
Lecture 2: Description of algorithms

- Pseudocode
 - How to specify the data
 - Some simple examples
 - The refining technique: decomposing a problem in subproblems. Algorithms and subalgorithms.
-

Problem solving methods are usually described in a mathematical language. Even if it is a rigorous language it is not always adequate for algorithms description since it does not allow specifying some aspects which are important in an algorithm. The algorithms can be described through:

- *Flowchart*. They allows describing the algorithm in a graphical manner. Each basic instruction has a corresponding graphical sign and the compound statements are described through connected signs. The connections are specified through arrows indicating in this way also the order of instructions execution.
- *Pseudocode*. Is a language based on a small vocabulary which contains keywords corresponding to statements and identifiers used to specify the data. As all languages it is based on some rules, but they are less restrictive than those used in programming languages. In pseudocode, expressions written in a mathematical form are accepted.

1 Pseudocode

The pseudocode which we shall use to describe algorithms allows describing simple and structured statements as follows.

Assignment. The assignment of a value (which can be the result of an expression evaluation) to a variable v can be described as:

$$v \leftarrow \langle \text{expression} \rangle$$

The expressions allow describing computations and consist of: *operands* and *operators*. The operands can be variables or constant values. The mostly used operators are:

- *arithmetical operators*: + (addition), - (subtraction), * (multiplication), / (division), ^ (power), DIV (integer quotient), MOD or % (remainder);
- *relational operators*: = (equality), < (strictly less than), ≤ or <= (less than or equal), > (strictly greater than), ≥ or >= (greater than or equal);
- *logical operators*: OR (disjunction), AND (conjunction), NOT (negation).

Read. To transfer an input data to a variable v one specify:

read v

Write. To transfer the value of a variable to the output device one specify:

write < expresie >

Sequential structure. A sequence of n statements is described through:

< statement 1 >
< statement 2 >
⋮
< statement n >

Conditional statement. When we have to choose between two alternative statements one specify:

if < condition > **then** < statement 1 > **else** < statement 2 > **endif**

At the execution of a conditional statement, if the condition evaluates to *true* then is executed *statement 1* otherwise (if the condition evaluates to *false*) is executed *statement 2*. The particular case of a conditional statement can be described as:

if < condition > **then** < statement > **endif**

In this case the specified statement is executed only if the condition is *true*. When the condition is *false* one passes to the next statement in the algorithm.

These conditional statements are illustrated in fig. 1

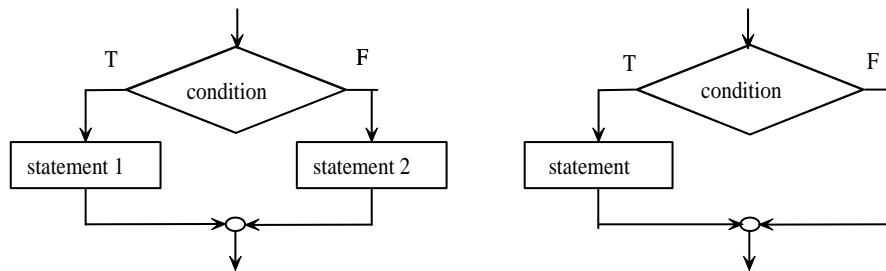


Figure 1: Conditional statements

Loop statement. It is used when some operations have to be repeated many times. There are two basic variants: the preconditioned variant (**while** ... **do**) and a postconditioned variant (**repeat** ... **until**).

The first variant can be specified as:

while < condition > **do** < statement > **endwhile**

This type of loop statement is illustrated in fig.2a. When the **while** loop is executed, the inner statement is repeated as long as the specified condition is **true**. If the condition is *false* from the beginning then the inner statement is not executed at all. If in the inner statement there are no operations which switches the condition from true to false then the loop repeats forever (this is a so-called infinite loop).

