5 Recursive Algorithms

5.1 Introduction

5.1.1 Definition. Iteration and Recursion

- A definition is said to be recursive, if it refers to an object which partially consists or is defined in terms of itself.
  - But a recursive definition as „a flower is flower” which can represent in poetry an entire universe, in science in general, and in mathematics in special, has no value.

- A very important characteristic of recursion is to specify the recursive definition in an evolving manner which avoid the circularity.
  - An example of a correct recursive definition is: "A bouquet of flowers is: (1) either a flower; (2) or a flower added to the bouquet".
    - The statement (1) is the initial condition which initiates the recursive definition.
    - The statement (2) specifies the recursive (evolutionary) definition itself.

- The iterative variant of the same definition is "a bouquet of flowers either consists in a flower, or in two flowers, or in three flowers, or..., etc."
  - As we can notice that, the recursive definition is simple and elegant, instead, the iterative definition is bearish and inelegant.

- Generally speaking, an object is said to be recursive, if it partially consists or is defined in terms of itself.

- Recursion is encountered not only in mathematics, but also in daily life.
  - Who has never seen an advertising picture which contains itself?

- Recursion is a particularly powerful technique in mathematical definitions.
  - A few familiar examples are those of natural numbers, tree structures, and of certain functions [5.1.1.a]:

- Natural numbers:
• (1) 0 is a natural number.

• (2) The successor of a natural number is a natural number.

• **Binary tree structure:**

  - (1) $\emptyset$ is an **empty binary tree**, containing no nodes.

  - (2) If $t_1$ and $t_2$ are binary trees, then the structures consisting of a node $r$ with two descendants $t_1$ and $t_2$ is also a binary tree.

  ![Binary Tree Diagram]

  \[ r \]
  \[ t_1 \quad \quad t_2 \]

• The **factorial function** $f(n) = n!$ (defined for positive integers):

  - (1) $f(0) = 1 \quad (0! = 1)$

  - (2) If $n > 0$, then $f(n) = n \times f(n-1) \quad (n! = n \times (n-1)!)$

• The **power of recursion** evidently lies in the possibility of defining an infinite set of objects by a finite statement, as we could observe above.

• Known as a fundamental concept in mathematics, recursion became a powerful programming facility, once with apparition of the high level programming languages which implements it as an intrinsic characteristic (ALGOL, PASCAL, C, JAVA etc.).

  - In the same manner, as in mathematics, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.

• In the programming context, recursion is intimately tied with iteration, and for avoid any confusion, we will define the two concepts from programming point of view.

• **Iteration** is the repeated execution of a portion of a program, until a certain specified condition is fulfilled, respectively while the condition is fulfilled.

  - One possible scenario (there are many others): Each execution is finalized, that means all the portion of code is executed, the condition is verified, and if the condition is not fulfilled, the execution is repeated.

  - Such classical examples are repetitive structures **WHILE, REPEAT** and **FOR**.

• **Recursion** presumes too, the repeated execution of a portion of program.
• In **contrast** with iteration, in **recursion**, the condition is verified **during** the execution of the program and not at its end as in iteration.

• If the condition is not fulfilled, the entire portion of program is called **again**, as **subprogram (procedure or function)** of itself, particularly as **subprogram** of the **unfinished original portion of code**.

• When the condition is **fulfilled**, the execution of the calling program is **resumed** exactly in the point in which it called itself.

• This happens for **all the calls** executed previous the fulfilling of the implied condition, in reverse order.

• **Recursive algorithms**, however, are primarily appropriate when the problem to be solved, or the function to be computed, or the data structure to be processed, are already **defined in recursive terms**.

• In general, a **recursive program** $P$ can be expressed as a composition $P$ of a sequences of statements (not containing $P$) $S_i$ and $S_j$ and $P$ itself [5.1.1.b].

• We have to specify that either $S_i$ or $S_j$ can be empty sequence.

\[
P \equiv P \{S_i, P, S_j\} \quad [5.1.1.b]
\]

---

### 5.1.2 The Recursion Mechanism

• The necessary and sufficient tool for expressing **programs recursively** is the **procedure** or **subroutine** or **function**, for it allows a statement to be given a name by which this statement may be invoked.

• If a procedure $P$ contains an **explicit reference** to itself, then it is said to be **directly recursive** (figure 5.1.2.a. (a)).

• If $P$ contains a reference to another procedure $Q$, which contains a (direct or indirect) reference to $P$, then $P$ is said to be **indirectly recursive** (figure 5.1.2.a.(b)).

• The use of recursion may therefore **not** be immediately apparent from the program text.

![Procedure A](a)
Intuitively, the **execution model of a recursive procedure** is presented in figure 5.1.2.b.

- Each associated representation of the procedure $P$, represents a **call instance** of this **procedure**.

- The instances call each other until the condition $c$ become false.

- In this moment, it **returns** on the **calls chain**, in **reverse order**.
  - For **each instance**, the execution is **continued** from the interrupting point, until the end, as the arrows indicate.

- **Overlapping** imaginary all this figures in a single one, we obtain the **execution model** of a **recursive procedure**.

- It is common to **associate** a **set of local objects** with a **procedure**, i.e., a set of variables, constants, types, and procedures which are defined locally to this procedure and have no existence or meaning outside this procedure.
  - Usually, this set of local objects includes the **calling parameters** of the procedure, too.
  - This set is named **context of the procedure**.
• Each time such a procedure is activated recursively, a new set of local, bound variables is created, and saved in the system stack. In other words, a new context of the procedure is created and saved in the system stack.

• Although the context’s variable have the same names as their corresponding elements in the set local to the previous instance of the procedure, their values are distinct, and any conflict in naming is avoided by the rule of scope of identifiers.

• The rule is:
  - The identifiers always refer to the most recently created set of variables.
  - The same rule holds for procedure parameters, which by definition are bound to the procedure.

• As recursive calls are executed, the context of each call is saved in the system stack.

• As the execution of the current instance is finished, the context of calling instance is restored, by eliminating the context from the head of the stack.

  • This mechanism for implementing recursion is illustrated in figure 5.1.2.c.

![Fig.5.1.2.c. Mechanism for implementing recursion](image)

• In connection with this mechanism, we have to underline some aspects:

  • (1) There are not problems regarding parameters transmission between recursive calling instances, if the parameters are transmitted by value, because they are saved in stack as instance’s context.

  • (2) If the parameters are transmitted by address (type VAR in Pascal, respectively type pointer in C), in order to make possible the transmission of the values of this kind of
parameters inter instances, is necessary to declare supplementary local parameters (surrogates), transmitted by value, one for each such parameter.

- This parameters will assure the transmission of the implicated values as in situation (1).

- Before returning from the recursive procedure, the implied calling parameters must be reassigned with the surrogate values [5.1.2.a].

```plaintext
PROCEDURE A(x: Type1; VAR y: Type2; VAR z: Type3);
VAR u,w: TypeX;
y1: Type2; {local (surrogate) parameter for y}
z1: Type3; {local (surrogate) parameter for z}
BEGIN
  ...
  A(x,y1,z1);
  ...
  y:= y1;  {reassign y}
  z:= z1;  {reassign z}
END; {A}
```

```plaintext
void a (type1 x, type2* y, type3* z){
typeX u,w;
type2 y1;  /* local (surrogate) parameter for y*/
type3 z1;  /* local (surrogate) parameter for z*/
  ...
  a(x,&y1,0);
  ...
  *y=y1;  /*reassign y*/
  *z=z1;  /*reassign z*/
} /*a*/
```

- (3) In the case of parameters of type reference is necessary to take into consideration the fact that, as result of the recursive procedure call, automatically, the actual reference parameters are assigned [5.1.2.b].

```plaintext
PROCEDURE A(r: TypeReference);
BEGIN
  ...
  A(r^.right);  {simultaneous with the call, r is assigned with r^.right value}
  ...
END; {A}
```

```plaintext
void a (type_reference r){
  /*[5.1.2.b]*/
  a(r->right);  /*simultaneous with the call, r is assigned with r^.right value*/
  ...
} /*a*/
```
• Like repetitive statements, recursive procedures introduce the possibility of **non-terminating computations**, and thereby also the necessity of considering the **problem of termination**.

  • A **fundamental requirement** is evidently that the recursive calls of \( P \) are subjected to a condition \( c \), which at some time becomes false.

• The scheme for **recursive algorithms** may therefore be expressed more precisely by one of the forms [5.1.2.c], [5.1.2.d] if \( S_i \) is null, respectively [5.1.2.e] if \( S_j \) is null.

\[
P \equiv P \begin{cases} 
S_i, & \text{IF } c \text{ THEN } P, S_j \end{cases} \quad [5.1.2.c]
\]

\[
P \equiv P \begin{cases} 
\text{IF } c \text{ THEN } P, S_j \end{cases} \quad [5.1.2.d]
\]

\[
P \equiv P \begin{cases} 
S_i, \text{IF } c \text{ THEN } P \end{cases} \quad [5.1.2.e]
\]

• **For repetitions**, the **basic technique** of **demonstrating termination** consists of:

  • (1) Defining a **function** \( f(x) \) (\( x \) shall be in the set of variables), such that \( f(x)<0 \) implies the terminating condition.

  • (2) **Proving** that \( f(x) \) decreases during each repetition step.

    • \( f \) is called the **variant of the repetition**.

• In the same manner, **termination of a recursion** can be proved by showing that each execution of \( P \) decreases some \( f(x) \), and that \( f(x)<0 \) implies \( \neg c \).

  • A particularly evident way to ensure termination is to associate a (value) parameter, say \( n \), with \( P \), and to recursively call \( P \) with \( n-1 \) as parameter value.

  • Substituting \( n>0 \) for \( c \) then guarantees termination.

• This may be expressed **formally** by the following program schemata [5.1.2.f]:

\[
P(n) \equiv P \begin{cases} 
S_i, \text{IF } n > 0 \text{ THEN } P(n-1), S_j \end{cases} \quad [5.1.2.f]
\]

• In practical applications it is mandatory to show that the ultimate **depth of recursion** is **not** only **finite**, but that it is actually **quite small**.

  • The reason is that upon each recursive activation of a procedure \( P \) some amount of storage is required to accommodate its variables (procedure instance context).

  • In addition to these **local variables**, the **current state of the computation** must be recorded too in the **context**, in order to be **retrievable** when the new activation of \( P \) is terminated and the old one has to be resumed.
Neglecting these aspects, the memory space allocated to the system stack can be easily overflowed, this happening often in recursive programming.

5.1.3 Example of Recursive Algorithms

To clarify the concepts presented above, we will present three simple recursive algorithms.

5.1.3.1 Simple Recursive Algorithm

The next program illustrates the working principle of a recursive algorithm.

For this purpose the procedure Reverse is defined, based on the following specification:

- Procedure Reverse reads a character and prints it.
- For this purpose the local variable z is declared.
- The procedure verifies if the read character is blanc:
  - If not, the procedure call itself recursively.
  - If yes, procedure print the character z. [5.1.3.1.a]

```pascal
PROCEDURE Reverse;
VAR z: char;
BEGIN [5.1.3.1.a]
  Read(z);
  IF z<>' ' THEN Reverse;
  Write(z)
END; {Reverse}
```

The execution of Reverse displays initially the characters in order, as they were read, until the first blanc character is read.

- Each self call presumes the saving of the call context in the system stack. In our case the context is only the local variable z.

```c
void reverse(){
    char z;
    scanf("%c", &z); /*[5.1.3.1.a]*/
    if (z !=" ")
        reverse();
    printf("%c", z);
} /*reverse*/
```

The context is only the local variable z.
• When the first blanc character is read, the recursive calls of the procedure are suspended, and the last instance is executed until its termination. This happens in reverse order for all previous recursive calls.

• In our case, this presumes displaying the current value of variable \( z \).

• This chain of returns will display at the beginning a blanc character, followed by the characters displayed in the reverse order of their reading.

• This happens because, each termination of the execution for current call instance, determines the return in the previous call, which presumes the restoration of the call context.

• Because, the contexts are stored in stack, their restoration is done in reverse order of their storing.

• In fact, for each introduced word terminated by a blanc, the Reverse procedure, supplies the word followed by the same word written in reverse order.

### 5.1.3.2 Algorithm for Linked Lists Traversal

• The next example presents an recursive algorithm for a linked list traversal\[5.1.3.2.a\].

```pascal
{Linked list traversal – Pascal variant}

TYPE
  TypeList = ^TypeNode;
  TypeNode = RECORD
    data: TypeData;
    next: TypeList
  END;

PROCEDURE ListTraversal(p:TypeList); [5.1.3.2.a]
  BEGIN
    IF p<>nil THEN
      BEGIN
        [1] Processing(p^.data);
        [2] ListTraversal(p^.next)
      END
  END; {ListTraversal}
```

```c
/* Linked list traversal – C variant */

typedef struct type_node* type_list;

typedef struct {
  int data;
  type_list next;
} type_node;

void list_traversal(type_list p) /*[5.1.3.2.a]*/
```c
{ 
    if (p!=0) {
        /*[1]*/ processing(p->data);
        /*[2]*/ list_traversal(p->next);
    }
} /*list_traversal*/
/*---------------------------------------------------------------*/
```

- The algorithm is not only very simple and in the same time very elegant, but by simple interchanging of the statements Processing and ListTraversal ([1] with [2]) we obtain the traversal of the list in reverse order.

- In the last case, the algorithm behavior can be explained as follows:
  - The list is parsed starting with current position until is end. After the parse is finished, the information from current node is processed.

- The consequence of the algorithm execution:
  - The data content of a certain position p in list is processed only after all the nodes succeeding to p have been processed.

5.1.3.3 Algorithm for Solving the Towers of Hanoi Problem

- The next example refers to the well known Towers of Hanoi problem, whose specification is the following:
  - Three rods A, B and C are considered.
  - We consider also a set on n perforated discs, each having another dimension, which are located in decreasing order from bottom-up on rod A (figure 5.1.3.3.a).

![Fig.5.1.3.3.a](image.png)

- The task is to move the n disks from rod A to rod C such that they are ordered in the original way.

- This has to be achieved under the specific constraints:
• (1) In each step exactly **one disk** is moved from one rod to another rod.

• (2) A disk **may never** be placed on top of a **smaller disk**.

• (3) Rod B may be used as an auxiliary store.

• The problem seem to be simple, but its classical solving requires patience, accuracy and a volume of time which increase exponential with \( n \).

  • A **recursive** approach is recommended because it’s simple and elegant.

• The **recursive solution** presume first to solve a **simple case**, followed by a corresponding **generalization**.

  • For the beginning we consider the **simple case** when exist only two disks numbered as 1 (the smallest, placed above) and 2 (the bigger, placed bellow).

  • The moving of the two disks presumes the following steps:
    • (1) Disk 1 is moved from A to B.
    • (2) Disk 2 is moved from A to C.
    • (3) Disk 1 is moved from B to C.

  • This simple case is **generalized** for \( n \) disks (figure 5.1.3.3.b):
    • (1) \( n-1 \) disks (placed above) are moved from A to B, by C.
    • (2) Disk \( n \) is moved from A to C.
    • (3) \( n-1 \) disks are moved from B to C, by A.

![Fig.5.1.3.3.b. Generalized model for Towers of Hanoi problem.](image)

• The **recursive implementation** of this **model** is presented in [5.1.3.3.a].
PROCEDURE  *TowersOfHanoi*(DisksNumber:integer;  A{from}, B(to), C{by}: TypeRod);
PROCEDURE  *MoveDisk*(x{from}, y{to}: TypeRod);
  BEGIN
        *move disk  from x to y
  END;  {MoveDisk}
BEGIN  {TowersOfHanoi}
  IF  DisksNumber =1  THEN
        MoveDisk(A,C)                       [5.1.3.3.a]
  ELSE
        BEGIN
              *TowersOfHanoi  (DisksNumber-1, A, B, C);
              MoveDisk(A,C);
              *TowersOfHanoi  (DisksNumber-1, B, C, A)
        END  {ELSE}
  END  {TowersOfHanoi}
-------------------------------------------------------------------
/* Towers of Hanoi algorithm - C variant */
void  *towers_of_hanoi*(int disks_number, type_rod a /*from*/,
                          type_rod b /*to*/, type_rod c /*by*/);

void  *move_disc*(type_rod x /*from*/, type_rod y /*to*/)
  {
        *move disk  from x to y;
  }      /*move_disk*/

void  *towers_of_hanoi*(int disks_number, type_rod a /*from*/,
                         type_rod b /*to*/, type_rod c /*by*/)
  {
        /*towers_of_hanoi*/
        if (disksNumber ==1)
                move_disk(a,c);                   /*[5.1.3.3a]*/
        else
                {
                        *towers_of_hanoi*(disks_number-1, a, b, c);
                        move_disk(a,c);
                        *towers_of_hanoi*(disks_number-1, b, c, a);
                } /*else*/
  } /*towers_of_hanoi*/
/*---------------------------------------------------------------*/

• The procedure  *MoveDisk* moves  a disk from a source to a destination rod.

• Procedure  *TowersOfHanoi* is based on the  generalized model and consists in two self recursive calls, and a call of the  *MoveDisk* procedure.

5.2  The Using of Recursion
5.2.1  General Case of Using Recursion
Recursive algorithms are particularly appropriate when the underlying problem or the data to be treated are defined in recursive terms.

- This does not mean, however, that such recursive definitions guarantee that a recursive algorithm is the best way to solve the problem.

- In fact, the explanation of the concept of recursive algorithm by such inappropriate examples has been a chief cause of creating widespread apprehension and antipathy toward the use of recursion in programming, and of equating recursion with inefficiency.

- In spite of such interpretations, recursion remains a fundamental technique in programming, with an application domain very well defined.

- Programs in which the use of algorithmic recursion is to be avoided can be characterized by a schema which exhibits the pattern of their composition.

  - The equivalent schemata are shown below. Their characteristic is that there is only a single call of P either at the end (or the beginning) of the composition.

- We will present an example for building an recursive program starting from the generic recursive model [5.2.1.a] derived from [5.1.2.d].

  
  \[
  P \equiv \text{IF } c \text{ THEN } P[S_i,P]\quad [5.2.1.a]
  \]

  
  This model is proper for calculating values defined by recursive relations.

- A classical example is represented by factorial numbers \( f_i = i! \) [5.2.1.b].

  
  \[
  i = 0,1,2,3,4,5,... \\
  f_i = 1,1,2,6,24,120,... \quad [5.2.1.b]
  \]

  
  The element with index 0 is defined as being equal to 1.

  - Each next element is defined in terms of its predecessor, using a recursive definition [5.2.1.c].

    \[
    f_{i+1} = (i+1) * f_i \quad [5.2.1.c]
    \]

  - Starting with this formula, two variable \( i \) and \( f \) are introduced. In the \( i \)-th level of recursion, \( S_i \) can be described as [5.2.1.d].
\[ S_i \equiv (i := i+1; f := i*f;) \quad [5.2.1.d] \]

- Substituting \( S_i \) in [5.2.1.a] with [5.2.1.d], we obtain the formal recursive procedure [5.2.1.e], to which the main program [5.2.1.f] is added.

\[ P \equiv \text{IF } i<n \text{ THEN } (i := i+1; f := i*f; P) \quad [5.2.1.e] \]

\[ i := 0; f := 1; P; \quad [5.2.1.f] \]

- An implementation variant of procedure \( P \) appears in [5.2.1.g].

{Computation of factorial – Variant 1 Pascal}

```pascal
PROCEDURE P;
BEGIN
    IF i<n THEN [5.2.1.g]
        BEGIN
            i:= i+1; f:= i*f; P
        END
END;{P}
```

{Computation of factorial – Variant 1 C}

```c
void p()
{
    if (i<n) { /*[5.2.1.g]*/
        i++;
        f*=i;
        p();
    }
} /*p*/
```

- As was mentioned, recursive algorithms are recommended to be used mainly to implement problems which are defined in a recursive manner.
  - Despite this fact, recursion can be used even for problems which have not a recursive nature.
- For example, in [5.2.1.h] is presented a recursive algorithm for a linear search in an array \( a \).
Recursive search in a linear array - Pascal variant

VAR  n: integer;  x: TypeElement;
     a: ARRAY[1..n] OF TypeElement;

FUNCTION Search(i: TypeIndex): TypeIndex;
BEGIN
IF i>n THEN Search:= 0 ELSE
  IF a[i]=x THEN Search:= i ELSE Search:= Search(i+1)
END;{Search}
{Initial call: j:=Search(1);}

/* Recursive search in a linear array - C variant */

int n; type_element x;
type_element *a;

type_index search(type_index i){
  type_index search_result;
  if (i>n)
    search_result=0;
  else /*[5.2.1h]*/
    if (a[i-1]==x)
      search_result=i;
    else
      search_result= search(i+1);
  return search_result;
} /* search*/
/*initial call: int j=search(1)*/

• It’s obvious that such a approach is possible but it’s **forced** and **inefficient**. In such situation, the **iterative** variant is more suitable.

5.2.2 An Recursive Algorithm for Calculating a Factorial Value

• Procedure \(P\) for calculating a **factorial value** [5.2.1.g], usually is implement as a **function subprogram** for the following reasons:

  • (1) A **function** can be used directly as constitutive of an expression.

  • (2) A **function** can be assigned with a calculated value.

• Under this considerations, variable \(f\) in procedure \(P\) becomes unnecessary, and \(i\) is replaced by the function calling parameter \(n\) [5.2.2.a].
FUNCTION Fact(n: integer): integer;
BEGIN
  IF n=0 THEN
    Fact:= 1                               \[5.2.2.a\]
  ELSE
    Fact:= n*Fact(n-1)
  END;{Fact}

/*Recursive function for computing a factorial value – C variant*/

int fact(int n)
{
  int fact_result;
  if (n==0)
    fact_result=1;                    \[5.2.2.a\]*/
  else
    fact_result=n*fact(n-1);
  return fact_result;
}  \/*fact*/

• The recursive function presented above, which calculates an integer value for factorial, is limited to a maximum value of n, from reasons related to representation of integers in a computing system.
  • The conversion to a variant which calculates a real value for a factorial, non affected by any constraint, is trivial.

• In figure 5.2.2.a an intuitive representation of recursive calls of function Fact for n=4, is presented [De89].

![Recursive calls of function Fact for n=4.](image-url)
In figure 5.2.2.b, are presented in the same intuitive manner, the **successive returns** from the recursive calls of function **Fact**, having as result the calculus of the factorial value.

![Diagram of recursive calls of function Fact](image)

**Fig.5.2.2.b.** Solving recursive calls of function **Fact** for \( n=4 \).

In figure 5.2.2.c appears the structure of recursive calls respectively of their solving in a **tree of calls** representation.
Fig. 5.2.2.c. Tree calls for function **Fact** (4).

- In this case, because the function **Fact** contains only one recursive call, the calls tree has a linear structure.
  - Each recursive call adds an element at the end of this list, and the solving of a recursive call reduces the list from the end to the beginning.
- We can notice that, for calculating the factorial of order \( n \), \( n+1 \) recursive calls of the function **Fact** are necessary.
- Of course, in the case of factorial computing, the recursion can be very simple replaced by an iterative loop [5.2.2.b].

