Cuprins

0.INTRODUCERE

Recurenţa este prezentă în lucrările de matematică destul de timpuriu, cel mai cunoscut exemplu fiind **numerele Fibonacci** care formează o secvenţă de numere întregi. Această secvenţă respectă o anumită regulă **recurentă:** fiecare număr este suma precedentelor sale două numere. Excepţia de la această regulă o reprezintă numerele de început ale secvenţei (0 şi 1):

0, 1, 1, 2, 3, 5, 8, 13, 21 34, 55, 89, 144,…

Regula recurentă este:

$\mathbf{F_n=F_{n-1}+F_{n-2},\ n>1}$ cu $\mathbf{F_0 = 1}$ şi $\mathbf{F_1 = 1}$

Secvenţa a fost numită după numele italianului Leonardo din Pisa, cunoscut şi sub pseudonimul Fibonacci, în lucrarea "Liber Abaci" apărută în 1202.

În anii următori, s-au dat apoi, în matematică şi fizică, nenumărate şiruri şi serii de numere folosind definiţii recurente, atât liniare cât şi de altă natură.

Odată cu apariţia limbajelor de programare de nivel înalt, **FORTRAN** fiind primul limbaj de nivel înalt şi dedicat rezolvării problemelor cu caracter ştiinţific, s-a pus problema folosirii acestor recurenţe. În programare noţiunea a suferit o mică modificare morfologică şi semantică: noţiunea se numeşte **recursivitate** şi înseamnă apelul unei subrutine (proceduri sau funcţii) din corpul ei de definiţie, deci **autoapelare.** Limbajul aminitit mai sus nu permitea recursivitatea, dar următorul limbaj **ALGOL** pe aceaşi linie cu **FORTRAN**-ul, a permis folosirea acestei modalităţi de programare. Folosirea recursivităţii în programare este accesibilă, în zilele noastre, în majoritatea limbajelor folosite: **PASCAL**, **C++**, **JAVA**, **PYTHON**, **GROVY**, **LISP**, **PROLOG**, etc. În programa liceală, la clasele de profil se folosesc limbajele **PASCAL** şi/sau **C/C++,** de aceea lucrarea va exemplifica algoritmii recursivi doar în aceste două limbaje de programare.

În matematică întâlnim multe alte exemple care se pot defini recursiv:

- ***şirul numerelor naturale*** (primul este 0, fiecare număr obţinându-se din precedentul său la care se adună 1);
- ***suma numerelor naturale*** (iniţial **suma** este zero, apoi suma (n) este suma primelor n-1 adunate cu **n**);
- ***cmmdc*** a două numere naturale atât prin scădere cât şi prin împărţire, vom reveni la acest exemplu;
- demonstrarea prin metoda inducţiei matematice a unor propoziţii matematice foloseşte de cele mai multe ori la pasul de demonstrare forme de recurenţă;
- etc.

Există şi fenomene de altă natură care se pot defini recursiv:

- succesiunea zilelor şi nopţilor;
- construcţia unor asambluri pe module;
- filmarea cu o camera a unei oglinzi; sau televizarea unei emisiuni cu filmarea unui televizor ce transmite aceea emisiune;
- etc.

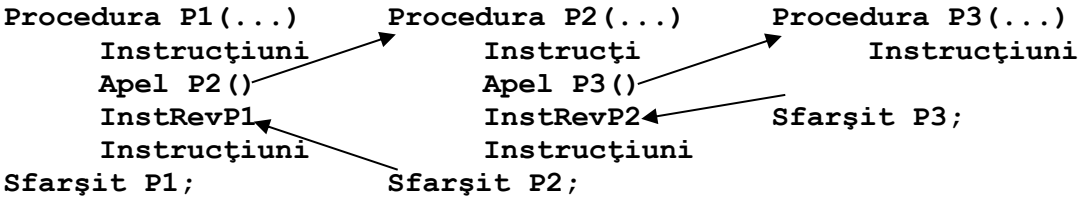
Primul capitol, va aborda şi sper că va clarifica noţiunea de recursivitate. Se vor da nenumărate exemple în **PASCAL** şi **C++,** nu numai din matematică dar şi probleme pe şiruri de caractere, de citiri/afişari de date, etc. Se vor da şi clasificări din diverse puncte de vedere, se va vorbi şi despre **stivă,** structura de dată logică care permite acest mecanism al recursivităţii. Programul apelant este şi program apelat, stiva trebuie gestionată foarte exact; la revenire se preia nu numai contextul de apel ci şi rezultatele intermediare, stocate de asemenea în stivă.

Al 2-lea capitol se va referi la metodologia predării mecanismului de recursivitate, trecerea de la algoritmii iterativi la cei recursivi, nefiind banală; dificultăţi în întelegere, exemple simple, lecţii cadru de predare, exerciţii şi probleme din tematica probei de informatică de la Bacalaureat, etc.

1. RECURSIVITATEA ÎN PROGRAMARE

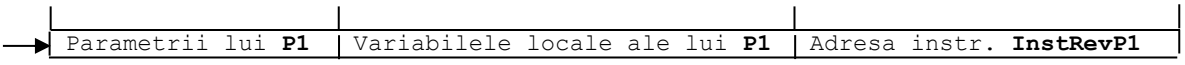
1.1. Motivaţia folosirii recursivităţii

Idea subprogramului (procedură sau funcţie) care apelează un alt subprogram este deja obişnuită în limbajele de programare. Majoritatea paradigmelor de dezvoltare a softului includ rafinarea problemei în subprobleme până când acestea din urmă se pot concretiza în module, proceduri sau funcţii. Apoi apelul acestor module, proceduri sau funcţii se face în mod natural în procesul rezolvării problemei iniţiale. Deci mecanismul apelării de subprograme este folosit în toate limbajele de programare moderne, la baza lui stând conceptul de stivă. Implementarea specificaţiilor unui limbaj de programare se face de obicei de compilatorul limbajului respectiv. Fără a intra în prea multe amănunte, stiva se defineşte ca o lista liniară care are principiul **LIFO** (ultimul element intrat în stivă este primul utilizat şi apoi scos din stivă). În stiva folosită de compilator se depune contextul apelului, în care pe lângă parametri de apel, variabilele locale ale subprogramului, etc. se depune şi adresa de revenire. Dacă se fac mai multe apeluri în cascadă, atunci se reţin în stivă adresele de revenire. La terminrea apelurilor se revine la fiecare adresă din stivă în mod invers depunerii în stivă, etc. Acest mecanism este ilustrat în cele ce urmează. Fie 3(trei) proceduri numite **P1**, **P2** şi **P3** şi secvenţa de pseudocod de mai jos:

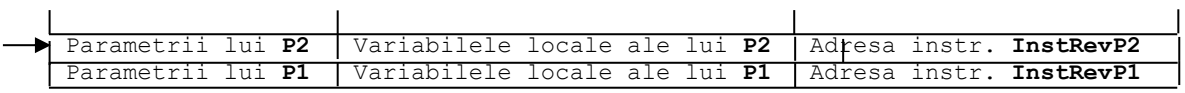


Prin **InstRevP1** s-a notat instrucţiunea de revenire din **P2**, iar prin **InstRevP2** s-a notat instrucţiunea de revenire din **P3**.

La apelul procedurii **P2** din **P1**, stiva are structura:



La apelul procedurii **P3** din **P2**, stiva se modifică astfel:



La revenirea din **P3** (în **P2**) stiva îşi reface prima formă, apoi la revenirea din **P2** (în **P1**) stiva se goleşte complet.

Fără a intra în alte amănunte amintim că un compilator al unui limbaj de programare foloseşte acest mecanism de stivă şi în alte procese de traducere a programului în limbaj maşină.

S-a pus problema, dacă un subprogram nu s-ar putea apela pe el însuşi, din corpul lui de definiţie? Raspunsul este pozitiv, şi pe parcursul dezvoltării a noi limbaje de programare

s-au implementat mecanismele de recursivitate. S-a luat măsura ca starea subprogramului din activitatea anterioară autoapelului să poată fi restaurată la terminarea apelului. Acesta este principala caracteristică prin care limbajelele recursive se deosebesc de cele nerecursive.

După cum se ştie unele procese matematice, fizice şi de altă natură se pot defini recurent, cu formule recurenţiale elegante. Să revenim la numerele lui Fibonacci, cu formula:

$$F_n = F_{n-1} + F_{n-2},\ n>1 \text{ cu } F_0 = 0 \text{ şi } F_1 = 1$$

În [1], se dă o formulă nerecurenţială pentru termenul F_n folosind funcţiile generatoare, se obţine formula:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \qquad (1)$$

Pentru a scrie o procedură **PASCAL**/**C++**, care determină termenul de ordinul **n**, al şirului lui Fibonacci, după formula (1), avem nevoie de calculul puterilor celor două numere iraţionale. În continuare dăm cele două programe:

Varianta **PASCAL:**

```
function Putere(a:real; n:integer):real;        {determina a la puterea n}
var Put:real;
    i  :integer;
begin
    Put:=1;
    for i:=1 to n do
      Put:=Put*a;
    Putere:=Put;
end;
function FibN(n:integer):longInt;          {determina F(n)                    }
var  phi1,phi2       : real;               {cele 2 constante din formula (1)  }
     phi1LaN,phi2LaN : real;               {constantele la puterea n          }
begin
     phi1:=(1+sqrt(5))/2;
     phi2:=(1-sqrt(5))/2;
     phi1LaN:=exp(n*ln(phi1));             {phi1>0 si se poate calcula puterea}
     phi2LaN:=Putere(phi2,n);              {phi2<0 si este necesar apelul     }
                                           {functiei Putere                   }
     FibN:=trunc((phi1LaN-phi2LaN)/sqrt(5));{formula (1)                      }
end;
var i:integer;
begin
   writeln;
   for i:=0 to 10 do  write(FibN(i),' ');
end.
```

Execuţia programului va tipări primele 11 numere Fibonacci:

```
0 1 1 2 3 5 8 13 21 34 55
```

Ca o observaţie să remarcăm faptul că mai avem nevoie şi de funcţia **Putere(x,n)** pentru calculul x^n. În varianta **C/C++** nu e necesar să se definească o astfel de functie care sa calculeze x^n, pentru că există o funcţie predefinită în fişierul header **math.h**. Funcţia se numeşte **pow** şi are prototipul:

double pow (double x,double y) - calculează x^y;

Varianta **C/C++:**

```
#include<iostream.h>
#include<math.h>
long FibN(int n)                      // determina F(n)
{ float     phi1= (1+sqrt(5.0))/2,  // cele 2 constante din formula (1)
            phi2= (1-sqrt(5.0))/2,
            phi1LaN, phi2LaN;       // constantele la puterea n
  phi1LaN= pow(phi1,n);
  phi2LaN= pow(phi2,n);
  return (phi1LaN-phi2LaN)/sqrt(5.0);     // formula (1)
}
void main()
{ int i;
  for (i=0;i<=10;i++)
    cout<<FibN(i)<<" ";
}
```

Execuţia programului va tipări primele 11 numere Fibonacci:

```
0 1 1 2 3 5 8 13 21 34 55
```

După cum se observă calculul termenului **F(n)** necesită expresii destul de complexe, calcule cu ridicări la putere, etc. Variantele recursive (folosind recurenţa definirii şirului lui Fibonacci) în **PASCAL** şi **C/C++,** care se dau în cele ce urmează, sunt mai simple.

Varianta recursivă în **PASCAL:**

```
function fibNRec (n:longint): longint;
begin
 if n in [0,1]
         then fibNRec:= n                          {se returneaza 0 sau 1 }
         else fibNRec:= fibNRec(n-2)+fibNRec(n-1){apelul dublu recursiv }
end;
var i:integer;
begin
 for i:=0 to 10 do
    write (' ', fibNRec(i));
end.
```

Execuţia programului va tipări primele 11 numere Fibonacci:

```
0 1 1 2 3 5 8 13 21 34 55
```

Varianta **C/C++:**

```
#include<iostream.h>
long fibNRec (long n)
{ if (n<2)  return n;                             //se returnează 0 sau 1
            return fibNRec(n-1) + fibNRec(n-2); //apelul dublu recursiv
}
void main (void)
{ int i;
  for (i=0;i<=10;i++)
    cout <<" "<<fibNRec(i);
}
```

Execuţia programului va tipări primele 11 numere Fibonacci:

```
0 1 1 2 3 5 8 13 21 34 55
```

Comparând variantele iterative cu cele recursive, se observă că cele din urmă sunt mai simple şi mai elegante. Evident că există şi variante iterative mai simple, cu determinarea secvenţei $F_n, n \geq 0$ într-un tablou, dar atunci, trebuie declarat tabloul de o dimensiune aprioric destul de mare; sau folosirea a 2(două) variabile, **a** şi **b** care reţin doi termeni consecutivi şi apoi calcularea sumei celor două variabile urmată de deplasarea valorilor, ceva de genul:

```
a←0                          {primul termen F(0)          }
Afiseaza (a," ");            {afisare F(0)                }
b←1                          {al 2-lea termen F(1)        }
Afiseaza (b," ");            {afisare F(1)                }
Pentru i←2,n executa
      Suma←a+b               {determinare termen F(i)     }
      Afiseaza (Suma," ");   {afisare F(i)                }
      a←b                    {deplasare valori            }
      b←Suma
SfPentru
```

În varianta recursivă se observă două aspecte importante:

-definirea unei ieşiri în procedura recursivă (când se calculează direct F_n);

-apelul recursiv la sfârşitul codului procedurii (de obicei);

Cele două aspecte sunt recomandate (primul obligatoriu, altfel am avea un apel recursiv infinit), de specialiştii Kormen, Leiserson, Rivest în [2].

O altă observaţie importantă este că trebuie gândită recursiv rezolvarea şi nu traducerea iterativă într-o variantă recursivă.

1.2. Clasificarea apelurilor recursive, exemple

Clasificarea apelurilor recursive se poate face din diverse puncte de vedere: tipul expresiei recurente, numărul de apeluri într-o expresie, apel intermediar recursiv.

Dacă formula recurenţială este liniară atunci avem **recursivitate liniară**, în caz contrar avem **recursivitate neliniară**. În ambele tipuri de mai sus putem avea apel **simplu recursiv** (când în formulă apare recurenţa o singură dată, recurenţa Fibonacci este dublu liniară) cât şi apel **multiplu recursiv**.

Recurenţa simplă liniară este ilustrată de calculul factorialului **`n!=1*2*…*n`**.

Formula recurenţială este:

$$factorial(n) = \begin{cases} 1 & pentru \quad n = 0 \\ n * factorial(n-1) & pentru \quad n > 0 \end{cases}$$

Varianta **`PASCAL`**:

```
function Factorial (n:byte): longint;
begin
      if n=0       then Factorial:=1
                   else Factorial:=n*Factorial(n-1);
end;
```

Varianta **`C/C++`**:

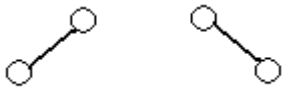
```
long Factorial(int n)
  { if (n==0)      return 1;
                   return n*Factorial(n-1);
  }
```

Vom ilustra recurenţa neliniară şi multiplă printr-un exemplu mai complex: calculul numărului de arbori binari cu **n** noduri notat cu b_n.

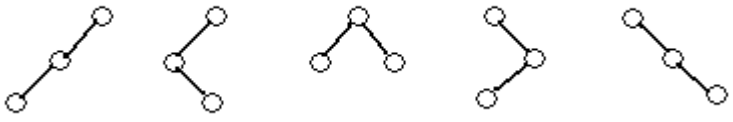
n=0, $\Rightarrow b_0=1$, prin convenţie se numără şi arborele binar vid, care nu are nici un nod.

n=1, $\Rightarrow b_1=1$, avem un arbore binar cu un nod, nodul rădăcină

n=2, $\Rightarrow b_2=2$, avem 2 arbori binari, conform schiţei:



n=3, $\Rightarrow b_3=5$, avem 5 arbori binari, ca în cele ce urmează:



Vom demonstra că:

$$b_n = \frac{1}{n+1} C_{2n}^{n} = \frac{1}{n+1}\binom{2n}{n}$$

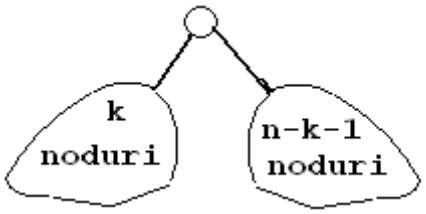
Vom folosi funcţia generatoare, conform [1]:

$B(z)=b_0+b_1z+b_2z^2+...+b_nz^n+...$

Funcţia generatoare este un polinom de grad infinit, având drept coeficienţi, chiar termenii şirului, b_n.
Presupunem că un arbore binar cu **n** noduri are:

un nod drept rădăcină
k noduri în subarborele stâng
n-k-1 noduri în subarborele drept

Figura următoare ilustrează acest fapt:



Dând lui **k** valorile **0,1,2,...,n-1** se obţine formula de recurenţă ce urmează:

$$b_n=b_0b_{n-1}+b_1b_{n-2}+...+b_{n-1}b_0$$

Este o recurenţa multiplă şi pătratică (neliniară). Se poate scrie un program recursiv, după această formulă, dar nu e recomandabil, pentru că apelul recursiv ar fi extrem de mult utilizat şi stiva aferentă se va depăşi repede.
În acest caz rezolvarea este matematică în primul rând şi e dată în continuare.

$B(z)=b_0+b_1z+b_2z^2+...+b_nz^n+...$
$B(z)=b_0+b_1z+b_2z^2+...+b_nz^n+...$

se înmulţeşte membru cu membru şi se obţine:

$B^2(z)=b_0+(b_0\ b_1+\ b_1\ b_0)z+...+(b_0b_n+...b_nb_0)z^n+...\Rightarrow$
$B^2(z)=b_1+b_2z+b_3z^2+...+b_{n+1}z^n+...$ |se înmulţeste cu z $\Rightarrow$
$zB^2(z)= b_1z+b_2z^2+b_3z^3+...+b_nz^n+... = B(z)- 1 \Rightarrow$
$zB^2(z)= B(z)- 1$

$B(z)=\frac{1-\sqrt{1-4z}}{2z}$, soluţia bună

$$B(z)=\frac{1-\sqrt{1-4z}}{2z}=\frac{1}{2z}\left(1-(1-4z)^{1/2}\right)=\frac{1}{2z}\left(1-\sum_{n\geq 0}C_{1/2}^{n}(-4z)^n\right)=\sum_{n\geq 0}C_{1/2}^{n+1}(-1)^n\bullet 2^{2n+1}z^n$$

$$\Rightarrow b_n=C_{1/2}^{n+1}(-1)^n\bullet 2^{2n+1}=...=\frac{1}{n+1}\frac{(2n)!}{n!n!}=\frac{1}{n+1}C_{2n}^{n}=\frac{1}{n+1}\binom{2n}{n}$$

Am obţinut ceea ce am dat mai înainte:

$$b_n=\frac{1}{n+1}C_{2n}^{n}=\frac{1}{n+1}\binom{2n}{n}$$

Apelul intermediar recursiv şi nu direct (ca şi cele care le-am folosit până acum) se obţine când apelul recursiv al unei proceduri **P1** nu se face în definiţia ei ci într-o altă procedură **P2**

apelată în prealabil în **`P1`**. Apoi din **`P2`** se apelează iar **`P1.`** Nu vom exemplifica această modalitate, pentru ca o considerăm prea dificilă pentru învăţământul preuniversitar.