A particular case of this type of loop statement is the so-called FOR loop statement. It is used when one apriori knows the number of repetitions. In this case it is used a counting variable. Its value varies from an initial value, $v1$ to a final one, $v2$, by increasing/decreasing it with a step value, $step$. This type of statement can be specified as:

for $v \leftarrow \overline{v1, v2, step}$ **do** < statement > **endfor**

and it is equivalent (for the case when $step$ is positive) with:

```

v ← v1
while v ≤ v2 do
  <statement>
  v ← v + step
endwhile

```

When $step$ has a negative value only the condition has to be changed with $v \geq v2$. When $step = 1$ then one can skip it in the description. This kind of loop is illustrated in fig. 2b.

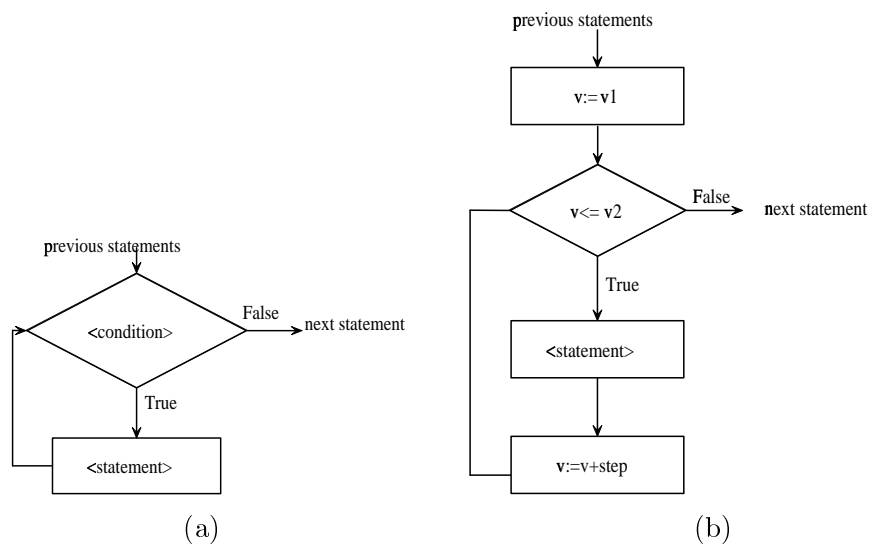


Figure 2: (a) WHILE statement; (b) FOR statement

The postconditioned variant can be specified as:

repeat < statement > **until** < condition >

When the **repeat** statement is executed, the inner statement is repeated until the condition becomes *true*. Thus, this statement is executed at least once, even if the condition is *true* from the beginning. If the condition does not become *true* then the loop repeats forever. The flowchart of this statement is illustrated in fig.3.

2 How to specify the data

Sometimes, when we describe the algorithms we want to specify also the nature of involved data. For instance, to specify simple data one can use: **integer** (for integers), **real** (for reals), **boolean**

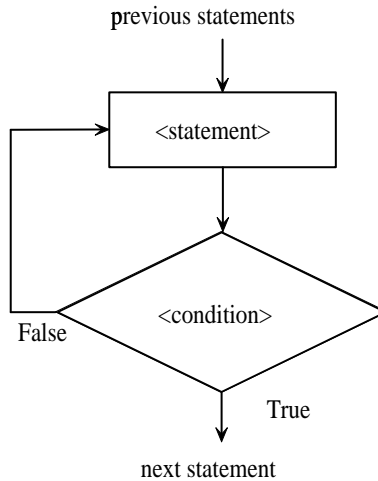


Figure 3: REPEAT statement

(for logical values which can have only one of the two truth value: **true** and **false**) and **char** (for characters).

Thus, if a is an integer, b a real, c a logical value and d a character value one can specify:

integer a
real b
boolean c
char d

The array structure can be described by specifying the type of the elements and the range of indices. For instance, for a one-dimensional array of integers one can specify **integer** $t1[n1..n2]$, while a bi-dimensional array of reals can be specified by **real** $t2[m1..m2, n1..n2]$.