```pascal
VAR i, fact: integer;
i := 0; fact := 1;
WHILE i < n DO
  BEGIN
    i := i + 1; fact := fact * i
  END;
```

```c
int i, fact;
i = 0;
fact = 1;
while (i < n) /*[5.2.2.b]*/
{
  i++; fact *= i;
}
```

- In figure 5.2.2.d. appears the graphical representation of the performance profile of the algorithm for factorial calculus in recursive and iterative variants.
  - In fact are represented the execution times of the both algorithms, on the same computing system, for different values for \( n \).
  - Is to be noticed that, whilst the both algorithms are linear with \( n \), that means they are \( O(n) \), the recursive algorithm performance deteriorates with \( n \), related to the iterative algorithm performance.
  - The analytical expressions of the execution times for the two situations appears in [5.2.2.c][De84].
5.2.2. Fibonacci Numbers

• There are other examples of recursive definition which are implemented more efficient as iterative algorithms.

• A classical example is the calculation of the Fibonacci numbers.

• Fibonacci numbers of first order are defined in a recursive manner [5.2.3.a]:

\[
\begin{align*}
F_{n+1} &= F_n + F_{n-1} \quad \text{for } n > 0 \\
F_1 &= 1; \quad F_0 = 0
\end{align*}
\]

• This definition suggests an immediate implementation using a recursive algorithm [5.2.3.b].

{Calculating Fibonacci numbers - recursive algorithm - Pascal}

FUNCTION Fib(n: integer): integer;
BEGIN
  IF n=0 THEN Fib:= 0 ELSE [5.2.3.b]
  IF n=1 THEN Fib:= 1 ELSE
    Fib:= Fib(n-1) + Fib(n-2)
  END; {Fib}

/*Calculating Fibonacci numbers - recursive algorithm - C variant*/
int fib(int n)
{
  int fib_result;
  if (n==0) fib_result=0; else /*[5.2.3.b]*/
  if (n==1) fib_result=1; else
fib_result = fib(n-1) + fib(n-2);
return fib_result;
} /*fib*/

/*-----------------------------------------------*/

- Unfortunately, the recursive model is inefficient because the numbers of recursive calls increases exponential with \(n\).

- In figure 5.2.3.a. is represented the calls tree for \(n=5\).

![Calls tree for procedure Fib (n=5)](image)

Fig.5.2.3.a. Calls tree for procedure Fib \((n=5)\).

- As we can notice:
  - The calls tree is a **binary tree** because in procedure Fib there are two recursive self calls.
  - In our case, the **calls tree** traversal requires 15 calls, every tree node presuming a call of the procedure.
  - The algorithm inefficiency grows with \(n\).

- It’s obvious, the **Fibonacci numbers** can be calculated more efficient using an **iterative algorithm** that avoids the recomputation of the same values by use of auxiliary variables \(x = \text{Fib}_i\) and \(y = \text{Fib}_{i-1}\) [5.2.3.c].

{Calculating Fibonacci numbers- iterative algorithm - Pascal}

\[
i := 1; \ x := 1; \ y := 0;
\text{WHILE} \ i < n \ \text{DO} \ \text{BEGIN}
\quad z := x; \ i := i+1;
\quad x := x+y; \ y := z
\text{END}
\]
```c
/* Calculating Fibonacci numbers- iterative algorithm - C */

i=1; x=1; y=0;
while (i<n)          /*[5.2.3.c]*/
{
    z=x; i++;
    x=x+y; y=z;
}
/*---------------------------------------------------------------*/
```

- The auxiliary variable z can be avoided if we use the following assignments \( x := x+y \) and \( y := x-y \).

### 5.2.4 Eliminating Recursion

- **Recursion** represents an excellent **programming facility** which allows a simple, concise and elegant formulation of recursive algorithms.
  - Nevertheless, when the **performance aspects are important**, to **avoid using of recursion** is **warmly recommended**.
  - Otherwise, **avoid of recursion** is **recommended** whenever we have at our disposal an **iterative variant**.
  - In fact, the implementation of recursion on equipments which are non recursive, proves that **any recursive algorithm** can be transformed in an **iterative** one.
  - Subsequently, we attack in a theoretically manner the problem of **conversion** a **recursive algorithm** in an **iterative** one.
  - In this approach we distinguish two cases:
    - (1) The case in which the **recursive call** appears at the **end of the procedure**, as its last statement, named "tail recursion".
      - In this situation, **recursion** can be replaced with a simple **iteration loop**.
      - This is possible because, the return from a nested call resume in the same time the **termination** of the correspondent call instance, as consequence, the call context doesn’t have to be restored.
      - Thus, if a procedure \( P (x) \) contains as its **last statement** a self call to the procedure itself \( P (y) \):
        - The **recursive** call can be replaced by an **assignment instruction** \( x := y \), followed by rerun (goto beginning) of the \( P \) code.
• $y$ can be an expression too.

• $x$ must be transmitted by **value**, its value being stored in a specific location associated to the call.

• $x$ can be transmitted by **reference** if $y$ is exactly $x$.

• If $P$ has more parameters, each will be processed as $x$ and $y$.

• This modification is valid because, the resume of the $P$ execution, with the new value of $x$, has the same effect as calling $P(y)$ and returning from this call.

• In general, this type of **recursive programs** can be converted in **iterative forms** as follows:

  • [5.2.4.a] becomes [5.2.4.b] and [5.2.4.c] becomes [5.2.4.d].

  \[
  P(x) \equiv P \ (S_i, \text{IF } c \text{ THEN } P(y)) \quad \text{[5.2.4.a]}
  \]

  \[
  P(x) \equiv P \ (S_i, \text{IF } c \text{ THEN } [x:=y; \text{ resume } P]) \quad \text{[5.2.4.b]}
  \]

  \[
  P(n) \equiv P \ (S_i, \text{IF } n>0 \text{ THEN } P(n-1)) \quad \text{[5.2.4.c]}
  \]

  \[
  P(n) \equiv P \ (S_i, \text{IF } n>0 \text{ THEN } [n:=n-1; \text{ resume } P]) \quad \text{[5.2.4.d]}
  \]

• Excellent examples in this sense the are iterative implementation of **factorial** or **Fibonacci numbers**.

• (2) Case 2 presumes that the recursive call(s) is (are) realized from the inner of the procedure [5.2.4.e].

  \[
  P \equiv P \ (S_i, \text{IF } c \text{ THEN } P, S_j) \quad \text{[5.2.4.e]}
  \]

• The iterative implementation of this situation presumes the **explicit processing** of the **calls stack** by the **programmer**.

  • For each call, the **call’s instance context**, must be saved in the **stack** by the **programmer**.

  • When the current call instance is **finished**, and it returns in the preceding instance, this **context** must be restored by the **programmer** too.

• The conversion activity in this case, has a **high degree of specificity**, in dependence of the nature of the recursive algorithm to be converted.

  • Usually this activity is more awkward, more complicated, and embarrasses the algorithm understanding.

  • An example in this sense is the **non recursive partitioning** presented in section &3.2.6.
• **Recursion** has its specific domains, very well defined, in which is used with big success.

• As a **general rule**, the **algorithms** whose **nature is recursive**, is recommended to be implemented as **recursive procedures**, as in the examples in this chapter and in the following.

### 5.3 Examples of Recursive Algorithms

#### 5.3.1. Algorithms which Implements Recursive Definitions

• We will present two examples of algorithms which implements recursive definitions.

  • The first one is **Euclid’s algorithm** for determining the **greatest common divisor** of two integer numbers.

  • The second is the algorithm for **conversion of a arithmetic expression** from **infix** to **postfix** format.

#### 5.3.1.1 Euclid’s Algorithm

• The **Euclid’s algorithm** determines the **greatest common divisor of two integers** (g.c.d).

  • The algorithm is defined in a **recursive** manner:

    • (1) If **one** of the numbers is **null**, their g.c.d is the other number.

    • (2) If **none** of the numbers is **null**, then g.c.d is not modified, if one of the numbers is replaced with the rest of its dividing with the other number.

  • Starting from this recursive definition, we can conceive a simple function for determining the g.c.d. of two integers [5.3.1.1.a].

------------------------------

{ Euclid’s algorithm implementation – Pascal variant }

FUNCTION Gcd(m,n: integer): integer;
BEGIN
  IF n=0 THEN Gcd:= m                       \[5.3.1.1.a\]
  ELSE Gcd:= Gcd(n, m MOD n)
END;{Gcd}
------------------------------

/* Euclid’s algorithm implementation – C variant */

int gcd(int m,int n)
{
  int gcd_result;
  if (n==0)
    gcd_result=m;                      \*[5.3.1.1.a]\*
\begin{verbatim}

else
    gcd_result = gcd(n, m\%n);
return gcd_result;

/*gcd*/
/*---------------------------------------------------------------*/

• In figure 5.3.1.1.a appears the execution trace of this algorithm for values \( m = 18 \) and \( n = 27 \).

<table>
<thead>
<tr>
<th>m</th>
<th>n</th>
<th>Gcd(n, m \ MOD n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>27</td>
<td>Gcd (27, 18 mod 27)</td>
</tr>
<tr>
<td>27</td>
<td>18</td>
<td>Gcd (18, 27 mod 18)</td>
</tr>
<tr>
<td>18</td>
<td>9</td>
<td>Gcd (9, 18 mod 9)</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>n = 0; Gcd = 9</td>
</tr>
</tbody>
</table>

Fig.5.3.1.1.a. Execution trace Euclid’s algorithm.

5.3.1.2 Recursive Algorithm for Transforming Arithmetic Expressions from Infix to Postfix Format (Polish Notation)

• Usual arithmetic expressions (infix format) can be defined in a recursive manner using syntactic diagrams (figure 5.3.1.2.a).

\end{verbatim}
• The indirect nature of the recursion of this definition, consists in the fact that the
definition of Factor includes again the definition of Expression in parentheses.

• Our problem is to develop an algorithm which accept an valid arithmetic expression in infix
format, and converts it in postfix format i.e. its corresponding Polish notation.

• We have to underline that Polish notation presumes the representation of a simple arithmetic
operation in the form "operand operand operator" and not as "operand operator operand"
presumed by infix notation.

• For example a+b becomes ab+ and a+b* (c-d) becomes abcd-*+ .

• The advantage of the polish notation is that it indicates the correct order of arithmetic operations
execution, without using parentheses, and the expression can be calculated by a simple parsing from
left to right.

• The problem of this transformation is to discover first the two operands (each could be
intricate) and then to place the operator after.

• From this reason, until the second operator is discovered, the additive operator
respectively the multiplicativ operator inside a term are stored in the local variables
op respectively op1.

• The required transformation can be realized building an individual conversion procedure
for each syntactical construction (Expression, Term, Factor).

• Because these syntactical constructions are defined recursively, the
 corresponding procedures must be defined in the same manner.

• The transformation algorithm appears in [5.3.1.2.a].

-------------------------------------------------------------------
{Recursive algorithm for transforming an arithmetic expression form
infix in postfix format – Pascal variant}

TYPE line = string[80];
VAR infix,post: line;
    c: char;
    i: integer; {index in infix expression}
    j: integer; {index in postfix format}

PROCEDURE ReadChar(ex: line; VAR ch: char);
{supplies the next character of the infix expression, eliminating
the blanks}
BEGIN
    REPEAT
        ch:= ex[i]; i:= i+1;
    UNTIL ch<>' '
END; {ReadChar}
PROCEDURE WriteChar(VAR ex: line, ch: char);
    {store the character in the specified area}
BEGIN
    ex[j]:= ch; j:= j+1
END; // WriteChar

PROCEDURE Expression;
    VAR op: char;
PROCEDURE Term;
    VAR op1: char;
PROCEDURE Factor
    BEGIN // Factor
        IF c = '(' THEN
            BEGIN
                ReadChar(infix,c); Expression
                {when returns from expression ch=')'"
            END ELSE
            BEGIN
                WriteChar(post,c)
            END;
        ReadChar(infix,c);
    END; // Factor
    BEGIN // Term
        Factor;
        WHILE (c = '*' OR c = '/') DO
            BEGIN
                op1:= c; ReadChar(infix,c);
                Factor;
                WriteChar(post,op1)
            END
        END; // Term
    BEGIN // Expression
        Termen;
        WHILE (c = '+' OR c = '-') DO
            BEGIN
                op:= c; ReadChar(infix,c);
                Term;
                WriteCar(post,op)
            END
        END; // Expression
    
/* Recursive algorithm for transforming an arithmetic expression
form infix in postfix format - C variant */

typedef char* line;
line infix,post;
char c;
int i; /*index in current expression*/
int j; /*index in postfix format*/

void expression();
static void term();
void read_char(line ex, char* ch)
/* supplies the next character of the infix expression, eliminating
the blanks */
{
    do {
        *ch=ex[i]; i++;    
    } while (!(*ch!=' '));
} /*read_char*/

void write_char(line* ex, char ch)
/* store the character in the specified area */
{
    (*ex)[j]=ch; j=j+1;
} /*write_char*/                       /*[5.3.1.2.a]*/

static void factor()
{
    if (c == '('){
        read_char(infix,&c);
        expression(); /* when returns from expression ch=')'*/
    } else{
        write_char(&post,c);
    }
    read_char(infix,&c);
} /*factor*/

static void term()
{
    char op1;
    factor();
    while ((c =='*')||(c =='/'))
    {
        op1=c; read_char(infix,&c);
        factor();
        write_char(&post,op1);
    } /*while*/
} /*term*/

void expression()
{
    char op;
    term();
    while ((c =='+')||(c =='-'))
    {
        op=c; read_char(infix,&c);
        term();
        write_char(&post,op);
    } /*while*/
} /*expression*/
/*---------------------------------------------------------------*/
• In the above conversion algorithm are defined:

  • The procedures Expression, Term and Factor in accordance with the diagrams from figure 5.3.1.2.a.

  • The local variables op and op1 used for storing the additive respectively multiplicative operators.

  • The procedure ReadChar which supplies the next character from the infix expression, eliminating the blanks.

  • The variable ch which stores in each moment the car supplied by procedure ReadChar.

  • The infix expression is stored as a string of characters in the array infix and scanned with index i.

  • The postfix format of the arithmetic expression is assembled in array post using index j.

  • The indices i and j are initialized in the main program which initially calls procedure Expression.

  • The initial arithmetic expression must be placed between parentheses.

5.3.2 Dividing Algorithms

5.3.2.1 Divide and Conquer Algorithms

• One of the fundamental methods for algorithms design is based on “divide and conquer” technique.

• The basic principle of this technique is the following:

  • (1) The problem to be solved is decomposed (divided) in a number of sub-problems, whose solving is simpler and from whose solutions can be assembled the solution for the initial problem.

  • (2) The step (1) is repeated recursively until the sub-problems become banal, and their solutions obviously.

• A typical application of the divide and conquer technique is based on the recursive model presented in [5.3.2.1.a].

{Divide and conquer – recursive solution – Pascal}

PROCEDURE Solve(x);
BEGIN
  IF *x is decomposable in sub-problems THEN
  BEGIN

When dividing a problem into more parts:
\[ x_1, x_2, \ldots, x_k; \]
\[ \text{Solve}(x_1); \text{Solve}(x_2), \ldots; \text{Solve}(x_k); \]
*combine the \( k \) partial solutions in a solution for \( x \)

\[ \text{END} \{ \text{IF} \} \]

\[ \text{ELSE} \]

* solve \( x \) directly

\[ \text{END}; \{ \text{Solve} \} \]

---

