl;
	delete[] s;
	return 0;
}
```

Note that in the second constructor, the formal parameters *husband* and *wife* have been declared as references to type *person*. If they had been declared as formal parameters of type *person*, then in the following situation:

```
Pair AB(A, B);
```

the copy constructor would have been called four times. In situations like this, temporary objects are first created using the copy constructor (two calls in this case), and then the constructors of the attributes having a class type are executed (other two calls).

### 3.3.3. The destructor

The destructor is the method called in case of object disposal. Global object destructor is called automatically at the end of the *main* function as part of the *exit* function. So using the *exit* function in a destructor is not recommended as it leads to an infinite loop. Local objects destructor is executed automatically when the block in which these objects were defined is finished. In case of dynamically allocated objects, the destructor is usually called indirectly via the *delete* operator (provided that the object has been previously created using the *new* operator). There is also an explicit way of calling the destructor and in this case the destructor name needs to be preceded by the class name and the scope resolution operator.

The destructor name starts with the ~ character followed by the class name. Like in the constructor case, the destructor does not return any value and using the *void* keyword is forbidden. The destructor call in various situations is shown in the following example:

File **destructExample.cpp**:

```
Person persGlobal("Toma", "Maria");

#include <iostream>
#include "person.h"

using namespace std;

void funct(){
	cout << "Function call " << endl;
	Person persLocal("Pop", "Ana");
}

int main() {
	Person* persDynamic = new Person("Moldovan", "Ioana");
	funct();
	cout << "Continue the main program" << endl;
	delete persDynamic;
	return 0;
}
```

# 4. Relationships between classes

## 4.1. Theoretical basis

The use of abstract data types creates an ensamble for managing data and operations on this data. By means of the abstract type class, data protection is also achieved, so usually the protected elements can only be accessed by the methods of the given class. This property of objects is called *encapsulation*.

In everyday life we do not see separate objects only, but also different relationships among these objects, and among the classes these objects belong to. In this way a class hierarchy is formed. The result is a second property of objects: *inheritance*. This means that all attributes and methods of the base class are inherited by the derived class, but new members (both attributes and methods) can be added to it. If a derived class has more than one base class, we talk about multiple inheritance.

Another important property of objects belonging to the derived class is that methods can be overridden. This means that an operation related to objects belonging to the hierarchy has a single signature, but the methods that describe this operation can be different. So, the name and the list of formal parameters of the method is the same in both the base and the derived class, but the implementation of the method can be different. Thus, in the derived class methods can be specific to that class, although the operation is identified through the same name. This property is called *polymorphism*.

## 4.2. Declaration of derived classes

A derived class is declared in the following way:

```
class name_of_derived_class : list_of_base_classes {
  //new attributes and methods
};
```

where `list_of_base_classes` is of the form: `elem_1, elem_2, ..., elem_n` and `elem_i` for each $1 \leq i \leq n$ can be:

```
public base_class_i
```

or

```
protected base_class_i
```

or

```
private base_class_i
```

The *public*, *protected* and *private* keywords are called inheritance access modifiers in this situation too. They can be missing, and in this case the default modifier is private. Access to elements from the derived class is presented in Table 1.

| **Access to elements from the base class** | **Inheritance access modifier** | **Access to elements from the derived class** |
|---|---|---|
| public | public | public |
| protected | public | protected |
| private | public | inaccesible |
| public | protected | protected |
| protected | protected | protected |
| private | protected | inaccesible |
| public | private | private |
| protected | private | private |
| private | private | inaccesible |

Table 1: *Access to elements from the derived class*

We can observe that *private* members of the base class are inaccesible in the derived class. *Protected* and *public* members become *protected* and *private*, respectively, if the inheritance access modifier is *protected* and *private*, respectively, and remain unchanged if the inheritance access modifier is *public*. This is why, generally, attributes and methods are declared *protected* and the inheritance access modifier is *public*. Thus, they can be accessed, but are protected in the derived class too.

### 4.3. Virtual functions

Polymorphism leads naturally to the problem of determining the method that will be called for a given object. Let us consider the following example. We declare a base class, called *base*, and a class derived from this class, called *derived*. The base class has two methods: *method_1* and *method_2*. Method *method_2* calls method *method_1*. In the derived class, *method_1* is overridden, but *method_2* is not. In the main program, an object of the derived class is declared and *method_2*, inherited from the base class, is called. In the C++ language, this example is written in the following way:

File `virtual1.cpp`:

```
#include <iostream>

using namespace std;

class base {
public:
	void method_1();
	void method_2();
};

class derived : public base {
public:
	void method_1();
};

void base:: method_1() {
	cout << "Call to method method_1" << " of base class" << endl;
}

void base:: method_2() {
	cout << "Call to method method_2" << " of base class " << endl;
	method_1();
}

void derived:: method_1() {
	cout << " Call to method method_1" << " of derived class" << endl;
}

int main() {
	derived D;
	D.method_2();
	return 0;
}
```

Executing the code, we will have the following result:

```
Call to method method_2 of base class
Call to method method_1 of base class
```

But this is not the desired result, because in the *main* function method *method_2*, inherited from the base class, was called, but method *method_1* called by *method_2* was determined at compile-time. Consequently, although *method_1* was overridden in the derived class, the method from the base class was called, not the overridden one.

This shortcoming can be overcome by introducing the notion of *virtual* methods. If a method is virtual, then for every call of it, the implementation corresponding to the class hierarchy will not be determined at compile-time, but at execution, depending on the type of the object on which the

call was made. This property is called *dynamic binding*, and if a method is determined at compile-time, we talk about static binding.

We have seen that if the `virtual1.cpp` program is executed, methods *method_1* and *method_2* from the base class are called. But *method_1* being overridden in the derived class, we wanted the overridden method to be called instead of the one from the base class.

This can be realised by declaring *method_1* as a virtual method. Thus, for each call of *method_1*, the implementation of the method that will be called is determined at execution time and not at compile-time. So, the method *method_1* is determined through dynamic binding.

In the C++ language a method is declared virtual in the following way: in the declaration of the class, the header of the method will start with the keyword virtual.

If a method is declared virtual in the base class, then the methods overriding it will be considered virtual in all derived classes of the hierarchy.

For the above example the declaration of the base class is modified in the following way:

```
class base {
public:
      virtual void method_1();
      void method_2();
};
```

The result of the execution becomes:

```
Call to method method_2 of base class
Call to method method_1 of derived class
```

So, *method_1* from the derived class is called indeed.

Further, we will present another example where the neccessity of introducing virtual methods appears. Let us define the class *fraction* referring to rational numbers, having as attributes the numerator and the denominator of the fraction. The class has to have a constructor, the default value for the numerator being 0 and for the denominator being 1, and two methods: *product*, for computing the product of two fractions and *multiply*, for multiplying the current object with a fraction given as parameter. Also, the *fraction* class has to have a method for displaying a rational number. Using class fraction as base class, we will define the derived class *fraction_write*, in which the *product* method will be overridden, so that besides executing the multiplication, the operation is displayed on *stdout*. The *multiply* method will not be overridden, but the performed operation has to be displayed on the standard output in this case too.

File `fvirt1.cpp`:

```
#include <iostream>
#include <string>
using namespace std;

class fraction {
protected:
	int numerator;
	int denominator;
public:
	fraction(int numerator1 = 0, int denominator1 = 1);
	fraction product(fraction& r); //computes the product of two fractions, but
                                       //does not simplify
	fraction& multiply(fraction& r);
	std::string toString();
};

fraction::fraction(int numerator1, int denominator1) {
	numerator = numerator1;
	denominator = denominator1;
}

fraction fraction::produs(fraction& r) {
	return fraction(numerator * r.numerator, denominator * r.denominator);
}

fraction& fraction::multiply(fraction& q) {
	*this = this->product(q);
	return *this;
}

std::string fraction::toString() {
	char* s = new char[5 + 3 + 5 + 1];
	if (denominator){
		std::string s1 = std::to_string(numerator);
		std::string s2 = std::to_string(denominator);
		std::string s = s1 + " / " + s2;
		return s;
	}
	else
		return "Incorrect fraction";
}


class fraction_write : public fraction{
public:
	fraction_write(int numerator1 = 0, int denominator1 = 1);
	fraction product(fraction & r);
};

inline fraction_write::fraction_write(int numerator1, int denominator1) : fraction
(numerator1, denominator1) {
}
```

```
fraction fraction_write::product(fraction& q) {
	fraction r = fraction(*this).product(q);
	cout << "(" << this->toString() << ") * (" << q.toString() << ") = " << r.toString()
<< endl;
	return r;
}

int main() {
	fraction p(3, 4), q(5, 2), r;
	r = p.multiply(q);
	cout << p.toString() << endl;
	cout << r.toString() << endl;
	fraction_write p1(3, 4), q1(5, 2);
	fraction r1, r2;
	r1 = p1.product(q1);
	r2 = p1.multiply(q1);
	cout << p1.toString() << endl;
	cout << r1.toString() << endl;
	cout << r2.toString() << endl;
	return 0;
}
```

Executing the code we will get:

```
15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8
```

We can observe that the result is not the desired one, since the multiplication operation was displayed only once, namely for the expression `r1 = p1.product(q1)`. In the case of the expression `r2 = p1.multiply(q1)` the multiplication was not displayed. This is caused by the fact that the *multiply* method was not overriden in the derived class, so the method inherited from class *fraction* was called. Inside the *multiply* method, the method *product* is called, but since this method was determined at compile-time, the one referring to class *fraction* was called and not the one from the derived class *fraction_write*. So, the operation was displayed only once.

The solution is, like for the previous example, to declare a virtual method, namely to declare method *product* virtual. So, the declaration of the base class is modified in the following way:

```
class fraction {
protected:
	int numerator;
	int denominator;
public:
	fraction(int numerator1 = 0, int denominator1 = 1);
```

```
    virtual fraction product(fraction& r); //computes the product of two fractions, but
                                           //does not simplify
    fraction& multiply(fraction& r);
    std::string toString();
};
```

After making these modifications, the result of the execution will be:

```
15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8
```

It can be observed that the operation was displayed twice, once for each expression. Virtual methods, just like other methods, do not neccessarily have to be overridden in the derived classes. If they are not overridden, the method from a superior level is inherited.

The corresponding implementation of virtual methods is determined based on some automatically built and managed tables. Objects of classes with virtual methods contain a pointer to this table. Because of this, managing virtual methods requires more memory and a longer execution time.

## 4.4. Abstract classes

In case of a complicated class hierarchy, the base class can have some properties which we know exist, but we can only define them for the derived classes. For example, let us consider the class hierarchy from Figure 1.

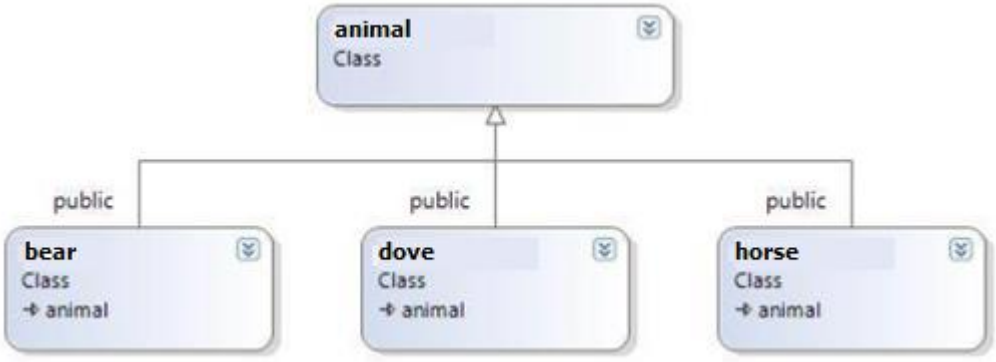



Figure 1. *Class hierarchy of animals*

We notice that we can determine some properties that refer to the derived classes, for example: average weight, lifespan and speed. These properties will be described using different

methods. Theoretically, average weight, lifespan and speed exist for the *animal* class too, but they are too complicated to determine, and are not important for us in such a general context. Still, for a uniform treatment, it would be good, if these three methods were declared in the base class and defined in the derived classes. For this purpose the notion of a *pure virtual method* is introduced.

A pure virtual method is a method which is declared in a given class, but is not defined in it. It *has to* be defined in a derived class. A pure virtual method is declared in the following way: the regular header of the method is preceeded by the *virtual* keyword, and the header ends with *= 0*. As its name and declaration show, a pure virtual method is a virtual method, so the selection of the implementation of the method from the class hierarchy will be done during the execution of the program.

Classes that contain at least one pure virtual method are called *abstract classes*.

Since abstract classes contain methods that are not defined, it is not possible to create objects that belong to an abstract class. If a pure virtual method was not defined in the derived class, than the derived class will also be abstract and it is impossible to define objects belonging to it.

Let us consider the above example and write a program that determines whether a *dove*, a *bear* or a *horse* is fat or skinny, fast or slow and old or young, respectively. The result will be displayed by a method of the *animal* class, which is not overridden in the derived classes.

File `abstract1.cpp`:

```
#include <iostream>
#include <string>
using namespace std;

class animal {
protected:
	double weight; // kg
	double age; // years
	double speed; // km / h
public:
	animal(double g, double v1, double v2);
	virtual double average_weight() = 0;
	virtual double average_lifespan() = 0;
	virtual double average_speed() = 0;
	int fat() {
		return weight > average_weight();
	}
	int fast() {
		return speed > average_speed();
	}
	int young() {
		return 2 * age < average_lifespan();
	}
	std::string toString();
```