The array $t1$ can be used to model a finite sequence (or a set) having $n2 - n1 + 1$ element, while the array $t2$ can be used for a matrix having $m2 - m1 + 1$ rows and $n2 - n1 + 1$ columns. The elements of an array are identified by their indices. For instance the i th element of $t1$ is specified as $t1[i]$, and that on the i th row and j th column of $t2$ is specified as $t2[i, j]$. In a similar manner one can specify sub-arrays containing all consecutive elements whose indices are between two values. For instance, if $n1 \leq i1 < i2 \leq n2$, by $t1[i1..i2]$ one specifies all elements of $t2$ having indices between $i1$ and $i2$. If $n1 > n2$ then the array $t[n1..n2]$ is considered to be empty.

3 Some simple examples

Sequential structure.

Problem. Compute the area of a triangle knowing its side lengths, a , b and c .

Solving method. We can use the well-known Heron formula $area = \sqrt{p(p-a)(p-b)(p-c)}$ where p is half of the triangle perimeter.

The algorithm.

```

real  $a, b, c, p, area$  // declaration of variables
read  $a, b, c$  //input data
 $p \leftarrow (a + b + c)/2$  //half perimeter computation
 $area \leftarrow \sqrt{p(p-a)(p-b)(p-c)}$  //area computation
write  $area$  //output the result

```

Conditional statements.

Problem. Compute the real roots of the equation $ax^2 + bx + c = 0$ for a, b and c real values.

Solving method. We can use the classical method for solving second degree equations, based on the relations:

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}, \quad \Delta = b^2 - 4ac.$$

The algorithm.

```

real  $a, b, c, delta, x1, x2$  // declaration of variables
read  $a, b, c$  //input data
 $delta \leftarrow b^2 - 4ac$  //discriminant computation
//Analysis of the discriminant's sign
if  $delta < 0$  then write "The equation doesn't have real roots"
else
  if  $delta = 0$  then
     $x1 \leftarrow -b/(2a)$ 
    write "Double root:",  $x1$ 
  else
     $x1 \leftarrow (-b + \sqrt{delta})/(2a)$ 
     $x2 \leftarrow (-b - \sqrt{delta})/(2a)$ 
    write  $x1, x2$ 
  endif
endif
endif
hline

```

Problem. Let us consider three real values stored in three variables a, b and c . (a) Find the smallest value; (b) Print the values in an increasing order.

Algorithms. (a) A first variant, based on comparing all pairs of variables is:

```

real  $a, b, c, min$  // variables declaration
read  $a, b, c$  //reading the input data
if  $a < b$  then
  if  $a < c$  then  $min \leftarrow a$ 
  else  $min \leftarrow c$ 
  endif
else
  if  $b < c$  then  $min \leftarrow b$ 
  else  $min \leftarrow c$ 
  endif
endif
write  $min$  //outputs the result

```

Another variant, which is more compact and can be easier extended to arrays is that based on supposing that the first value is the minimum and then comparing this minimum with the other values. When one finds a value smaller than the current minimum this value becomes the new minimum.

```

real  $a, b, c, min$ 
read  $a, b, c$ 
 $min \leftarrow a$  // the initial value of the minimum
if  $b < min$  then  $min \leftarrow b$  endif // modification of the minimum if it is necessary
if  $c < min$  then  $min \leftarrow c$  endif
write  $min$ 

```

(b) A first variant is that based on the identification of all six possible situations. A simple conditional statement is associated to each situation:

```

real  $a, b, c$ 
read  $a, b, c$ 
if  $a < b$  AND  $b < c$  then write  $a, b, c$  endif
if  $a < c$  AND  $c < b$  then write  $a, c, b$  endif
if  $c < a$  AND  $a < b$  then write  $c, a, b$  endif
if  $c < b$  AND  $b < a$  then write  $c, b, a$  endif
if  $b < a$  AND  $a < c$  then write  $b, a, c$  endif
if  $b < c$  AND  $c < a$  then write  $b, c, a$  endif