```
/* Divide and conquer – recursive solution – C */

void solve(x);
{
if (*x is decomposable in sub-problems)
{
*divide x in more parts: \( x_1, x_2, \ldots, x_k; \)
solve (x1);
solve (x2);
... /*[5.3.2.1.a]*)/
solve (xk);
*combine the \( k \) partial solutions in a solution for \( x \)
}/*if*/
else
*solve x directly;
}; /*solve*/
```

---

- If the recombining of the partial solutions is substantial simpler than solving the problem in its integrity, this technique leads to designing algorithms which are really efficient.

- Because the model contains \( k \) recursive calls, the associated calls tree of procedure \text{Solve} is of order \( k \).

### 5.3.2.2 Dividing Algorithms Analysis

- We presume that execution time for solving a problem of dimension \( n \) is \( T(n) \).

- By successive dividing, the problem reduces its dimension, so we can consider that for \( n \leq c \) (\( c \) is a constant), determination of solution requires an execution time which is a constant \( \Theta(1) \).

- We denote by \( D(n) \) the time necessary to divide the problem, and by \( C(n) \) the time necessary to combine partial solutions.

- If the problem is divided in \( k \) sub-problems, each having the dimension \( 1/b \) from the dimension of the initial problem, we obtain the following recursive formula for \( T(n) \) [5.3.2.2.a]: 
\[ T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ k \cdot T(n/b) + D(n) + C(n) & \text{for } n > c \end{cases} \]  

5.3.2.3 Example of Dividing Algorithm

- A first example of dividing algorithm is Quicksort (&.3.2.6).
- In this case, the initial problem is divided each time in two sub-problems, resulting a binary calls tree.
- The combination of the two partial solutions is not necessary because, the objective is reached by the modifications produced by determining the partial solutions.

5.3.2.4 Algorithm for Determining the Extreme Values of a Linear Array. Iterative and Recursive Solution.

- The algorithm for determining the extreme values of a linear array is conceived as a procedure Domain\((a, left, right, small, big)\) which assigns to parameters small and big respectively with the values of the minimum and maximum element of the array \(a\), in the domain delimited by the indices left and right, respectively \((a[left]..a[right])\).
- An evident iterative implementation of the algorithm is presented in [5.3.2.4.a] (DomainIt).
void domain_it(type_array const a, int left, int right, int* small, int* big)
{
    int k;
    *small=a[left]; *big=a[left];
    for( k=left+1; k<=right; k++)        /*[5.3.2.4.a]*/
    {
        if (a[k]>*big) *big=a[k];
        if (a[k]<*small) *small=a[k];
    }
} /*domain_it*/
/**----------------------------------------*/

- The procedure scans the array comparing each element with the smaller and the biggest current element.

- It’s simple to see that the cost of procedure DomainIt in terms of number of element comparisons is $2n-2$ for an array with $n$ elements.

- To be noticed, that each element of $a$ is compared twice, once for minimum and then for maximum.

- The algorithm doesn’t account that an element which is candidate for minimum, can never be a candidate for the maximum and vice versa, excepting the initialization condition.

- Thus, a considerable effort is spent examining twice each element. This waste can be avoid.

- A recursive solution based on dividing technique is the following [AH85]:

    - The array is divided in two parts.

    - The minimum and maximum values of each of the parts are determined by recursive calls of the procedure.

    - The obtained extreme values are compared and the absolute minimum and maximum values are determined.

    - Further, it proceeds in the same manner, reducing each time the dimension of the processed domains at half.

    - For dimension 1 or 2 of the arrays, the solution is trivial.

- An example of this approach is presented in the form of recursive procedure Domain [5.3.2.4.b].

{Algorithm for determining the extreme values of a vector – recursive solution – Pascal variant}

CONST n = ...;
TYPE TypeArray = ARRAY[1..n] OF integer;
VAR a: TypeArray;
PROCEDURE Domain(a: TypeArray; left, right: integer;
    VAR small, big: integer);

VAR middle, small1, big1, small2, big2: integer;
BEGIN
  IF right <= small + 1 THEN (*array of dimension 1 or 2*)
  BEGIN
    IF a[left] < a[right] THEN
      BEGIN
        small := a[left]; big := a[right]
      END
    ELSE
      BEGIN
        small := a[right]; big := a[left]
      END
  END (*IF*)
  ELSE
  BEGIN
    middle := (left + right) DIV 2;
    Domain(a, left, middle, small1, big1);
    Domain(a, middle + 1, right, small2, big2);
    IF big1 > big2 THEN big := big1 ELSE big := big2;
    IF small1 < small2 THEN small := small1 ELSE small := small2
  END (*ELSE*)
END; (*Domain*)

/* Algorithm for determining the extreme values of a vector –
recursive solution – C variant */

#define n 100
typedef int type_array[n];
type_array a;

void domain(type_array const a, int left, int right, int* small, int* big)
{
  int middle, small1, big1, small2, big2;
  if (right <= left + 1) /*array of dimension 1 or 2*/
  {
    if (a[left - 1] < a[right - 1])
      {
        *small = a[left - 1]; *big = a[right - 1];
      }
    else
      {
        /*[5.3.2.4.b]*/
        *small = a[right - 1]; *big = a[left - 1];
      }
  } /*if*/
  else
  {
    middle = (left + right) / 2;
    domain(a, left, middle, &small1, &big1);
    domain(a, middle + 1, right, &small2, &big2);
    if (big1 > big2)
*big=big1;
else
    *big=big2;
if (small1<small2)
    *small=small1;
else
    *small=small2;
} /*else*/
} /*domain*/
/*----------------------------------------------------------------*/

- We can notice the **perfect analogy** with the generic structure presented in [5.3.2.1.a].

- In figure 5.3.2.4.a. appears a **graphical representation** of the working manner of procedure **Domain**.

![Graphical representation of the working manner of procedure Domain.](image)

**Fig.5.3.2.4.a.** Graphical representation of the working manner of procedure **Domain**.

- The **execution trace** of procedure **Domain** for \( n = 10 \) appears in figure 5.3.2.4.b.
In figure 5.3.2.4.b. appears the limits of the current domain, for each of the procedure calls.

- In the border are specified the domains of length 1 or 2, which presume direct comparisons.

- The calls tree is a binary tree, because the algorithm presumes two recursive self calls.

- If we analyze more deeply the implementation manner we realize that:
  
  - The call contexts are salved in the system stack in the order resulted from preorder traversal of the calls tree.

  - The call contexts are extracted from the stack as resulted from postorder traversal of the calls tree.

  - The context restoration for each call, make possible the ordered scanning of all the possibilities relieved by the calls tree.

  - Overall, the overhead of the recursive algorithm is higher than the iterative one. There are necessary 11 calls for n = 10.

- Though, the authors demonstrate that, from the number of element comparisons point of view, the algorithm is more efficient than the iterative one.

- Noting that:
  
  - (1) The procedure includes direct comparisons only for domains of length 1 or 2.

  - (2) For the longer domains, it proceeds to dividing them, comparing only their extremes.

  - In [AH85] it indicates an approximate value for comparison number equal to \((3/2)n-2\) where n is the dimension of the processed array.

- The quantitative analysis of algorithm Domain can be done by customizing the relation [5.3.2.2.a].

  - We notice immediately that \(k=2, \ b=2, \) and \(D(n)\) and \(C(n)\) are both \(\Theta(1)\).

- As result, the resulting equation is [5.3.2.4.c] whose solution is [5.3.2.4.d].

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 2 \\
2 \cdot T(n/2) + 2 & \text{if } n > 2 
\end{cases}
\]
5.3.3 Reduction Algorithms

• Another category of algorithms which are suitable for a recursive approach are the reduction algorithms.
  
  • This algorithms are based on recursive manner reduction of the degree of difficulty of the problem to be solved, step by step, until the problem became banal.
  
  • Further, it returns in the same recursive manner, assembling the integral solution of the problem.

• To this category belongs the algorithm for computing the factorial and the algorithm for solving the towers of Hanoi problem.

• We have to notice that, unlike the backtracking algorithms (&5.4), in this case there is no question to return in the case of a failure, that is the reason for this algorithms are classified in a different category.

• Further we present an example of reduction algorithm.

5.3.3.1 Algorithm for Determining the Permutations of $n$ Given Numbers

• For a given sequence of $n$ numbers, it requires to design an algorithm for determining all the permutation of the $n$ numbers.

• For this purpose the reduction technique can be used.

• The main idea is:
  
  • To obtain the permutation of $n$ elements it’s sufficient to fix in turn an element and to permute all the remaining $n-1$ elements.
  
  • Repeating the above action in a recursive manner we obtain permutations of 1 element which are trivial.

• This technique is implemented by the procedure $\text{Permut}(k)$.

• The pseudocode format of this algorithm appears in [5.3.3.1.a.].
Pseudocode format of the algorithm for determining the permutations of n given numbers

Procedura Permut(k:integer);
  if k=1 then
    *display the array {a permutation is finalized}
  else
    for i=1 to k execute
      *interchange a[i] with a[k];
      Permut(k-1); {recursive call}
    *interchange a[i] with a[k] {state restoring}
BEGIN

The given numbers are stored in array a[i] (i=1,2,...,n).

If \( k \neq 1 \), then each of the array elements, situated on positions which are smaller than \( k \), are fixed on \( k \)-th position, by interchanging position \( i \) and \( k \) of the array, in a repetitive loop for \( i=1,2,...,k \).

Initial \( k \) takes value \( n \) for permuting \( n \) numbers.

For each change, the routine is called recursively with parameter \( k-1 \).

- After returning, the initial state is restored, resuming in sense inverse the interchange of the positions \( i \) and \( k \).

- This is necessary because the fixing of the next element, requires the restoration of the initial state for parsing in an ordered manner, all the possibilities for each element.

- When \( k=1 \), a call is considered finalized, and the array containing a permutation is displayed.

- This is the condition which limits the depth of recursive calls, determining the return from the anterior calls.

In [5.3.3.1.b.] are presented two implementation of the algorithm for determining the permutations of \( n \) numbers; one in Pascal and another in C.

Algorithm for determining the permutations of n given numbers – Pascal variant

PROCEDURE Permut(k: integer);
  VAR i,x: integer;
  BEGIN
    IF k=1 THEN *display a ELSE
    BEGIN
      FOR i:=1 TO k DO
      BEGIN
        *interchange a[i] with a[k] {state restoring}
      END
    END
BEGIN
```c
void permut(int k)
{
    int i, x;
    if (k == 1)
        *display a;
    else
    {
        for (i = 1; i <= k; i++) /* [5.3.3.b] */
        {
            x = a[i]; a[i] = a[k]; a[k] = x;
            permut(k - 1);
            x = a[i]; a[i] = a[k]; a[k] = x;
        }
    }
} /*permut*/
/* *-------------------------------------------------------------------*/
```

In figure 5.3.3.1.a is represented the partially calls tree for procedure `Permut(4)`, for permutation obtained by fixing the first element.
For the rest of the elements, from missing space reasons, are suggested only the corresponding sub-trees.

The structure of the calls tree is more complicated because the recursive calls are realized from a FOR loop with a variable limit $k$, which decrease once with increasing of the tree level.

The height of the calls tree is equal to $n$.

5.3.4 Recursive Algorithms for Determining All the Solutions of Some Problems

• The recursive algorithms has the propriety to reveal in an ordered manner all the possibilities related to a given situation.

We present here two examples of such recursive algorithms:

• First example reveals in an ordered manner all the possibilities related to a given situation.

• The second example, selects from de set of all possibilities, those which present interest.

5.3.4.1 Algorithm for Determining All the Possibilities for Cutting a Thread with a Given Length

• The following algorithm determines all the cutting possibilities of a given length thread $(n)$ in parts of length 1 or 2.

• The algorithm is based on the following working technique:

  • For a length $n>1$ there are two possibilities:

    • It cuts a part of length 1 and the rest $(n-1)$ is cut in all possible manners.

    • It cuts a part of length 2 and the rest $(n-2)$ is cut in all possible manners.

  • For length $n=1$ we have the simple case of a cut of length 1.

  • For length $n=0$ there is no possibility to cut.

• The pseudocode variant of this algorithm appears in [5.3.4.1.a].
{Pseudocode variant of the algorithm for cutting a thread}

Procedure Cut(lengthThread);
    if lengthThread>1 then
        *cut a part of length 1;
        Cut(lengthThread-1);
        *cut a part of length 2;
        Cut(lengthThread-2);
        *the cut is cancelled
    else
        if lengthThread=1 then
            *cut a part of length 1;
            *display

In [5.3.4.1.b] are presented the implementations of this algorithm in Pascal and in C.

In both implementations:

- Procedure Cut implements the above presented technique. When the length become $n=1$ or $n=0$, the procedure is considered finished and the sequence of generated cuts is displayed.

- The cuts are presented in graphical manner using the character '.' for the segment of length 1 and the character '_' for the segment of length 2.

- For storing the cuts, is used an array of characters $z$, which is managed with index $k$.

{Pascal variant of the algorithm for cutting a thread}

VAR i,k,x: integer;
    z: ARRAY[1..9] OF char;

PROCEDURE Cut(lengthThread: integer);
BEGIN
    IF lengthThread>1 THEN
        BEGIN
            k:= k+1;
            z[k]:= ".";  {cut a part of length 1}
            Cut(lengthThread-1);
            z[k]:= ";";  {cut a part of length 1}
            Cut(lengthThread-2);
            k:= k-1  {cancel the cut}
        END
    ELSE
        BEGIN
            Write('     ');
            FOR i:= 1 TO k DO Write(z[i]);
            IF lengthThread=1 THEN Write('.');
```c
int i, k, x;
char z[9];

void cut(int length_thread)
{
    if (length_thread > 1)
    {
        k = k + 1;
        z[k-1] = '.';  /* cut a part of length 1 */
        cut(length_thread - 1);
        z[k-1] = '_';  /* cut a part of length 2 */
        cut(length_thread - 2);
        k = k - 1;    /* canceling the cut */
    }  /* if */
    else
    {                                /* [5.3.4.1.b] */
        printf("     ");
        for (i = 1; i <= k; i++)
            printf("%c", z[i-1]);
        if (length_thread == 1)
            printf(".");
        printf("\n");
    }  /* else */
}  /* cut */
```

- In figure 5.3.4.1.a appears the results of the call of procedure `Cut` for `n=4` respectively `n=5`.
  - From this figure is easy to deduce the execution manner of the algorithm, knowing that in graphical representation:
    - Character ' . ' represents the call of procedure `Cut` for `n-1` (a cut of length 1).
    - Character ' _ ' represents the call of procedure `Cut` for `n-2` (a cut of length 2).