1.3. Aspecte matematice, formalizare

Cel mai ilustrativ exemplu utilizat pentru recursivitate este definirea factorialului; reluăm definirea recursivă:

$$factorial(n) = \begin{cases} 1 & pentru \quad n = 0 \\ n * factorial(n-1) & pentru \quad n > 0 \end{cases}$$

Pentru a calcula **factorial(n)** trebuie să înmulţim **n** cu **factorial(n-1)**, care e definit cu ajutorul **factorial(n-2),...,** până când ajungem la **factorial(0)**, care e definit explicit având valoarea **1**. Pentru o astfel de definiţie poate că e potrivită notaţie expresia condiţională introdusă de McCarty [1], noaţie care foloseşte expresii condiţionale de forma:
`

$$[b_1:e_1,\ b_2:e_2,\ ...,\ b_{n-1}:e_{n-1},\ e_n]$$

Cu $b_i, i=1,...,n-1$, am notat condiţii booleene, care pot fi adevărate sau false, iar cu $e_i, i=1,...,n$ s-au notat expresii (calcule, formule, funcţii, etc.)
Evaluarea expresiei condiţionale se va face astfel:

- se evaluează pe rând condiţiile b_i, până când găsim primul b_i adevărat, şi atunci valoarea expresiei condiţionale va fi expresia e_i, corespunzătoare lui b_i;
- dacă nu s-a găsit nici un b_i cu valoarea logică adevărat, atunci valoarea expresiei condiţionale va fi valoarea expresiei e_n.

Revenind la definiţia factorialului, ea poate fi transcrisă în felul următor:

```
factorial(n)=[(n=0):1, n*factorial(n-1)]
```

Dacă extindem definiţia pentru toate valorile intregi a lui **n** avem:

```
factorial(n)=[(n<=0):1, n*factorial(n-1)]
```

În acest exemplu se ştie numărul de paşi, de exemplu pentu **factorial(4)** se apelează recursiv de **3** ori. Dar în cazul algoritmului lui Euclid de determinare a celui mai mare divizor comun a două numere întregi şi pozitive, numărul de apeluri recursive, nu e aprioric cunoscut; definiţia cu ajutorul expresiei condiţionale poate fi:

```
CMMDC(n,m)=[(m=0):n, CMMDC(m, MODULO(n,m))]
```

Spre exemplu:
CMMDC(345,1920)=CMMDC(1920,345)=CMMDC(345,195)=CMMDC(195,150)=
CMMDC(150,45)=CMMDC(45,15)=CMMDC(15,0)=15
Deci apelul recursiv a fost utilizat de 6 ori.

Un alt exemplu este definiţia funcţiei **Bessel**:

$$J_{n+1}(x) = (2*n/x) * J_n(x) - J_{n-1}(x)$$

Rescriem în $J_n(x)$ în forma simetrică **J(n,x)** şi înlocuim **n** cu **n-1**, relaţia de recurenţă va deveni:

J(n,x)=(2*n/x)*J(n-1,x)-J(n-2,x)

Dacă pentru un argument particular **x** ştim valoare lui J_0 şi J_1 atunci putem defini:

J(n,x)=[(n=0):J(0,x), (n=1):J(1,x), ((2*(n-1)/x*J(n-1,x)-J(n-2,x)]

Un alt exemplu este funcţia Ackermann: **ac:NxN→ N**

$$ac = \begin{cases} n+1, & \text{daca } m = 0 \\ ac(m-1,1), & \text{daca } n = 0 \\ ac(m-1, ac(m, n-1)), & \text{altfel} \end{cases}$$

Definită cu ajutorul expresiei condiţionale avem:

ac(m,n)=[(m=0):n+1, (n=0):ac(m-1,1), ac(m-1,ac(m,n-1))]

1.4. Probleme simple

Vom da câteva exemple de proceduri recursive care utilizează la intrare un singur parametru **n** de tip întreg şi pozitiv şi determină:

a) numărul cifrelor sale
b) suma cifrelor numărului
c) produsul cifrelor sale
d) oglinditul său (oglinditul lui 123 este 321);
e) cifra de pe poziţia **i** a unui număr natural.

Aceste rezolvări simple pot să familiarizeze elevul cu modul recursiv de gândire.
Pentru a determina numărul cifrelor lui **n** ne gândim dacă am putea da o formulă recurentă, ceva în genul şirului **Fibonacci**:

- pentru **n∈{0,1,2,3,4,5,6,7,8,9}** evident numărul de cifre este **1**;
- pentru **n ≥ 10,** se poate gândi că numărul de cifre a lui n este **1**+ numarul de cifre a numărului obţinut din n, din care am tăiat ultima cifră (de fiecare dată se taie cifra unităţilor); tăierea ultimei cifre se obţine prin împărtirea întreagă la **10**.

Exemplu: **NrCifre(123)= 1+ NrCifre(12)= 1+(1+NrCifre(1))=1+1+1=3.**

Formula recurentă ar putea sa fie:

$$NrCifre(n) = \begin{cases} 1 & 0 \le n \le 9 \\ 1 + NrCifre(n/10) & n \ge 10 \end{cases}$$

unde prin **n/10** am notat câtul întreg al împărţirii **n/10**, deci $\lfloor n/10 \rfloor$.

Varianta **Pascal:**

```
type Natural = 0..MaxLongInt;

function NrCifre(n:Natural):Natural;
begin
      if n in [0..9]
            then NrCifre:=1
            else NrCifre:=1 + NrCifre(n div 10);
end;
```

Varianta **C/C++:**

```
long NrCifre(long n)
{  if (n>=0 && n<10)    return 1;
                        return 1 + NrCifre(n/10);
}
```

Pentru a determina suma cifrelor gândim analog:

- pentru **n∈{0,1,2,3,4,5,6,7,8,9}** evident suma cifrelor este **n**;
- pentru **n ≥ 10,** se poate gândi că suma cifrelor lui **n** este ultima cifră + suma cifrelor numărului obţinut din n, din care am tăiat ultima cifră (de fiecare dată se taie cifra unităţilor); tăierea ultimei cifre se obţine prin împărtirea întreagă la 10 iar ultima cifră se poate obţine prin restul împărţirii întregi a lui n la 10.

Exemplu: **SumaCifre(123)= 3+ SumaCifre(12)= 3+(2+SumaCifre(1))=3+2+1=6.**

Formula recurentă ar putea sa fie:

$$SumaCifre(n) = \begin{cases} n & 0 \le n \le 9 \\ (\text{n mod } 10) + SumaCifre(n/10) & n \ge 10 \end{cases}$$

unde prin **n/10** am notat câtul întreg al împărţirii **n/10**, deci $\left[n/10 \right]$;

iar prin **(n mod 10)** am notat restul împărţirii lui **n** la **10.**

Varianta **Pascal:**

```
type Natural = 0..MaxLongInt;
function SumaCifre(n:Natural):Natural;
begin
      if n in [0..9]
            then SumaCifre:= n
            else SumaCifre:= n mod 10 + SumaCifre(n div 10);
end;
```

Varianta **C/C++:**

```
long SumaCifre(long n)
{  if (n>=0 && n<10)     return n;
                         return n%10 + SumaCifre(n/10);
}
```

Pentru a determina produsul cifrelor gândim analog:

- pentru **n∈{0,1,2,3,4,5,6,7,8,9}** evident produsul cifrelor este **n**;
- pentru **n ≥ 10,** se poate gândi că produsul cifrelor lui **n** este ultima cifră * produsul cifrelor numărului obţinut din n, din care am tăiat ultima cifră (de fiecare dată se taie cifra unităţilor); tăierea ultimei cifre se obţine prin împărtirea întreagă la **10** iar ultima cifră se poate obţine prin restul împărţirii întregi a lui **n** la **10**.

Exemplu: **ProdusCifre(123)= 3* ProdusCifre(12)= 3*(2*ProdusCifre(1))=3*2*1=6.**

Formula recurentă ar putea să fie:

$$\text{Pr}odusCifre(n) = \begin{cases} n & 0 \le n \le 9 \\ (\text{n mod } 10) * \text{Pr}odusCifre(n/10) & n \ge 10 \end{cases}$$

unde prin **n/10** am notat câtul întreg al împărţirii **n/10**, deci $\left[n/10 \right]$;

iar prin **(n mod 10)** am notat restul împărţirii lui **n** la **10.**

Varianta **Pascal**

```
type Natural = 0..MaxLongInt;
function ProdusCifre(n:Natural):Natural;
begin
      if n in [0..9]
            then ProdusCifre:= n
            else ProdusCifre:= n mod 10 + ProdusCifre(n div 10);
end;
```

Varianta **C/C++:**

```
long ProdusCifre(long n)
{  if (n>=0 && n<10)     return n;
                         return n%10 + ProdusCifre(n/10);
}
```

Pentru a determina oglinditul cifrelor gândim altfel:

- pentru **n∈{0,1,2,3,4,5,6,7,8,9}** evident oglinditul lui **n** este **n**;
- pentru **n ≥ 10,** se poate gândi că oglinditul lui **n** este ultima cifră * $10^{\texttt{NrCifre(n/10)}}$ adunată cu oglinditul lui **n / 10**

Exemplu: **Oglinda(123)=3*10²+(Oglinda(12))=300+(2*10¹ +Oglinda(1))=300+20+1=321.**

Formula recurentă ar putea sa fie:

$$Oglinda(n)=\begin{cases} n & 0\le n\le 9 \\ (n \bmod 10)*10^{NrCifre(n/10)}+Oglinda(n/10) & n\ge 10 \end{cases}$$

unde prin **n/10** am notat câtul întreg al împărţirii **n/10**, deci $\left[n/10\right]$;

iar prin (**n mod 10**) am notat restul împărţirii lui **n** la **10.**

Varianta **Pascal:** Dăm programul complet care verifică dacă un număr dat **n** este palindrom sau nu.

```
program Palindrom;
 uses fdelay,crt;
 type Natural = 0..MaxLongInt;

function NrCifre(n:Natural):Natural;
begin
    if n in[0..9] then NrCifre:=1
                  else NrCifre:=1+NrCifre(n div 10);
end;
function Oglinda (n:Natural): Natural;
var aux:Natural;
begin
    if n in [0..9]
      then Oglinda:= n
      else
        begin
          aux    :=NrCifre(n div 10);
          Oglinda:=trunc((n mod 10)*exp(aux*ln(10))) + Oglinda(n div 10);
        end;
end;

var n,nrcif:Natural;
begin
  clrscr;
  write ('dati n=');
  readln(n);
  nrcif :=NrCifre(n);
  writeln("oglinditul=",Oglinda(n));
  if n=Oglinda(n)
      then writeln (n,' este palindrom   ')
      else writeln (n,' nu este palindrom');
  readkey;
end.
```

Execuţia programului:

```
dati n=15249
oglinditul=94251
15249 nu este palindrom
```

Varianta **C/C++:** Dăm programul complet care verifică dacă un număr dat **n** este palindrom sau nu.

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
long NrCifre(long n)
{if (n>=0 && n<10)return 1;
                  return 1+NrCifre(n/10);
}
long Oglinda (long n)
{int aux;
 if (n >=0 && n<10)
       return n;
 else
      {aux= NrCifre(n / 10);
       return ((n % 10)*pow(10,aux) + Oglinda(n / 10));
      }
}
void main()
{long  n;
 clrscr();
 cout<<"dati n=";cin>>n;
 cout<<Oglinda(n)<<endl;
 if (n==Oglinda(n))
      cout<<n<<" este palindrom";
 else cout<<n<<" nu este palindrom";
 getch();
}
```

Execuţia programului:

```
dati n=152251
oglinditul=152251
152251 este palindrom
```

```
Program Oglinda;

Altă variantă oglindit;

function Ogli(N:longint; Og:longint):longint; {la apel Og este 0}
begin
   if(N=0)
     then Ogli:=Og
     else Ogli:=Ogli(N div 10, Og*10+ N mod 10);
end;

var N,Og:longint;
begin
  write('da N=');
  readln(N);
  Og:=0;
  write('oglinda lui ',N,' este ',Ogli(N,Og));
  readln;
end.
```

Problema e) Cifra de pe poziţia **poz (**de la dreapta la stânga**)** a unui număr natural **n** poate fi gândită astfel:

- se scad numărul de împărţiri ale lui **n** la **10** din variabila **poz**; dacă am ajuns la 0 se returnează (**n mod 10**)

Varianta **Pascal:**

```
program Pozit;
 uses fdelay,crt;
 type Natural = 0..MaxLongInt;
function Pozitia (n,Poz:Natural): byte;
begin
    if Poz=1
      then Pozitia:=n mod 10
      else
        begin
           Poz:=Poz-1;
           Pozitia:=Pozitia(n div 10,Poz);
        end;
end;
var n,Poz:Natural;
begin
  write   ('dati n=');
  readln  (n);
  write   ('a cata cifra =');
  readln  (Poz);
  writeln (”cifra este:”,Pozitia(n,Poz));
  readkey;
end.
```

Execuţia programului:

```
dati n=15249
a cata cifra =4
cifra este:5
```

Varianta **C/C++**:

```
#include<iostream.h>
#include<conio.h>
int Pozitia (long n,long Poz)
{if (Poz==1)
         return n % 10;
         return Pozitia(n / 10,--Poz);
}

void main()
{ long n;
  int  Poz;
  clrscr();
  cout <<"dati n=";        cin >>n;
  cout <<"a cata cifra ="; cin >>Poz;
  cout <<Pozitia(n,Poz);
  getch();
}
```

Execuţia programului:

```
dati n=15249
a cata cifra =3
cifra este:2
```

Algoritmului lui Euclid prin impărţire

Varianta **Pascal:**

```
program EuclidImpartire;
uses fdelay,crt;
type Natural=1..MaxLongInt;

function Cmmdc (a,b:Natural): Natural;
Begin
 if a mod b = 0 then Cmmdc:= b
                else Cmmdc:= Cmmdc (b, a mod b)
end;
var x,y:Natural;
begin
 clrscr;
 write  ('dati x,y=');
 readln (x,y);
 write  ('Cmmdc(',x,',',y,')=',Cmmdc (x,y));
 readkey
end.
```

Execuţia programului:

```
dati x,y= 100 12
Cmmdc(100,12)=4
```

Varianta **C/C++:**

```
#include <iostream.h>
#include <conio.h>
long Cmmdc(long a, long b)
{ if (!(a % b)) return b;
           else return Cmmdc(b, a % b);
}

void main(void)
{ long x,y;
  clrscr();
  cout << "dati un  numar natural=";
  cin  >> x;
  cout << "dati alt numar natural=";
  cin  >> y;
  cout << "cmmdc(" << x << "," << y << ")=" << Cmmdc (x,y);
}
```

Execuţia programului:

```
dati un  numar natural=100
dati alt numar natural=12
cmmdc(100,12)=4
```

Algoritmului lui Euclid prin scădere

Varianta **PASCAL:**

```
program EuclidScadere;
uses crt;
var x,y:longint;
function Cmmdc (a,b:longint): longint;
Begin
 if a = b then Cmmdc:= a
          else begin
                 if a>b then  Cmmdc:= Cmmdc (a-b,b)
                        else  Cmmdc:= Cmmdc (a,b-a);
               end
end;
begin
 clrscr;
 write ('dati x,y=');
 readln(x,y);
 write ('cmmdc(',x,',',y,')=',Cmmdc (x,y));
 readkey
end.
```

Execuţia programului:

```
dati x,y= 1000 12
cmmdc(1000,12)=4
```

Varianta **C/C++:**

```
#include <iostream.h>
#include <conio.h>
long Cmmdc(long a, long b)
{ if (a == b)  return a;
      if (a>b) return  Cmmdc (a-b,b)
               return  Cmmdc (a,b-a);
}
void main(void)
{ long x,y;
  clrscr();
  cout << "dati un  numar natural=";  cin  >> x;
  cout << "dati alt numar natural=";  cin  >> y;
  cout << "cmmdc(" << x << "," << y << ")=" << Cmmdc (x,y);
}
```

Execuţia programului:

```
dati un  numar natural=1000
dati alt numar natural=12
cmmdc(1000,12)=4
```

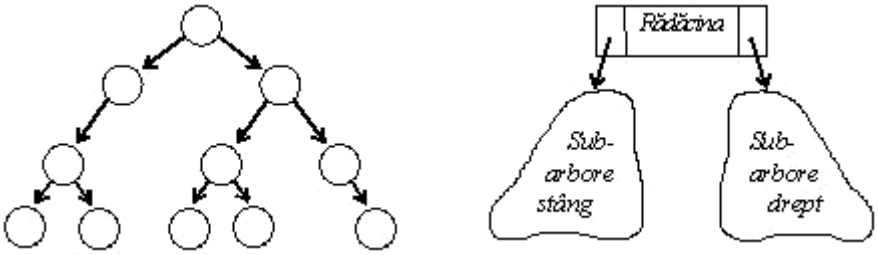
1.5. Probleme complexe

Vom da câteva exemple de proceduri recursive care se aplică arborilor binari..

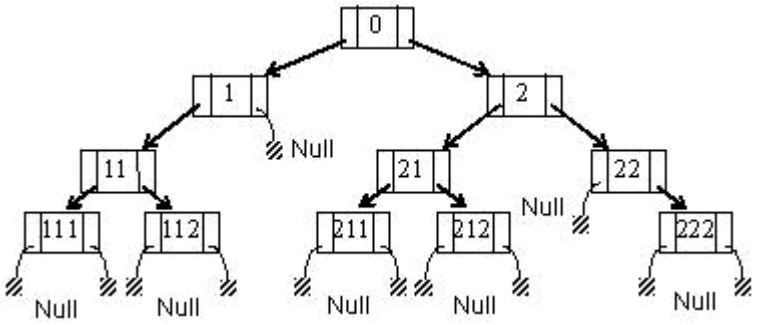
O colecţie de elemente are o structură de tip ***arborescent*** dacă elementele componente sunt în relaţie ***unu la mai multe***, adică un element este în relaţie cu mai multe elemente.

Elementele unei astfel de structuri se numesc ***noduri*** sau ***vârfuri.*** Acestea au un unic predecesor numit ***părinte*** şi mai mulţi succesori numiţi ***fii.*** Un arbore este format din mulţimea nodurilor şi legăturile dintre acestea. Un nod al unui arbore poate fi ***nod rădăcină*** (dacă nu are predecesor), poate fi ***nod intern*** (dacă are un singur predecesor şi mai mulţi succesori) sau poate fi ***nod terminal*** sau ***frunză*** (dacă nu are niciun succesor). Un arbore particular este ***arborele binar*** pentru care relaţia dintre elemente este de tip ***unu la două,*** adică un element poate avea maxim doi succesori.

Un arbore binar poate fi definit recursiv ca fiind o mulţime (colecţie) de noduri vidă sau formată din nodul ***Rădăcină, SubarboreStâng şi SubarboreDrept.***



La reprezentarea în memorie a unui arbore binar, pe lânga informaţia propriu-zisă se vor memora în fiecare nod şi adresele de legătură spre nodurile succesoare conform reprezentarii grafice următoare:



Prin ***traversarea*** unui arbore binar vom înţelege parcurgerea tuturor vârfurilor arborelui, trecând o singură dată prin fiecare nod.

În funcţie de ordinea (disciplina) de vizitare a nodurilor unui arbore binar, traversarea poate fi în ***preordine,*** în ***inordine,*** în ***postordine*** sau pe ***nivele.*** Traversarea pe nivele este de tip **breathfirst**, celelalte sunt de tip **depthfirst.**

Traversarea în ***preordine*** impune parcurgerea întâi a nodului rădăcină, apoi a subarborelui stâng şi după aceea a subarborelui drept. Deci parcurgerea are loc în ordinea: ***Rădăcină, SubArboreStâng, SubArboreDrept*** (notată ***RSD***). Evident, definiţia este recursivă, traversarea unui subarbore fiind făcută după aceeaşi regulă, începând cu rădăcina. O procedură în Pseudocod corespunzatoare traversării ***RSD*** se dă în continuare.