```

However, this variant works only in the case of distinct values. Another variant is based on two by two comparisons:

```

real  $a, b, c, min$ 
read  $a, b, c$ 
if  $a < b$  then
    if  $b < c$  then write  $a, b, c$ 
        else if  $a < c$  write  $a, c, b$ 
            else write  $c, a, b$ 
        endif
    endif else
    if  $a < c$  then write  $b, a, c$ 
        else if  $b < c$  then write  $b, c, a$ 
            else write  $c, b, a$ 
        endif
    endif
endif

```

(c) The basic idea of this variant is to put the smallest value in a and to swap, if it is the case, the variables b and c .

```

real  $a, b, c$ 
read  $a, b, c$ 
if  $a > b$  then  $a \leftrightarrow b$  endif
if  $a > c$  then  $a \leftrightarrow c$  endif
if  $b > c$  then  $b \leftrightarrow c$  endif

```

Loop statements. The basic elements of a loop statement are: (i) the initial values of the variable involved in the loop; (ii) the statement(s) which are repeated; (iii) the stopping condition.

Summations. Compute the following sums: (a) $\sum_{i=1}^n i^3$, $n \in N^*$; (b) $\sum_{i=1}^n x^i/i!$ for $x \in (0, 1)$ and $n \in N^*$; (c) $\sum_{i=1}^{n(\epsilon)} x^i/i!$, where $x \in (0, 1)$, $\epsilon > 0$ and $n(\epsilon) \in N^*$ is the smallest natural number for which $x^{n(\epsilon)}/n(\epsilon) < \epsilon$ (this sum can be considered as an approximation - with a precision of ϵ - of the infinite sum $\sum_{i=1}^{\infty} x^i/i!$).

Algorithms. (a) The repeated operation should be the addition of a term. The loop is terminated when all terms have been summed. This process can be described using anyone from the three loop statements:

```

integer  $S, i, n$ 
read  $n$ 
 $S \leftarrow 0$ 
 $i \leftarrow 1$ 
while  $i \leq n$ 
     $S \leftarrow S + i^3$ 
     $i \leftarrow i + 1$ 
endwhile
write  $S$ 

```

```

integer  $S, i, n$ 
read  $n$ 
 $S \leftarrow 0$ 
for  $i \leftarrow 1, n$  do  $S \leftarrow S + i^3$  endfor
write  $S$ 

```

```

integer  $S, i, n$ 
read  $n$ 
 $S \leftarrow 0$ 
 $i \leftarrow 1$ 
repeat
     $S \leftarrow S + i^3$ 
     $i \leftarrow i + 1$ 
until  $i > n$ 
write  $S$ 

```

(b) The sum can be rewritten as $\sum_{i=1}^n T(i)$ where $T(i) = x^i/i!$. The problem could be solved in the same manner as in the previous example, by computing for each $i = \overline{1, n}$ the value of the term $T(i)$. However one can remark that there exists a relationship between successive terms. This relationship allows us to compute the value of $T(i)$ starting from $T(i-1)$:

$$T(i) = \frac{x^i}{i!} = \frac{x^{i-1}}{(i-1)!} \cdot \frac{x}{i} = T(i-1) \cdot \frac{x}{i}$$

Since $n \in N^*$ it follows that the sum contains at least one term, thus the computation can be described using a **repeat** statement:

```

integer  $i, n$ 
real  $x, S$ 
read  $x, n$ 
 $S \leftarrow 0$ 
 $T \leftarrow 1$ 
 $i \leftarrow 1$ 
repeat
   $T \leftarrow T * x/i$  // finding the value of the current term
   $S \leftarrow S + T$  // adding the term to the sum
   $i \leftarrow i + 1$  // passing to the next term
until  $i > n$ 
write  $S$ 

```

(c) Since the number of terms is not known one have to use another stopping condition: the loop is stopped after that the first term smaller than ϵ has been added to the sum (since the sequence $T(i)$ is decreasing all terms $T(i)$ for $i > n(\epsilon)$ are smaller than ϵ). The algorithm can be described as:

```

integer  $i$ 
real  $x, S, \epsilon$ 
read  $x, \epsilon$ 
 $S \leftarrow 0$ 
 $T \leftarrow 1$ 
 $i \leftarrow 1$ 
repeat
   $T \leftarrow T * x/i$  // computing the current term
   $S \leftarrow S + T$  // adding the term to the sum
   $i \leftarrow i + 1$  // next term
until  $T < \epsilon$ 
write  $S$ 

```

Greatest common divisor of two natural numbers. Let a and b be two strictly positive integers. The greatest common divisor (g.c.d) is the largest integer that divides both numbers.