```
lengthThread = 4                      lengthThread = 5

    ....                          ....
   .   _                          .   _
   .   .                          .   .
  .     _                          .     _
    _     _                        _     _
```
Fig.5.3.4.1.a. Execution of procedure **Cut** for **n=4** and **n=5**.

- In figure 5.3.4.1.b is presented the calls tree of procedure **Cut** for **n=4**.

![Calls tree of procedure Cut](image)

**Fig.5.3.4.1.b. Calls tree of procedure Cut**

- In each node of the calls tree are specified:
  - The sequence of cut parts.
  - The recursive call.
  - The current value of \( k \).
  - In bolded border are represented the sequences that are displayed.

- This model can be easy extended to determine the cutting possibilities for any specified number of cuts, an for any structure of dimensions.
5.3.4.2 Algorithm for Determining All Exit Paths from a Maze

- The next algorithm presumes a **maze**.
  - The maze is modeled by a **two dimensions array** of characters \( m_{(n+1),(n+1)} \).
  - The **walls** of the maze are represented by character '★'.
  - The **lanes** of the maze are represented by blanc character ' '.
  - The **starting point** is the **center of the maze**.

- The **exit path** is searched by a **recursive procedure** `Search(x, y)` where \( x \) and \( y \) are the **coordinates** of the current place in the maze and in the same time, indices in the array which models the maze.

- **The search** description is the following:
  - (1) If the value of the current place is ' ':  
    - It enters a potential exit path.
    - The current place is marked with character ' # '.
    - If the a border is reached, that means an **exit path** was found, and the array \( m \) is displayed (the maze and the exit path).
  - (2) If the value of the current place is not ' ':  
    - The procedure `Search` is called for the four points situated in the immediate neighborhood of the current place, after the direction of the coordinate axes (right, up, left, down).

- As we can remark, for each successful step, the path is marked with character ' # '.

- This mark assures from one side the **marking** of the exit path, and by the other side, avoids **retrace** a selected path.
  - Retracing the same path my produce an **infinite** cycle.

- It’s very important to observe, that the marker is deleted as soon as an **impasse** is detected, or if an exit path was found.
  - In both situations it retraces the actual path until the proximal position which permits the selection of a new path possibility.

- **To delete a marker**, a character ' ' is generated on its position before to return from procedure.
• This corresponds to the “rolling up the thread” method suggested by Ariadne the daughter of King Minos of Crete and used by Theseus who killed the Minotaur in the labyrinth build by Daedalus for Minos in Crete.

• In [5.3.4.2.a] is presented the pseudocode format of procedure \texttt{Search} and in [5.3.4.2.b] its implementation in \texttt{Pascal} respectively \texttt{C}.

---

\textbf{Pseudocode format of the algorithm for searching all exit paths from a maze}

\begin{verbatim}
Procedure Search(x,y: place_coordinates);
  if *the place is free then \[5.3.4.2.a\]
    *mark the place
  if *the border was detected then *display the path
  else
    Search(x+1,y); \{right\} Search(x,y+1); \{up\}
    Search(x-1,y); \{left\} Search(x,y-1) \{down\}
  *delete the marker
\end{verbatim}

---

\textbf{Pascal variant of the algorithm for searching all exit paths from a maze}

\begin{verbatim}
PROCEDURE Search(x,y: place_coordinates);
BEGIN
  IF m[x,y]=' ' THEN \[5.3.4.2.b\]
    BEGIN
      m[x,y]:= '#';
      IF ((x MOD n)=0)OR((y MOD n)=0) THEN *display the maze
      ELSE
        BEGIN
          Search(x+1,y); Search(x,y+1); Search(x-1,y);
          Search(x,y-1)
        END;
      m[x,y]:= ' '
    END
END;{Search}
\end{verbatim}

---

\textbf{C variant of the algorithm for searching all exit paths from a maze \*/}

\begin{verbatim}
void search(int x, int y)
{
  if (m[x][y]==' ')
  {\[5.3.4.2.b\]*/}
    m[x][y]='#';
    if (((x % n)==0)||((y % n)==0)) ;
    *display the maze
  else
    {
      search(x+1,y);
    }
\end{verbatim}
search(x, y+1);  
search(x-1, y);  
search(x, y-1);  
}  
m[x][y]=' ';  
}  
*/search*/  
/*---------------------------------------------------------------*/

- If during the search process, one of the borders is reached – the values 0 or n on x or y axis, – an exit path was found and it is displayed.

- In figure 5.3.4.2.a is presented an execution example of this algorithm.

Fig.5.3.4.2.a. Execution example for procedure Search

5.3.5 Algorithms for Drawing Some Recursive Curves

- **Recursive curves** represents another application domain of **recursive algorithms**.

  - Such curves can be elegantly generated using a computing system, because they presume a reduced number of graphical modules, which are assembled successively, in a recursive manner, based on a set of simple and well defined **rules**.

- The purpose of this section is to present two recursive algorithms for generating such curves.

- Apparently, the study of recursive curves, doesn’t have an immediate applicability, but it contributes in a decisive manner to fundament, to consolidate and to understand the concept of recursion.

- We have to mention that these curves can be also included in the category of **fractals**.

5.3.5.1 Recursive Algorithm for Generating the Hilbert Curves

- In figure 5.3.5.1.a appears four curves notated $H_0, H_1, H_2$ and $H_3$ which represents the Hilbert curves of order 0, 1, 2 and 3.
• We can notice that:
  
  • A Hilbert curve $H_{i+1}$ is obtained by assembling four instances of Hilbert curves $H_i$.
  
  • The 4 instances has each half of the initial dimension.
  
  • Each instance is rotated in a specific manner, respecting a pattern.
  
  • The instances are connected by three linking lines which are represented bolded in the figure.

\[
\begin{array}{cccc}
(A_0) & (A_1) & (A_2) & (A_3) \\
\circ & \square & \square & \square \\
H_0 & H_1 & H_2 & H_3
\end{array}
\]

\textbf{Fig.5.3.5.1.a.} Hilbert curves of order 0, 1, 2 and 3

• To be notice that $H_1$ can be considerate as consisting of 4 instances of the empty curve $H_0$, connected by the same linking lines.

• This are the Hilbert curves dating from 1891.

• Presuming we dispose a graphical library which supplies the next graphical operators:
  
  • $\text{MoveTo}(x, y)$ which sets the graphical cursor in the points having the coordinates $x$ and $y$;
  
  • $\text{LineTo}(x, y)$ which draws a straight line between the current position of the graphical cursor and the position indicated by the coordinates $x$ and $y$.

• We intend to conceive a simple program for generating and displaying the curves of Hilbert of different orders superimposed.
Because each Hilbert curve $H_i$ consists of 4 halved copies of $H_{i-1}$, it’s natural that the procedure which draws $H_i$, to consist of 4 parts, each drawing a curve $H_{i-1}$ rotated properly and at required dimension.

By generalization, are deduced the four recursive patterns which are represented in figure 5.3.5.1.b.

The four parts are named A, B, C and D, and the connecting lines are represented by arrows indicating the direction of drawing.

Fig.5.3.5.1.b. Recursive patterns for Hilbert curves.

Starting from these recursive patterns, the recursive schemes for generating Hilbert curves are presented in [5.3.5.1.a].

---
A:D ← A ↓ A → B
B:C ↑ B ← B ↓ A
C:B → C ↑ C → D
D:A ↓ D ← D ↑ C
[5.3.5.1.a]
---

If the length of line unit is denoted by variable $h$, the procedure reflecting the pattern A can be expressed using recursive calls to its self, as well as to procedures implementing the patterns D and B conceived in the same manner [5.3.5.1.b].

{Procedure for recursive drawing of pattern A of a Hilbert curve – Pascal variant}

PROCEDURE A(i: integer);
BEGIN
IF \( i > 0 \) THEN
BEGIN
  D(i-1); x:= x-h; LineTo(x,y);
  A(i-1); y:= y-h; LineTo(x,y); \[5.3.5.1.b\]
  A(i-1); x:= x+h; LineTo(x,y);
  B(i-1)
END
END; \{A\}

- The procedure initialized in the main program for each Hilbert curve which will be superimposed.
  - On the same draw are superimposed Hilbert curves of different successive orders.
  - The result is likeness with fractal representation.
- The main program determines the initial starting point of each curve and the specific length \( h \).
- Variable \( h_0 \) which represents the maximum value of the connecting lines length, must be a power of two \( 2^k \) where \( k > n \), \( n \) being the maximum order of curves which are superimposed.
- The program for drawing the Hilbert curves \( H_1, H_2, \ldots, H_n \) appears in [5.3.5.1.c] in Pascal respectively C variant.

{Program for recursive drawing of some Hilbert curves of different orders –Pascal variant}

PROGRAM Hilbert;
  \{draws hilbert curves of order 1,2,...,n\}
CONST n=4; h0=512;
VAR i,h,x,y,x0,y0: integer;
PROCEDURE B(i: integer); FORWARD;
PROCEDURE C(i: integer); FORWARD;
PROCEDURE D(i: integer); FORWARD;
PROCEDURE A(i: integer); \{draws pattern A\}
BEGIN
  IF \( i > 0 \) THEN
    D(i-1); x:= x-h; LineTo(x,y);
    A(i-1); y:= y-h; LineTo(x,y);
    A(i-1); x:= x+h; LineTo(x,y);
    B(i-1)
  END
END; \{A\} \[5.3.5.1.c\]

PROCEDURE B(i: integer); \{draws pattern B\}
BEGIN
  IF \( i > 0 \) THEN
    BEGIN
      C(i-1); y:= y+h; LineTo(x,y);
      B(i-1); x:= x+h; LineTo(x,y);
      B(i-1); y:= y-h; LineTo(x,y);
    END
END; \{B\}
PROCEDURE C(i: integer); {draws pattern C}
BEGIN
  IF i>0 THEN
  BEGIN
    B(i-1); x:= x+h; LineTo(x,y);
    C(i-1); y:= y+h; LineTo(x,y);
    C(i-1); x:= x-h; LineTo(x,y);
    D(i-1)
  END;
END; {C}

PROCEDURE D(i: integer); {draws pattern D}
BEGIN
  IF i>0 THEN
  BEGIN
    A(i-1); y:= y-h; LineTo(x,y);
    D(i-1); x:= x-h; LineTo(x,y);
    D(i-1); y:= y+h; LineTo(x,y);
    C(i-1)
  END;
END; {D}

BEGIN {main program}
i:= 0; h:= h0; x0:= h DIV 2; y0:= x0;
REPEAT {draws hilbert curve of order i}
i:= i+1; h:= h DIV 2;
x0:= x0+(h DIV 2); y0:= y0+(h DIV 2);
x:= x0; y:= y0; MoveTo(x,y);
A(i)
UNTIL i=n;
END.

/* Program for recursive drawing of some Hilbert curves of different orders – C variant */
*/

#include <graphics.h>

#define n 4
#define h0 512

int i,h,x,y,x0,y01;

void b(int);
void c(int);
void d(int);
void graphic_init(void);
void graphic_close(void);

void a(int i) /*draws pattern A*/
{ 
    if (i>0) 
    { 
        d(i-1); x=x-h; lineto(x,y); 
        a(i-1); y=y-h; lineto(x,y); 
        a(i-1); x=x+h; lineto(x,y); 
        b(i-1); 
    } 
} /*a*/                              /*[5.3.5.1.c]*/

void b(int i) /*draws pattern B*/
{ 
    if (i>0) 
    { 
        c(i-1); y=y+h; lineto(x,y); 
        b(i-1); x=x+h; lineto(x,y); 
        b(i-1); y=y-h; lineto(x,y); 
        a(i-1); 
    } 
} /*b*/

void c(int i) /* draws pattern C*/
{ 
    if (i>0) 
    { 
        b(i-1); x=x+h; lineto(x,y); 
        c(i-1); y=y+h; lineto(x,y); 
        c(i-1); x=x-h; lineto(x,y); 
        d(i-1); 
    } 
} /*c*/

void d(int i) /* draws pattern D*/
{ 
    if (i>0) 
    { 
        a(i-1); y=y-h; lineto(x,y); 
        d(i-1); x=x-h; lineto(x,y); 
        d(i-1); y=y+h; lineto(x,y); 
        c(i-1); 
    } 
} /*d*/

int main(int argc, const char* argv[]) 
{     /*main program*/
    graphic_init();
    i=0;
    h=h0;
    x0=h/2;
    y01=x0;
    do {/*draws hilbert curve of order i*/
        i++;
        h=h/2;
        x0=x0+(h/2);
y01=y01+(h/2);
x=x0; y=y01;
moveto(x,y);
a(i);
} while (!(i==n));
getchar();
graphic_close();
return 0;
}

/*OS specific code follows*/
/*BorlandC 3.1 exemple*/
void graphic_init()
{ /* request auto detection */
  int gdriver =DETECT, gmode, errorcode;
  /* initialize graphics mode */
  initgraph(&gdriver, &gmode, "");
  /* read result of initialization */
  errorcode =graphresult();
  if (errorcode !=grOk) /* an error occurred */
  {
    printf("Graphics error: %s
",
            grapherrormsg(errorcode));
    exit(1);
  }
}

void graphic_close()
{
  closegraph();
}

/*---------------------------------------------------------------*/
5.3.5.2 Recursive Algorithm for Generating Sierpinski Curves

- A more complicated example is the Sierpinski curves which are represented in figure 5.3.5.2.a \((S_1 \text{ and } S_2)\).

![Recursive patterns for Sierpinski curves of order 1 and 2.](image)

- The difference between the two categories of curves is the fact that the Sierpinsky curves are \textit{closed} curves, while the Hilbert curves are \textit{open} one.

- As a consequence, in the case of Sierpinsky curves:
  - The \textbf{basic recursive pattern} is an \textit{open curve}.
  - The \textbf{four parts} are connected by four \textbf{closing lines} which doesn’t belong to the recursive pattern.
  - The \textbf{closing lines} consist in four biased lines situated in the corners of the figure, and are represented bolded (fig.5.3.5.2.a).
  - From this representation, we can deduce the \textbf{assembling mode} of the four inferior order \textbf{patterns}, to obtain a curve of higher order (fig.5.3.5.2.a).
• Now, we can specify the four \textbf{generic recursive patterns} of the \textit{Sierpinski} curves.

• They are named $A$, $B$, $C$ and $D$, and internal connecting lines are explicitly represented as arrows (fig.5.3.5.2.b).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{sierpinski_patterns.png}
\caption{Basic model and recursive patterns of Sierpinski curves.}
\end{figure}

• The four \textbf{generic recursive patterns} are \textbf{identical} excepting the fact, that they are rotated with 90° each.

• As we can notice, the \textbf{generic patterns} are \textbf{open curves}.

• Its important to underline that the four patterns are assembled using a \textbf{basic model} consisting in four \textbf{closing lines}, represented above in the same figure 5.3.5.2.b.

• From these \textbf{recursive patterns}, we can be simple deduce the \textbf{recursive schemes} which allow to generate the curves.

• In [5.3.5.2.a] is represented the \textbf{scheme of the basic model}, and in [5.3.5.2.b] the \textbf{recursive schemes} of the component curves, where an double arrow specifies a line of double length.

\begin{equation}
\begin{array}{l}
S: A \rightarrow B \leftarrow C \rightarrow D \\
A: A \rightarrow B \Rightarrow D \rightarrow A \\
B: B \leftarrow C \downarrow A \rightarrow B \\
C: C \leftarrow D \leftarrow B \rightarrow C \\
D: D \rightarrow A \uparrow C \leftarrow D \\
\end{array}
\end{equation}
• Considering the same representation facilities as in the previous case, we can simply develop the routines which draws Sierpinsky curves.

• Thus, the procedure A can be build starting from the first line of the recursive scheme [5.3.5.2.b]. The same for la procedures B, C and D.

• In [5.3.5.2.c] we present an example for implementation of procedure A.

---

{Example of implementation of pattern A of Sierpinski curves – Pascal variant}

PROCEDURE A(i: integer);
BEGIN
  IF i>0 THEN [5.3.5.2.c]
    BEGIN
      A(i-1); x:= x+h; y:= y-h; LineTo(x,y);
      B(i-1); x:= x+2*h; LineTo(x,y);
      D(i-1); x:= x+h; y:= y+h; LineTo(x,y);
      A(i-1)
    END
  END;{A}
---

• The main program is conceived in accordance with the basic model [5.3.5.2.a].

• The task of the main program is to initiate generation of the successive superimposed Sierpinsky curves of increasing orders, establishing for each order, the value of \( h \) and the coordinates of the starting point \( x_0 \) and \( y_0 \) in accordance with initial dimension of the displaying area.

• The integral program appears in [5.3.5.2.d].

  • The elegance and the opportunity of using recursion in this situation is obviously.

  • The correctness of the program can be simple deduced from the patterns structure and configuration.

  • The recursion depth is controlled by means of parameter \( i \), assuring in this manner the program termination.

  • The iterative variant of the same program is more complicated and bearish.

---

{Program for drawing Sierpinski curves of different orders –Pascal variant}

PROGRAM Sierpinski;
  {draws Sierpinski curves of order 1,2,....,n}

  CONST n=4; h0=512;

  VAR i,h,x,y,xo,yo: integer;
PROCEDURE B(i: integer); FORWARD; [5.3.5.2.d]
PROCEDURE C(i: integer); FORWARD;
PROCEDURE D(i: integer); FORWARD;

PROCEDURE A(i: integer); {draws pattern A}
BEGIN
  IF i>0 THEN
  BEGIN
    A(i-1); x:= x+h; y:= y-h; LineTo(x,y);
    B(i-1); x:= x+2*h; LineTo(x,y);
    D(i-1); x:= x+h; y:= y+h; LineTo(x,y);
    A (i-1)
  END
END;{A}

PROCEDURE B(i: integer); {draws pattern B}
BEGIN
  IF i>0 THEN
  BEGIN
    B(i-1); x:= x-h; y:= y-h; LineTo(x,y);
    C(i-1); y:= y-2*h; LineTo(x,y);
    A(i-1); x:= x+h; y:= y-h; LineTo(x,y);
    B (i-1)
  END
END;{B}

PROCEDURE C(i: integer); {draws pattern C} [5.3.5.2.d]
BEGIN
  {. . .}
END;{C}

PROCEDURE D(i: integer); {draws pattern D}
BEGIN
  {. . .}
END;{D}

BEGIN {main program}
i:= 0; h:= ho DIV 4; xo:= 2*h; yo:= 3*h;
REPEAT
  i:= i+1; xo:= xo-h;
  h:= h DIV 2; yo:= yo+h;
  x:= xo; y:= yo; MoveTo(x,y);
  A(i); x:= x+h; y:= y-h; LineTo(x,y);{draws linking lines}
  B(i); x:= x-h; y:= y-h; LineTo(x,y);
  C(i); x:= x-h; y:= y+h; LineTo(x,y);
  D(i); x:= x+h; y:= y+h; LineTo(x,y)
UNTIL i=n;
END.

-------------------------------------------------------------------
/* Program for drawing Sierpinski curves of different orders – C variant * /

#include <graphics.h>
/* draws Sierpinski curves of order 1,2,...,n */

enum {n =4, h0 =512};
int i,h,x,y,xo,yo;

void b(int i);
void c(int i);
void d(int i);
void graphic_init(void);
void graphic_close(void);

void a(int i) /*draws patern A*/ {
    if (i>0){
        a(i-1); x=x+h; y=y-h; lineto(x,y);
        b(i-1); x=x+2*h; lineto(x,y);
        d(i-1); x=x+h; y=y+h; lineto(x,y);
        a(i-1);
    }
}  /*a*/

void b(int i) /*draws patern B*/ {
    if (i>0){
        b(i-1); x=x-h; y=y-h; lineto(x,y);
        c(i-1); y=y-2*h; lineto(x,y);
        a(i-1); x=x+h; y=y-h; lineto(x,y);
        b(i-1);
    }
}  /*b*/

void c(int i) /*draws patern C*/ /*[5.3.5.2.d]*/ {
    /* . . .*/
}  /*c*/

void d(int i) /*draws patern D*/ {
    /* . . .*/
}  /*d*/

int main(int argc, const char* argv[]) {
    /*main program*/
    graphic_init();
    i=0;
    h=h0/4;
    xo=2*h;
    yo=3*h;
    do {
        i++; xo=xo-h;
        h=h/2; yo=yo+h;
        x=xo; y=yo; moveto(x,y);
        a(i); x=x+h; y=y-h; lineto(x,y); /*draws linking lines*/
        b(i); x=x-h; y=y-h; lineto(x,y);
    }
\[c(i); x=x-h; y=y+h; \text{lineto}(x,y);\]
\[d(i); x=x+h; y=y+h; \text{lineto}(x,y);\]
} \textbf{while} (! (i==n));
\textbf{return} 0;
\}

/*OS specific code follows*/
/*BorlandC 3.1 exemple*/
\textbf{void} graphic\_init()
{ /* request auto detection */
  \textbf{int} gdriver =DETECT, gmode, errorcode;
  /* initialize graphics mode */
  \textbf{initgraph}(\&gdriver, \&gmode, "");
  /* read result of initialization */
  errorcode =\textbf{graphresult}();
  \textbf{if} (errorcode !=grOk) /* an error occurred */
  \{
    \textbf{printf}("Graphics error: %s\n",
        grapherrormsg(errorcode));
    \textbf{exit}(1);
  \}
}
\textbf{void} graphic\_close()
{ \textbf{closegraph}();
}

/*-----------------------------------------------*/

- In figure 5.3.5.2.c appears a graphical example of execution of \textbf{Sierpinsky} program for \(n=4\).
5.4 Backtracking Algorithms

- A particularly intriguing programming endeavor is the subject of so-called general problem solving.
  - The idea is to determine algorithms for finding solutions to specific problems not by following a fixed rule of computation, but by trial and error.

- The common pattern is to decompose the trial-and-error process onto partial tasks.
  - Often these tasks are most naturally expressed in recursive terms and consist of the exploration of a finite number of subtasks.
  - We may generally view the entire process as a trial or search process that gradually builds up and scans (prunes) a tree of subtasks.
  - If the process determines partial or final solutions which doesn’t satisfy the requirements, the search returns in a recursive manner and the process is resumed from the proximal potential point, until the final solution is obtained.
  - That is the reason for this kind of algorithms are named backtracking algorithms.

- In many problems this search tree grows very rapidly, often exponentially, depending on a given parameter. The search effort increases accordingly.
• Frequently, the search tree can be pruned by the use of heuristics only, thereby reducing computation to tolerable bounds.

• It is not our aim to discuss general heuristic rules in this text.

• Rather, we will discuss:
  
  • (1) The general principle of breaking up such problem-solving tasks into subtasks.
  
  • (2) The use of recursion in this context.

• We start out by demonstrating the underlying technique by using an example, namely, the well known knight's tour.

### 5.4.1 Knight’s Tour

• The knight’s tour problem specification is:

  • Is given a \( n \times n \) chess board with \( n^2 \) fields.

  • A knight being allowed to move according to the rules of chess, is placed on the field with initial coordinates \( x_0, y_0 \).

  • The problem is to find a covering of the entire board, if there exists one, i.e. to compute a tour of \( n^2-1 \) moves such that every field of the board is visited exactly once.

• The obvious way to reduce the problem of covering \( n^2 \) fields is to consider the next problem:

  • At a given moment it either tries to perform the next move or finds out that none is possible. In the last situation it returns to the previous step.

• Let us therefore define an algorithm trying to perform a next move. A first approach is the following [5.4.1.a]:

```plaintext
{Pseudocode form of next knight’s move algorithm – Refinement step 0 – Pascal variant}

PROCEDURE TryNextMove;
BEGIN
  *initialize the list of moves
  REPEAT
    *select the next candidate from list of moves
    IF *acceptable THEN
      BEGIN
        [5.4.1.a]
        *record current move;
        IF *board not full THEN
          BEGIN
            TryNextMove;
            IF NOT *successful move THEN
```

```
// Pseudocode form of next knight's move algorithm – Refinement step 0 – C variant *

void try_next_move()
{
    /*initialize the list of moves*/
    do {
        /*select next candidate from list of moves*/
        if (*acceptable)
        {
            /*record current move;*/
            if (*board not full)
            {
                try_next_move();
                if (!*successful move)
                    /*[5.4.1.a]*/
                    *erase previous move;
            }
            else;
            *successful move
        }
    } while (!((*successful move*) || (*no more candidates in the list of next moves)));
} /*try_next_move*/

• If we wish to be more precise in describing this algorithm, we are forced to make some decisions on data representation.

• An obvious step is to represent the board by a matrix, say t. Let us also introduce a type to denote index values TypeIndex [5.4.1.b].

{Defining data structures: the chess board}

TYPE TypeIndex = 1..n; [5.4.1.b]
    TypeBoard = ARRAY[TypeIndex,TypeIndex] OF integer;

enum {n=8};
typedef unsigned type_index; /*[5.4.1.b]*/
typedef int type_board[n][n]; /*-------------------------------------------*/
The decision to represent each field of the board \( t \) by an integer instead of a Boolean value denoting occupation is because we wish to keep track of the history of successive board occupations.

The following convention is an obvious choice [5.4.1.c]:

\[
\begin{align*}
t[x,y] &= 0; \quad \text{\{field \( (x,y) \) has not been visited\}} \\
t[x,y] &= i; \quad \text{\{field \( (x,y) \) has been visited in the \( i \)-th move (\( 1 \leq i \leq n^2 \))\}}
\end{align*}
\]

The next decision concerns the choice of appropriate call parameters. They are to determine:

1. The starting conditions for the next move.
2. To report on its success or failure (output parameter).

The former task (1) is adequately solved by specifying:

(a) The coordinates \( x, y \) from which the move is to be made.
(b) The number \( i \) of the move (for recording purposes).

The second task (2) requires a Boolean result parameter \( q \), with \( q = \text{true} \) meaning the move was successful.

The next step is to refine the statements preceded by \( * \) in [5.4.1.a]. Which statements can now be refined on the basis of these decisions?

First, certainly \( * \text{board not full} \) can be expressed as \( i < n^2 \).

Moreover, if we introduce two local variables \( u \) and \( v \) to stand for the coordinates of possible move destinations determined according to the jump pattern of knights in the chasse play

- Then the predicate \( * \text{acceptable} \) can be expressed as the logical conjunction of the conditions that the new field lies on the board, i.e. \( 1 \leq u \leq n \) \text{ and } \( 1 \leq v \leq n \), and that it had not been visited previously, i.e., \( t[u,v] = 0 \).

The statement \( * \text{record current move} \) becomes \( t[u,v] = i \).

The statement \( * \text{erase current move}, \) can be expressed as \( t[u,v] = 0 \).

If a local Boolean variable \( q_1 \) is introduced and used as the result parameter in the recursive call of this algorithm, then \( q_1 \) may be substituted for \( * \text{successful move} \).

Thereby we arrive at the following formulation [5.4.1.d]:
PROCEDURE TryNextMove(i: integer; x,y: index;
    VAR q: boolean);
VAR u,v: integer; q1: boolean;
BEGIN
    *initialize the list of moves
    REPEAT
        *let u,v be the coordinates of the next move
        IF (1<=u<=n)AND(1<=v<=n)AND(t[u,v]=0) THEN
            BEGIN
                t[u,v]:= i;                          [5.4.1.d]
                IF i<n*n THEN
                    BEGIN
                        TryNextMove(i+1,u,v,q1);
                        IF NOT q1 THEN t[u,v]:= 0
                    END
                ELSE
                    q1:= true
            END
        UNTIL q1 OR (no more candidates in
          the list of next moves);
    q:= q1
END; {TryNextMove}

---------
/* Refinement of algorithm TryNextMove - step 1 – C variant */

typedef type_index index;
typedef unsigned int boolean;
#define true (1)
#define false (0)

void try_next_move(int i, type_index x, type_index y, boolean* q)
{
    int u,v; boolean q1;

    * initialize the list of moves
    do {
        *let u,v be the coordinates of the next move
        if ((1<=u<=n)&&(1<=v<=n)&&(t[u][v]==0))
        {
            t[u][v]=i;                       /*[5.4.1.d]*/
            if (i<n*n)
            {
                try_next_move(i+1,u,v,&q1);
                if (! q1)  t[u][v]=0;
            }
        }
    } while (!(q1 || (no more candidates in the list of next
        moves)));
    *q=q1;
}/*try_next_move*/
Relative to this refining step we can notice the followings:

The used **technique** is known in the literature as "**look-ahead**". Based on this technique:

1. The procedure **TryNextMove** is called with the **coordinates** \( x \) and \( y \).
2. A **new move** with **coordinates** \( u \) and \( v \) is selected from the next moves list, in fact the next move from the 8 possible in the list.
3. It tries to achieve the **next move** starting from the position \( u, v \).
   - If the move is **not** successful, respectively all the 8 possibilities of the next move list starting from position \( u \) and \( v \) where unsuccessfully trials, the **previous move** \( (u, v) \) is **erased** because it has no perspective (**REPEAT** loop).

Looking for the **search perspective**, each of the 8 possibilities is treated in the same manner:

1. Starting from each selected move, the search process advances as far as possible.
2. In the case of a failure (**not successful move**), the next move from the list is tried until all the possibilities are exhausted.
3. If all the possibilities where exhausted, the **previous move** is **cancelled**.