```
Procedură  PreOrdine (R:Arbore);
 Daca R ≠ NULL
   Atunci
       Prelucrare (R.Info);
       PreOrdine  (R.SubArboreStang);
       PreOrdine  (R.SubArboreDrept)
 SfDaca
SfProcedura;
```

La traversarea în ***inordine*** se parcurge întâi subarborele stâng, apoi nodul rădăcină şi după aceea subarborele drept. Deci se traversează arborele în ordinea: ***SubArboreStâng***, ***Rădăcină***, ***SubArboreDrept*** (notată ***SRD***), conform procedurii:

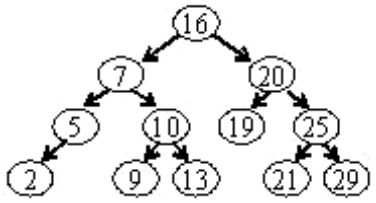
```
Procedură  InOrdine (R:Arbore);
 Daca R ≠ NULL
   Atunci
       InOrdine  (R.SubArboreStang);
       Prelucrare(R.Info);
       InOrdine  (R.SubArboreDrept)
 SfDaca
SfProcedura;
```

Traversarea în ***postordine*** este aceea în care se parcurge întâi subarborele stâng, apoi subarborele drept şi după aceea nodul rădăcină. Ordinea de parcurgere este următoarea: ***SubArboreStâng***, ***SubArboreDrept***, ***Rădăcină*** (notată ***SDR***). O procedură în Pseudocod corespunzătoare traversării ***SDR*** este:

```
Procedură  PostOrdine (R:Arbore);
 Daca R ≠ NULL
   Atunci
       PostOrdine (R.SubArboreStang);
       PostOrdine (R.SubArboreDrept)
       Prelucrare (R.Info);
 SfDaca
SfProcedura;
```

Traversarea pe ***nivele*** se face vizitând întâi *rădăcina* (nivelul 0), *succesorii rădăcinii* (deci nodurile de pe nivelul 1), de la stânga spre drepta, se continuă cu *succesorii nodurilor de pe nivelul 1*, de la stânga spre dreapta ş.a.m.d.

Pentru arborele reprezentat grafic mai jos, ordinea nodurilor corespunzătoare celor patru tipuri de traversări (parcurgeri) este următoarea :



a) în *preordine* : 16,7,5,2,10,9,13,20,19,25,21,29;
b) în *inordine* : 2,5,7,9,10,13,16,19,20,21,25,29;
c) în *postordine* : 2,5,9,13,10,7,19,21,29,25,20,16;
d) pe nivele: 16,7,20,5,10,19,25,2,9,13,21,29.

Observaţie: Cele trei parcurgeri *preordine*+*postordine*+*inordine* definesc în mod unic aborele binar. Se pune întrebarea dacă sunt suficiente două parcurgeri pentru o definire unică a arborelui binar. Răspuns: *preordine*+*inordine*, respectiv *inordine*+*postordine* definesc în mod unic arborele, dar *preordine*+*postordine* nu.

Exemplu: Se implementează şi se utilizează **TAD** ***arbore binar*** având ca informaţie în noduri un caracter, cu operaţiile specifice:

- creare arbore binar în preordine;
- creare arbore binar pe nivele;
- copierea unui arbore binar în alt arbore binar;
- verificarea egalităţii a doi arbori binari;
- parcurgerea nerecursivă pe nivele;
- parcurgere recursivă şi nerecursivă în preordine;
- parcurgere recursivă şi nerecursivă în inordine;
- parcurgere recursivă şi nerecursivă în postordine;
- căutare informaţie în arborele binar;
- eliberarea zonei de memorie alocate dinamic pentru arbore.

Variantă **C++:**

Fişierul **arbinar.h**:

```
class ARB;                              //declaratie clasa arbore binar
class NOD                               //definitie clasa nod arbore
{protected:
      char info;                        //informatia din nod;
      NOD *st;                          //adresa fiului stang;
      NOD *dr;                          //adresa fiului drept;
  public:
      NOD(){st = dr = NULL;}            //constructor implicit;
      NOD(char c)                       //constructor;
         {info = c; st = dr = NULL;}
      friend class ARB;           //clasa arbore prietena a clasei nod
};
class ARB                    //definitie clasa arbore binar
 {protected:
      NOD* rad;              //adresa nodului radacina a arborelui binar;
      NOD* crePre(NOD*);     //functii protejate recursive care
      void inOrdRec(NOD*);   //sunt apelate de functiile omonime
      void preOrdRec(NOD*); //de interfata;
      void postOrdRec(NOD*);//este nevoie de aceste functii
      int  egal(NOD*,NOD*); //deoarece functiile de interfata nu pot
      NOD* copiere(NOD*);    //fi recursive, dar trebuie sa modeleze
      void elib(NOD*);       //o prelucrare recursiva a arborelui binar
      NOD* cree();
  public:
      ARB(){rad = NULL;}     //constructor implicit;
      ARB(ARB&);             //constructor de copiere;
      ~ARB();                //destructor;
      void crePreOrd();      //creare arbore in preordine;
      void creNivele();      //creare arbore pe nivele;
      void preOrdine();      //parcurgere nerecursiva in preordine;
      void inOrdine();       //parcurgere nerecursiva in inordine;
      void postOrdine();     //parcurgere nerecursiva in postordine;
      void inOrdRec();       //parcurgere recursiva in inordine;
      void preOrdRec();      //parcurgere recursiva in preordine;
      void postOrdRec();     //parcurgere recursiva in postordine;
      void nivele();         //parcurgere arbore pe nivele;
      int egal(ARB&);        //verifica egalitatea a doi arbori;
      int caut(char c);      //cauta in arbore nodul cu informatia c;
};
```

Pentru parcurgerea/crearea pe nivele se utilizează o coadă de aşteptare, iar pentru celelate parcurgeri nerecursive se utilizează o stivă. **COADA<tip>** şi **STIVA<tip>** sunt cele două clase parametrizate (generice) descrise în secţiunea 7 din lucrarea [4] şi care vor fi instanţiate pentru:

tip = adresa nod arbore(NOD*).

Următorul fişier **arbinar.cpp** are conţinutul:

```
#include <iostream.h>
#include <conio.h>
#include "stiva_templ.h"    //vezi [4]
#include "coada_templ.h"    //vezi [4]
#include "arbinar.h"
NOD* ARB::cree()            //citeste informatia, creeaza nod
{ char c;                   //arbore si returneaza adresa acestuia;
  cin >> c;
  if(c == '$') return(NULL);
  return new NOD(c);
}
void ARB::crePreOrd() //functie de interfata care creeaza arbore in
{rad = crePre(NULL);} //preordine prin apelarea unei functii recursive

NOD* ARB::crePre(NOD* p)    //functie recursiva care creeaza
                            //un arbore binar in preordine;
{ if(p == NULL) cout << "\nDati radacina:";
  if(p = cree())
    {cout << "\n Dati fiul stang ($ pentru NULL) al lui "
          << p->info << ": ";
     p->st = crePre(p);
     cout << "\n Dati fiul drept ($ pentru NULL) al lui "
          << p->info<<": ";
     p->dr = crePre(p);
    }
 return p;
}
void ARB::creNivele()       //creeaza un arbore binar pe nivele folosind
{                           //o coada de asteptare in care sunt memorate
  NOD* p;                   //adrese de noduri;
  COADA<NOD*> coada(50);
  cout << "\nDati radacina :  ";
  if(!(p = cree()))
          {rad = NULL;
           return ;}
  coada.Adaug(p);
  rad = p;
  while(coada.Nevida())
     { coada.Extrag(p);
       cout << "\nDati fiul stang ($ pentru NULL) al lui "
            << p->info <<" : ";
       p->st = cree();
       if(p->st) coada.Adaug(p->st);
       cout << "\nDati fiul drept ($ pentru NULL) al lui "
            << p->info << " : ";
       p->dr = cree();
       if(p->dr) coada.Adaug(p->dr);
     }
}
```

```
void ARB::preOrdine()       //parcurgerea nerecursiva in preordine
{ NOD* p;                   //folosind o stiva de adrese de noduri;
  STIVA<NOD*> stiva(50);
  if(!rad)
     { cout << " Arbore vid";
       return; }
  stiva.push(rad);
  while(stiva.Nevida())
     {stiva.pop(p);
      cout << p->info;
      if(p->dr) stiva.push(p->dr);
      if(p->st) stiva.push(p->st);
     }
}
void ARB::inOrdine()        //parcurgerea nerecursiva in inordine
{ NOD* p;                   //folosind o stiva de adrese de noduri;
  STIVA<NOD*> stiva(50);
  if(!rad)
      { cout << " Arbore vid";
        return;}
  p = rad;
  while(p || stiva.Nevida())
  { while(p)
       { stiva.push(p);
         p = p->st;
       }
    stiva.pop(p);
    cout << p->info;
    p = p->dr;
 }
}

typedef struct
  { NOD* p;           //p este adresa de tip NOD;
    int k;            //indicator cu valori 0 si 1;
  } el;

void ARB::postOrdine()      //parcurgerea nerecursiva in postordine
{                           //folosind o stiva in care sunt memorate
                            //elemente de tipul el;
  NOD *p;
  el x;
  STIVA<el> stiva(50);
  if(!rad)
   {cout << " Arbore vid";
    return;
   }
  p = rad;
  while(p || stiva.Nevida())
   {while(p)
     {
      x.p = p;   x.k = 0;
      stiva.push(x);
      p = p->st;
     }
   stiva.pop(x);
   p = x.p;
```

```
   if(x.k == 0)  //nu s-a parcurs inca subarborele drept al nodului
                 //cu adresa p;
      { x.k = 1; //se trece la parcurgerea subarborelui drept al
                 //nodului cu adresa p;
       stiva.push(x);
       p = p->dr;
       }
      else
       {cout << p->info;
        p = NULL;
       }
 }
}
void ARB::nivele()    //parcurgerea pe nivele folosind o coada de
{ NOD* p;             //asteptare care memoreaza adrese de noduri;
  COADA<NOD*> coada(50);
  if(!rad)
   {cout << " Arbore vid";
    return;
   }
  coada.Adaug(rad);
  while(coada.Nevida())
   {coada.Extrag(p);
    cout << p->info;
    if(p->st) coada.Adaug(p->st);
    if(p->dr) coada.Adaug(p->dr);
   }
}
void ARB::preOrdRec()       //functia de interfata pentru parcurgerea
{ preOrdRec(rad);           //recursiva in preordine;
}
void ARB::preOrdRec(NOD* p)//parcurgere recursiva arbore binar in
{ if(p == NULL) return;     //preordine;
  cout << p->info;
  preOrdRec(p->st);
  preOrdRec(p->dr);
}
void ARB::inOrdRec()        //functia de interfata pentru parcurgerea
{ inOrdRec(rad);            //recursiva in inordine;
}

void ARB::inOrdRec(NOD* p)  //parcurgere recursiva arbore binar
{if(p == NULL) return;      //in inordine;
 inOrdRec(p->st);
 cout << p->info;
 inOrdRec(p->dr);
}

void ARB::postOrdRec()      //functia de interfata pentru parcurgerea
{ postOrdRec(rad);          //recursiva in postordine;
}

void ARB::postOrdRec(NOD* p)     //parcurgere recursiva arbore binar
{ if(p == NULL) return;          //in postordine;
  postOrdRec(p->st);
  postOrdRec(p->dr);
  cout << p->info;
}
```

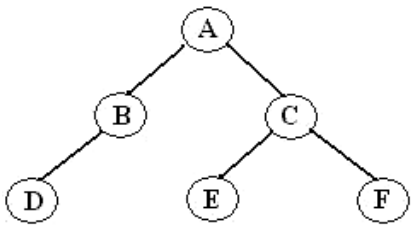
```
int ARB::caut(char x)       //se foloseste parcurgerea in preordine
{ NOD* p;                   //nerecursiva pentru cautarea nodului
  STIVA<NOD*> stiva(50);    //cu informatia x, returneaza 1 la gasire
  if(!rad) return 0;        //si 0 in caz contrar;
  stiva.push(rad);
  while(stiva.Nevida())
    {stiva.pop(p);
     if(p->info == x) return 1;
     if(p->dr) stiva.push(p->dr);
     if(p->st) stiva.push(p->st);
     }
  return 0;
}
int ARB::egal(ARB& a)       //functia de interfata pentru verificarea
{ return egal(rad,a.rad);   //egalitatii a doi arbori binari;
}

ARB::ARB(ARB& a)       //constructorul de copiere care apeleaza o functie
{rad=copiere(a.rad);   //recursiva ce realizeaza copierea efectiva;
}
                       //returneaza valoarea 1 daca subarborii cu
                       //adresele p1 si p2 sunt egali si 0 in caz contrar;
int ARB::egal(NOD* p1,NOD *p2)
{ if(!p1 && !p2) return 1;       //cei doi subarbori sunt nuli;
  if( p1 && p2 && (p1->info==p2->info) &&
      egal(p1->st,p2->st) && egal(p1->dr,p2->dr)) return 1;
  return 0;
}
NOD* ARB::copiere(NOD* p1) //copiaza arborele cu adresa p1 într-un
{                          //arbore cu adresa p2 si returneaza p2;
  NOD* p2;
  if(!p1) return NULL;
  p2 = new NOD(p1->info);
  p2->st = copiere(p1->st);
  p2->dr = copiere(p1->dr);
  return p2;
}
ARB::~ARB()            //destructorul care apeleaza o functie recursiva
{elib(rad);}           //pentru dealocarea zonei de memorie alocate
                       //arborelui;

void ARB::elib(NOD* p)      //elibereaza zona alocata dinamic pentru
{                           //arbore prin parcurgere recursiva in
  if(p == NULL) return;     //postordine;
  elib(p->st);
  elib(p->dr);
  delete p;
}
```

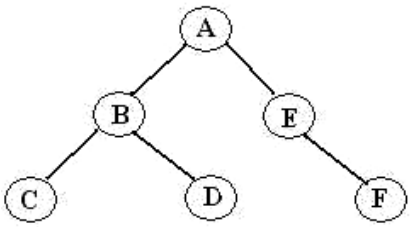
Fişierul care exploatează cele 2 fişiere anterioare:

```
// program client care apeleaza TAD - arbore binar
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include "arbinar.cpp"
```

```
void main()
{
 char c;
 cout <<"\n..... TAD - ARBORE BINAR - informatia din noduri
        un caracter.....\n";
 ARB arb1, arb2;
 cout << "\n    Creare arbore binar pe nivele";
 arb1.creNivele();
 cout << "\n Parcurgere nerecursiva in inordine:\n";
 arb1.inOrdine();
 cout << "\n Parcurgere nerecursiva in preordine:\n";
 arb1.preOrdine();
 cout << "\n Parcurgere nerecursiva in postordine:\n";
 arb1.postOrdine();
 cout << "\n Parcurgere pe nivele:\n";
 arb1.nivele();

 cout << "\n\n Dati informatia de cautat in arbore:";
 cin  >> c;
 if(arb1.caut(c))
     cout << "\nNodul cu informatia " << c << " exista in arbore";
   else
     cout << "\nNodul cu informatia " << c << " nu exista in arbore";

 ARB arb1c(arb1);
 cout << "\n\n S-a copiat arborele binar creat in alt arbore
           si se verica egalitatea lor";
 if(arb1.egal(arb1c))
     cout << "\n Cei doi arbori sunt egali";
   else
     cout << "\n Cei doi arbori sunt diferiti";

 cout << "\n\n Creare arbore binar in preordine  \n";
 arb2.crePreOrd();
 cout << "\n Parcurgere recursiva in inordine:\n";
 arb2.inOrdRec();
 cout << "\n Parcurgere recursiva in preordine:\n";
 arb2.preOrdRec();
 cout << "\n Parcurgere recursiva in postordine:\n";
 arb2.postOrdRec();

 cout << "\n\n Se verifica egalitate arbore creat pe nivele cu
          cel creat in preordine";
 if(arb1.egal(arb2))
     cout << "\n Cei doi arbori sunt egali";
   else
     cout<<"\n Cei doi arbori sunt diferiti";
}
```

Se consideră arborii din figura următoare care se vor crea şi prelucra.





Rezultate execuţie:

```
... TAD - ARBORE BINAR - informatia din noduri un caracter...