Euclid's method. The method proposed by Euclid to compute the g.c.d is one of the oldest algorithms known (it appeared in Euclid's Elements around 300 BC). The Euclid's method can be described as follows. Given two natural numbers a and b , first check if b is zero. If yes, then the g.c.d of a and b is a . If no, compute r , the remainder obtained by dividing a by b . Replace a with b , b with r , and start the process again. The process continues until one obtains a remainder equal to zero. Then the previous remainder (which, obviously, is not zero) will be the g.c.d. The sequence of operations described above is:

$$\begin{aligned}
 a &= bq_1 + r_1, & 0 \leq r_1 < b \\
 b &= r_1q_2 + r_2, & 0 \leq r_2 < r_1 \\
 r_1 &= r_2q_3 + r_3, & 0 \leq r_3 < r_2 \\
 &\vdots \\
 r_{i-2} &= r_{i-1}q_i + r_i, & 0 \leq r_i < r_{i-1} \\
 &\vdots \\
 r_{n-1} &= r_nq_{n+1} + r_{n+1}, & r_{n+1} = 0
 \end{aligned}$$

It is easy to remark that the method is an algorithm since the sequence of divisions is a finite one because the remainders' sequence, r_i , is a strictly decreasing sequence of naturals. Thus after a

finite number of divisions the remainder becomes 0.

The algorithm. The method can be described by using a **while** loop statement as:

```
integer  $a, b, d, i, r$ 
read  $a, b$ 
 $d \leftarrow a$  // dividend's initialization
 $i \leftarrow b$  // divisor's initialization
 $r \leftarrow d \text{ MOD } i$  //remainder computation
while  $r \neq 0$  do
   $d \leftarrow i$  //this is the new dividend
   $i \leftarrow r$  //this is the new divisor
   $r \leftarrow d \text{ MOD } i$  //this is the new remainder
endwhile
write  $i$  //outputs the greatest common divisor
```

An equivalent form described by using a **repeat** loop statement is:

```
integer  $a, b, d, i, r$ 
read  $a, b$ 
 $d \leftarrow a$ 
 $i \leftarrow b$ 
repeat
   $r \leftarrow d \text{ MOD } i$ 
   $d \leftarrow i$ 
   $i \leftarrow r$ 
until  $r = 0$ 
write  $d$ 
```

Computing the average of a finite sequence of numbers. Let us consider a finite sequence of real numbers (x_1, x_2, \dots, x_n) . The average value of these numbers, \bar{x} satisfies:

$$\bar{x} = \left(\sum_{i=1}^n x_i \right) / n.$$

The algorithm. The sequence can be represented as an array $x[1..n]$ and the computation is based on a loop having a fixed number of repetitions (n):

```
real  $x[1..n], \text{average}$ 
integer  $n, i$ 
read  $x[1..n]$  //inputs the array's elements
 $\text{average} \leftarrow 0$  // initialization
for  $i \leftarrow \overline{1, n}$  do //sum's computation
   $\text{average} \leftarrow \text{average} + x[i]$  // adding a new element
endfor
 $\text{average} = \text{average} / n$  //division by  $n$ 
write  $\text{average}$ 
```

4 Decomposing a problem in subproblems. Algorithms and sub-algorithms

One of the most simple strategy for algorithmic problem solving is to decompose the problem into some subproblems and to try to solve them independently. Thus the algorithm corresponding to the entire problem will consist on some subalgorithms each one corresponding to a subproblem. Sometimes the subproblems are similar thus they could be solved by using the same method perhaps applied to different data. Thus we can consider that the subalgorithms are applied to some *generic data* which will be replaced by actual values only when the subalgorithm is *called* and executed. These generic data are usually called formal parameters. A subalgorithm will either return a result value or modify some of the algorithm's variables. The general structure of a subalgorithm is:

```
<subalgorithm name> (< generic data >)  
  < local data >  
  < statements >  
  return < results >
```

Example 1.