We can notice that, the searching process extends over **three searching levels**:

1. We start with initial position \( x, y \), a new position \( u, v \) is selected from the move list, and from this position the next move is attempted to be achieved.
2. If a failure is detected, the searching process can **return** to select a new way.

The associate **calls tree** has the next characteristics:

1. It’s of order 8 (from each point it can select 8 moving possibilities).
2. It’s **height** is \( n^2 \) (the steps number for reaching the final solution). This explains the exponential complexity of the process which determines the solution of the knight’s tour problem.

For the next and **last refinement step** there are few points to be specified.

We should note that so far the program was developed completely independently of the laws governing the **jumps of the knight**. Now is the time to take them into account.

- Given a starting coordinate pair \( <x, y> \) there are eight potential candidates \( <u, v> \) of the next destination. They are numbered 1 to 8 in figure 5.4.1.a.
A simple method of obtaining \( u, v \) from \( x, y \) is by addition of the coordinate differences stored in either an array of difference pairs or in two arrays of single differences.

Let \( a \) and \( b \), the two arrays appropriately initialized, the first for \( x \) coordinate, the second for \( y \).

Then an index \( k \) may be used to number the next candidate in the moves list \((1 \leq k \leq 8)\).

In [5.4.1.e] are represented the contents of these arrays.

\[
\begin{align*}
  a[1] &:= 2; & b[1] &:= 1; \\
  a[2] &:= 1; & b[2] &:= 2; \\
  a[3] &:= -1; & b[3] &:= 2; \\
  a[4] &:= -2; & b[4] &:= 1; \\
  a[5] &:= -2; & b[5] &:= -1; \\
  a[7] &:= 1; & b[7] &:= -2; \\
  a[8] &:= 2; & b[8] &:= -1; \\
\end{align*}
\]

The details are shown in [5.4.1.f,g], as Pascal respectively C variants.

The recursive procedure is initiated by a call with the coordinates \( x_0, y_0 \) of that field, as parameters from which the tour is to start.

This field must be given the value 1; all others are to be marked free.

We make the following mention: the variable \( t[u,v] \) exists only if \( u \) and \( v \) are in domain \( 1..n \).

In variant Pascal variant this requirement was implemented using the set type (the set variable \( S \) and the set operator \( \text{IN} \)) [5.4.1.f].

In C variant, a straight implementation based on simple comparisons, was used [5.4.1.f].

\{Knight’s Tour – final Pascal variant\}

```pascal
PROGRAM Knight’sTour;
  CONST n=5;
```
TYPE TypeIndex = 1..n;
VAR i,j: TypeIndex;
    q: boolean;
S: SET OF integer;
a,b: ARRAY[1..8] OF integer;
t: ARRAY[TypeIndex, TypeIndex] OF integer;

PROCEDURE TryNextMove(i: integer; x,y: TypeIndex;
    VAR q: boolean);

VAR k,u,v: integer;
k: integer;
q1: boolean;
BEGIN
    k:= 0;
    REPEAT
        k:= k+1; q1:= false;
        u:= x+a[k]; v:= y+b[k];
        IF (u IN S) AND (v IN S) THEN
            IF t[u,v]=0 THEN
                BEGIN
                    t[u,v]:= i;                      \[5.4.1.f\]
                    IF i<n*n THEN
                        BEGIN
                            TryNextMove(i+1,u,v,q1);
                            IF NOT q1 THEN t[u,v]:= 0
                        END
                    ELSE
                        q1:= true
                END
            END
        UNTIL q1 OR (k=8);
    q:= q1
END;\{TryNextMove\}

BEGIN \{main program\}
    S:= [1,2,3,4,5,6,7,8];
a[1]:= 2; b[1]:= 1;
a[2]:= 1; b[2]:= 2;
a[3]:= -1; b[3]:= 2;
a[4]:= -2; b[4]:= 1;
a[5]:= -2; b[5]:= -1;
a[6]:= -1; b[6]:= -2;
a[7]:= 1; b[7]:= -2;
a[8]:= 2; b[8]:= -1;
FOR i:=1 TO n DO
    FOR j:= 1 TO n DO t[i,j]:= 0;
t[1,1]:= 1; TryNextMove(2,1,1,q);
    IF q THEN
        BEGIN
            FOR j:= 1 TO n DO Write(' ',t[i,j]);
            writeln
        END
    ELSE writeln(' there is no solution ')
END.
/* Knight’s Tour – final C variant */

#include <limits.h>
#include <stdarg.h>
#include <stdlib.h>

enum {n=8};
typedef unsigned char type_index;
typedef unsigned int boolean;
#define true (1)
#define false (0)
type_index i,j;
boolean q;
int a[8],b[8];
int t[n][n];

void try_next_move(int i, type_index x, type_index y, boolean* q)
{
    int k,u,v;
    boolean q1;
    k=0;
    do {
        k=k+1;
        q1=false;
        u=x+a[k-1]; v=y+b[k-1];
        if ((0<=u<=n-1)&&(0<=v<=n-1))
            if (t[u-1][v-1]==0)
                {t[u-1][v-1]=i; /*[5.4.1.e]*/}
            if (i<n*n)
                try_next_move(i+1,u,v,&q1);
            if (!q1)
                t[u-1][v-1]=0;
            else
                q1=true;
    } while (!(q1||(k==8)));
    *q=q1;
} /*try_next_move*/

int main(int argc, const char* argv[])
{ /*main program*/
    a[0]=2; b[0]=1;
    a[1]=1; b[1]=2;
    a[2]=-1; b[2]=2;
a[7]=2; b[7]=-1;
for( i=1; i <=n; i++)
  for( j=1; j <=n; j++)
    t[i-1][j-1]=0;
t[0][0]=1;
try_next_move(2,1,1,&q);
if (q)
  for(i=1; i<=n; i++)
    {
      for(j=1; j<=n; j++)
        printf(" %3i", t[i-1][j-1]);
        printf("\n");
    }
else printf("there is no solution \n");
getch();
return 0;

/*---------------------------------------------------------------*/

• In figure 5.4.1.b are presented the results of program Knight’sTour for initial positions (1,1), (3,3) for n = 5 and (1,1) for n = 6.

```
 1  6  15 10 21  23  10 15  4  25
14  9  20  5  16  16  5  24  9  14
19  2  7  22 11  11  22  1  18  3
 8 13  24 17  4  6  17  20 13  8
25 18  3 12 23  21 12  7  2 19
```

**Fig.5.4.1.b.** Execution examples of Knight’sTour

• The essential characteristic of the knight’s tour algorithm is:
  
  • It builds the final solution step by step exploring and recording the way.
  
  • If at a certain moment, it realizes that the selected paths leads to an impasse or doesn’t possibly lead to the total solution, it returns erasing the trace of its steps until the proximal point which promises a new selection possibility is reached.
    
    • This in fact is the essence of the ”trial and error” technique.
This general approach to solve a problem in a such manner is named “backtracking”, and the corresponding algorithms are named backtracking algorithms.

5.4.2 Problems (n, m). Determining a Solution

The general pattern of a backtracking algorithm, it’s very well fitted to solve a class of problems for which:

- (1) The total solution presumes n successive steps.
- (2) Each step can be selected from m possibilities.

A such a problem we will denote as a problem of type (n, m).

In [5.4.2.a] appears the basic solving model of such a problem as procedure Try.

The model offers a single solution of the problem.

---

{Basic model for solving a problem of type (n,m). Determining a solution}

Procedure Try;
BEGIN
*initialize the selection of possibilities;
REPEAT
*select the next possibility;
IF *acceptable THEN
BEGIN
*record it as current;
IF *incomplete solution THEN
BEGIN
Try *next step;
IF *not successful THEN
*erase current record
END
ELSE
*successful step (found solution)
END
UNTIL (*successful step) OR (*there are not more possibilities)
END;{Try}
---

/* Basic model for solving a problem of type (n,m). Determining a solution */

void try()
{
*initialize the selection of possibilities;
do
*select the next possibility;
if (acceptable) /*[5.4.2.a]*/
{record it as current; 
    if (*incomplete solution ) 
        { 
            try *next step; 
            if (*not successful step)  
                *erase current record 
        } 
    else 
        *successful step (complete solution) 
} /*if */
while(!(*successful step)||(!(*there are not more possibilities)) 
} /*try*/
/*-----------------------------------------------*/

• This basic model can be materialized in different forms.
  • We present two of them.
  • (1) In first variant:
    • Procedure Try1 has as parameter the number of the current step.
    • The exploration of possibilities is accomplished in internal REPEAT loop [5.4.2.b].

{Solving a problem (n,m).Determining a solution - Variant 1 Pascal}

Procedure Try1(i: TypeStep);
VAR possibility: TypePossibility;
BEGIN
    possibility:= 0;{initialize the selection of possibilities}
    REPEAT
        possibility:= possibility+1; {select the next possibility}
        IF *acceptable THEN
            BEGIN [5.4.2.b]
                *record it as current;
                IF i<n THEN {incomplete solution}
                    BEGIN
                        Try1(i+1); {try next step}
                        IF *not successful THEN
                            *erase current record
                    END
                ELSE
                    *complete solution (display)
            END
        UNTIL *complete solution OR (possibility=m)
    END;{Try1}

/*Solving a problem (n,m).Determining a solution - Variant 1 C*/
void try1(type_step i)
{
    type_possibility possib;
    possib=0; /*initialize selection of possibilities*/
    do
        possib=possib+1; /*select the next possibility*/
        if (*acceptable)
            { /*[5.4.2.b]*/
                *record it as current;
                if (i<n) /*incomplete solution*/
                    {
                        try1(i+1); /*try next step*/
                        if (*not successful)
                            *erase current record
                    }
                else
                    *complete solution (display)
            }
        while (!*complete solution || (possib!=m))
    }/*try1*/
/*---------------------------------------------------------------*/

• (2) In the second variant:
  
• Procedure Try2 has as parameter a selection possibility.
  
• The construction of the solution is accomplished calling in recursive manner the procedure, successively, for each possibility [5.4.2.c].

• From the finality point of view, the both variants are identical, they differ by form.

{Solving a problem (n,m).Determining a solution - Variant 2 Pascal}

Procedure Try2(possibility: TypePossibility);
BEGIN
    IF *acceptable THEN
        BEGIN
            *record it as current;
            IF *incomplete solution THEN
                BEGIN [5.4.2.c]
                    Try2(possibility1);
                    Try2(possibility2);
                    ...
                    Try2(possibilitym);
                    *erase current record
                END
            ELSE
                *complete solution (display)
        END
    END; {Try2}
/* Solving a problem (n,m). Determining a solution - Variant 2 C */

void try2(type_possibility posib)
{
    if (*acceptable)
    {
        *record it as current;
        if (*incomplete solution)
        {
            /*[5.4.2.c]*/
            try2(possibility_1);
            try2(possibility_2);
            ...
            try2(possibility_m);
            *erase current record
        }
        else
        {
            *complete solution (display)
        }
    } /*try2*/
} /*-----------------------------------------------*/

• We presume that at each step the number of possibilities is fix (m), and the procedure is called initially from the main program as Try2(1).

• Hereinafter will be presented some applications of backtracking algorithms, which are perfectly suitable for recursive approach.

5.4.3 The Eight Queens Problem

• The problem of the eight queens is a well-known example of the use of trial-and-error methods and of backtracking algorithms.

• It was investigated by C. F. Gauss in 1850, but he did not completely solve it.

• This should not surprise anyone. After all, the characteristic property of these problems is that they defy analytic solution.

    • Instead, they require large amounts of exacting labor, patience, and accuracy.

    • Such algorithms have therefore gained relevance almost exclusively through the automatic computer, which possesses these properties to a much higher degree than people, and even geniuses, do.

• The eight queens problem is stated as follows:

    • Eight queens are to be placed on a chess board in such a way that no queen checks against any other queen.

• We can remark immediately that is a problem of type (n, m):
• There are 8 queens which has to be placed on the chess board, therefore the solution requires 8 steps.

• For each queen there are, as we will see, 8 possibilities to be placed.

• Using the schema [5.4.2.a] of the previous section 5.4.2 as a template, we readily obtain the following crude version of a solution[5.4.3.a] :

---

{Solving the eight queens problem – basic model}

PROCEDURE Try(i: queen);
BEGIN
  *initialize the selection of the place for the i-th queen
  REPEAT
    *select next place
    IF *safe place THEN
    BEGIN
      [5.4.3.a]
      *place queen i
      IF i<8 THEN
      BEGIN
        Try(i+1);
        IF *unsuccessful attempt THEN *remove the queen
      END
      ELSE
        *successful attempt (i=8)
      END
    END
  UNTIL *successful attempt OR (*there are no more places)
END;{Try}

/*---------------------------------------------------------------*/

/* Solving the eight queens problem – basic model */

void try(queen i)
{
  *initialize the selection of the place for the i-th queen
  do
    *select next place
    if (*safe place) 
    {
      /*[5.4.3.a]*/
      *place queen i
      if (i<8)
      {
        try(i+1);
        if (*unsuccessful attempt) *remove the queen
      }
    else
      *successful attempt (i=8)
  }
  while (!( *successful attempt) || !( *there are no more places))
} /*try*/
/*--------------------------------------------------------------------*/
In order to proceed, it is necessary to make some commitments concerning the data representation.

- Since we know from the rules of chess that a queen checks all other figures lying in either the same column, row or diagonal on the board, we infer that each column may contain one and only one queen.

- That the choice of a position for the \(i\)-th queen may be restricted to the \(i\)-th column.

As result:

- The parameter \(i\) therefore becomes the column index.

- The selection process for positions ranges over the eight possible values for a row index \(j\) in column \(i\).

In conclusion:

- We have a typical problem \((8, 8)\).

- The solution requires 8 steps (placing 8 queens).

- Each queen can be placed in one of the 8 positions of its proper column (8 possibilities).

- The associated calls tree is of order 8 and has the height 8.

There remains the question of representing the eight queens on the board.

- An obvious choice would again be a square matrix \(t[8, 8]\) to represent the board.
  - But a little inspection reveals that such a representation would lead to fairly cumbersome operations for checking the availability of positions.
  - This is highly undesirable since it is the most frequently executed operation.

- We should therefore choose a data representation which makes checking as simple as possible.

- The best recipe is to represent as directly as possible that information which is truly relevant and most often used.

  - In our case this is not the position of the queens, but whether or not a queen has already been placed along each row and diagonal.

  - We already know that exactly one queen is placed in each column \(i\) for \(1 \leq i \leq 8\).

  - This leads to the following choice of data structures [5.4.3.b]:

{The eight queens problem – defining data structures}
VAR x: ARRAY[1..8] OF integer;
a: ARRAY[1..8] OF boolean;
b: ARRAY[b1..b2] OF boolean;         [5.4.3.b]
c: ARRAY[c1..c2] OF boolean;

/* The eight queens problem - defining data structures */

int x[8];
boolean a[8];       /*[5.4.3.b]*/
boolean b[b2];
boolean c[c2];

/*---------------------------------------------------------------*/

• Supposing queen i is placed in position (i, j) on the chess board, the significance of this action is presented in [5.4.3.c].

---------------------------------------------------------------

x[i]:= j   denote the position j of the queen in column i
a[j]:= true no queen lies in the j row        [5.4.3.c]
b[k]:= true no queen occupies the diagonal /k
b[k]:= true no queen occupies the diagonal \k

/*---------------------------------------------------------------*/

• On the chess board there are 15 diagonals / (inclined to right) and 15 diagonals \ (inclined to left) (figure 5.4.3.a).

• We note that in a /-diagonal for all fields the sum of their coordinates i and j is a constant.

• In a \- diagonal the coordinate differences i - j are constant.

• In figure 5.4.3.a appears represented the two types of diagonals.

• We observe that for the diagonals / the sums i+j are in the range [2, 16].

• For the diagonals \ the differences belongs to domain [-7, 7].
Based on this, we can choose the values for $b_1, b_2, c_1, c_2$, the limits of the arrays $b$ and $c$ [5.4.3.b].

- One of the possibilities is to chose the following values: $b_1 = 2, b_2 = 16, c_1 = 0, c_2 = 14$.

- For $b_1$ and $b_2$ where selected exactly the margins of the sums interval.

- The interval for differences has been translated to right with value 7, to obtain positive values for the indices in array $c$.

- In other words:

  - The access in array $b$ designated to diagonals / is realized by $b[i+j]$. 

Fig. 5.4.3.a. Types of diagonals in eight queens problem.
• The access in array \( c \) designated to diagonals \( \) is realized by \( c[i-j+7] \).

• Initial all the locations in arrays \( a, b \) and \( c \) are set to \textit{true}.

• With these details, the statement \textit{place queen \( i \) in position \( (i,j) \), \( i \) being the proper column became} [5.4.3.d]:

\[
\begin{align*}
\{ \text{\textit{place queen \( i \) in position \( (i,j) \)}} \} \\
\text{x}\[i]\:=\ j; \ a\[j]\:= \textit{false}; \ b[i+j]\:= \textit{false}; \ c[i-j+7]:= \textit{false} \\
\end{align*}
\]

\[
\begin{align*}
\{ \text{\textit{place queen \( i \) in position \( (i,j) \)}} \} \\
\text{x}\[i]\:=\ j; \ a\[j]\:=\textit{false}; \ b[i+j]\:=\textit{false}; \ c[i-j+7]:=\textit{false} \\
\end{align*}
\]

• In the same context, the statement \textit{remove queen} is refined in [5.4.3.e]:

\[
\begin{align*}
\{ \text{\textit{remove queen}} \} \\
a\[j]\:=\textit{true}; \ b[i+j]\:=\textit{true}; \ c[i-j+7]:=\textit{true} \\
\end{align*}
\]

\[
\begin{align*}
\{ \text{\textit{remove queen}} \} \\
a\[j]=\textit{true}; b[i+j]=\textit{true}; c[i-j+7]=\textit{true} \\
\end{align*}
\]

• The condition \textit{safe place} is fulfilled if the selected field \( (i,j) \) belongs to a row and two diagonals which are free \( (\text{true}) \). This situation can be expressed as a logical expression [5.4.3.f]:

\[
\begin{align*}
\{ \text{\textit{safe place}} \} \\
\text{a}[j] \text{ AND } \text{b}[i+j] \text{ AND } \text{c}[i-j+7] \\
\end{align*}
\]

\[
\begin{align*}
\{ \text{\textit{safe place}} \} \\
a[j] \&\& b[i+j] \&\& c[i-j+7] \\
\end{align*}
\]

• The associate program is presented in [5.4.3.g].

\[
\begin{align*}
\{ \text{The eight queens problem - determining a solution - final variant} \} \\
\text{PROGRAM Queens1;} \\
\quad \{ \text{find a solution to 8 queens problem} \}
\end{align*}
\]

\[
\begin{align*}
\text{VAR} \quad i: \text{integer}; q: \text{boolean}; \\
\quad a: \text{ARRAY}[1..8] \text{ OF boolean}; \\
\quad b: \text{ARRAY}[2..16] \text{ OF boolean}; \\
\quad c: \text{ARRAY}[0..14] \text{ OF boolean};
\end{align*}
\]
x: ARRAY[1..8] OF integer;

PROCEDURE Try(i: integer; VAR q: boolean);
VAR j: integer;
BEGIN
  j:= 0;
  REPEAT
    j:= j+1; q:= false;
    IF a[j] AND b[i+j] AND c[i-j+7] THEN
      BEGIN
        x[i]:= j;                          [5.4.3.g]
        a[j]:= false; b[i+j]:= false;
        c[i-j+7]:= false;
        IF i<8 THEN
          BEGIN
            Try(i+1,q);
            IF NOT q THEN
              BEGIN
                a[j]:= true; b[i+j]:= true;
                c[i-j+7]:= true
              END
          END
        ELSE
          q:= true
      END
    UNTIL q OR (j=8)
END; {Try}

BEGIN {main program}
  FOR i:=1 TO 8 DO a[i]:= true;
  FOR i:=2 TO 16 DO b[i]:= true;
  FOR i:=0 TO 14 DO c[i]:= true;
  Try(1,q);
  IF q THEN
    FOR i:=1 TO 8 DO Write(x[i]);
    Writeln
END. {main program}

#include <stdio.h>
/*---------------------------------------------------------------*/
/* find a solution to 8 queens problem */

typedef unsigned boolean;
#define true (1)
#define false (0)

int i; boolean q;
boolean a[8];
boolean b[15];
boolean c[15];
int x[8];

void try(int i, boolean* q)
{
```c
int j;
j=0;
do {
  j=j+1; *q=false;
  if (a[j-1]&b[i+j-2]&c[i-j+7]){
    x[i-1]=j;                         /*[5.4.3.g]*/
    a[j-1]=false;
    b[i+j-2]=false;
    c[i-j+7]=false;
    if (i<8){
      try(i+1,q);
      if (!*q){
        a[j-1]=true;
        b[i+j-2]=true;
        c[i-j+7]=true;
      }
    }
    else
      *q=true;
  }
  } while (!(*q||(j==8)));
} /*try*/

int main(int argc, const char* argv[])
{
  /*main program*/
  for(i=1; i<=8; i++)
    a[i-1]=true;
  for(i=2; i<=16; i++)
    b[i-2]=true;
  for(i=0; i<=14; i++)
    c[i]=true;
  try(1,&q);
  if (q)
    for(i=1; i<=8; i++)
      printf("%i ", x[i-1]);
  printf(\n"
return 0;
}

/*---------------------------------------------------------------*/
```

- The solution determined for the program is $x = (1, 5, 8, 6, 3, 7, 2, 4)$ and appears in graphical representation in figure 5.4.3.b

![Diagram](image-url)
5.4.4 Determining All the Solutions for a Problem \((n, m)\). Generalization of the Eight Queens Problem

- The model of the algorithm which determines a solution for a problem of type \((n, m)\) can be easy extended to determine all the solutions.