    Creare arbore binar pe nivele
Dati radacina :  A
Dati fiul stang ($ pentru NULL) al lui A : B
Dati fiul drept ($ pentru NULL) al lui A : C
Dati fiul stang ($ pentru NULL) al lui B : D
Dati fiul drept ($ pentru NULL) al lui B : $
Dati fiul stang ($ pentru NULL) al lui C : E
Dati fiul drept ($ pentru NULL) al lui C : F
Dati fiul stang ($ pentru NULL) al lui D : $
Dati fiul drept ($ pentru NULL) al lui D : $
Dati fiul stang ($ pentru NULL) al lui E : $
Dati fiul drept ($ pentru NULL) al lui E : $
Dati fiul stang ($ pentru NULL) al lui F : $
Dati fiul drept ($ pentru NULL) al lui F : $

 Parcurgere nerecursiva in inordine:DBAECF
 Parcurgere nerecursiva in preordine:ABDCEF
 Parcurgere nerecursiva in postordine:DBEFCA
 Parcurgere pe nivele:ABCDEF
 Dati informatia de cautat in arbore:E
 Nodul cu informatia E exista in arbore
 S-a copiat arborele binar creat in alt arbore si se verica egalitatea
 lor
 Cei doi arbori sunt egali

    Creare arbore binar in preordine
Dati radacina:A
Dati fiul stang ($ pentru NULL) al lui A: B
Dati fiul stang ($ pentru NULL) al lui B: C
Dati fiul stang ($ pentru NULL) al lui C: $
Dati fiul drept ($ pentru NULL) al lui C: $
Dati fiul drept ($ pentru NULL) al lui B: D
Dati fiul stang ($ pentru NULL) al lui D: $
Dati fiul drept ($ pentru NULL) al lui D: $
Dati fiul drept ($ pentru NULL) al lui A: E
Dati fiul stang ($ pentru NULL) al lui E: $
Dati fiul drept ($ pentru NULL) al lui E: F
Dati fiul stang ($ pentru NULL) al lui F: $
Dati fiul drept ($ pentru NULL) al lui F: $
 Parcurgere recursiva in inordine:CBDAEF
 Parcurgere recursiva in preordine:ABCDEF
 Parcurgere recursiva in postordine:CDBFEA

 Se verifica egalitatea arbore creat pe nivele cu cel creat in preordine
 Cei doi arbori nu sunt egali
```

1.6. Rezolvarea recurenţelor de tip Divide et Impera

Câteva aspecte de bază legate de algoritmi:

1) **Eficienţa algorimilor**
   - constă în măsurarea cantităţii de resurse utilizate şi anume **spaţiu** de memorie şi **timp** de execuţie;
   - există două metode de măsurare a eficienţei:
     - analiză matematică a algoritmului (analiză asimptotică) şi nu execuţii exacte de timp;
     - analiză empirică, care măsoară timpul exact de rulare pentru date specifice;

2) **Complexitatea programării** (unele **SD** sunt mai simplu de implementat, altele mai greu)
   - ce structură de date se alege pentru o problemă concretă ?
   - aspecte legate management şi inginerie soft (implementare, flexibilitate, întreţinere)

Timpul de execuţie

- dat fiind un program, timpul său de execuţie nu este un număr fix, ci mai degrabă o funcţie care depinde de dimensiunea **SD** (structurilor de date) folosite;
- pentru intrari diferite (instanţe a **SD**) se pot obţine timpi diferiţi de rulare;
- dacă notăm cu **n = dim(SD)** atunci timpul de execuţie depinde de **n**, îl notăm cu **T(n)**;
- în general, noţiunea de timp de execuţie trebuie să fie independentă de maşina fizică; se va măsura **numărul de paşi** pe care algoritmul îi execută, (sau numărul de instrucţiuni ale codului pe care algorimul îl execută)
- timpul **T(n)** obţinut în acest fel nu e **timpul real de execuţie** dar este proporţinal cu acesta (deci timpul real ar fi **T(n)* c**, **c** constantă mică);
- de obicei **T(n)** ne va interesa în două cazuri:
  - timpul maxim de execuţie (cazul cel mai nefavorabil), pentru toate valorile lui **n**;
  - timpul mediu de execuţie = media timpilor de execuţie pentru toate valorile lui **n**;

- **T(n)** dă ceea ce se cheamă **complexitatea algoritmului**.
- De obicei se va face o analiză asimptotică a lui **T(n)**
  Exemplu:
  Presupunem că am analizat un algoritm şi am obţinut:

$$T(n) = 13n^3 + 42n^2 + 2n\log_2(n) + 3\sqrt{n}$$

  - termenul n^3 este dominant pentru valori mari ale lei **n**, deci ceilalţi termeni ai sumei se pot neglija;
  - constanta **13** este relativ mică şi se poate ignora şi ea;
  - deci **T(n)** creşte odată cu $n^3 \Rightarrow T(n) \in O(n^3)$.

Să reamintim câteva definiţii ale notaţiilor asimptotice.

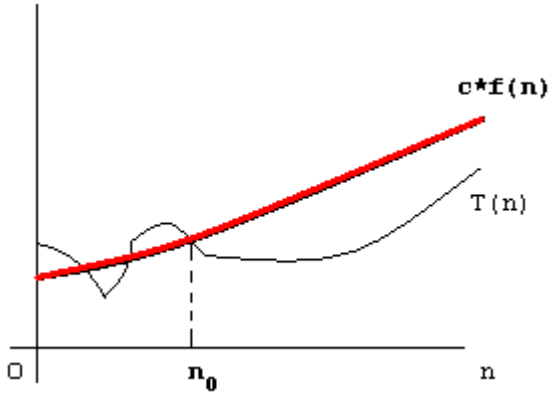
1.6.1. Notaţii asimptotice

Definiţie (1) Margine asimptotică superioară

Spunem că $T(n) \in O(f(n))$ (sau că $T(n)=O(f(n))$) ddacă $\exists\ c, n_0>0$ constante (deci nu depind de **n**) astfel încât:

$$0 \le T(n) \le c*f(n),\ \forall\ n \ge n_0.$$

Vizual avem schema următoare:



O definiţie alternativă ar fi $T(n)=O(f(n))$, dacă $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \begin{cases} 0 & \text{sau} \\ \text{constanta finita} \end{cases}$

Să observă de exemplu că dacă $T(n)=O(n^3)$ atunci $T(n)$ este şi $O(n^4)$, $O(n^5)$... $O(n^k), k\ge 3$.

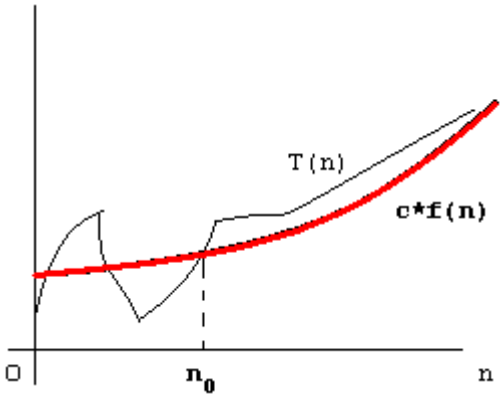
Mai general avem $O(n^k)+O(n^l)=O(n^{max(k,l)})$, vezi [2].

Definiţie (2) Margine asimptotică inferioară

Spunem că $T(n) \in \Omega(f(n))$ (sau că $T(n) = \Omega(f(n))$) ddacă $\exists\ c, n_0>0$ constante (deci nu depind de **n**) astfel încât:

$$0 \le c*f(n) \le T(n),\ \forall\ n \ge n_0.$$

Vizual avem schema următoare:



O definiţie alternativă ar fi $T(n)=\Omega(f(n))$, dacă $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \begin{cases} \infty & \text{sau} \\ \text{constanta finita} & \text{dar nu } 0 \end{cases}$

Să observă de exemplu că dacă $T(n)=\Omega(n^3)$ atunci $T(n)$ e şi $\Omega(n^2), \Omega(n)$... $\Omega(n^k), 0 \le k \le 3$.

Mai general avem $\Omega(n^k)+\Omega(n^l)=\Omega(n^{\min(k,l)})$, vezi [2].

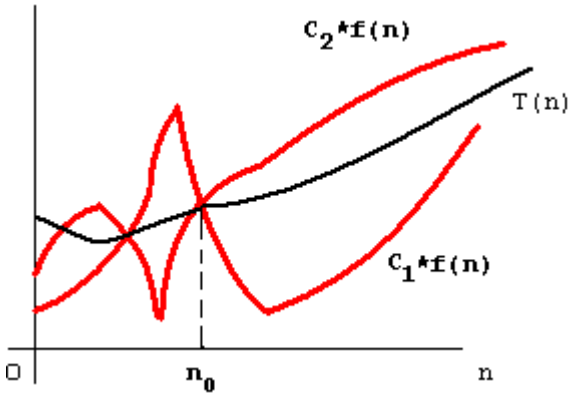
Definiţie (3) Margine asimptotică strânsă

Spunem că $T(n)\in\Theta(f(n))$ (sau că $T(n)=\Theta(f(n))$) ddacă $\exists\ c_1,c_2,n_0>0$ constante (deci nu depind de n) astfel încât:

$$0 \le c_1*f(n)\le T(n)\le c_2*f(n), \quad \forall\ n\ge n_0.$$

Sau $T(n)\in\Theta(f(n))$ ddacă $T(n)=O(f(n))$ şi $T(n)=\Omega(f(n))$

Vizual avem schema următoare:



O definiţie alternativă ar fi $T(n)=\Theta(f(n))$, dacă $\lim_{n\to\infty}\frac{T(n)}{f(n)}=$ constanta $\neq 0$

Evident că este cea mai „tare” dintre aproximări asimptotice.
Timpul de execuţie al unui algoritm este $\Theta(f(n))$ ddacă timpul de execuţie în cazul cel mai nefavorabil este $O(f(n))$ şi timpul său de execuţie în cazul cel mai favorabil este $\Omega(f(n))$.

Exemple:

- $T(n)=O(1)$ – complexitate constantă;
- $T(n)=O(\log_2(\log_2 n))$ – timp f. rapid (aproape ca cel constant);
- $T(n)=O(\log_2 n)$ – complexitate logaritmică, f.bună;
- $T(n)=O((\log_2 n)^k)$ – complexitate polilogaritmică, k constant, nu e rău;
- $T(n)=O(n)$ – complexitate liniară;
- $T(n)=O(n.\log_2 n)$ – complexitatea pentru sortarea prin compararea termenilor unei secvenţe;
- $T(n)=O(n^2)$ – complexitate pătratică (quadratică), când $n\ge 10^6$ e rău deocamdată;
- $T(n)=O(n^k)$ – complexitate polinomială, k constant, relativ bun;
- $T(n)=O(2^n),O(n^n),O(n!)$ – complexitate exponenţială, bune pentru n cu valori mici.

Sume importante în analiza algoritmilor iterativi şi recursivi

a) preupunem că avem structura repetivă:

```
Pentru  i ← 1, n execută
      CorpulCiclului
SfPentru;
```

Dacă corpul se execută în p(i) paşi atunci $T(n)=\sum_{i=1}^{n} p(i)$

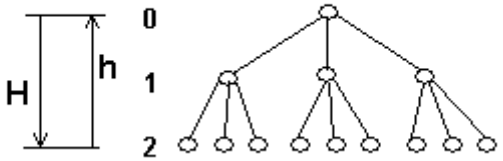
b) $\sum_{i=1}^{n} 1 = n$, sumă constantă; rezultat liniar

c) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, sumă liniară; rezultat polinomial pătratic;

d) $\sum_{i=1}^{n} \frac{1}{i} = \ln(n) + r(n)$, suma armonică (H(n)), rezultat logaritmic;

e) $\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, c \neq 1$, suma progresiei geometrice, destul de importantă;

Exemplu. Să se afle câte noduri are un arbore **ternar complet** de înălţime h (sau adâncime H).



H (adâncimea) exprimă numărul maxim de arce (nivele) de la rădacină la o frunză.
h (înălţimea) exprimă numărul maxim de arce (nivele) de la o frunză la rădacină; evident **H=h.**

$$\Rightarrow \sum_{i=0}^{h} 3^i = \frac{3^{h+1}-1}{3-1} = \frac{3^{h+1}-1}{2} \in O(3^h)$$

Invers: dacă se ştie numărul de noduri n ale unui arbore ternar, care este înălţimea **h**?
Avem:

$$\frac{3^{h+1}-1}{2} = n \Rightarrow 3^{h+1} = 2n+1 \Rightarrow h+1 = \log_3(2n+1) \Rightarrow h = \log_3(2n+1)-1 \Rightarrow h \in O(\log_3 n)$$

Recursiv:

$$N(k) = \begin{cases} 1, & k = 0 \\ 3N(k-1)+1, & k \geq 1 \end{cases}$$

Avem: $N(h) = 3N(h-1)+1 = 3(3N(h-2)+1)+1 = 3^2N(h-2)+3+1 = \ldots =$

$$= 3^hN(0) + (3^{h-1} + \ldots + 1) = \sum_{i=0}^{h} 3^i = \frac{3^{h+1}-1}{3-1} = \frac{3^{h+1}-1}{2} \in O(3^h)$$

1.6.2. Recurenţele de tip Divide et Impera

În general o algoritmul **Divide et Impera** se aplică unui tablou (vector) **V = <v_1,...,v_n>** (sau **V[1..n], n=dim[V]**), asupra căruia vrem sa aplicăm o operaţie oarecare (sortare, determinarea valorii maxime, determinarea cmmdc, etc). Deoarece nu se poate aplica operaţia direct se împarte secvenţa în două subsecvenţe şi se încearcă rezolvarea celor două şi apoi combinarea rezultatelor şi procesul continuă. În momentul când operaţia se poate aplica direct (de exemplu un număr este sortat, suma a două numere se determină direct, sau chiar suma unui număr este chiar numărul, etc) se rezolvă subsecvenţa şi se face o revenire din succesivă din apel.
Avem schiţată în pseudocod procedura recursivă:

```
Procedura DivImp(V,p,q)                    //V secventa 1<=p<=q<=n
                                           //(p,q) determina o subsecventa
      Daca q-p ≤ ε atunci  Rezolva(V,p,q)
                   altfel  m=(p+q) div 2  //m se poate determina si altfel
                           DivImp(V,p,m)  //apel pentru (p,m)
                           DivImp(V,m+1,q)//apel pentru (m+1,q)
                           Combina(V,p,m,q)
      SfDaca
SfProcedura
```

Observaţii generale

1°. Se divide problema în subprobleme (**2,3,..., a, a** finit);

2°. Stăpâneşte subproblemele prin

- rezolvarea nerecursivă (directă), dacă dimensiunea lor **≤ ε** (prag)
- rezolvă subproblemele recursiv, dacă dimensiunea lor **> ε.**

3°. Combină soluţiile tuturor subproblemelor pentru a ajunge la soluţia finală;

4°. Apel din exteriorul procedurii se face cu **DivImp(V,1,n);**

5°. Pentru sortare avem **ε=1** şi **a=2**, iar procedura **Combina** face interclasarea;

6°. Pentru sortare avem şi varianta recursivă care urmează:

```
Procedura SortMerge(V,p,q)
      Daca p ≤ q atunci
                       m=(p+q) div 2
                       SortMerge(V,p,m)
                       SortMerge(V,m+1,q)
                       InterClaseaza(V,p,m,q)
      sfDaca
sfProcedura
```

1.6.3. Analiza algoritmului tip Divide et Impera

Vom încerca să deducem o recurenţă

$$T(n)=\begin{cases}\theta(1), & n\le\varepsilon\\ aT\left(\frac{n}{b}\right)+D(n)+C(n), & n>\varepsilon\end{cases}$$

Unde: – **ε** este acel **prag** de la care problema este suficient de mică şi se poate rezolva direct şi/sau iterativ;
- **a** reprezintă numărul de subprobleme;
- **1/b** reprezintă dimensiunea unei subprobleme din problema iniţială;
- **D(n)** = timpul necesar pentru a divide în **a** subprobleme;
- **C(n)** = timpul necesar pentru a combina soluţiile celor **a** subprobleme în soluţia problemei iniţiale.

Exemplu: La algoritmul de sortare prin interclasare avem
- divizarea în **2** subsecvenţe de lungime **n/2**; deci **a=2**, iar dimensiunea unei probleme este **1/2**;
- dacă **p=q** avem timp constant **θ(1)** (**n=1**, deci **ε=1**);
- **D(n)=θ(1)**, pentru că se determină indicele de mijloc al secvenţei **(p,q)**;
- **C(n)=θ(n)**, (interclasarea a **2** secvenţe de lungime **p**, respectiv **q** dă **θ(p+q-1)**)

Avem deci:

$$T(n)=\begin{cases}\theta(1), & n=1\\ 2T\left(\frac{n}{2}\right)+\theta(1)+\theta(n), & n>1\end{cases}\text{ sau }T(n)=\begin{cases}\theta(1), & n=1\\ 2T\left(\frac{n}{2}\right)+\theta(n), & n>1\end{cases}$$

În final avem $T(n)=\begin{cases}0, & n=1\\ 2T\left(\frac{n}{2}\right)+n, & n>1\end{cases}$

1.6.4. Metode de rezolvarea a recurenţelor

Există 3 metode importante de soluţionare a tipurilor de recurenţă descrise anterior:

1. medoda **iteraţiei**; prin iteraţii succesive şi reduceri se deduce formula

   Exemplu 1) să luăm formula $T(n)=2T\left(\frac{n}{2}\right)+n$ pentru $n=2^k$, avem succesiv

   $T(2^k)=2T(2^{k-1})+2^k=2(2T(2^{k-2})+2^{k-1})+2^k=2^2T(2^{k-2})+2^k+2^k=\ldots=$
   $=2^kT(2^0)+2^k+\ldots+2^k=(k+1)2^k$. Revenind la n rezultă:
   $T(n)=n(\log_2 n+1)=O(n\log_2 n)$.
   Pentru n natural oarecare ∃ **k>0** astfel că: $n\le 2^k$, deci $T(n)\le T(2^k)$ şi de aici $T(n)\le n(\log_2 n+1)$. Deci $T(n)=O(n\log_2 n)$ pentru n oarecare.

   Exemplul 2) fie $T(n)=3T(n/4)+n$
   O iterăm astfel:
   $T(n)=n+3T(n/4)=n+3(n/4+3T(n/16))=n+3n/4+9T(n/16)=\ldots$

Se observă că al i-lea termen din serie este $3^i\lfloor n/4^i \rfloor$. Iteraţia ajunge la final când $\lfloor n/4^i \rfloor = 1$ sau echivalent când i depăşeşte $\log_4 n$. Continuând iterarea până în acest punct şi utilizând delimitarea $\lfloor n/4^i \rfloor \le n/4^i$, descoperim că suma conţine şi o serie geometrică descrescătoare:

$$T(n) = n + 3n/4 + 9n/16 + ... + 3^{\log_4 n}\Theta(1) \le n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta\left(n^{\log_4 3}\right) = 4n + O(n) = O(n)$$

2. metoda **substituţiei**, prin intuiţie se deduce formula folosind diverse substituţii iar apoi se demonstrează prin inducţie;

   Exemplu: fie recurenţa: $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log_2 n$, care pare dificilă.

   Putem simplifica recurenţa printr-o schimbare de variabilă, fie m=$\log_2$n şi se obţine

   $$T(2^m) = 2T(2^{m/2}) + m$$

   Acum redenumim **S(m)=T(2^m)** şi obţinem o nouă recurenţă:

   $$S(m) = 2S(m/2) + m$$

   Această recurenţă este asemănatoare recurenţei de la exemplul 1, metoda iteraţiei şi are soluţia **S(m)=O(mlog$_2$m)**

   Revenind la n, obţinem **T(n)=T(2^m)=S(m)= O(mlog$_2$m)=O(log$_2$n log$_2$(log$_2$n))**, ceea ce este o complexitate f. bună.

3. metoda **master.**

   Această metodă se aplică problemelor recursive care au timpul de execuţie de forma:

   $$T(n) = aT(n/b) + f(n), \text{ cu } a \ge 1, b > 1, f(n) > 0 \text{ (asimptotic)}$$

   Avem 3 cazuri (demonstrate în [2])

   1^o Dacă $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ pentru o anumită constantă **ε**, $\Rightarrow$ $T(n) = \Theta\left(n^{\log_b a}\right)$

   2^o Dacă $f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a}\log_2 n\right)$

   3^o Dacă $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$, pentru o anumită constantă **ε > 0**, şi dacă $af(n/b) \le cf(n)$ (numită şi condiţie de regularitate) pentru o constantă c şi n suficient de mari, atunci $T(n) = \Theta(f(n))$

   Observăm că practic se compară **f(n)** cu $n^{\log_b a}$ şi soluţia o va da cea mai mare dintre ele; **f(n)** trebuie să fie polinomial mai mică (în primul caz) şi polinomial mai mare (în cazul 3) decât $n^{\log_b a}$.

Utilizarea metodei master.

Exemplul 1.

Fie $T(n) = 9T(n/3) + n$

Avem **a=9**, **b=3** şi **f(n)=n**.

Calculăm $n^{\log_b a} = n^{\log_3 9} = n^2$, deci **f(n)=n=n$^{2-1}$ =** $f(n) = n = n^{2-1} = n^{\log_3 9 - 1}$,deci suntem în cazul 1^o şi soluţia este: $T(n) = \Theta(n^2)$.

Exemplul 2.

Fie $T(n) = T(2n/3) + 1$

Avem **a=1**, **b=3/2** şi **f(n)=1**.

Calculăm $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$ **= f(n)**, deci suntem în cazul 2^o şi soluţia este $T(n) = \Theta(n^0 \log_2 n)$, adică $T(n) = \Theta(\log_2 n)$.

Exemplul 3.

Fie $T(n) = 3T(n/4) + n\log_2 n$

Avem **a=3**, **b=4** şi **f(n)=nlog$_2$n**.

Calculăm $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$iar **f(n)=nlog$_2$n**; avem $f(n) = n\log_2 n = n^{\log_4 3 + \varepsilon}$, deci suntem în cazul 3^o şi soluţia ar fi $T(n) = \Theta(n \log_2 n)$, dacă se verifică condiţia de regularitate:

$af(n/b) = 3(n/4)\log_2(n/4) \le \frac{3}{4} n\log_2 n = cf(n)$, cu $c = \frac{3}{4}$.

Exemplul 4.

Fie $T(n) = 2T(n/2) + n\log_2 n$

Avem **a=2, b=2** şi **f(n)=nlog$_2$n**.

$n^{\log_b a} = n^{\log_2 2} = n$ ar fi cazul 3^o dar **f(n)=nlog$_2$n** este asimptotic mai mare ca **n** dar nu polinomial mai mare deci nu putem rezolva.

1.6.5. Exemple Divide et Impera

Determinarea minimului

Varianta **PASCAL:**

```
program DIMinim;
uses crt;

function COMB(m1,m2:integer):integer;     {functia de combinare a rezultatelor }
 begin
  if(m1<m2) then COMB:=m1                 {determina minimul dintre 2 numere   }
            else COMB:=m2;
end;
const LimitaNrElem=100;                   {doar de aici e nevoie de tabloul x  }
var x:array[1..LimitaNrElem] of integer;
function DI(p,q:integer):integer;
  var m1,m2,mp,m:integer;
  begin
    if p=q then   DI:=x[p]                {cand secventa are lungime 1, minimul}
          else begin                      {este chiar elementul secventei      }
                 m :=(p+q) div 2;
                 m1:=DI(p,m);
                 m2:=DI(m+1,q);
                 DI:=COMB(m1,m2);
               end;
end;

var i,n :integer;
begin                                     {programul principal                 }
  clrscr;
  write  ('dati numarul de elemente n= ');{1 <= n <= LimitaNrElem                }
  readln (n);
  for i:=1 to n do begin
                     write ('x[',i,']=');
                     readln(x[i]);
                   end;
  write('minimul=',DI(1,n):3);            {apelul functiei DI                  }
  readkey;
end.
```

Execuţia programului:

```
dati numarul de elemente n=5
x[1]=7
x[2]=3
x[3]=2
x[4]=100
x[5]=75
minimul=  2
```

Varianta **C/C++:**

```
#include<iostream.h>
#include<conio.h>

int COMB(int m1,int m2)             //functia de combinare a rezultatelor
{ if(m1<m2) return m1;              //determina minimul dintre 2 numere
            return m2;
}

int const LimitaNrElem=100;         //doar de aici e nevoie de tabloul x
int x[LimitaNrElem+1];
```

```
int DI(int p, int q)
{ int m1,m2,m;
  if (p==q)  return x[p];            //cand secventa are lungime 1, minimul
            {m      =(p+q) / 2;      //este chiar elementul secventei
             m1     =DI(p,m);
             m2     =DI(m+1,q);
             return COMB(m1,m2);
            }
}
void main()                          //functia principala main
{int i,n;
 cout<<"dati numarul de elemente n= ";
 cin >>n;                            //1 <= n <= LimitaNrElem
 for (i=1;i<=n;i++)
   {cout << "x[" << i << "]=";
    cin  >> x[i];
   }
 cout << "minimul=" << DI(1,n);      //apelul functiei DI
}
```

Execuţia programului:

```
dati numarul de elemente n=5
x[1]=7
x[2]=3
x[3]=2
x[4]=100
x[5]=75
minimul=  2
```

Determinarea maximului

Varianta **PASCAL:**

```
program DIMaxim;
uses crt;
function COMB(m1,m2:integer):integer; {functia de combinare a rezultatelor   }
 begin                                {determina maximul dintre 2 numere     }
  if(m1>m2) then COMB:=m1
            else COMB:=m2;
end;
const LimitaNrElem=100;              {doar de aici e nevoie de tabloul x    }
var x:array[1..LimitaNrElem] of integer;
function DI(p,q:integer):integer;
 var m1,m2,mp,m:integer;
 begin
   if p=q then   DI:=x[p]            {cand secventa are lungime 1, maximul  }
         else                        {este chiar elementul secventei        }
              begin
                m :=(p+q) div 2;
                m1:=DI(p,m);
                m2:=DI(m+1,q);
                DI:=COMB(m1,m2);
              end;
end;
var i,n :integer;
begin                                {programul principal                   }
  write ('dati numarul de elemente n= ');
  readln(n);                         {1 <= n <= LimitaNrElem                }
  for i:=1 to n do begin
                     write ('x[',i,']=');
                     readln(x[i]);
                   end;
  write('minimul=',DI(1,n):3);       {apelul functiei DI                     }
end.
```

Execuţia programului:

```
dati numarul de elemente n=5
x[1]=7
x[2]=3
x[3]=2
x[4]=100
x[5]=75
minimul=100
```

Varianta **C/C++:**

```
#include<iostream.h>
#include<conio.h>

int COMB(int m1,int m2)                    //functia de combinare a rezultatelor
{ if(m1>m2) return m1;                     //determina maximul dintre 2 numere
            return m2;
}

int const LimitaNrElem=100;                //doar de aici e nevoie de tabloul x
int x[LimitaNrElem+1];

int DI(int p, int q)
{ int m1,m2,m;
  if (p==q) return x[p];                   //cand secventa are lungime 1, maximul
                                           //este chiar elementul secventei
           {m      =(p+q) / 2;
            m1     =DI(p,m);
            m2     =DI(m+1,q);
            return COMB(m1,m2);
           }
}

void main()                                //functia principala
{int i,n;
 clrscr();
 cout<<"dati numarul de elemente n= ";     //1 <= n <= LimitaNrElem
 cin >>n;
 for (i=1;i<=n;i++)
   {cout << "x[" << i << "]=";
    cin  >> x[i];
   }
 cout<< "maximul=" << DI(1,n);             //apelul functiei DI
 getch();
}
```

Execuţia programului:

```
dati numarul de elemente n=5
x[1]=7
x[2]=3
x[3]=2
x[4]=100
x[5]=75
maximul=100
```

Suma elementelor unui vector

Varianta **PASCAL:**

```
program DISuma;                          {Suma elementelor unui vector        }
uses crt;
const nrelem=100;
var x:array[1..LimitaNrElem] of integer;
function DI(p,q:integer):integer;
var m1,m2,m:integer;
 begin
   if p=q then   DI:=x[p]                {cand secventa are lungime 1, suma   }
         else begin                      {este chiar elementul secventei      }
                m :=(p+q) div 2;
                m1:=DI(p,m);
                m2:=DI(m+1,q);
                DI:=m1+m2;               {combinarea inseamna suma            }
              end;
end;
var  i,n :integer;
begin                                    {programul principal                 }
  write ('dati numarul de elemente n= ');
  readln(n);                             { 1 <= n <= LimitaNrElem             }
  for i:=1 to n do begin
                     write ('x[',i,']=');
                     readln(x[i]);
                   end;
  write('Suma= ',DI(1,n):3);             {apelul functiei DI                  }
end.
```

Execuţia programului:

```
dati numarul de elemente n=5
x[1]=7
x[2]=3
x[3]=2
x[4]=100
x[5]=75
Suma=187
```

Varianta **C/C++:**

```
#include<iostream.h>                     //Suma elementelor unui vector
#include<conio.h>
int const LimitaNrElem=100;
int x[LimitaNrElem+1];
int DI(int p, int q)
{ int m1,m2,m;
  if (p==q) return x[p];                 //cand secventa are lungime 1, suma
           {m      =(p+q) / 2;           //este chiar elementul secventei
            m1     =DI(p,m);
            m2     =DI(m+1,q);
            return m1+m2;                //combinarea inseamna suma
           }
}
void main()
{int i,n;
 clrscr();
 cout<<"dati numarul de elemente n= ";
 cin >>n;                                // 1<= n <= LimitaNrElem
 for (i=1;i<=n;i++)
   {cout << "x[" << i << "]=";
    cin  >> x[i];
   }
```

```
 cout<< "Suma=" << DI(1,n);                //apelul functiei DI
 getch();
}
```

Execuţia programului:

```
dati numarul de elemente n=5
x[1]=7
x[2]=3
x[3]=2
x[4]=100
x[5]=75
Suma=187
```

Determinarea produsului elementelor unui vector

Varianta **PASCAL:**

```
program DIProdus;                    {Produsul elementelor unui vector de numere}
uses fdelay,crt;
const LimitaNrElem=100;
type Vector = record                 {declararea tipului vector                  }
                 Dim:byte;
                 V  :array[1..LimitaNrElem] of integer;
              end;
var X:vector;

function DI(p,q:integer):integer;
var m1,m2,m:integer;
begin
  with X do
  if p=q
     then DI:=V[p]                   {cand secventa are lungimea 1, produsul     }
     else begin                      {este chiar elementul secventei             }
            m :=(p+q) div 2;
            m1:=DI(p,m);
            m2:=DI(m+1,q);
            DI:=m1*m2;
          end;
end;
var  i :integer;
begin
  clrscr;
  with X do
    begin
      write ('dati numarul de elemente = ');
      readln(X.Dim);                 { 1 <= X.Dim <= LimitaNrElem                }
      for i:=1 to Dim do
        begin
          write ('X[',i,']=');
          readln(V[i]);
        end;
      write('Produs= ',DI(1,dim):3);      {apelul functiei DI                    }
    end;
  readkey;
end.
```

Execuţia programului:

```
dati un numarul de elemente n=5
x[1]=1
x[2]=2
x[3]=3
x[4]=4
x[5]=5
Produs=120
```

Varianta **C/C++:**

```
#include<iostream.h>                //produsul elementelor unui vector
#include<conio.h>
int const LimitaNrElem=100;
int       x[LimitaNrElem+1];
int DI(int p, int q)
{ int m1,m2,m;
  if (p==q) return x[p];            //cand secventa are lungimea 1, produsul
           {m      =(p+q) / 2;      //este chiar elementul secventei
            m1     =DI(p,m);
            m2     =DI(m+1,q);
            return m1*m2;
           }
}
void main()
{int i,n;
 clrscr();
 cout<<"dati numarul de elemente n= ";
 cin >>n;                           // 1 <= n <= LimitaNrElem
 for (i=1;i<=n;i++)
   {cout << "x[" << i << "]=";
    cin  >> x[i];
   }
 cout<< "Produs=" << DI(1,n); //apelul functiei DI
}
```

Execuţia programului:

```
dati un numarul de elemente n=5
x[1]=1
x[2]=2
x[3]=3
x[4]=4
x[5]=5
Produs=120
```

Algoritmul lui Euclid, cmmdc dintr-un vector de numere naturale

Varianta **PASCAL:**

```
program DIEuclid;
uses crt;
const LimitaNrElem=100;
function CMMDC(a,b:longint): longint;     {functia CMMDC este definita recursiv}
Begin
 if a = b then CMMDC:= a
          else begin
                 if a>b then  CMMDC:= CMMDC (a-b,b)
                        else  CMMDC:= CMMDC (a,b-a);
               end
end;
var x:array[1..LimitaNrElem] of integer;  {doar de aici e nevoie de tabloul x  }
function DI (p,q:integer):integer;
var m1,m2,m:integer;
 begin
   if p=q then DI:=x[p]
               else begin
                      m :=(p+q) div 2;
                      m1:=DI(p,m);
                      m2:=DI(m+1,q);
                      di:=CMMDC(m1,m2);
                    end;
end;
```

```
var i,n :integer;
begin
  clrscr;
  write ('dati numarul de elemente n= ');
  readln(n);                                  {1 <= n <= LimitaNrElem            }
  for i:=1 to n do begin
                     write ('x[',i,']=');
                     readln(x[i]);
                   end;
  write('cmmdc=',DI(1,n):3);                  {apelul functiei CMMDC             }
  readkey;
end.
```

Execuţia programului:

```
dati un numarul de elemente n=5
x[1]=10
x[2]=20
x[3]=40
x[4]=120
x[5]=50
cmmdc= 10
```

Varianta **C/C++:**

```
#include<iostream.h>
#include<conio.h>
int const LimitaNrElem=100;
int x[LimitaNrElem+1];
long CMMDC(long a, long b)
{if (a==b)         return a;
 else {if (a>b)    return CMMDC(a-b,b);
                   return CMMDC(a,b-a);
      }
}
int DI(int p, int q)
{ int m1,m2,m;
  if (p==q) return x[p];
            {m      =(p+q) / 2;
             m1     =DI(p,m);
             m2     =DI(m+1,q);
             return CMMDC(m1,m2);
           }
}
void main()
{int i,n;
 clrscr();
 cout<<"dati numarul de elemente n= ";
 cin >>n;
 for (i=1;i<=n;i++)                          // 1 <= n <= LimitaNrElem
   {cout << "x[" << i << "]=";
    cin  >> x[i];
   }
 cout<<"CMMDC="<<DI(1,n);                    //apelul functiei DI
 getch();
}
```

Execuţia programului:

```
dati un numarul de elemente n=5
x[1]=10
x[2]=20
x[3]=3300
x[4]=400
x[5]=50
CMMDC=10
```

Problema turnurilor din Hanoi.

Având **n** discuri aşezate (descrescător după dimensiune) pe una din cele trei tije (numerotate cu **1**,**2**,**3**), se cere să le mutăm pe altă tijă luând câte un singur disc (cel de deasupra) de pe o tijă şi mutându-l pe o alta respectând ordinea (nu se poate depune un disc mai greu peste unul mai uşor).
Pentru a rezolva problema vom utiliza un subalgoritm care mută primele **n-1** discuri de pe tija **a** pe tija **b**. Dacă mutăm primele **n**, înseamnă că le mutăm pe toate şi problema este rezolvată.
Primele **n** discuri se mută astfel: se mută primele **n-1** discuri de pe tija **a** pe tija **c=6-a-b**, apoi se mută discul **n** (baza) pe tija destinaţie (**b**), iar în final se mută din nou primele primele **n-1** discuri de pe tija **c=6-a-b** pe tija **b**.