```
};

animal::animal(double g, double v1, double v2){
	weight = g;
	age = v1;
	speed = v2;
}

std::string animal::toString(){
	return fat() ? "fat, " : "skinny, ";
	return young() ? "young, " : "old, ";
	return fast() ? "fast" : "slow";
}


class dove : public animal {
public:
	dove(double g, double v1, double v2) : animal(g, v1, v2) {}
	double average_weight() { return 0.5; }
	double average_lifespan() { return 6; }
	double average_speed() { return 90; }
};


class bear : public animal {
public:
	bear(double g, double v1, double v2) : animal(g, v1, v2) {}
	double average_weight () { return 450; }
	double average_lifespan () { return 43; }
	double average_speed () { return 40; }
};


class horse : public animal {
public:
	horse(double g, double v1, double v2) : animal(g, v1, v2) {}
	double average_weight () { return 1000; }
	double average_lifespan () { return 36; }
	double average_speed () { return 60; }
};


int main() {
	dove d(0.6, 1, 80);
	bear b(500, 40, 46);
	horse h(900, 8, 70);
	cout << d.toString() << endl;
	cout << b.toString() << endl;
	cout << h.toString() << endl;
	return 0;
}
```

We notice that, although the *animal* class is abstract, it is useful to introduce it, since there are many methods that can be defined in the base class and inherited without modifications in the three derived classes.

## 4.5. Interfaces

The C++ language has no notion of interface, which exist in Java or C# languages. But any abstract class that contains only pure virtual methods can be considered an interface. Obviously, in this case no attributes will be declared inside the class. The *animal* abstract class contains both attributes and nonvirtual methods, so it cannot be considered an interface.

Further, we will introduce an abstract class, *Vehicle*, which contains only pure virtual methods, and two classes derived from it.

File `vehicle.cpp`:

```
#include <iostream>
using namespace std;

class Vehicle {
public:
	virtual void Start() = 0;
	virtual void Stop() = 0;
	virtual void Go(int km) = 0;
	virtual void Stand(int min) = 0;
};

class Bicycle : public Vehicle {
public:
	void Start();
	void Stop();
	void Go(int km);
	void Stand(int min);
};

void Bicycle::Start() {
	cout << "The bicycle starts." << endl;
}

void Bicycle::Stop() {
	cout << "The bicycle stops." << endl;
}

void Bicycle::Go(int km) {
	cout << "The bicycle goes " << km << " kilometres." << endl;
}

void Bicycle::Stand(int min) {
	cout << "The bicycle stands " << min << " minutes." << endl;
}
```

```
class Car : public Vehicle{
public:
	void Start();
	void Stop();
	void Go(int km);
	void Stand(int min);
};

void Car::Start() {
	cout << "The car starts." << endl;
}

void Car::Stop() {
	cout << "The car stops." << endl;
}

void Car::Go(int km) {
	cout << "The car goes " << km << " kilometres." << endl;
}

void Car::Stand(int min) {
	cout << "The car stands " << min << " minutes." << endl;
}

void Route(Vehicle *v) {
	v->Start();
	v->Go(3);
	v->Stand(2);
	v->Go(2);
	v->Stop();
}