- For this is necessary that, the steps which build the solution, to be generated in an ordered manner which guarantee that each can be generated only once.

  - This propriety correspond to search in the calls tree in a systematic manner, thus each node is visited only once.

- As soon as, a solution was found and recorded, the algorithm doesn’t stop, it continues to determine the next solution, based on a systematical selection until all the possibilities where exhausted.

- The general scheme derived from [5.4.2.a], which implement this approach, is presented in [5.4.4.a].

- Surprisingly, founding all the solutions of a problem of type \((n, m)\) presumes a simpler algorithm that the case of a single solution.

---

{Model for solving a problem of type \((n,m)\) – determining all the solutions}

Procedure Try1;
    BEGIN
        FOR *all selection possibilities DO
            IF *selection is acceptable THEN
                BEGIN
                    *record it as current          [5.4.4.a]
                    IF *incomplete solution THEN
                        Try1 *next step
                    ELSE
                        *display solution;
                        *erase current record
                    END
                END; {Try1}
    END; {Try1}

/* Model for solving a problem of type \((n,m)\) – determining all the solutions */

void try1()
{
    for (*all selection possibilities)
        if (*selection is acceptable)
As in the previous case we have two implementation variants:

(1) In the first variant:

- The procedure Try1 has as call parameter the number of the current step and explores all the possibilities in internal loop FOR [5.4.4.b].

- For reveling all the possibilities in each step, all the values of k=[1,m] must be completed. As result the REPEAT cycle was replaced by a FOR loop.

**{Solving a problem of type (n,m) – determining all the solutions – Variant 1}**

PROCEDURE Try1(i: TypeStep);
VAR possib: TypePossibility;
BEGIN
  FOR possib:= 1 TO m DO
    IF *acceptable THEN
      BEGIN
        *record it as current;
        IF i<n THEN
          Try1(i+1) [5.4.4.b]
        ELSE
          *display solution;
          *erase record
      END
    END; {Try1}
END;

/* Solving a problem of type (n,m) – determining all the solutions – Variant 1 */

void try1(type_step i)
{
  type_possibility possib;
  for (possib=1 until m)
    if (*acceptable) {
      *record it as current;
      if (i<n) /*[5.4.4.b]*/
        try1(i+1)
      else
        *display solution;
      *erase record
```
(2) In the second variant:

- Procedure Try2 has as calling parameter a selection possibility.
- The construction of the solution is achieved calling the procedure in turn, for each possibility [5.4.4.c].

```plaintext
{ Solving a problem of type (n,m) – determining all the solutions – Variant 2}

Procedura Try2(possib: TypePossibility);
BEGIN
  IF *acceptable THEN
    BEGIN
      *record it as current;
      IF *incomplete solution THEN
        BEGIN
          Try2(possibility_1);
          Try2(possibility_2);
          ...
          Try2(possibility_m);
        END
      ELSE
        *display solution
        *erase current record
    END
  END;
END;{Try2}

{ Solving a problem of type (n,m) – determining all the solutions – Variant 2}

void try2(type_possibility possib)
{
  if (*acceptable)
  {
    *record it as current;
    if (*incomplete solution)
      {
        try2(possibility_1);
        try2(possibility_2);
        ...
        try2(possibility_m);
      }
    else
      *display solution
      *erase current record
  }
}
As example, is presented the generalization of the eight queens problem for determining all the solutions[5.4.4.d].

{The eight queens problem – determining all the solutions – Pascal variant}

PROGRAM EightQueens;
VAR i: integer;
a: ARRAY[1..8] OF boolean;
b: ARRAY[2..16] OF boolean;
c: ARRAY[0..14] OF boolean;
x: ARRAY[1..8] OF integer;

PROCEDURE Display;
VAR k: integer;
BEGIN
FOR k:=1 TO 8 DO Write(x[k]); Writeln
END;{Display}

PROCEDURE Try(i: integer);
VAR j: integer;
BEGIN
FOR j:=1 TO 8 DO IF a[j] AND b[i+j] AND c[i-j+7] THEN
BEGIN
x[i]:= j;
a[j]:= false; b[i+j]:= false; c[i-j+7]:= false;
IF i<8 THEN
Try(i+1)
ELSE
Display;
a[j]:= true; b[i+j]:= true; c[i-j+7]:= true
END;{Try}
END {main program}
FOR i:=1 TO 8 DO a[i]:= true;
FOR i:=2 TO 16 DO b[i]:= true;
FOR i:=0 TO 14 DO c[i]:= true;
Try(1)

;/* The eight queens problem – determining all the solutions – C variant */
#include <stdio.h>

typedef unsigned boolean;
#define true (1)
#define false (0)
int i;
boolean a[8];
boolean b[15];
boolean c[15];
int x[8];

void display()
{
    int k;
    for(k=1; k<=8; k++)
        printf("%i ", x[k-1]); /*[5.4.4.d]*/
    printf("\n");
} /*display*/

void try(int i)
{
    int j;

    for(j=1; j<=8; j++)
        if (a[j-1] && b[i+j-2] && c[i-j+7])
            {
                x[i-1]=j;
                a[j-1]=false; b[i+j-2]=false; c[i-j+7]=false;
                if (i<8)
                    try(i+1);
                else
                    display();
                a[j-1]=true; b[i+j-2]=true; c[i-j+7]=true;
            } /*if*/
} /*try*/

int main(int argc, const char* argv[])
{
    /*main program*/
    for(i=1; i<= 8; i++) a[i-1]=true;
    for(i=2; i<=16; i++) b[i-2]=true;
    for(i=0; i<=14; i++) c[i]=true;
    try(1);
    return 0;
} /*-----------------------------------------------*/

• The above algorithm determines all the 92 solutions of the 8 queens problems.

• In reality exist only 12 distinct solutions which are represented in figure 5.4.4.a. The rest of the solutions can be deduced by symmetry.
Fig.5.4.4.a. The solutions of the 8 queens problem.

5.4.5 The Stable Marriage Problem

- The stable marriage problem generic specification is:
  - Two distinct disjunctive sets $A$ and $B$ of equal cardinality $n$ are given.
  - It requires to find a set of $n$ pairs $<a,b>$ so that $a \in A$ and $b \in B$, which have to satisfy certain constraints.

- There are many criteria to compose pairs of elements satisfying constrains, but one of the most known is the "stable marriage rule".
  - Assume that $A$ is a set of men and $B$ is a set of women.
  - Each man and each woman has stated distinct preferences for their possible partners.
  - If the $n$ couples are chosen such that there exists a man and a woman who are not married, but who would both prefer each other to their actual marriage partners, then the assignment is unstable.
  - If no such pair exists, it is called stable.

- This situation characterizes many related problems in which assignments have to be made according to preferences such as, for example, the choice of a school by students, the choice of recruits by different branches of the armed services, etc.

- The example of marriages is particularly intuitive.
  - Note, however, that the stated list of preferences is invariant and does not change after a particular assignment has been made.
• This assumption simplifies the problem, but it also represents a grave distortion of reality (called abstraction).

• One way to search for a solution is to try pairing off members of the two sets one after the other until the two sets are exhausted.

• Setting out to find all stable assignments, we realize that this is a problem of type \((n, m)\) and we can readily sketch a solution by using the generic algorithm which determines all the solution of this type of problem [5.4.4.a].

• Let \textbf{Try} \((m: \text{man})\) denote the algorithm to find a partner for man \(m\).

• The idea is to take the men one by one and to distribute each a woman based on the man’s list of stated preferences.

• The first version based on these assumptions is [5.4.5.a]:

---

\textit{The stable marriage problem – refinement step 1}

\begin{verbatim}
PROCEDURE Try (m: TypeMan);
 VAR r: TypeRank;
 BEGIN
 FOR r:=1 TO n DO
 BEGIN
 *select the r-th preference of the man m
 IF *acceptable THEN
 BEGIN
 *record the marriage;
 IF m is not the last man THEN
 Try(Successor(m))
 ELSE
 *display the stable set;
 *cancel the marriage
 END{IF}
 END{FOR}
 END{Try}
\end{verbatim}

---

\textit{The stable marriage problem – refinement step 1}

\begin{verbatim}
void try(type_man m);
{
 type_rank r;
 for(r=1; r<=n; r++)
 {
 *select the r-th preference of the man m
 if(*acceptable)
 /*[5.4.5.a]*/
 {*
 *record the marriage;
 if(m is not the last man)
 try(successor(m));
 else
 *display the stable set;
 }
*cancel the marriage
} /*if*/
} /*for*/
} /*try*/
/*-----------------------------------------------*/
• For the beginning we will denote the data structures.

  • From the clarity reasons, we introduce three identical scalar types but with different names [5.4.5.b].

• The initial data are represented by two matrices that indicate the men's and women's preferences.

  • PrefMen[m] represents the preferences list of man m, respective PrefMen[m,r] is the woman which occupies the rank r in the list of man m.

  • PrefWomen[w] is the preferences list of woman w, and PrefWomen[w,r] is the man which is the r-th choice in her list (fig.5.4.5.a).

-------------------------------------------------------------------
{The stable marriage problem – defining data structures}

TYPE TypeMan   = 1..n;
   TypeWoman = 1..n;                         [5.4.5.b]
   TypeRank  = 1..n;

VAR PrefMen: ARRAY[TypeMan,TypeRank] OF TypeWoman;
   PrefWomen: ARRAY[TypeWomen,TypeRank] OF TypeMan;

------------------------------------------------------------------- /* The stable marriage problem – defining data structures */

typedef unsigned type_man;
typedef unsigned type_woman; /*[5.4.5.b]*/
typedef unsigned type_rank;

type_woman pref_men[n][n];
type_man pref_woman[n][n];
/*-----------------------------------------------*/

• An example of input data for stable marriage problem appears in figure 5.4.5.a.

• The result of the stable marriage algorithm execution is represented as an array of women x, so that x[m] stores the partner of man m.

• For symmetry between men and women reasons, we introduce one more array of men y, so that y[w] stores the partner of woman w [5.4.5.c].

-------------------------------------------------------------------
{ The stable marriage problem – defining output data structures }

VAR x: ARRAY[TypeMan] OF TypeWoman;         [5.4.5.c]
y: ARRAY[TypeWoman] OF TypeMan;

-------------------------------------------------------------------
/* The stable marriage problem – defining output data structures */

type_woman x[n];                      /*[5.4.5.c]*/

type_man  y[n];

/*---------------------------------------------------------------*/

<table>
<thead>
<tr>
<th>rank</th>
<th>select the woman</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>man 1</td>
<td></td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rank</th>
<th>select the man</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>woman 1</td>
<td></td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>6</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>7</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig.5.4.5.a. Input data for stable marriage problem.

• Actually, y is **redundant**, since it represents information that is already present through the existence of x.

  • In fact, the relations \(x[y[w]]=w\) and \(y[x[m]]=m\) hold for all m and w who are married.

  • Thus, the value \(y[w]\) could be determined by a simple search of x, the array y, however, clearly improves the **efficiency** of the **algorithm**.

• The information represented by x and y is needed to determine **stability** of a **proposed set of marriages**.

  • Since this set is constructed stepwise by marrying individuals and testing stability after each proposed marriage, x and y are needed even before all their components are defined.

• In order to keep track of the **candidates**, we may introduce Boolean arrays **SingleMan** and **SingleWoman**[5.4.5.d].

{The stable marriage problem – defining auxiliary data structures (1)}
SingleMan: \texttt{ARRAY[TypeMan] OF boolean}; \hspace{1cm} \texttt{[5.4.5.d]}

SingleWoman: \texttt{ARRAY[TypeWoman] OF boolean;}

\texttt{SingleMan[m]= false } \{\text{the man } m \text{ has a pair}\}
\texttt{SingleWoman[w]= false } \{\text{the woman } w \text{ has a pair}\}

---------------------------------------------

\texttt{/* The stable marriage problem – defining auxiliary data structures (1)*/}

\begin{verbatim}
boolean single_man[type_man]; \hspace{1cm} \texttt{[5.4.5.d]}\nbolean single_woman[woman];

\texttt{single_man[m]==false; } \{\text{the man } m \text{ has a pair}\}
\texttt{single_woman[w]==false; } \{\text{the woman } w \text{ has a pair}\}
\end{verbatim}

---------------------------------------------

\begin{itemize}
  \item The significance of the two fields is presented also in \texttt{[5.4.5.d]}.
    \begin{itemize}
      \item We can notice that \texttt{SingleMan[m]=false} implies that \texttt{x[m]} is defined, thus man \texttt{m} has a pair.
      \item In the same idea \texttt{SingleWoman[w]=false} implies that \texttt{y[w]} is defined, thus woman \texttt{w} has a pair.
    \end{itemize}
  \item The construction of the pairs is achieved starting the men which are selected in their initial order (\texttt{Successor(m)} in \texttt{[5.4.5a]}), the fact that a man \texttt{k} was or wasn’t selected, can be determined from the actual value of \texttt{m} in conformity with relation \texttt{[5.4.5.e]}:
    \begin{verbatim}
    NOT SingleMan[k] \equiv k<m \hspace{1cm} \texttt{[5.4.5.e]}
    \end{verbatim}
    \begin{itemize}
      \item This suggests that, in fact the array \texttt{SingleMan} is not necessary, as consequence, a single array \texttt{SingleWoman} will be used.
      \item Condition \texttt{*acceptable} can be refined using the Boolean variable \texttt{SingleWoman[w]} and the condition \texttt{*stable}, where \texttt{stable} will be detailed later.
      \item Applying these developments the next refinement becomes \texttt{[5.4.5.f]}.
    \end{itemize}
\end{itemize}

\begin{verbatim}
{ The stable marriage problem – refinement 2}

PROCEDURE Try(m: TypeMan);
  VAR r: TypeRank; f: TypeWoman;
  BEGIN
    FOR r:=1 TO n DO BEGIN
      w:= PrefMen[m,r];
      IF SingleWoman[w] AND *stable THEN \hspace{1cm} \texttt{[5.4.5.f]}
      BEGIN
        x[m]:= w; y[w]:= m; SingleWoman[w]:= false
        IF m<n THEN
          Try(Successor(m))
        ELSE
      END
    END
  END

\end{verbatim}
*display the stable set;
    SingleWoman[w] := true
END
END; {Try}

/* The stable marriage problem - refinement 2 */

void try(type_man m);
{
    type_rank r;
    type_woman w;
    for (r=1; r<=n; r++)
    {
        w=pref_men[m,r];
        if (single_woman[f] && *stable) /*[5.4.5.f]*/
            if (m<n )
                try(successor(m));
            else
                *display the stable set;
        single_woman[f]=true
    }
} /*try*/

• The crucial task is now the refinement of the algorithm to determine stability.

  • Unfortunately, it is not possible to represent stability by such a simple expression as the safety of a queen's position.

  • The first detail that should be kept in mind is that stability follows by definition from comparisons of ranks.

  • The ranks of men or women, however, are nowhere explicitly available in our collection of data established so far.

    • Surely, the rank of woman w in the mind of man m can be computed, but only by a costly search of w in PrefMen and vice versa.

  • Since the computation of stability is a very frequent operation, it is advisable to make this information more directly accessible. To this end, we introduce two new data structures [5.4.5.g]:

    • The matrix RWm in which RWm[m,w] represents the rank of woman w in the list of the man m.

    • The matrix RMw in which RMw[w,m] represents the rank of man m in the list of the woman w.
• It is plain that the values of these auxiliary arrays are constant and can initially be determined from the values of PrefMen and PrefWomen.

{ The stable marriage problem – defining auxiliary data structures (2)}

RWM: ARRAY[TypeMan,TypeWoman] OF TypeRank; {the rank of women in the lists of men}

RMW: ARRAY[TypeWoman,TypeMan] OF TypeRank; {the rank of men in the lists of women}

/* The stable marriage problem – defining auxiliary data structures (2)*/

type_rank rwm[n][n]; /* the rank of women in the lists of men */

[type_rank rmw[n][n]; /* the rank of men in the lists of women */

• The process of determining the predicate stable now proceeds strictly according to its original definition.

  • Recall that we are trying the feasibility of marrying the man \(m \in [1,n]\) and \(w\), where \(w = \text{PrefMen}[m,r]\), i.e., \(w\) is the \(r\)-th choice of man \(m\).

  • Being optimistic, we first presume that stability still prevails, and then we set out to find possible sources of trouble. Where could they be hidden?

  • There are two symmetrical possibilities:

    • Source (1) Presuming that man \(m\) was married with woman \(w\), there might be a women \(pw\), preferred to \(w\) by \(m\), who herself prefers \(m\) over her husband \(m1\) [5.4.5.h. (1)]

    • Source (2) Presuming that woman \(w\) is wife of man \(m\), there might be a man \(pm\), preferred to \(m\) by \(w\), who himself prefers \(w\) over his wife \(w1\) [5.4.5.h. (2)].

(1) \(m \rightarrow w\)

\(m1 \rightarrow pw\)

\(y[pw]\)

(2) \(w \rightarrow m\)

\(w1 \rightarrow pm\)

\(x[pm]\)

[5.4.5.h]
• Initial we analyze trouble source (1):

  • For all women \( pw \) having a lower rank in list of \( m \), i.e. are more preferred than his wife \( w \), it verifies if there is a woman which prefers more \( m \) than her actual assigned husband.

  • For this, are compared the ranks \( RMW[pw, m] \) and \( RMW[pw, y[pw]] \) for all the women \( pw \) which are more preferred by \( m \) than \( w \), that means all \( pw = PrefMen[m, i] \) for \( 1 \leq i < r \), where \( r \) is the rank of woman \( w \), the actual wife of \( m \).

  • In fact, all women \( pw \) are already married, because if any of them should has been single, he should has assigned to \( m \) previously.

  • We have to note that a rank with a bigger value means a weaker preference.

  • The comparison result is the Boolean variable \( s \) which stands for stability. As long as \( s \) remains true, the relation is stable.

  • The described process can be implemented as a simple linear search [5.4.5.i].

{Verifying trouble source (1)}

\[
\begin{align*}
  s &:= \text{true}; i := 1; \\
  \text{WHILE} &\ (i < r) \ \text{AND} \ s \ \text{DO} \ [5.4.5.i] \\
  \text{BEGIN} &
  \begin{array}{l}
  \text{pw} := \text{PrefMen}[m, i]; i := i + 1; \\
  \text{IF NOT SingleWoman[pw] THEN} \\
  s := \text{RMW}[pw, m] > \text{RMW}[pw, y[pw]]
  \end{array} \\
  \text{END;}
\end{align*}
\]

/* Verifying trouble source (1)*/

\[
\begin{align*}
  \text{s} &= \text{true}; i = 1; \\
  \text{while} &\ ((i < r) \ \&\& \ s) \ /*[5.4.5.i]*/ \\
  \{ &
  \text{pw} = \text{pref\_men}[m, i]; i++; \\
  \text{if} &\ (!\text{single\_woman}[pw]) \\
  s &:= \text{rmw}[pw, m] > \text{rmw}[pw, y[pw]]?1:0;
  \}
\end{align*}
\]

• Now we analyze trouble source (2):

  • For all men \( pm \) having a lower rank in list of \( w \), i.e. are more preferred than her husband \( m \), it verifies if there is a man which prefers more \( w \) than his actual assigned wife.

  • For this purpose, all the preferred men \( pm = PrefWomen[w, i] \) for \( 1 \leq i < RMW[w, m] \), are investigated.