```
Subalgoritmul Hanoi (n,a,b) este:
      Dacă n>0 Atunci Hanoi (n-1,a,6-a-b);
                      Mută (a,b);           {trebuie definita}
                      Hanoi (n-1,6-a-b,b);
      SfDacă
SfHanoi
```

Merge Sort (sortare prin interclasare) .

Să se ordoneze un vector **X** cu **n** componente. Se va apela subalgoritmul de mai jos sub forma **MergeSort (X,1,n)**:

```
Subalgoritmul MergeSort (X,i,j) este: { Ordonează subşirul xi,…,xj }
      Dacă j-i>ε Atunci
                      m:=(i+j)/2;
                      MergeSort (X,i,m);
                      MergeSort (X,m+1,j);
                      Interclasare (X,i,m, X,m+1,j, X,i,j)
      SfDacă
SfMergeSort.
```

QuickSort (sortare rapidă).

Să se ordoneze un vector **X** cu **n** componente. Se va apela subalgoritmul de mai jos sub forma **QuickSort (X,1,n)** :

```
Subalgoritmul QuickSort (X,i,j) este: { Ordonează subşirul xi,…,xj }
      Dacă j-i>ε Atunci
                  m:=Split (X,i,j);     {trebuie definita procedura Split}
                  QuickSort (X,i,m);
                  QuickSort (X,m+1,j);
      SfDacă
SfQuickSort.
```

1.7. Heapul binar

Structura de date numită **heap binar** este un vector cu anumite proprietăţi:
- corespunde unui arbore binar aproape complet;
- arborele este plin exceptând eventual nivelul inferior care este plin de la stânga , la dreapta, apoi până la un anumit loc;

Un vector **A** care reprezintă un heap are **2** atribute :
-**Lungime[A]** = numărul elementelor din vector
-**DimHeap[A]** = numărul elememententelor Heap-ului memorat în vectorul **A.**

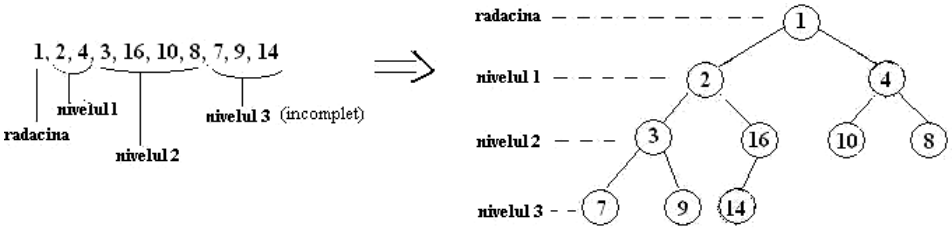
Proprietatea de heap

$$\forall\ i\ ,\ i>1,\ i<DimHeap[A] \Rightarrow A[PARINTE(i)] \geq A[i]$$

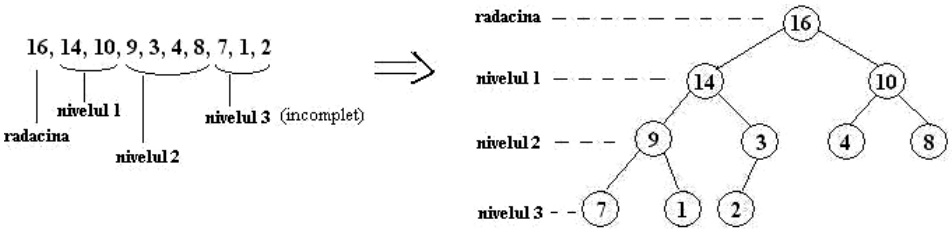
Radăcina arborelui este conţinută de **A[1]**.

Exemplu:

Secvenţa: **1,2,4,3,16,10,8,7,9,14** nu e o structură de **HEAP:**



Aceeaşi secvenţă modificată: **16,14,10,9,3,4,8,7,1,2** are o structura de **HEAP:**



Se observă că orice nod părinte are valoarea mai mare decât fiii săi.

Structura de **HEAP** se poate folosi la sortare eficientă (cu complexitatea $O(n*\log_2 n)$) şi la utilizarea cozilor cu priorităţi (exemplu gestiunea sarcinilor unui sistem de operare).

Pentru un indice i, corespunzator unui nod, se pot determina uşor indicii **PARINTE[i]** şi ai fiilor **STANGA[i]** şi **DREAPTA[i].**

```
PARINTE(i)
   returneaza [i/2]      -o singură instrucţiune translatand valoarea binara a
                                lui i la dreapta cu 1 poziţie - Θ(1)
SfParinte
```

```
STANGA(i)
   returneaza  2*i      -shift la  stânga cu 1 poziţie - Θ(1)
SfStanga

DREAPTA(i)
   returneaza 2*i+1     -shift la stanga cu 1 poziţie şi bitul nou setat pe 1 - Θ(1)
SfDreapta
```

- înălţimea unui **nod ≤ $\log_2(n)$**, unde **n=DimHeap[A]**

5 Operaţii:

ReMakeHeap	- pentru întreţinerea proprietăţii de heap;
BuildHeap	- generează un heap dintr-un vector neordonat, furnizat la intrare.
HeapSort	- ordonează un vector în spaţiul alocat acestuia.
extrage – max insereaza	- pentru cozi de prioritate.

Reconstituirea proprietaţii de **Heap** (**ReMakeHeap**).

-se presupune că la apel pentru nodul **i**, subarborii cu radacina **STANGA(i)** şi **DREAPTA(i)** sunt heap-uri.

-sarcina procedurii este de a ,"scufunda" in **HEAP** pe **A[i]** astfel încât subarborele, care are radăcina valorii **A[i]** să devină un heap.

```
Procedure ReMakeHeap(A,i)
      l ← STANGA(i)
      r ← DREAPTA(i)
      Daca l ≤ DimHeap[A]si A[l] > A[i]          // determinarea mimimului
            Atunci  maxim ← l                    // dintre A[i], A[l], A[r]
            Altfel  maxim ← i
      SfDaca
      Daca r ≤ DimHeap[A] si A[r] > A[maxim]
            Atunci      maxim ← r
      SfDaca
      Daca maxim ≠ i
            Atunci
                  A[i] ↔ A[maxim]
                  ReMakeHeap(A,maxim)            //apelul recursiv
      SfDaca
SfProcedura.
```

$T(n) \le T(2n/3)+\Theta(1) \Rightarrow T(n)=O(\log_2 n)$ - vezi cazul 2 al teoremei **master**.

Suntem în cazul 2 al teoremei **master**:

a=1, b=3/2, f(n) = Θ(1)

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = f(n) = \Theta(1)$ **=> T(n)=O($\log_2$n)** sau pentru înălţimea **h** avem **O(h)**.

Dăm în continuare procedura de construire a unui heap dintr-un vector:

```
Procedure BuildHeap(A)
      DimHeap[A] ← Lungime[A]
      Pentru i ← [Lungime[A]/2],1,-1 executa
            ReMakeHeap(A,i)
      SfPentru
SfProcedura.
```

Complexitate

Pentru o înalţime **h** oarecare (interioară, înălţimea arborelui fiind $\log_2 n$) a unui arbore binar aproape complet de n noduri, există cel mult $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ noduri de înălţime **h**.

Avem $\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(\frac{1}{2} n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{h}{2^h} \right\rceil \right) \le O\left(\frac{1}{2} n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$

deoarece:

$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$, aplicând formula $\sum_{k=0}^{\infty} k \bullet x^k = \frac{x}{(1-x)^2}$, pentru **x=1/2.**

Dăm în continuare procedura de sortare a unui heap:

```
Procedure HeapSort(A)
      BuildHeap(A)
      Pentru i ← Lungime[A],2,-1 executa
            A[1] ↔ A[i]
            DimHeap[A] ← DimHeap[A]-1
            ReMakeHeap(A,1)
      SfPentru
SfProcedura.
```

1.8. BackTracking recursiv

Metoda **backtracking** (căutare cu revenire) se utilizează dacă soluţia problemei se poate reprezenta sub forma unui vector $X=(x_1,x_2,\dots,x_n)$. X este vector din spaţiul $S=S_1\times S_2\times\dots\times S_n$, care este spaţiul tuturor valorilor posibile, iar S_i sunt mulţimi finite ($1\le i\le n$). Mulţimea rezultatelor este $R = \{X\in S \mid X$ îndeplineşte **condiţiile interne**$\}$.

Condiţiile interne sunt relaţii între componentele $x_1,x_2,\dots,x_n$ ale unui vector X pe care acestea trebuie să le îndeplinească pentru ca vectorul X să fie soluţie.

Într-o problemă se pot cere toate soluţiile sau o singură soluţie care optimizează o funcţie obiectiv dată. Prin metoda backtracking nu se generează toate valorile posibile din spaţiul stărilor (S). Vectorul X se completează secvenţial, atribuind componentei x_k o valoare numai după ce au fost atribuite deja valori componentelor $x_1,x_2,\dots,x_{k-1}$. Nu se va trece la componenta x_{k+1}, decât dacă sunt verificate **condiţiile de continuare** pentru $x_1,x_2,\dots,x_k$. **Condiţiile de continuare** sunt necesare pentru a ajunge la rezultat cu primele k alegeri făcute. Altfel spus, dacă **condiţiile de continuare** nu sunt îndeplinite, atunci indiferent cum am alege valori pentru următoarele componente $x_{k+1}, x_{k+2},\dots,x_n$, nu vom ajunge la rezultat, deci nu are sens să continuăm completarea vectorului X. Se poate observa că cu cât aceste **condiţii de continuare** sunt mai restrictive cu atât algoritmul va fi mai eficient, deci o bună alegere a **condiţiilor de continuare** va duce la reducerea numărului de căutări.

Condiţiile de continuare sunt o generalizare a **condiţiilor interne** pentru orice subşir $x_1,x_2,\dots,x_k$ $(1 \le k \le n)$. Aceasta înseamnă că pentru $k=n$ **condiţiile de continuare** coincid cu **condiţiile interne** (lucru care trebuie demonstrat teoretic). Dacă $k=n$ şi sunt îndeplinite condiţiile de continuare se poate da ca rezultat vectorul X, pentru că au fost îndeplinite condiţiile interne (deci X este soluţie).

Dacă nu sunt îndeplinite condiţiile de continuare se alege altă valoare din mulţimea S_k, pentru componenta x_k, iar dacă în mulţimea S_k nu mai sunt alte valori (pentru că mulţimile sunt finite), atunci se revine la componenta anterioară x_{k-1}, încercând o nouă alegere (din mulţimea S_{k-1} pentru componenta x_{k-1}). Dacă s-a ajuns la x_1 şi nu se mai poate executa o revenire ($k=0$), atunci algoritmul se termină.

Subalgoritmul backtracking este următorul:

```
Subalgoritmul Bt(X,k,n)
      Pentru ∀e∈Sk execută
            Xk:=e;
            Dacă Cond_Cont (X,k)
                  Atunci
                        Dacă k=n   Atunci Rezultate(X,n)
                            {k<n}  Altfel Bt(X,k+1,n)
                  SfDacă
            SfDacă
      SfPentru
SfBt.
```

Procedura `Rezultate(X,n)` va afişa componentele soluţiei X.

Dacă dorim să completăm vectorul **X** începând cu prima componentă, apelul subalgoritmului **Bt(X,k,n)** (care completează vectorul **X** din poziţia **k** până în poziţia **n**) este **Bt(X,1,n).**

În cele ce urmează vom da un exemplu ilustrativ.

Problema submulţimilor de sumă dată

Fiind date numerele pozitive $a_1, a_2, \ldots, a_n$ se cere să se determine subşirul de sumă dată T. Fie $x_i \in \{0,1\}$ astfel: 0 reprezintă faptul că ai nu intră în sumă, iar 1 în sens contrar. Se observă că $S=\{0,1\}^n$ este finită.

Condiţiile interne: $x_1*a_1+x_2*a_2+\ldots+x_n*a_n = T$.

Condiţiile de continuare: $x_1*a_1+x_2*a_2+\ldots+x_k*a_k \le T$, şi

$x_1*a_1+x_2*a_2+\ldots+x_k*a_k + a_{k+1}+a_{k+2}+\ldots+a_n \ge T$.

deci:

$$T - (a_{k+1}+a_{k+2}+\ldots+a_n) \le x_1*a_1+x_2*a_2+\ldots+x_k*a_k \le T.$$

Se observă că pentru **k=n condiţiile de continuare** devin **condiţii interne**.

Varianta **Pascal:**

```
Program Bt_Subm_Suma_Data;
      Const   n=10;
      Type    Rez=Array[1..n] Of Byte;
      Const A : Array[1..n] Of Word=(10,20,30,40,50,100,150,1000,5000,10000);
      T       : Integer=11150;
Function Cc(X:Rez;k:Byte):Boolean;
  Function Sum (k:Byte):Word;
     Begin If k>n Then Sum :=0
                  Else Sum :=Sum(k+1)+a[k]
     End;
   Function Suma (k:Byte):Word;
     Begin If k=0 Then Suma:=0
                  Else Suma:=Suma(k-1)+X[k]*a[k]
     End;
Begin
      Cc:=(T-Sum(k+1)<=Suma(k)) And (Suma(k)<=T)
End;
Procedure Tip(X:Rez;n:Byte);
Begin
      If n>0 Then
                Begin If X[n]=1 Then Write(a[n],'+');
                      Tip(X,n-1)
                End
             Else Writeln(#8+' = ',T)
End;
Procedure Bt(X:Rez; k,n:Byte);
Var v:Byte;
Begin
     For v:=0 To 1 Do
       Begin
             X[k]:=v;
             If Cc(X,k) Then
                    If k=n Then Tip(X,k)
                           Else Bt(X,k+1,n)
       End
End;
```

```
Var X:Rez;
Begin
   Writeln('*Start*');
   Bt(X,1,n);
   Write('*Stop*');
  Readln
End.
```

Execuţia programului:

```
*Start*
1000+100+10+5 = 1115
1000+100+10+3+2 = 1115
1000+100+10+4+1 = 1115
1000+100+5+4+3+2+1 = 1115
*Stop*
```

Varianta **C/C++:**

```
#include<iostream.h>
#include<conio.h>
unsigned char  n=10;
enum  Boolean  {False, True};
unsigned char X[11];                        //alocare pe cate un byte
unsigned long a[]={0,1,2,3,4,5,10,20,100,500,1000};
unsigned long T  =1115;
unsigned long Sum (unsigned char k)
{ if (k>n) return 0;
           return Sum(k+1)+a[k];
}
unsigned long Suma (unsigned char k)
{ if (k==0) return 0;
            return Suma(k-1)+X[k]*a[k];
}
Boolean Cc(unsigned char k)
{if (((T-Sum(k+1))<=Suma(k)) && (Suma(k)<=T)) return True;
                                              return False;
}
void Tip (unsigned char n)
{ if (n>0)
     { if (X[n]==1) cout<<a[n]<<"+";
       Tip(n-1);
     }
  cout<<"+ = "<<T;
}
void Bt(unsigned char k, unsigned char n)
{unsigned char v;
 for (v=0;v<=1;v++)
   {X[k]=v;
    if (Cc(k))
       if (k==n)  Tip(k);
       else       Bt(k+1,n);
   }
}

void main()
{clrscr();
 cout<<"*Start*";
 Bt(1,n);
 cout<<"*Stop*";
 getch();
}
```

Execuţia programului:

```
*Start*
1000+100+10+5 = 1115
```

```
1000+100+10+3+2 = 1115
1000+100+10+4+1 = 1115
1000+100+5+4+3+2+1 = 1115
*Stop*
```

2. METODICA PREDĂRII

2.1. Conceptul de metodă

Conceptul de metodă derivă din grecescul methodos (calea către, drum spre). Esenţa metodei de învăţământ rezultă din esenţa însăşi a activităţii de învăţare, ca formă specifică a cunoaşterii umane.

Se consideră a fi modernă metoda care duce la promovarea creativităţii elevilor. Modernitatea unei metode este dată de măsura în care aceasta reuşeşte să cultive însuşirile fundamentale necesare omului de azi şi de mâine: independenţă, spirit critic, gândire creatoare, aptitudini.

Există o relaţie între perfecţionarea procesului predare-învăţare (sistem) şi metodele utilizate (subsistem).

În funcţie de specificul disciplinei se poate face următoarea clasificare:

*a)**metode de comunicare orală***

a1) **metode expozitive** (prelegerea, explicaţia, descrierea, povestirea);

a2) **metode interogative** (conversaţia, dialogul didactic, problematizarea, dezbaterea, asaltul de idei);

*b) **metode de comunicare scrisă*** (învăţarea după textul scris, munca cu manualul)

*c) **metode obiective*** (bazate pe contactul cu realitatea)

c1) **metode experimentale**

c2) **metode de modelare**

c3) **metode demonstrative**

*d) **metode bazate pe acţiune***

d1) **algoritmizarea**

d2) **exerciţiul** şi **rezolvările de probleme**

d3) **studiul de caz** (metoda cazurilor)

d4) **proiectul** sau **tema de cercetare**

d5) **metode de simulare** (jocuri didactice, învăţarea pe simulator)

2.1.1. Subiectele metodicii predării informaticii

Gruparea subiectelor o vom realiza după un plan nu neapărat clasic, realizând astfel un model flexibil şi care va putea accepta îmbunătăţiri în ceea ce priveşte ponderea unor teme, respectiv punctul de trecere din care se tratează acestea.

1)Obiectivele disciplinei informatică;

2) Analiza critică a unor concepte false privind disciplina;

3) Domeniile informaticii, clasificarea temelor şi gruparea lor după categorii de vârstă;

4) Domeniile informaticii prevăzute în programele şcolare; cunoaşterea şi analiza programelor şcolare;

5) Metode specifice de predare a tehnicilor de programare, a limbajelor de programare, a utilitarelor şi a sistemelor de operare;

6) Instrumente didactice;

7) Metode de evaluare; proiecte, lucrări semestriale, extemporale, corectarea lor; clase de greşeli tipice, corectarea, depanarea programelor, strategii de testare;

8) Concursuri, olimpiade, simpozioane, sesiuni de comunicări; compunerea de probleme, metodologii de desfăşurare etc.;

9) Dotarea laboratoarelor, echipament şi soft; mobilarea laboratoarelor;

10) Informatica în şcoală – în afara orelor.

2.1.2 Metode specifice de predare a informaticii

Subdomeniile informaticii nu pot fi predate apelând la o singură metodă.

În cele ce urmează se vor prezenta metode specifice de abordare a predării diferitelor domenii din informatică; metodele clasice de predare (prin expunere, descoperire dirijată etc.) urmând să fie prezentate ulterior. Bineînţeles, metodele didactice se vor combina, de asemenea se vor dezvolta pe baza experienţei fiecărui profesor. Dar există câteva elemente care neapărat trebuie luate în considerare în momentul în care se alege o metodă sau o combinaţie a mai multora:

- domeniul propriu-zis al disciplinei;
- conţinutul ştiinţific;
- categoria de vârstă;
- obiectivele generale;
- nivelul clasei;

- personalitatea clasei;
- personalitatea profesorului;
- convingerile profesorului.

Metodele care vor fi prezentate se vor analiza cu avantajele şi dezavantajele lor. Se recomandă să nu se fixeze nicicând pentru totdeauna o anumită metodă aleasă la un moment dat pentru predarea unei anumite părţi din materie. S-ar plictisi profesorii, s-ar plictisi elevii, iar rezultatul nu va fi cel scontat.

Metode

A) Metodele predării algoritmilor şi tehnicilor de programare

1) Orientată pe algoritmi

Întregul proces de programare este privit ca fiind ceva indivizibil, dar accentul se pune pe conceperea algoritmului, restul activităţii de programare se realizează în planul doi, având rol de verificare. În această categorie intră disciplina Algoritmi de programare.

2) Orientată pe tipuri de probleme

Se formează un set de probleme având dificultate treptată, dezvoltate una din cealaltă sau înlănţuite pe baza unei anumite proprietăţi comune dintr-o clasă de probleme şi pe parcursul rezolvării acestora se introduc cunoştinţele necesare de programare. Se va lucra atunci când trebuie introduse structuri de date noi sau structuri de control noi. La fel se va proceda în cazul învăţării funcţiilor predefinite sau componentelor de grafică. Chiar şi recursivitatea poate fi introdusă astfel.

3) Orientată pe limbaj

Această metodă porneşte din posibilităţile limbajului de programare. Se prezintă riguros, mergând până în toate detaliile, elementele limbajului, într-o succesiune „oarecare" şi în funcţie de instrumentarul învăţat se prezintă şi cunoştinţe de programare. Procesul are loc invers decât la punctul 1) deoarece în acest caz problema de rezolvat este o anexă, rezolvarea ei foloseşte în scopul verificării cunoaşterii limbajului.

S-a precizat anterior că limbajul de programare este un instrument – ca de altfel şi calculatorul – deci scopul constă nu în a învăţa limbaje de programare , ci în a rezolva

probleme, deci în a gândi. Rezultă clar sfatul de a pune accent nu pe limbaj, ci pe rezolvarea de probleme.

4) Orientată pe structuri şi instrucţiuni

Metoda seamănă cu cea precedentă, dar nu îşi fixează atenţia pe un singur limbaj, ci pe concepte generale, valabile pentru o clasă de limbaje sau medii. Latura bună a acestei metode constă în faptul că pune accent pe prezentarea şi învăţarea unor concepte generale cum sunt de exemplu, programarea structurată, structurile abstracte de date, programarea orientată pe obiecte etc. În cazul sistemelor de gestiune a bazelor de date, de asemenea vor fi concepte care trebuie clarificate în termeni generali, independent de implementare (modelul bazelor de date, limbajul de descriere, limbajul de manipulare).

5) Orientată pe matematică

Această metodă se orientează pe necesităţile impuse de dorinţa de a rezolva probleme de matematică prin folosirea celor două instrumente: limbajul şi calculatorul. De exemplu, dacă profesorul îşi propune să predea teoria numerelor şi doreşte să rezolve probleme din acest domeniu, atunci va preda cunoştinţele necesare rezolvării acestor probleme (matematică, algoritm, limbaj – eventual şi elemente de hard) ţinând cont doar de ceea ce are nevoie în scopul rezolvării acestor probleme.

Metoda orientată pe matematică se poate aplica mai rar şi doar pentru atingerea anumitor obiective clar şi imperativ impuse de problemă. Oricum, în ultima vreme sunt „la modă" enunţurile „îmbrăcate", ele nu se mai formulează, decât foarte rar în termeni de matematică pură. Metoda orientată pe matematică a fost şi este criticată atunci când se utilizează abuziv şi toată predarea se finalizează prin rezolvarea unor probleme de matematică cu ajutorul calculatorului.

6) Orientată pe specificaţii

Această metodă se bazează pe considerentul că partea esenţială în rezolvarea de probleme constă în formalizarea problemei. Din această formalizare, respectând riguros specificaţiile, se deduce „automat" algoritmul, apoi din nou „pe robot automat", respectând reţete rigide de codificare se transformă acest algoritm în program. Această metodă nu este recomandată deloc în şcoala generală, dar nici în liceu nu se va utiliza prea des. Se

recomandă în învăţământul universitar, în cazul cursurilor de specializare etc. În liceu apare necesitatea demonstrării corectitudinii unui anumit algoritm, dar acest proces necesită un alt gen de formalizare şi se va impune relativ rar.

7) Orientată pe hardware

Cei care utilizează această metodă afirmă că nu se poate învăţa algoritmizare, decât dacă există deja cunoştinţe riguroase de programare şi se cunoaşte bine un limbaj de programare; dar un limbaj de programare nu se poate cunoaşte în detaliile sale fără cunoştinţe aprofundate privind assemblerul; de asemenea nici assemblerul nu poate fi cunoscut fără cunoaşterea limbajului maşină, respectiv a procesoarelor ajungând la aspecte de hardware.

Această metodă se alege de cei care au devenit profesori din profesionişti (analişti programatori care au lucrat la dezvoltare de soft de bază, ingineri etc.) dar şi de tineri absolvenţi de învăţământ superior „în care nu încap cunoştinţele" şi, mânaţi de cele mai bune intenţii vor să-i înveţe pe copii tot ce ştiu ei. Evident, este discutabil dacă au dreptate sau nu. Probabil există probleme ale căror rezolvare necesită o asemenea abordare, dar numai într-un mediu de şcoală adecvat.

2.2. Metode didactice utilizate în predarea recursivităţii

a) Expunerea - constă în prezentarea de către profesor a unor cunoştinţe noi, pe cale orală, în structuri bine închegate. Este o metodă utilizată în lecţiile teoretice despre recursivitate. Pentru că este o metodă ce predispune la pasivism i se poate imprima o notă activă prin recursul la întrebări, încercări de problematizare pe anumite secvenţe etc. Această metodă permite transmiterea unui volum mare de informaţii într-o unitate de timp determinată.

Explicaţia este forma de expunere în care „predomină argumentarea raţională". Astfel, în prezentarea unor algoritmi ce nu pot fi descoperiţi de către elev, cum ar fi algoritmii de descriere a procedurilor şi funcţiilor din secţiunea de implementare, se va explica mecanismul pe cazuri concrete, abia apoi trecându-se la simbolizarea sa.

b) Conversaţia euristică - este o metodă de dialog, de incitare a elevilor prin întrebări. Elevii sunt invitaţi să realizeze o incursiune în propriul univers şi să facă o serie de conexiuni care să faciliteze dezvăluirea de noi aspecte ale realităţii. Dintre situaţiile în care se utilizează acesta metodă se pot aminti: în cadrul conversaţiei care poate preceda predarea unei teme noi, pentru ca profesorul să-si poată da seama la ce nivel trebuie concepută predarea ca atare; pe tot parcursul predării subiectului nou, cu rol de feed-back, la încheierea predării unei lecţii, prin întrebări recapitulative, care să reia, în mare, aspectele reprezentative din materialul predat. De asemenea conversaţia euristică este utilă în lecţiile de recapitulare şi sistematizare.

c) Demonstraţia - este o metodă în cadrul căreia mesajul de transmis către elev se cuprinde într-un obiect concret, o acţiune concretă (de exemplu reprezentarea grafică). Astfel, utilizând reprezentarea grafică, asimilarea cunoştinţelor teoretice pe baza acestei surse intuitive este mult mai uşoară.

d) Observaţia - reprezintă una din metodele de învăţare prin cercetare şi descoperire. Este o metodă des utilizată, având în vedere că formularea „se observă că..." duce la desprinderea unor concluzii utile în construirea algoritmilor.

e) Exerciţiul - constituie o modalitate de executare repetată şi conştientă a unei acţiuni în vederea achiziţionării sau consolidării unor cunoştinţe sau abilităţi. Exerciţiul contribuie la realizarea unor sarcini cum ar fi:

- adâncirea înţelegerii noţiunilor, regulilor principiilor şi teoriilor învăţate;

- consolidarea cunoştinţelor şi deprinderilor însuşite;

- dezvoltarea operaţiilor mintale şi constituirea lor în structuri operaţionale;

- sporirea capacităţii operatorii a cunoştinţelor, priceperilor şi deprinderilor;

- prevenirea uitării şi evitarea apariţiei confuziilor.

Fiecare noţiune teoretică este exemplificată pe unit-uri concrete, iar algoritmii sunt verificaţi pentru cât mai multe exemple şi chiar cerându-le elevilor să rezolve probleme care necesită modificări uşoare ale algoritmilor predaţi. Studiul de caz-metodă ce constă din confruntarea elevului cu o situaţie, prin a cărei observare, înţelegere, interpretare, urmează să realizeze un progres în cunoaştere.

La conceperea algoritmilor de prelucrare a datelor folosind recurivitatea se încearcă analiza tuturor cazurilor posibile iar la verificarea programelor ele se vor rula pentru cât mai multe cazuri. Unii dintre algoritmi recursivi sunt valabili doar pentru anumite cazuri.

g) Învăţarea prin descoperire - constă în crearea condiţiilor de reactualizare a experienţei şi a capacităţilor individuale, în vederea desluşirii unor noi situaţii problemă. Premisa iniţială constă în delimitarea a ceea ce este util şi oportun să fie oferit elevului şi ce este necesar să fie lăsat acestuia să descopere prin proprie iniţiativă.

În funcţie de relaţia care se stabileşte între profesor şi elev, se pot delimita două feluri de descoperire:

- descoperire ***independentă,*** elevul este actorul principal, profesorul doar supraveghind şi controlând acest proces.
- descoperire ***dirijată,*** profesorul conduce descoperirea prin sugestii, puncte de sprijin, întrebări, soluţii parţiale.

Pornind de la relaţia care se stabileşte între cunoştinţele anterioare şi cele la care se ajunge se disting:

- descoperirea ***inductivă,*** când pe baza unor date şi cunoştinţe particulare sunt dobândite cunoştinţe şi se efectuează operaţii cu un grad mai înalt de generalitate. Astfel, construirea unui algoritm poate pleca de la analiza paşilor ce se parcurg în situaţii concrete (particulare).

- descoperirea ***deductivă,*** prin trecerea de la general la particular. Odată însuşit un algoritm general de rezolvare a unui întreg tip de probleme el se poate particulariza pentru rezolvarea unei anumite probleme (vezi **Divide et Impera** în [4],[5],[6].)

h) Problematizarea (predare prin rezolvare de probleme) - este o metodă didactică ce constă din punerea în faţa elevului a unor dificultăţi create în mod deliberat, în depăşirea cărora, prin efort propriu, elevul învaţă ceva nou.

Situaţiile problematice se pot ordona pe tipuri:

- când există un dezacord între vechile cunoştinţe ale elevului şi cerinţele impuse de rezolvarea unei noi situaţii;
- când elevul trebuie să aleagă dintr-un lanţ sau sistem de cunoştinţe, chiar incomplete, numai pe cele necesare în rezolvarea unei situaţii date, urmând să completeze datele necunoscute;
- când elevul este pus în faţa unei contradicţii între modul de rezolvare posibil din punct de vedere teoretic şi dificultatea de aplicare a lui în practică;
- când elevul este solicitat să sesizeze dinamica mişcării chiar într-o schemă aparent statică;
- când elevului i se cere să aplice, în condiţii noi, cunoştinţe deja asimilate.

i) Modelarea - este o metodă de predare-însuşire în cadrul căreia mesajul ce urmează transmis este cuprins într-un model.

j) Algoritmizarea este definită ca metoda de predare-învăţare constând din utilizarea şi valorificarea algoritmilor. Ea înseamnă găsirea de către profesor a înlănţuirii necesare a operaţiilor fiecărei activităţi de învăţat, ce se pretează unei astfel de ordonări, în anumite situaţii

algoritmii pot fi construiţi de către elevi (de exemplu pentru rezolvarea unei probleme cu ajutorul calculatorului).

Astfel, modul de concepere şi realizare a unei lecţii presupune stabilirea unor paşi ce urmează să se desfăşoare într-un timp limitat.

De exemplu o lecţie mixtă (care urmăreşte realizarea mai multor scopuri: comunicare, sistematizare, fixare, verificare) se poate desfăşura conform următorilor paşi:

- moment organizatoric;
- verificarea conţinuturilor însuşite (verificarea temei, cunoştinţelor, deprinderilor, priceperilor dobândite de elev);
- pregătirea elevilor pentru receptarea noilor cunoştinţe;
- precizarea titlului şi obiectivelor
- comunicarea/însuşirea noilor cunoştinţe, printr-o strategie metodică adaptată obiectivelor, conţinutului temei şi elevilor, şi prin utilizarea acelor mijloace de învăţământ care pot facilita şi eficientiza realizarea acestei sarcini didactice;
- fixarea şi sistematizarea conţinuturilor predate prin repetare şi exerciţii aplicative;
- explicaţii pentru continuarea învăţării acasă şi pentru realizarea temei.

În rezolvarea de probleme algoritmizarea are două aspecte: prezentând modele de rezolvări de probleme punem la îndemâna elevului un instrument simplu şi operativ în care găseşte un sprijin permanent. Pe de altă parte prin repetare şi conştientizare se pot decripta alte soluţii algoritmice, ajungându-se, astfel, la o fază nouă de învăţare, cea euristică, de descoperire şi probare a unor noi scheme de procedură.

Informatica este o disciplină care are printre obiective însuşirea de algoritmi pentru rezolvarea diferitelor tipuri de probleme, alegerea dintre mai mulţi algoritmi de rezolvare a unei probleme pe cel optim în raport cu specificul problemei, resursele calculatorului, etc. Algoritmii sunt elaboraţi cu scopul programării calculatoarelor, prin codificarea lor într-un limbaj de programare.

k) Brainstorming-ul este o metodă ce urmăreşte formarea la elevi a unor calităţi imaginative, creative şi chiar a unor trăsături de personalitate (spontaneitate, altruism). Se poate utiliza în special în cadrul orelor de laborator, când, după enunţarea problemei de rezolvat elevii sunt invitaţi să emită soluţii, în mod spontan, fără preocuparea validităţii acestora. Evaluarea propriu-zisă a soluţiilor preconizate se realizează după un anumit timp, prin compararea şi selectarea ideilor valabile sau prin compararea acestora din punct de vedere al complexităţii algoritmului.

l) Studiul individual - când, în urma exerciţiilor rezolvate împreună cu profesorul elevului i se cere să rezolve singur probleme asemănătoare prin tema primită pentru acasă sau în cadrul orelor de laborator. Avantajele lucrului individual în cadrul orelor de laborator rezultă şi din respectarea următoarelor principii:

- principiul paşilor mici şi al progresului gradat (stabilirea operaţiilor ce se pot face cu numere raţionale şi implementarea procedurilor sau funcţiilor respective)
- principiul participării active (elevul rezolvă şi pune întrebări atunci când are nelămuriri);
- principiul verificării imediate (algoritmul este verificat imediat prin rularea programului pe calculator);
- principiul respectării ritmului individual (elevul gândeşte şi implementează algoritmul de rezolvare a problemei în funcţie de posibilităţile sale).

În desfăşurarea lecţiei sau a altor forme de activitate un rol important revine verificării şi evaluării cunoştinţelor şi deprinderilor prin funcţiile pe care acestea le îndeplinesc:

- moment al conexiunii inverse;

- măsură a progresului realizat de elevi;

- valoarea motivaţională;

- moment al autoevaluării, al formării conştiinţei de sine;

- factor de reglare a activităţii atât pentru profesor cât şi pentru elev.

Evaluarea şi notarea presupun un cadru de raportare, de comparaţie, întâi se poate considera elevul în raport cu ***el însuşi,*** remarcând progresele sau regresele. În al doilea rând, putem clasifica elevul având în vedere prestaţia ***grupului-clasă*** sau a unui grup reprezentativ (se promovează în acest caz competiţia cultivând motivaţia pentru reuşita individuală), în al treilea rând, evaluăm elevul în raport cu obiectivele, respectiv, temele înscrise în programe, estimând distanţa ce-l separă de aceste obiective. Acoperirea unui obiectiv operaţional, respectiv stăpânirea unei teme are în vedere o serie de trepte ce constituie criterii de evaluare ce vor cuantifica standardele de nivel minim, mediu sau superior.

Aceste trepte sunt:

- a avea informaţia necesară, a cunoaşte din memorie;

- a opera cu aceste cunoştinţe într-un context similar celui de la lecţie;
- a integra cunoştinţele în sisteme, a face asocieri/combinări intra şi inter sistemice, respectiv între noţiuni din capitole vecine ori îndepărtate;

- a opera într-un context nou, diferit de cel de la lecţie (element de creativitate).

Astfel, dacă obiectivul propus este rezolvarea unei probleme atunci standardul minim este atins prin reproducerea şi înţelegerea enunţului, standardul mediu prin identificarea metodei şi tehnicii de realizare iar standardul superior prin identificarea algoritmului optim conform criteriilor prestabilite.

Metode de verificare si evaluare

a) Observaţia curentă, în activitatea de fiecare zi la clasă, în contextul muncii de predare şi de contacte cu elevii, profesorul sesizează contribuţiile spontane ale copiilor, modul cum îşi realizează tema de acasă, calitatea prestaţiilor în munca independentă şi la fixarea cunoştinţelor, manifestări de neatenţie, dificultăţi şi greşeli semnificative etc. fără ca acestea să facă explicit obiectul notării. Aceste constatări pot fi consemnate în fişa de observare în vederea individualizării procesului sumativ de evaluare, dar şi a celui de învăţare.

b) Chestionarea sau examinarea orală - este o formă particulară a conversaţiei prin care se verifică gradul de însuşire a cunoştinţelor şi deprinderilor, priceperea de a interpreta şi prelucra date, stăpânirea operativă a materialului în cadrul aplicaţiilor practice. Prin acest tip de evaluare, chiar dacă presupune un consum mare de timp, profesorul are posibilitatea de a clarifica imediat eventualele erori sau neînţelegeri ale elevului. Clasa nu poate rămâne în afara câmpului de observaţie al profesorului, fiind chemată să participe prin completări, aprecieri, soluţii inedite etc. Pe de altă parte se va evita fracţionarea excesivă a examinării elevului pentru a nu crea acestuia o stare de tensiune sau de dependenţă de profesor.

c) Probele scrise - sunt o modalitate mai economică de verificare (evaluarea unui număr mare de elevi, într-un timp relativ scurt). Elevii au şansa să-şi prezinte achiziţiile educaţiei fără intervenţia profesorului, raportarea rezultatelor se va face la un criteriu unic de validare, constituit din conţinutul lucrării scrise. Elevii pot elabora răspunsurile într-un ritm propriu, fiind avantajaţi cei timizi şi care se exprimă defectuos pe cale orală. Verificarea scrisă implică un feed-back mai slab, în sensul că unele erori sau neîmpliniri nu pot fi eliminate operativ, prin intervenţia profesorului.

Probele scrise pot fi date fără ca elevii să fie avertizaţi, atunci când se urmăreşte verificarea cunoştinţelor din lecţia de zi, sau anunţate şi eventual pregătite prin lecţii recapitulative (spre exemplu tezele de sfârşit de semestru).

Se dă în continuare *un exemplu* de test de evaluare al cărui *obiectiv* este verificarea modului în care elevii şi-au însuşit noţiunile de bază despre recursivitate şi a modului în care reuşesc să opereze cu aceste noţiuni în cadrul unor secvenţe simple de program.

Test

1. Fie schema generală a algoritmului **DIVIDE et IMPERA**, prezentat la pagina 36.

   a) implementaţi procedurile de adunare, respectiv înmulţire a două tablouri numere întregi (în limbajul învăţat Pascal sau C/C++)

   b) scrieţi un program care să testeze procedurile de la pct. a).

2. Fie 2 numere naturale, fiecare având un număr de 100 cifre.

a) implementaţi procedura recursivă de adunare a două numere naturale descrise în enunţ în cadrul unui unit «Naturale »

b) scrieţi un program care să testeze procedura de la pct. a).

d) Probele practice ocupă un loc însemnat în verificarea priceperilor şi deprinderilor formate în cadrul activităţilor aplicative, a capacităţii de aplicare în practică a cunoştinţelor dobândite în cadrul orelor de teorie. Evaluarea practică se poate desfăşura şi prin observaţia sistematică a activităţii fiecărui elev dar şi prin lucrări practice individuale, când fiecare elev primeşte spre rezolvare o anumită problemă, cu un barem de notare stabilit, aprecierea de către profesor făcându - se în urma verificării pe calculator a rezultatelor.

Barem de notare (dat de profesor, elevul având şi posibilitatea de autoevaluare):

- Transpunerea problemei în termeni matematici: *2 puncte*
- Identificare operaţiilor matematice recursive: *2 puncte*
- Specificarea antetelor procedurilor şi funcţiilor corespunzătoare operaţiilor: *2 puncte*
- Descrierea efectivă în partea a procedurilor şi funcţiilor: *2 puncte*
- ilustrarea funcţionalităţii printr-un program de test: *1 puncte*

e) Proiectul reprezintă o activitate de evaluare ce se desfăşoară pe parcursul mai multor ore de laborator completat eventual şi cu o muncă desfăşurată acasă. El constă în dezvoltarea unei aplicaţii a cărei temă este aleasă de către elev sau propusă de profesor şi care utilizează cunoştinţele însuşite de elev pe parcursul unui semestru sau an şcolar. Această modalitate de evaluare constituie în prezent şi una dintre probele pe care elevii trebuie să le susţină în vederea dobândirii atestatului profesional. Evaluarea proiectului se face în urma susţinerii lui vizându-se: modul de prezentare, motivaţia alegerii temei, aspectul comercial, utilitatea sa în practică şi eventual complexitatea algoritmului utilizat.

Un exemplu de temă de proiect ar putea fi:

Unit Polinom

Obiective: realizarea unui unit care să gestioneze prin proceduri recursive un **TAD Polinom.**

Prezentăm, în continuare, un exemplu de proiect didactic pentru tipuri diferite de lecţii, punând în evidenţă unele aspecte prezentate în acest capitol al lucrării.

PROIECT DIDACTIC

Disciplina: informatică

Clasa: a IX-a

Unitatea de învăţare: limbajul **`Turbo Pascal`** sau **`Dev C++`**;

Tema: Recursivitatea în limbajul **`Turbo Pascal`** sau **`Dev C++`.**

Tipul lecţiei: Dobândire de noi cunoştinţe.

Locul de desfăşurare: Laboratorul de informatică.

Nivelul iniţial al clasei:

- elevii şi-au însuşit toate noţiunile teoretice despre structura unui program în **`Turbo – Pascal`** sau **`Dev C++`**;
- elevii şi-au însuşit cunoştinţele despre tipurile de variabile şi declararea acestora;
- elevii şi-au însuşit noţiunile despre instrucţiunea de atribuire.
- elevii şi-au însuşit cunoştinţele despre lucrul cu subprograme recursive (proceduri şi funcţii)

Obiectiv cadru: realizarea de programe recursive.

Obiective referinţă:

- să realizeze programe în limbajul **`Turbo Pascal`** sau **`Dev C++`** pentru rezolvarea de probleme;
- să urmărească etapele de realizare a unei aplicaţii **Divide et Impera**;

Obiective educaţionale:

- **Obiective cognitive:**

  - să definească corect noţiunile teoretice însuşite;
  - să folosească corect unit-urile;

- **Obiective afective:**

  - să aleagă corect programele care se pot rezolva prin utilizarea unit-urilor;
  - să aprecieze corect soluţiile oferite de colegi;
  - să manifeste interes faţă de problemele puse şi dorinţa de învăţare prin
  - descoperirea proprie a adevărului ştiinţific;
  - să se implice cu plăcere şi interes la toate etapele lecţiei;
  - să se bucure de rezultatele muncii depuse;

- **Obiective psihomotorii:**

  - să utilizeze corect noţiunile teoretice însuşite anterior;

  - să-şi formeze deprinderi de lucru specifice temei de studiu;

  - să-şi dezvolte gândirea logică, capacitatea de generalizare şi problematizare;

- **Obiective operaţionale:**

  - să reproducă şi să explice forma generală a unui subprogram recursiv;

  - să justifice necesitatea utilizării unui subprogram recursiv;

  - să înţeleagă exemplele date şi să elaboreze programe corecte pentru aplicaţiile date;

  - să utilizeze corect apelul unui subprogram recuriv;

  - să conceapă (compună) noi aplicaţii (exemple) care necesită utilizarea recursivităţii.

Strategii didactice:

Principii didactice:

- principiul participării şi învăţării active;
- principiul asigurării progresului gradat al performanţei;
- principiul conexiunii inverse (feedback);

Metode de învăţământ:

- metode de comunicare orală: expunere, conversaţie, problematizare;
- metode de acţiune: exerciţiul, învăţare prin descoperire;

Procedee de instruire:

- explicaţia în etapa de comunicare;
- învăţarea prin descoperire, prin rezolvarea de aplicaţii;
- conversaţia de consolidare în etapa de fixare a cunoştinţelor;

Forme de organizare: frontală şi individuală;

Forme de dirijare a învăţării:

- Dirijată de profesor sau prin materiale didactice;
- Independentă;

Resurse materiale:

- Fişe de lucru;
- Calculator;
- Video – proiector/

Material bibliografic:

- MARIANA MILOŞESCU, Informatică – Profilul real, Editura DIDACTICĂ ŞI PEDAGOGICĂ,2004.

Metode de evaluare:

- Evaluare sumativă;
- Evaluare continuă pe parcursul lecţiei(calculator);
- Evaluare formativă.

Desfăşurarea lecţiei:

- **Moment organizatoric:**

- pregătirea lecţiei:
- întocmirea proiectului didactic;
- pregătirea setului de întrebări;
- pregătirea setului de aplicaţii;
- pregătirea temei;
- organizarea şi pregătirea clasei: verificarea frecvenţei;
- captarea atenţiei clasei:
    - anunţarea subiectului pentru tema respectivă;
    - anunţarea obiectivelor urmărite;
    - anunţarea modului de desfăşurare a activităţii;

- **Reactualizarea cunoştinţelor:**

Se realizează un set de întrebări pentru reactualizarea cunoştinţelor teoretice de mai jos:

Întrebare	Răspuns aşteptat
Care sunt structurile învăţate până acum ?	Structura liniara, structura alternativa simpla (if…then …else), structura alternativă generalizată (in case …) , structura repetitivă cu număr cunoscut de paşi (for…), structura cu număr necunoscut de paşi condiţionată anterior (while..), structura cu număr necunoscut de paşi condiţionată posterior(repeat ….until)
Cum se clasifică datele în funcţie de fluxul de informaţie?	Date de intrare, date intermediare, date de ieşire
Care sunt tipurile de date folosite?	Întreg, real, logic, şir de caractere.
Ce sunt tipurile instrucţiuni pretabile la recursivitate ?	Tipurile repetitive se pot înlocui cu structuri recursive, atenţie trebuie regândită problema şi nu transformarea repetiţiei în recursie

- **Comunicarea noilor cunoştinţe:**

Care este forma generală a unui apel recursiv? Cum arată stiva?	Vezi schemele generale de la pagina 7 din această lucrare.

Aplicaţii:

În continuare este prezentat un program complet în care se pot defini câteva funcţii recursive de interes general, de exemplu **Divide et Impera.**