Problem. Let us consider a matrix $(a_{ij})_{i=\overline{1,m}, j=\overline{1,n}}$. Find the greatest value of the minimal values of the matrix rows, i.e.: $\max_{i=\overline{1,m}} \min_{j=\overline{1,n}} a_{ij}$.

Method. We construct an array containing the minimal value of each row in the matrix:

$$b_i = \min_{j=\overline{1,n}} a_{ij}, \quad i = \overline{1,m}$$

Then we compute the minimal value of this array. Thus the problem can be decomposed in two subproblems: that of finding the minimal value of an array and that of finding the maximal value of an array.

Subalgorithms. To find the minimal value of an array we initialize a variable which finally will contain the minimum with the first element of the array. Then this value is successively compared with the array's elements. When a value smaller than the current minimum is found then the value of the variable is replaced with this one. This computation can be described as:

```
minimum (real  $x[1..n]$ )    {the formal parameter is an array with  $n$  elements}  
real  $min$     // the variable which will contain the minimal value  
integer  $i$     // counter variable used to scan the array  
 $min \leftarrow x[1]$     // variable's initialization  
for  $i \leftarrow \overline{2, n}$  do    // scanning the array  
  if  $min > x[i]$  then  $min \leftarrow x[i]$  endif    //the current minimal value is replaced if it is the case  
endfor  
return  $min$     // return the result
```

The subalgorithm which finds the maximum value of an array with m elements:

```

maximum (real  $y[1..m]$ ) //the formal parameter is an array with  $n$  elements


---


real  $max$  // the variable which will contain the minimal value
integer  $i$  // counting variable used to scan the array
 $max \leftarrow y[1]$  // initialization of maximum
for  $i \leftarrow 2, m$  do
    if  $max < y[i]$  then  $min \leftarrow x[i]$  endif {the current minimal value is replaced if it is the case}
endfor
return  $max$ 

```

The algorithm. The algorithm consists in constructing (by using the **minimum** subalgorithm) the array containing the rows minimum and in finding the maximum of this array (by using the **maximum** subalgorithm).

```

real  $a[1..m, 1..n]$  //the matrix
real  $b[1..m]$  //the array containing the rows minimum
real  $c$  the variable which will contain the result
integer  $m, n, i$ 
read  $a[1..m, 1..n]$  //matrix's input
for  $i \leftarrow 2, n$  do //constructing the minimal values array
     $b[i] \leftarrow \text{minimum}(a[i, 1..n])$  // finding the minimum of  $i$ th row.
endfor
 $c \leftarrow \text{maximum}(b[1..m])$  //finding the minimum of  $b$ 
write  $c$  //result's output

```

Example 2. *The problem.* Sort in a decreasing order a sequence by only reversing the order of the elements in a final subsequence (in particular the entire sequence). For instance starting from the sequence 4, 5, 3, 6, 1, 2 by applying the transformations: 4, 5, 3, 6, 1, 2 \rightarrow 4, 5, 3, 2, 1, 6 \rightarrow 6, 1, 2, 3, 5, 4 \rightarrow 6, 1, 2, 3, 4, 5 \rightarrow 6, 5, 4, 3, 2, 1 one obtains the sorted sequence.

Solving method. The basic idea is the following: identify the maximal value in the sequence and reverse the order of elements in the subsequence starting with this maximal value. Then reverse the entire sequence. Thus the largest value arrives on the first position. Apply the same steps for the subsequence starting with the second element and so on.