    • In an analog manner with another trouble source, is necessary to compare the ranks \( RWM[pm, w] \) and \( RWM[pm, x[pm]] \).
• We have to avoid the comparisons with the men $p_m$ which are not already married.

• The condition is fulfilled for $p_m < m$, because all the men preceding $m$ are already married.

• The comparison result is the Boolean variable $s$ which stands for stability. As long as $s$ remains true, the relation is stable.

• The described process can be implemented as a simple linear search [5.4.5.i].

• The code implementing the verification of the second trouble source appears in [5.4.5.j].

{ Verifying trouble source (2)}

\[
i := 1; \text{lim} := \text{RMW}[w,m]\\
\text{WHILE } (i < \text{lim}) \text{ AND } s \text{ DO }\]
BEGIN
\[
\text{pm} := \text{PrefWomen}[w,i]; i := i + 1;\\
\text{IF } \text{pm} < m \text{ THEN } s := \text{RWM}[\text{pm},w] > \text{RWM}[\text{pm},x[\text{pm}]]
\]
END;

/* Verifying trouble source (2)*/

{The stable marriage problem – final variant – Pascal}

PROGRAM StableMarriage;
CONST n=8;
TYPE TypeMan = 1..n; TypeWoman = 1..n; TypeRank = 1..n;

VAR b: TypeMan; f: TypeWoman; o: TypeRank;
PrefMen: ARRAY[TypeMan,TypeRank] OF TypeWoman;
PrefWomen: ARRAY[TypeWoman,TypeRank] OF TypeMan;
RWM: ARRAY[TypeMan,TypeWoman] OF TypeRank;
RMW: ARRAY[TypeWoman,TypeMan] OF TypeRank;
x: ARRAY[TypeMan] OF TypeWoman;
y: ARRAY[TypeWoman] OF TypeMan;
SingleWoman: ARRAY[TypeWoman] OF boolean;

PROCEDURE Display;
VAR m1: TypeMan;
    rm, rw: integer;
BEGIN
    rm := 0; rw := 0;
    FOR m1 := 1 TO n DO
        BEGIN
            Write(x[m1]);
            rm := rm + RWM[m1, x[m1]]; rw := rw + RMW[x[m1], m1]
        END;
    Writeln(rm, rw)
END;

PROCEDURE Try(m: TypeMan);
VAR r: TypeRank; w: TypeWoman;

FUNCTION Stable: boolean;
VAR pm: TypeMan; pw: TypeWoman;
i, lim: TypeRank; s: boolean;
BEGIN
    s := true; i := 1; {source (1)}
    WHILE (i < r) AND s DO
        BEGIN
            pw := PrefMen[m, i]; i := i + 1;
            IF NOT SingleWoman[pw] THEN
                s := RMW[pw, m] > RMW[pw, w]
        END;
    i := 1; lim := RMW[w, b]; {source (2)}
    WHILE (i < lim) AND s DO
        BEGIN
            pm := PrefWomen[w, i]; i := i + 1;
            IF pm < m THEN
                s := RWB[pm, w] > OFB[pm, x[pm]]
        END;
    Stable := s
END;

BEGIN {Try}
    FOR r := 1 TO n DO
        BEGIN
            f := PrefMen[m, r];
            IF SingleWoman[f] THEN
                IF Stable THEN
                    BEGIN
                        x[m] := w; y[w] := m; SingleWoman[w] := false;
                        IF m < n THEN
                            Try(Successor(m))
                        ELSE
                            Display;
                        SingleWoman[w] := true
                    END
        END;
END; {Try}

BEGIN {main program}
    FOR m := 1 TO n DO
FOR $r:=1$ TO $n$ DO
BEGIN
    Read(PrefMen[$m$, $r$]);
    RWM[$m$, PrefMen[$m$, $r$]] := $r$
END;
FOR $w:=1$ TO $n$ DO
BEGIN
    Read(PrefWomen[$w$, $r$]);
    RMW[$w$, PrefWomen[$w$, $r$]] := $r$
END;
FOR $w:=1$ TO $n$ DO SingleWoman[$w$] := true;
Try(1)
END.

/* The stable marriage problem – final variant – C */

defined enum {n =8};
defined typedef unsigned char type_man;
defined typedef unsigned char type_woman;
defined typedef unsigned char type_rank;
defined typedef int type_rank

defined typedef unsigned boolean;
defined #define true (1)
defined #define false (0)

type_man b;
type_woman f;
type_rank o;
type_woman pref_men[n][n];
type_man pref_women[n][n];
type_rank ofb[n][n];
type_rank obf[n][n];
type_woman x[n];
type_man y[n];
boolean single_woman[n];

void display()
{
    type_man m1;
    int rm, rw;
    rm=0; rw=0;  /*[5.4.5.k]*/
    for (m1=1; m1<=n; m1++){
        printf("%i", x[m1-1]);
        rm=rm+rwm[m1][x[m1-1]-1];
        rw=rw+rmw[x[m1-1]][m1-1];
    }
    printf("%i%i\n", rm,rw);
} /*display*/

void try(type_man b);

static boolean stable(type_rank* r, type_man* m, type_woman* w)
{  
    type_man pm; type_woman pw;
    type_rank i, lim; boolean s;
    boolean stable_result;

    s=true; i=1; /*source (1)*/
    while ((i<*r) && s){
        pw=pref_men[*m][i-1]; i++;
        if (! single_woman[pw-1])
            s=rmw[pw][*m-1]>rmw[pw][y[pw-1]-1];
    }
    i=1; lim=rmw[*w][*m-1]; /*source (2)*/
    while ((i<lim) && s){
        pm=pref_women[*w][i-1]; i++;
        if (pm<*m)
            s=rwm[pm][*w-1]>rwm[pm][x[pm-1]-1];
    }
    stable_result=s;
    return stable_result;
} /*stable*/

void try(type_man m)
{
    type_rank r; type_woman w;
    /*try*/
    for(r=1; r<=n; r++){
        w=pref_men[m][r-1];
        if (single_woman[w-1])
            if (stable(&r, &m, &w)) {
                x[m-1]=w; y[w-1]=m;
                single_woman[w-1]=false;
                if (m<n)
                    try(m++);
                else
                    display();
                single_woman[w-1]=true;
            }
    }
} /*try*/

int main(int argc, const char* argv[])
{
    /*main program*/
    for(m=1; m<=n; m++)
        for(r=1; r<=n; r++)
        {
            scanf("%c", &pref_men[m][r-1]);
            rwm[m][pref_men[m][r-1]-1]=r;
        }
    for(m=1; m<=n; m++)
        for(r=1; r<=n; r++)
        {
            scanf("%c", &pref_women[w][r-1]);
            rmw[w][pref_women[w][r-1]-1]=r;
        }
for(w=1; w<=n; w++) single_woman[w-1]=true;
try(1);
return 0;

/*---------------------------------------------*/

- For the input data in figure 5.4.5.a are obtained the stable solutions presented in figure 5.4.5.b.
- This algorithm is based on a straightforward backtracking scheme.
- Its efficiency primarily depends on the sophistication of the solution tree pruning scheme.
  - A somewhat faster, but more complex and less transparent algorithm has been presented by McVitie and Wilson, who also have extended it to the case of sets (of men and women) of unequal size [MW71].
- Algorithms of the kind of the last two examples, which generate all possible solutions to a problem, given certain constraints, are often used to select one or several of the solutions that are optimal in some sense.
  - In the present example, one might, for instance, be interested in the solution that on the average best satisfies the men, or the women, or everyone.
- The table in figure 5.4.5.b. appear in specific columns the sum of the ranks of all women in the preference lists of their husbands \((rm)\) and the sum of the ranks of all men in the preference lists of their wives \((rw)\), for each found solution.
- These values are calculated in accordance with the relations [5.4.5.1].

\[
\begin{align*}
\text{rm} := \sum_{m=1}^{n} RWM[m,x[m]] \\
\text{rw} := \sum_{m=1}^{n} RMW[x[m],m]
\end{align*}
\] [5.4.5.1]

<table>
<thead>
<tr>
<th></th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
<th>x6</th>
<th>x7</th>
<th>x8</th>
<th>rm</th>
<th>rw</th>
</tr>
</thead>
<tbody>
<tr>
<td>sol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>22</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>31</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>26</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>35</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>29</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>38</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>34</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>43</td>
<td>11</td>
</tr>
</tbody>
</table>

**Fig.5.4.5.b. Solutions of StableMarriage algorithm**

- The solution with the least value \(rm\) is called the male-optimal stable solution.
The solution with the smallest \( r_w \) is the **female-optimal stable solution**.

- Due to the nature of the *chosen search strategy* the **good solutions** from the **men's** point of view are **generated first** and the **good solutions** from the **women's** perspective appear toward **the end**. In this sense, the algorithm is based toward the male population.

- This can quickly be changed by systematically interchanging the role of men and women, i.e., by merely interchanging PrefWomen with PrefMen and interchanging \( RMW \) with \( RWM \).

- We refrain from extending this program further and leave the incorporation of a search for an **optimal solution** to the next and last example of a backtracking algorithm.

### 5.4.6 The Optimal Selection Problem

- The last example of a **backtracking algorithm** is a logical extension of the previous examples.

  - First we were using the principle of backtracking to find a **single solution** to a given problem of type \( (n, m) \). This was exemplified by the **knight's tour** and the **eight queens**.

  - Then we tackled the goal of finding **all solutions** to a given problem of type \( (n, m) \). The example was the **eight queens** generalization.

  - The next step was to find **all the solutions** affected by some **constraints** of a given problem \( (n, m) \), i.e. **stable marriage**.

- Now we wish to find an **optimal solution**.

  - To this end, the next strategy is used: **all possible solutions** are generated and in the course of generating them, is **retained** the one that is **optimal** in some specific sense.

- Assuming that **optimality** is defined in terms of some positive valued **function** \( f(s) \), the algorithm is derived from the general schema for finding **all the solutions** of a problem of type \( (n, m) \) [5.4.4.a], replacing the statement *display solution* by the statement [5.4.6.a].

\[
\text{IF } f(\text{solution}) > f(\text{optimum}) \text{ THEN optimum:= solution} \quad [5.4.6.a]
\]

- The variable **optimum** records the best solution so far encountered.

- Naturally, it has to be properly initialized; moreover, it is customary to record to value \( f(\text{optimum}) \) by another variable in order to avoid its frequent re-computation.

- An example of the **general problem** of finding an **optimal solution** to a given problem follows:

  - We choose the important and frequently encountered problem of finding an **optimal selection** out of a given **set of objects** subject to **constraints**.
• Selections that constitute acceptable solutions are gradually built up by investigating individual objects from the base set.

• The first refinement of procedure Try describes the process of investigating the suitability of one individual object [5.4.6.b].

• The procedure is called recursively (to investigate the next object) until all objects have been considered.

• We note that the consideration of each object, called candidate in previous examples, has two possible outcomes:
  
  • (1) Either the inclusion of the investigated object in the current selection,
  
  • (2) Or its exclusion from the current selection.

• This makes the use of a REPEAT or FOR statement inappropriate; instead, the two cases may as well be explicitly written out [5.4.6.b].

---

{The optimal selection problem – basic scheme of the algorithm}

PROCEDURE Try(i: TypeObject);
BEGIN
  IF *inclusion is acceptable THEN
    BEGIN
      *include i-th object;
      IF i<n THEN
        Try(i+1)
      ELSE
        *verify optimality;
        *eliminate i-th object
    END;
  IF *exclusion is acceptable THEN
    IF i<n THEN
      Try(i+1)
    ELSE /*[5.4.6.b]*/
      *verify optimality
  END;
{Try}

/* The optimal selection problem – basic scheme of the algorithm */

void try(type_object i);
{
  if (*inclusion is acceptable)
  {
    *include i-th object;
    if (i<n )
      try(i+1);
    else /*[5.4.6.b]*/
      *verify optimality;
      *eliminate i-th object
  }
if (*exclusion is acceptable)
  if ( i<n )
    try(i+1);
  else
    *verify optimality
} /*try*/
/*---------------------------------------------------------------*/

• We assume that the objects are numbered 1,2,...,n.
  • From this pattern it is evident that there are \(2^n\) possible sets.

• Clearly, appropriate acceptability criteria must be employed to reduce the number of investigated candidates very drastically.

• In order to elucidate this process, let us choose a concrete example for a selection problem with the following specification:

  • Consider a basic set consisting of \(n\) objects \(A_1,A_2,...,A_n\).

  • Let each of the \(n\) objects \(A_i\) be characterized by its weight \(w_i\) and its value \(v_i\).

  • Let the optimal set be the one with the largest sum of the values of its components.

  • Let the constraint be a limit on the sum of their weight.

• This is a problem well known to all travelers who pack suitcases by selecting from \(n\) items in such a way that:

  • (1) Their total value is optimal.

  • (2) Their total weight does not exceed a specific allowance.

• It is also known as knapsack problem.

• We are now in a position to decide upon the representation of the given facts in terms of global variables [5.4.6.c].

-------------------------------------------------------------------

{Optimal selection – defining data structures}

**TYPE**

  TypeIndex = 1..n;
  TypeObject = RECORD
    w,v: integer
  END;

**VAR**

  objects: ARRAY[TypIndex] OF TypeObject; [5.4.6.c]
  limw,totv,maxv: integer;
  s,opts: SET OF TypeIndex;

-------------------------------------------------------------------

/* Optimal selection – defining data structures */

enum {n=...};
typedef unsigned type_index;
typedef int *set;

typedef struct{
    int w,v;
}type_object;

type_object objects[n]; /*[5.4.6.c]*/
int limw,totv,maxv;
set s,opts;
/*---------------------------------------------------------------*/

• Variable limw denotes the **weight limit** imposed to selection.

• Variable totv denotes **total value** of all n objects.
  
  • These two values are actually **constant** during the entire selection process.

• s represents the **current selection** of objects in which each object is represented by its name (index).

• opts is the **optimal selection** so far encountered.

• maxv is its value of opts.

• Now we will specify the **acceptance criteria** for an object in the **current selection**.

  • (1) If we consider **inclusion**, then an object is **selectable**, if it fits into the **weight allowance**.
    
    • If it does not fit, we may stop trying to add further objects to the current selection.

  • (2) If we consider **exclusion**, then the **criterion for acceptability**, i.e., for the continuation of building up the current selection, is that the **total value** which is still achievable after this exclusion, is **not less** than the **value of the optimum** so far encountered.
    
    • Non including the current candidate means practically its exclusion from current selection.

    • In this case, the continuation of the search, although it may produce some solution, will **not** yield the **optimal solution**.

    • Hence **any further search** on the current path is **fruitless**.

• From these **two conditions** we determine the **relevant quantities** to be computed for each step in the selection process:

  • (1) The **total weight** $t_w$ of the selection $s$ so far made.
    
    • Variable $t_w$ is initialized with value 0.
- Each time an object is included in the current selection, its weight is added to tw.

- The condition *inclusion is acceptable it’s fulfilled if the new value of tw remains smaller than limw [5.4.6.d].

- (2) The still achievable value av of the current selection s.
  - Initial av takes the maximum possible totv obtained by summing the values of all n objects of the set.
  - At each exclusion, av is decremented by the value of the excluded object.
  - When the current selection is ended, av stores its value.
  - Condition *exclusion is acceptable is true if the new value of av remains greater than maxv, where vmax is the value of the current optimum solution [5.4.6.e].
    - The explanation is the following: even eliminating the current object from the actual selection, the potential value still achievable, is greater than the value of the current optimal solution. That means, the actual selection can still yield to an optimal solution.

- The entities tw and av are parameters of the procedure Try.

\begin{align*}
\text{refinement of statement *inclusion is acceptable} \\
\text{tw} + \text{objects}[i].w &\leq \text{limw} \quad [5.4.6.d] \\
\text{refinement of statement *exclusion is acceptable} \\
\text{av} - \text{objects}[i].v &> \text{maxv} \quad [5.4.6.e]
\end{align*}

- For coding reasons, the difference av-objects[i].v is stored in variable av1.

- *verify optimality and the registration of the optimal solution, if the case, is refined in [5.4.6.f].

\begin{align*}
\text{verify optimality} \quad \text{IF} \quad \text{av}>\text{maxv} \text{ THEN} \\
\quad \text{BEGIN} \quad \{\text{we have a new optimum which is registered}\} \\
\quad \quad \text{sopt}:= s; \text{maxv}:= \text{av} \quad [5.4.6.f] \\
\quad \text{END;}
\end{align*}

- The integral program OptimalSelection in Pascal and C variant appears in [5.4.6.g].
• In the Pascal variant, the inclusion and exclusion have an elegant implementation due to type `SET` and its specific operators defined in language.

• In the C variant, the set operators are implemented in an explicit manner in the form of defined type `set` and its associated operators `inset(int,set)`, `setof(int,...)`, `join(set,set)` and `difference(set,set)`.  

---

```plaintext
PROGRAM OptimalSelection;

CONST n=10;
TYPE TypeIndex = 1..n;
 TypeObject = RECORD
   w,v: integer
 END;

VAR i: TypeIndex;
 objects: ARRAY[TypeIndex] OF TypeObject;
 limw,totv,maxv: integer;
 w1,w2,ratio: integer;
 s,sopt: SET OF TypeIndex;
 z: ARRAY[boolean] OF char;               [5.4.6.g]
b: boolean;

PROCEDURE Try(i: TypeIndex; tw,av: integer);
VAR av1: integer;
 BEGIN {trying inclusion of object i}
   IF tw+objects[i].w<=limw THEN
     BEGIN
       s:= s+[i];
       IF i<n THEN
         Try(i+1,tw+objects[i].w,av)
       ELSE
         IF av>maxv THEN
           BEGIN
             maxv:= av;
             sopt:= s
           END;
         s:= s-[i]
     END;

   {trying exclusion of object i}
   av1:= av-objects[i].v;
   IF av1>maxv THEN
     BEGIN
       IF i<n THEN
         Try(i+1,tw,av1)
       ELSE
         BEGIN
           maxv:= av1;
           sopt:= s
         END
     END
   END; {Try}
```
BEGIN {main program}
totv:= 0;
FOR i:=1 TO n DO
  WITH objects[i] DO
    BEGIN
      Read(w,v);
      totv:= totv+v
    END;
  Read(w1,w2,ratio);
z[true]:= '*';
z[false]:= '  ';
Write('   Weight  ');
FOR i:=1 TO n DO Write('   ',objects[i].w);
Writeln;
Write('    Value  ');
FOR i:=1 TO n DO Write('  ',objects[i].v);
Writeln;
REPEAT
  limw:= w1;
  maxv:= 0;
  s:= [];
  sopt:= [];
  Try(1,0,totv);
  Write('         ',limg);
  FOR i:=1 TO n DO
    BEGIN
      b:= i IN sopt;
      Write('    ',z[b])
    END;
  Writeln;
  w1:= w1+ratio
UNTIL w1>w2
END.