int main() {
	Vehicle *b = new Bicycle;
	Route(b);
	Vehicle *c = new Car;
	Route(c);
	delete c;
	delete b;
}
```

In the *main* function two dynamic objects of type *Bicycle* and *Car*, respectively, are declared, and in this way, calling the *Route* function we will get different results, although this function has as formal parameter only a pointer to the abstract class *Vehicle*.

# 5. Proposed problems

1. Write a program in one of the programming languages Python, C++, Java, C# that:
    a. Defines a class **Student** with:
        – an attribute ***name*** of type string;
        – an attribute ***grades*** containing a list of grades (integer numbers),

    constructurs, access methods and a methods that computes the average grade of the student.
    b. Defines a function that gets an object of type **Student** and returns True if all the grades of the student are greater than 4.
    c. Write the specifications for the methods defined in class **Student** and also for the function defined at b.

2. Write a program in one of the programming languages Python, C++, Java, C# that:
    a. Defines a class **Student** with:
        – an attribute ***name*** of type string;
        – an attribute ***grades*** containing a list of grades (integer numbers),

    constructurs, access methods and a methods that computes the average grade of the student.
    b. Defines a subprogram that gets an object of type **Student** and prints the name of the students and the grades of the student in descending order.
    c. Write the specifications for the methods defined in class **Student** and also for the function defined at b.

3. Write a program in one of the programming languages Python, C++, Java, C# that:
    a. Defines a class **Point2D** with the following protected attributes:
        – ***name*** of type char;
        – ***coordX*** of type real number;
        – ***coordY*** of type real number,

    and the following public methods:
        – constructor with parameters to initialize all attributes;
        – method ***toString*** that returns the following string representation: *name(coordX, coordY)*, for example *A(2, 3)*;

- method with no arguments ***distanceToOrigin*** that computes and returns the Euclidian distance from the current point to the origin (0, 0) of a 2D coordinate system.

b. Defines a class **SetOfPoints** with the following private attributes:
   - ***noPoints*** of type integer;
   - ***points*** of type array (list) with elements of type **Point2D**,

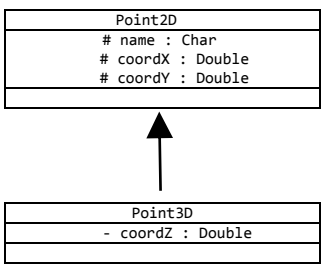
   and the following public methods:
   - constructor with no parameters;
   - getter methods to access both attributes,
   - method ***add(p)*** to add a point *p* to the array *points*,
   - method ***filterPoints(limit)*** where limit is a real number that keeps in the list of points only those that have distance to origin greater than *limit*,
   - method ***sortPoints*** that sorts points alphabetically based on *name*.

c. Defines a function ***display(setpoints)***, where *setpoints* is of type **SetOfPoints**, that prints on the screen the points from *setpoints*.

d. Defines a function ***processing1()*** that:
   - creates an object of type **SetOfPoints**, containing the following points: C(1,2), A(2,3), B(1,2), A(2,4), D(2,5),
   - sorts these points alphabetically (using method ***sortPoints***), and
   - prints the sorted set of points (using function ***display***).

e. Defines a class **Point3D** derived from class **Point2D** with one private attribute:
   - **coordZ** of type real number,

   and the following public methods:
   - constructor with parameters to initialize all attributes;
   - method **toString** tht returns the following string representation: *name(coordX, coordY, coordZ)*, for example *B(5,2,4)*;
   - method with no arguments ***distanceToOrigin*** that computes and returns the Euclidian distance from the current point to the origin (0, 0, 0) of a 3D coordinate system.



f. Defines a function **processing2(val)** that:
   - creates an object of type **SetOfPoints**, containing the following points: A(1,2), B(1,2,3), C(1,2), D(3,4,5);
   - determines and prints on the screen the points from the set having a distance to origin greater than *val*.

g. Defines the main function of the program and:
   - call the function ***processing1()***;
   - call the function ***processing2(10)***.

# General bibliography


1. Alexandrescu, *Programarea modernă in C++. Programare generică si modele de proiectare aplicate*, Editura Teora, 2002.
2. Angster Erzsébet: *Objektumorientált tervezés és programozás Java*, 4KÖR Bt, 2003.
3. Bjarne Stroustrup: *A C++ programozási nyelv*, Kiskapu kiadó, Budapest, 2001.
4. Bjarne Stroustrup: *The C++ Programming Language Special Edition*, AT&T, 2000.
5. Boian F.M. Frentiu M., Lazăr I. Tambulea L. *Informatica de bază*. Presa Universitară Clujeana, Cluj, 2005
6. Bradley L. Jones: *C# mesteri szinten 21 nap alatt*, Kiskapu kiadó, Budapest, 2004.
7. Bradley L. Jones: *SAMS Teach Yourself the C# Language in 21 Days*, Pearson Education,2004.
8. Cormen, T., Leiserson, C., Rivest, R., *Introducere în algoritmi*, Editura Computer Libris Agora, Cluj, 2000
9. Eckel B., *Thinking in C++*, vol I-II, http://www.mindview.net
10. Ellis M.A., Stroustrup B., *The annotated C++ Reference Manual*, Addison-Wesley, 1995
11. Frentiu M., Lazăr I. *Bazele programării*. Partea I-a: Proiectarea algoritmilor
12. Herbert Schildt: Java. *The Complete Reference*, Eighth Edition, McGraw-Hill, 2011.
13. Robert Sedgewick: *Algorithms*, Addison-Wesley, 1984
14. Simon Károly, *Kenyerünk Java. A Java programozás alapjai*, Presa Universitară Clujeană, 2010.