The algorithm can be described as follows:

```

sort(real  $x[1..n]$ )
integer  $i, imax$ 
for  $i \leftarrow 1, n - 1$  do
     $imax \leftarrow \text{maxim}(x[i..n])$ 
    if  $imax \neq i$  then
        reverse( $x[imax..n]$ )
        reverse( $x[i..n]$ )
    endif
endfor

```

where **maximum** and **reverse** are described as follows:

```

maximum(real x[s..d])
integer imax, i
imax ← s
for i ← s + 1, d do
    if x[imax] < x[i] then imax ← i endif
endfor
return imax

```

```

reverse(real x[s..d])
integer mid, i
mid = ⌊(s + d)/2⌋
for i ← s, mid do
    x[i] ↔ x[s + d - i]
endfor

```

Example 3.

Problem. Let us consider a natural number of 10 distinct digits (for instance, 6309785421). Compute the next number (in increasing order) in the sequence of all naturals consisting of 10 distinct digits.

Solving method. We consider that the number is represented by the array of its digits, $x[1..10]$ when the more significant digit is placed on the first position. As long as the digits of the number are not in a decreasing order (meaning 9876543210) the number which we are searching for can be obtained by executing the following steps:

Step 1. Search from right to left for the first pair (x_{i-1}, x_i) having the property $x_{i-1} < x_i$. If the initial number is not 9876543210, then such a pair exists.

Step 2. Find

$$x_k = \min\{x_j | j = \overline{i, n} \text{ with } x_j > x_{i-1}\}$$

i.e. the smallest element of the subarray $x[i..n]$ which is greater than x_{i-1} (the existence of such an element is ensured by the property $x_i > x_{i-1}$).

Step 3. Swap x_{i-1} with x_k . Thus one obtains a number greater than the initial number.

Step 4. Sort increasingly the subarray $x[i..n]$. The aim of this sorting step is to obtain the smallest number (consisting of distinct digits) which is greater than the initial number.

Algorithm. The algorithm can be decomposed in subalgorithms corresponding to the processing steps specified above:

- **Identify:** used to find the first pair (x_{i-1}, x_i) having the property $x_{i-1} < x_i$. The subalgorithm receives as an input parameter the array x and returns the value of the index i . If $i = 1$ then the initial number does not have a successor satisfying the requested restrictions.
- **Minimum:** used to find the smallest value in the subarray $x[i..n]$ which is greater than x_{i-1} . The subalgorithm will receive as parameters the array $x[i..n]$ and the index k of the smallest element in $x[i..n]$ which is larger than x_{i-1} . It will return the index of the minimum.
- **Sorting:** used to sort increasingly the subarray $x[i..n]$. In this case, since $x[i..n]$ is already decreasingly sorted it suffices to reverse the order of the elements, by using the same subalgorithm as in the previous example.

The general structure of the algorithm is:

```
Successor(integer  $x[1..n]$ ) integer  $i, k$   
 $i \leftarrow \text{Identify}(x[1..n])$   
if  $i = 1$  then write "It does not exist"  
else  
     $k \leftarrow \text{Minimum}(x[1..n], i)$   
     $x[i - 1] \leftrightarrow x[k]$   
    Reverse( $x[i..n]$ )  
    write  $x[1..n]$   
endif
```

The subalgorithms **Identify** and **Minimum** can be described as follows:

```
Identify (integer  $x[1..n]$ )  
integer  $i$   
 $i \leftarrow n$   
while ( $i > 1$ ) and ( $x[i] < x[i - 1]$ ) do  $i \leftarrow i - 1$  endwhile  
return  $i$ 
```

```
Minimum (integer  $x[i..n]$ )  
integer  $j$   
 $k \leftarrow i$   
for  $j \leftarrow i + 1, n$  do  
    if  $x[j] < x[k]$  and  $x[j] > x[i - 1]$  then  $k \leftarrow j$   
return  $k$ 
```

In the following, most of the algorithm will be described as subalgorithms and, to simplify the description, the type of parameters and local variables will not be specified.