/*--------------------------------------------------------*/
/* Optimal Selection */

#include <limits.h>
#include <stdarg.h>
#include <stdlib.h>
enum { n =10};
typedef unsigned char type_index;
typedef unsigned int boolean;
#define false (0)
#define true (1)
typedef int* set;
#define eos INT_MIN
typedef struct{
  int w,v;
} type_object;
type_index i;
type_object objects[n];
int limw,totv,maxv;
int w1,w2,ratio;
set s,sopt;
char z[2]; /*[5.4.6.g]*/
boolean b;

boolean inset(int, set);
set setof(int,...);
set join(set,set);
set difference(set,set);

void try(type_index i, int tw, int av)
{
    int av1;
    /* trying the inclusion of object i */
    if (tw+objects[i-1].w<=limw){
        s=join(s, setof(1,i));
        if (i<n)
            try(i+1,tw+objects[i-1].w,av);
        else
            if (av>maxv){
                maxv=av;
                sopt=s;
            }
        s=difference(s, setof(1,i));
    }
/* trying the exclusion of object i */
    av1=av-objects[i-1].v;
    if (av1>maxv){
        if (i<n)
            try(i+1,tw,av1);
        else{
            maxv=av1;
            sopt=s;
        }
    }
}
/*try*/

int main(int argc, const char* argv[])
{
    /*main program*/
    totv=0;
    for(i=i; i<=n; i++){
        type_object* with=&objects[i-1];
        scanf("%i%i", &with->w,&with->v);
        totv=totv+with->v;
    }
    scanf("%i%i%i", &g1,&g2,&ratio);
    z[true]='*';
    z[false]=" ";
    printf("  Weight ");
    for( i=1; i <=n; i++)
printf("%3i", objects[i-1].w);
printf("\n");
printf(" Value ");
for( i=1; i <=n; i++)
printf("%3i", objects[i-1].v);
printf("\n");
do {
  limw=w1;
  maxv=0;
  s=setof(0);
  sopt=setof(0);
  try(1,0,totv);
  printf("%3i", limw);
  for( i=1; i<=n; i++){
    b=inset(i, sopt);
    printf("%4c", z[b]);
  }
  printf("\n");
  w1=w1+ratio;
} while (!(w1>w2));
return 0;

/*---------Implementation of set operators----------------------*/

boolean inset(int x, set s)
{
  int i=0;
  for (i=1; i<=s[0]; i++)
    if (s[i] ==x)
      return true;
  return false;
}

set setof(int x,...)
{
  va_list ap;
  set s;
  int i=0,j=0,t,f;
  if ((s=malloc((x+1)*sizeof(int)))==NULL){
    printf("eroare!");
    exit(1);
  }
  s[0]=0;
  if (x){
    va_start(ap,x);
    for (i=1; i<=x; i++){
      t=va_arg(ap,int);
      f=1;
      for (j=1; j<i; j++)
        if ( t ==s[j])
          f=0;
      if (f){
        s[i] =t;
      }
    }
  }
}


```c
    s[0]++;
} }
va_end(ap);
} return s;
}

set join(set s1, set s2) {
    int i=1;
    int j=1;
    set s;
    s=realloc(NULL, (s1[0]+s2[0]+1)*sizeof(int));
    s[0]=0;
    for (i =1; i<=s1[0]; i++) {
        s[i]=s1[i];
        s[0]++;
    }
    for (j =1; j<=s2[0]; j++)
        if ( inset(s2[j],s) ==false ) {
            s[i++]=s2[j];
            s[0]++;
        }
    return s;
}

set difference(set s1, set s2) {
    set s;
    int i;
    int j;
    s=realloc(NULL, (s1[0]+s2[0]+1)*sizeof(int));
    for (i=0; i<=s1[0]; i++)
        s[i]=s1[i];
    for (j =1; j<s2[0]; j++)
        if (!inset(s2[j],s)) {
            s[0]++;
            s[i++]=s2[j];
        }
    return s;
}
/*-----------------------------------------------*/

o In figure 5.4.6.c are listed the results of the optimal selection algorithm for the specified objects, with weight allowances ranging from 10 to 120 with ratio 10.
```
Fig.5.4.6.c. The results of the **OptimalSelection** program execution

- The **algorithms** based on this **backtracking scheme** with a **limitation factor** curtailing the growth of the potential search tree is also known as **branch and bound algorithms**.

### 5.5 Recursive Data Structures

#### 5.5.1 Static and Dynamic Data Structures

- In Chapter 1 we have presented the fundamental data structures respectively the **unstructured types** enumeration and **predefined (standard) primitive types**, as well as the **structured data types** array, record, union and sequence.

- They are called **fundamental** because:
  
  - They constitute the **building blocks** out of which more complex structures are formed.
  
  - In practice they do occur most frequently.

- The purposes of **defining a data type**, and of thereafter specifying that certain **variables** be of that type, are:
  
  - (1) To fix the **range of values** assumed by these variables.
  
  - (2) To specify the **structure**, **dimension** and **location in memory** of their storage patterns.
  
  - (3) To specify the **operators** which can operate with these variables.

- All these characteristics are fixed once and for all inside the current program.
• Hence, variables declared in this way are said to be static because all the underlined associate characteristics are established by the compiler at the very beginning and can’t be changed except their values.

• However, there are many problems which involve far more complicated information structures.

• The characteristic of these problems is that not only the values but also the structures of variables change during the computation.

• They are therefore called dynamic structures.

• Naturally, the components of such structures are, at some level of resolution, static, i.e., of one of the fundamental data types.

• Generally speaking, in any programming language, a static data structure occupies during all the existence period of the program to which it belongs, a fix memory area, at a fix address, with a constant volume.

• The memory necessary for a static structure is allocated during compilation, before the execution of the program.

• Some remarks are necessary:

  • (1) Are considered static those data structures which has the memory volume effectively constant.

  • (2) Are considered static too, those structures with variable memory volume, for which the programmer can estimate an upper limit for the dimension, and for which the volume of memory is statically allocated at compilation in accordance with this limit.

• In general, any variable declared in the variable declarations area, excepting those of type sequence, represents a static structure, for which the compiler allocates a specific constant volume of memory.

• By declaring a variable, the programmer specifies the name of the variable and its type.

  • The variable name is an identifier used to refer the variable – known as logical access.

  • The variable name has associated the address of the static memory area allocated to the variable – physical access.

  • The dimension of the allocated memory area is related to its type.

• By contrast, for a dynamic data structure, the necessary memory volume can’t be known in the compilation phase, because it depends on factors which are known only at the execution time.

• In other words, the memory volume can increase or decrease in dependence of factors determined during execution.
• As result, the memory allocation for a dynamic structure is realized during the execution of the program.

• Until now, the only dynamic structures we tackled were of type sequence.
  • Because usually sequences are stored in external memories, they are considered special dynamic structures and they are out of our interest at this moment.

• As we have mentioned, in fact the dynamic structures, as well as static structures are composed in last instance, of constant volume components, belonging to the fundamental data types.
  • Usually, for the dynamic structures this components are named nodes.

• The dynamic nature of these structures resides in the fact that the nodes number is variable during the program execution.

• The dynamic structures and their individual nodes, differ from static structures, by the fact that they are not properly declared.
  • As result they can’t be referred by name, because in fact they have no name.

• To refer such structures, a special type of static variable named indicators (pointers) are used.
  • The variables referred in this manner are named indicated variables.

5.5.2 Abstract Data Type Indicator

5.5.2.1 Definition of ADT Indicator

• The using of dynamic data structures beside other special applications impose the definition of a special data type named indicator.
  • The associated variables are named indicator variables.
  • The values of this variables are not effective data, but they determine memory locations which in their turn, store effective data.

• In other words, the value of a such indicator variable represents a reference to a variable of a specified type, named indicated variable.

• In order to manage this type of variable a new abstract data type named indicator was introduced.

• The principle description of this type appears in [5.5.2.a].

ADT Indicator

Mathematical model:
Consists of a set of values which indicate the memory addresses of some variables belonging to a specified type,
named indicated variables. The set includes the empty indicator pointing none variable.

Notations:

\[ p, q \] - variables of IndicatorType;  
\[ e \] - variable of IndicatedType;  
\[ b \] - Boolean value.  

Operators:

\textbf{New} (\( p \)) - operator which allocates memory for an indicated variable and places the value of the allocated memory address in variable \( p \) (the indicator);

\textbf{Dispose} (\( p \)) - operator which dispose the memory area (location) allocated to variable indicated by \( p \);

\textbf{StoreIndicator} (\( p,q \)) - copies indicator \( p \) in \( q \);

\textbf{StoreIndicatedValue} (\( p,e \)) - copies the value of \( e \) in the location indicated by \( p \);

\( e := \textbf{RetreiveIndicatedValue} (p) \) - function which returns the value stored in location indicated by \( p \);

\( b := \textbf{IdenticalIndicator} (p,q) \) - Boolean function which returns true if \( p \equiv q \). That means the two variables indicate the same memory location.

---

- The \textbf{abstract data type indicator} can be implemented in different manners.
- We present three of this: one based on \textbf{pointers}, the second on \textbf{cursors} and the third on \textbf{references}.

5.5.2.2 Implementation of ADT Indicator Using Pointers

- As we have mentioned, \textbf{pointer variables} are usual \textbf{static variables} which are declared as any other type of variable.
  - The particularity of this kind of variables is they are \textit{declared} of \textbf{type pointer}.
- What means a variable of \textbf{type pointer}?
  - The \textbf{value} of a such variable is in fact a \textbf{memory address}, which can point a \textbf{dynamic structure} or a \textbf{component} of a such structure.
  - In consequence, at the \textbf{assembler level}, the access to an indicated variable pointed by a indicator variable is achieved by \textbf{indirect addressing} of the last.
• In the **high level languages**, the same thing is achieved by attaching some special characters to the name of the **pointer variable**, depending on the used language [5.5.2.2.a].

\[
\text{IndicatedVariable} \equiv \text{PointerVariable}^\wedge \quad [5.5.2.2.a]
\]

// **Accessing an indicate variable – C variant**

\[
\text{IndicatedVariable} \equiv *\text{PointerVariable} \quad [5.5.2.2.a]
\]

• In addition, in C language was defined the operator `&` which supply the **address** of an **indicated variable**. In the same time in C was developed a **specific pointers’ arithmetic**.

• A very important rule stipulates that, in contrast with the **assembling languages**, in the **high level languages** a **rigid link** is established between any **pointer variable** and the type of **indicated variable**.

  • A declared **pointer variable** refers in a compulsory mode to an **indicated variable** of a certain **specified type**.

  • As result, when a **pointer type** is declared, the **type of the indicated variable** is also specified.

• This restriction increases the **reliability** in programming and in the same time constitutes an evident difference between the **pointer variables** defined in the **high level programming languages** and the **usual addresses** in the **assembling languages**.

• A **type pointer** is defined specifying the type of the indicated variable preceded by the character "\^\" in Pascal respectively the character "\*\" in C [5.5.2.2.b].

\[
\text{TYPE} \quad \text{PointerType} = ^\wedge\text{IndicatedType}; \quad \text{(Pascal)} \quad [5.5.2.2.b]
\]

// **Defining a pointer type – C variant**

\[
\text{indicated_type} \quad *\text{pointer_variable}; \quad \text{(C)} \quad \text{//[5.5.2.2.b]}
\]

• In C this restriction can be avoided using the generic type **void** which allows the declaration of a **generic pointer** which is **not** associated with a specified data type.

\[
\text{void} \quad *\text{pointer_variable}; \quad [5.5.2.2.c]
\]
• The allocation and the freeing of memory of a dynamic structure during the execution, is the programmer task and is achieved by means of some standard operators which depend on the programming language.

• In [5.5.2.2.d] and [5.5.2.2.e] are presented the implementations of the ADT Indicator in Pascal respectively C.
  
  • Specific standard operators defined at the language level are used.

-------------------------------------------------------------------
{ADT Indicator - Pascal implementation}

TYPE PointerType = ^IndicatedType;
VAR p,q: PointerType;
e: IndicatedType; b: boolean;

New(p);         {New(p)}
Dispose(p);     {Dispose(p)}
p:= q;          {StoreIndicator(p,q)}
p^:= e;         {StoreIndicatedValue(p,e)}
e:= p^;         {e:=RetrieveIndicatedValue(p)}
b:= p=q;        {b:=IdenticalIndicator(p,q)}
-------------------------------------------------------------------
// ADT Indicator - C implementation

indicated_type *p,*q, e;
int b;                [5.5.2.2.e]

p=malloc(sizeof(indicated_type));  {New(p)}
free(p);       {Dispose(p)}
p= q;          {StoreIndicator(p,q)}
*p= e;         {StoreIndicatedValue(p,e)}
e= *p;         {e:=RetrieveIndicatedValue(p)}
b=(p==q);      {b:=IdenticalIndicator(p,q)}
-------------------------------------------------------------------

5.5.2.3 Implementation of ADT Indicator Using Cursors

• Some languages, such as Fortran and Algol, do not have pointers.

• If we are working with such a language, we can simulate pointers with cursors, that is, with integers that indicate positions in arrays.

• A cursor is an integer variable used to indicate a location inside an array of indicated variables.
  
  • As a method, a cursor is perfect equivalent with a pointer, but it can be used in languages which do not define the type indicator.

  • Interpreting the value of an integer variable as an index in an array, in an effectively manner, this variable indicates the respective location in the array.
• The indicated variable is stored in the corresponding location in the array.

• Using this technique, can be practically implemented all the data structures presuming links, as we will see in the next chapters.

• It’s obvious, that the task of managing the memory area which is allocated dynamically, is achieved exclusively by the programmer. In other words, the programmer must define its own operators of type allocate-free or new-dispose.

• In the same time is the programmer task to statically reserve the required memory space for the array used as support for dynamic allocation of memory.

• Such an example is presented in the next chapter and refers to list structure.

5.5.2.4 Implementation of ADT Indicator Using References

• Object oriented programming has imposed new developments of the possibilities to access the elements used by a programmer inside a program.

• Thus, the object oriented languages as C++ or JAVA define the so-called references which are in fact an implementation form of type indicator.

  • A reference appears as being similar in many aspects with a pointer, but in fact it isn’t a pointer.

  • A reference is an alias for a specified variable.

    • A reference is in fact, a name (an identifier) which can be used instead of the original variable.

    • Because a reference is an alias and not a pointer, the variable for which the reference is declared, must be specified at the moment of reference declaration.

    • In addition, unlike a pointer, a reference can’t be modified to refer another variable.

• Example 5.5.2.4. Consider in C++ a variable declared as follows [5.5.2.4.a]:

```cpp
long number = 0;                         [5.5.2.4.a]
```

• A reference for this variable can be declared using the next construction [5.5.2.4.b]:

```cpp
long& ref_number = number; //reference declaration [5.5.2.4.b]
```

• In this case we can say that ref_number is a reference to the variable number of type long.
• The reference can be subsequently used instead of the original variable. For example the assignment [5.5.2.4.c]

```
ref_number += 13;                        [5.5.2.4.c]
```

• Has as effect incrementing the variable number value, by 13.

• Opposed to the above defined reference, a pointer to the same variable can be declared as [5.5.2.4.d]:

```
long* pointer_number = &number;           [5.5.2.4.d]
```

• Thus permits to increment the variable number as follows [5.5.2.4.e]:

```
*pointer_number += 13;                   [5.5.2.4.e]
```

• There is an obvious difference between using a pointer and a reference.

  • The pointer is evaluated at each utilization, and in accordance with its current contained address, is accessed the indicated variable.

  • The reference isn’t evaluated at each utilization. Once defined, it can’t be changed.

  • A reference is perfectly equivalent with the referred variable.

At a summary analysis, the reference seems to be only an alternative notation for a certain variable.

• In reality the true valences of this concept are emphasized in the case of using objects.

  • Thus, in the object oriented programming languages, when objects are assigned to variables or are send as parameters for methods, only the references of this objects are used, and not the objects themselves or their copies.

  • This has a very particular importance in the process of developing object oriented applications.

  • For example in JAVA doesn’t exist pointers nor so called pointer’s arithmetic, only references. This fact doesn’t restraint the programming freedom or the developing possibilities, on contrary.
5.5.3 Recursive Data Structures

• We have discussed until now about recursion as a property of algorithms, implemented in form of some procedures or functions which call themselves.

• It’s the time now to extend this property over the data types in the form of so called recursive data structures.

  • By analogy with algorithms, by a recursive data structure we will understand a structure which has at least a component of the same type as the structure itself.

• In this case too, we can have for such data types, direct or indirect recursion (see §5.1).

• A simple example of recursive data structure is linked list, formally defined in [5.5.3.a].

{Example 1 of recursive data structure - structure linked list}

```plaintext
TYPE TypeList = RECORD
  info: TypeInfo;  [5.5.3.a]
  next: TypeList   (incorrect)
END;
```

/* Example 1 of recursive data structure - structure linked list */

typedef struct {
  type_info info;       /*[5.5.3.a]*/
  type_list next;       /*incorrect*/
} type_list;

/* Example 2 of recursive data structure - structure family pedigree */

{Example 2 of recursive data structure - structure family pedigree}

```plaintext
TYPE TypePedigree = RECORD
  name: string;                    [5.5.3.b]
  father, mother: TypePedigree     (incorrect)
END;
```

/* Example 2 of recursive data structure - structure family pedigree */

typedef struct {
  char* name;                      /*[5.5.3.b]*/
```

• A second example of a recursive information structure is the family pedigree.

  • Let a pedigree be defined by the name of a person and the two pedigrees of the parents.

  • This definition leads inevitably to an infinite structure.

• This can be formally expressed as [5.5.3.b]:

{Example 2 of recursive data structure - structure family pedigree}
In the above examples, due to fields next, respectively father and mother, of the same type as the structure itself, the defined data types are recursive.

We have to mention that the above definitions are incorrect. In the both cases, the identifier TypeList respectively TypePedigree are used before their definitions have been completed.

- There is an important programming rule: a type can be used only if it’s definition has been completed.

- This aspect will be tackled soon.

As we have already mentioned, the two types define infinite structures.

In reality neither the lists nor the family pedigree are infinite.

- Real pedigrees for example, are bounded because at some level of ancestry information is missing.

Taken into account this aspect, the TypePedigree will be modified as follows:

- If the person is unknown, the structure will contain a single field noted as 'XX'.

- In any other case, the structure contains the three mentioned fields.

In figure 5.5.3.a is represented a structure of TypePedigree which respects the modified definition.

![Fig.5.5.3.a. An example of recursive structure](image-url)
• With respect to **recursive data structures**, in order to avoid **infinite structures**, the **recursive type definition** must contain a **condition** which regulates the presence or the absence of the component(s).

  • As in above example, under certain circumstances, the components having the same type as the structure itself, can miss.

  • Under this circumstances, there are **finite recursive structures**.

• The **recursive structures** can be implemented in advanced programming languages **only** as **dynamic structures**.

  • By definition, any **component** having the type identical with the **whole structure**, is **replaced** with a **pointer** which indicates this component.

• In [5.5.3.c,d] appear the correct definitions for the recursive structure of TypePedigree in Pascal respectively C variants.

```
{Recursive data structure - structure family pedigree - Pascal variant}

TYPE PointerPedigree = ^TypePedigree;
  TypePedigree = RECORD
    name: string;                [5.5.3.c]
    father,mother:PointerPedigree
  END;

// Recursive data structure - structure family pedigree - C variant

struct pedigree {
  char *name;                     [5.5.3.d]
  struct pedigree *father, *mother;
}
```

• As we have already mentioned, under certain conditions, a component of a recursive structure can miss.

• For this purpose, the set of values for any **reference type** is extended with **null reference** (null pointer) which indicates no variable.

  • This pointer is noted as **NIL** respectively **NULL**, and it belongs by **definition** to any **indicator type**.

  • The **absence** of a **recursive component** will be indicated by assigning the **pointer** to this component with the **null reference**.

• With these specifications, the recursive structure from figure 5.5.3.a can be figure 5.5.3.b.
An other example of recursive structures are arithmetic expressions.

In this case, recursion is used to reflect the possibility of nesting, i.e., of using parenthesized arithmetic expressions as operands in expressions.

In the following example, we will consider the simple case when by arithmetic expression we will understand either:

- A simple operand represented as an identifier consisting of one single letter.
- An arithmetic expression (operand), followed by an operator, followed by an arithmetic expression (operand).

Such a structure can be defined as in [5.5.3.e].

---

**Recursive data structure – arithmetic expression – Pascal variant**

```
TYPE PointerExpression= ^TypeExpression;
TypeExpression = RECORD
[5.5.3.e]
    Operator: CHAR;
    Operand1,Operand2: PointerExpression
END;
```

---

**Recursive data structure – arithmetic expression – C variant */

```
typedef struct type_expression* pointer_expression;
typedef struct { /*[5.5.3.e]*/
    char operator_;
    pointer_expression operand1,operand2;
}type_expression;
```

---

In [5.5.3.e] we consider that:
• If the field operator stores one of the characters '+' , '-' , '*' or ' / ', signifying an arithmetic operation, the other two fields are pointers to other arithmetic expressions.

• If the field operator stores a letter, signifying an arithmetic operand, the other two fields store the NIL value.

• We have shown in &5.2.4, that any recursive algorithm can be replaced by an equivalent iterative one.

• This propriety is valuable for the recursive data structures too.

  • Any recursive structure can be replaced with an iterative structure of type sequence.

• Thus, a recursive data structure defined as in [5.5.3.f]:

```plaintext
{Recursive data structure}

TYPE StructureIndicator= ^TypeStructure
  TypeStructure = RECORD
    content: TypeContent;
    link: StructureIndicator
  END;
```

• Is equivalent with the sequential type [5.5.3.g]:

```plaintext
{Iterative variant of the recursive data structure [5.5.3.f]}

TYPE TypeStructure = FILE OF TypeContent [5.5.3.g]
```

• We underline that a recursive structure can be immediately replaced by a sequential one, if and only if, the name of the recursive field appears in the self recursive definition, only once, at its end (beginning).

  • The observation is similar with those for tail recursive procedures or functions (see &5.2.4).

• If the recursive structure has a higher degree of complexity, (usually based on different order trees), the transformation in a sequential structure can be achieved by using a specific manner of traversal (in, pre or post order for example).