```
Procedura DivImp(V,p,q)                    //V secventa 1<=p<=q<=n
                                           //(p,q) determina o subsecventa
      Daca q-p ≤ ε atunci  Rezolva(V,p,q)
                   altfel  m=(p+q) div 2  //m se poate determina si altfel
                           DivImp(V,p,m)  //apel pentru (p,m)
                           DivImp(V,m+1,q)//apel pentru (m+1,q)
                           Combina(V,p,m,q)
      SfDaca
SfProcedura
```

Observaţii generale

1°. Se divide problema în subprobleme (**2,3,..., a, a** finit);

2°. Stăpâneşte subproblemele prin

- rezolvarea nerecursivă (directă), dacă dimensiunea lor **≤ ε** (prag)
- rezolvă subproblemele recursiv, dacă dimensiunea lor **> ε.**

3°. Combină soluţiile tuturor subproblemelor pentru a ajunge la soluţia finală;

4°. Apel din exteriorul procedurii se face cu **DivImp(V,1,n);**

5°. Pentru sortare avem **ε=1** şi **a=2**, iar procedura **Combina** face interclasarea;

6°. Pentru sortare avem şi varianta recursivă care urmează:

```
Procedura SortMerge(V,p,q)
      Daca p ≤ q atunci
                      m=(p+q) div 2
                      SortMerge(V,p,m)
                      SortMerge(V,m+1,q)
                      InterClaseaza(V,p,m,q)
      sfDaca
sfProcedura
```

Să se ordoneze un vector **X** cu **n** componente. Se va apela subalgoritmul de mai jos sub forma **MergeSort (X,1,n)**:

```
Subalgoritmul MergeSort (X,i,j) este: { Ordonează subşirul xi,…,xj }
      Dacă j-i>ε Atunci
                      m:=(i+j)/2;
                      MergeSort (X,i,m);
                      MergeSort (X,m+1,j);
                      Interclasare (X,i,m, X,m+1,j, X,i,j)
      SfDacă
SfMergeSort.
```

QuickSort (sortare rapidă).

Să se ordoneze un vector **X** cu **n** componente. Se va apela subalgoritmul de mai jos sub forma **QuickSort (X,1,n)** :

```
Subalgoritmul QuickSort (X,i,j) este: { Ordonează subşirul xi,…,xj }
      Dacă j-i>ε Atunci
                  m:=Split (X,i,j);     {trebuie definita procedura Split}
                  QuickSort (X,i,m);
                  QuickSort (X,m+1,j);
      SfDacă
SfQuickSort.
```

Asigurarea feedback-ului şi evaluarea performantei

Se vor implementa 2 din cele 3 proceduri recursive în limbaj de programare.

Tema pentru acasă

A 3-a procedura se va da tema, precum si un program de utilizare.

2.3. Exemple

Prof. Pop Vasile

Recursivitate. Test 1.

BILET NR. 1

1. Considerând funcţia f de mai jos, ce se va întâmpla în urma apelului f(12,8)? Ce reprezintă rezultatul?

```
Function f(a,b:integer):integer;
Begin
If a>b then f:=f(a-b,b)
       else f:=f(a,b-a)
f:=a;
End;
```

(2.50p)

2. Definiţi funcţia Ackermann, pentru două numere întregi a şi b. Calculaţi valoarea funcţiei pentru a=1 şi b=1. Parcurgeţi paşii funcţiei.

(2p)

Scrieţi un subprogram recursiv pentru următoarele probleme(precizaţi şi apelul subprogramului):

3. $\prod_{k=1}^{n}(4k+1)$

(1.50p)

4. a) Determină poziţia elementului maxim din şir

b) Determină numărul elementelor negative din şir

(3p)

Prof. Pop Vasile

BILET NR. 2

1. Considerând funcţia f de mai jos, ce se va întâmpla în urma apelului f(9,6)? Ce reprezintă rezultatul?

```
Function f(i:integer):integer;
Begin
   r:=a mod b;
   f:=f(b,r);
   if r=0 then f:=b;
End;
```

(2.50p)

2. Definiţi funcţia Fibonacci, recursiv pentru un număr n dat. Calculaţi valoarea funcţiei pentru n=10. Parcurgeţi paşii funcţiei.

(2p)

Scrieţi un subprogram recursiv pentru următoarele probleme(precizaţi şi apelul subprogramului):

3. $\frac{1}{1+2}+\frac{1}{1+4}+\frac{1}{1+6}+...+\frac{1}{1+2n}$

(1.50p)

4. a) Determină elementul maxim din şir

b) Determină numărul elementelor pozitive din şir

(3p)

Prof. Pop Vasile

BILET NR. 3

1. Considerând funcţia f de mai jos, ce se va întâmpla în urma apelului f(237)? Ce reprezintă rezultatul?

```
Function f(n:integer):integer;
Var a:integer;
Begin
   a:=0;
   while n<>0 do begin
     a:=a+f(n div 10)+n mod 10;
     n:=n div 10;
                 end;
     f:=a;
End;
```

(2.50p)

2. Definiţi funcţia putere, recursiv, având doi parametrii **a** şi **n** dat. Calculaţi valoarea funcţiei pentru **a=3** şi **n=4**. Parcurgeţi paşii funcţiei.

(2p)

Scrieţi un subprogram recursiv pentru următoarele probleme (precizaţi şi apelul subprogramului):

3. $1 \cdot 2+2 \cdot 3+\ldots+n \cdot (n+1)$

(1.50p)

4. a) Determină dacă un element dat aparţine sau nu şirului

b) Determină dacă şirul este sortat crescător

(3p)

Prof. Pop Vasile

BILET NR. 4

1. Considerând funcţia f de mai jos, ce se va întâmpla în urma apelului f(237)? Ce reprezintă rezultatul?

```
Function f(n:integer):integer;
Var a:integer;
Begin
   a:=1;
   while n<>0 do
    begin
       a:=a+f(n div 10)+1
       n:=n div 10;
    end;
   f:=a;
End;
```

(2.50p)

2. Definiţi funcţia factorial, recursiv pentru un număr n dat. Calculaţi valoarea funcţiei pentru n=8. Parcurgeţi paşii funcţiei.

(2p)

Scrieţi un subprogram recursiv pentru următoarele probleme(precizaţi şi apelul subprogramului):

3. $2+2^2+2^3+\ldots+2^{20}$

(1.50p)

4. a) Generează şirul **v** cu **n** valori aleatoare

b) Tipăreşte din şirul **v** primele **n** elemente

(3p)

Recursivitate. Test 5.

BILET NR. 5

1. Considerând funcţia f de mai jos, ce se va întâmpla în urma apelului f(237)? Ce reprezintă rezultatul?

```
Function f(n:integer):integer;
Var a:integer;
Begin
   a:=0;
   while n<>0 do
     begin
       a:=a+f(n div 10)+n mod 10;
       n:=n div 10;
     end;
   f:=a;
End;
```

(2.50p)

2. Definiţi funcţia putere, recursiv, având doi parametrii **a** şi **n** dat. Calculaţi valoarea funcţiei pentru **a=3** şi **n=4**. Parcurgeţi paşii funcţiei.

(2p)

Scrieţi un subprogram recursiv pentru următoarele probleme(precizaţi şi apelul subprogramului):

3. $1\cdot2+2\cdot3+\ldots+n\cdot(n+1)$

(1.50p)

4. a) Determină dacă un element dat aparţine sau nu şirului

c) Determină dacă şirul este sortat crescător

(3p)

Recursivitate. Test 6.

BILET NR. 6

1. Considerând funcţia f de mai jos, ce se va întâmpla în urma apelului f(237)? Ce reprezintă rezultatul?

```
Function f(n:integer):integer;
Var a:integer;
Begin
   a:=1;
   while n<>0 do
    begin
       a:=a+f(n div 10)+1
       n:=n div 10;
    end;
   f:=a;
End;
```

(2.50p)

2. Definiţi funcţia factorial, recursiv pentru un număr n dat. Calculaţi valoarea funcţiei pentru n=8. Parcurgeţi paşii funcţiei.

(2p)

Scrieţi un subprogram recursiv pentru următoarele probleme(precizaţi şi apelul subprogramului):

3. $2+2^2+2^3+\ldots+2^{20}$

(1.50p)

4. a) Generează şirul **v** cu **n** valori aleatoare

b) Tipăreşte din şirul **v** primele **n** elemente

(3p)

Recursivitate. Test 7.

BILET NR. 7

1. Considerând funcţia f de mai jos, ce se va întâmpla în urma apelului f(12,8)? Ce reprezintă rezultatul?

```
Function f(a,b:integer):integer;
Begin
If a>b then f:=f(a-b,b)
       else f:=f(a,b-a)
f:=a;
End;
```

(2.50p)

2. Definiţi funcţia Ackermann, pentru două numere întregi a şi b. Calculaţi valoarea funcţiei pentru a=1 şi b=1. Parcurgeţi paşii funcţiei.

(2p)

Scrieţi un subprogram recursiv pentru următoarele probleme(precizaţi şi apelul subprogramului):

3. $\prod_{k=1}^{n}(4k+1)$

(1.50p)

4. a) Determină poziţia elementului maxim din şir;
b)Determină numărul elementelor negative din şir

(3p)

Recursivitate. Test 8.

BILET NR. 8

1. Considerând funcţia f de mai jos, ce se va întâmpla în urma apelului f(9,6)? Ce reprezintă rezultatul?

```
Function f(i:integer):integer;
Begin
   r:=a mod b;
   f:=f(b,r);
   if r=0 then f:=b;
End;
```

(2.50p)

2. Definiţi funcţia Fibonacci, recursiv pentru un număr n dat. Calculaţi valoarea funcţiei pentru n=10. Parcurgeţi paşii funcţiei.

(2p)

Scrieţi un subprogram recursiv pentru următoarele probleme(precizaţi şi apelul subprogramului):

3. $\frac{1}{1+2}+\frac{1}{1+4}+\frac{1}{1+6}+...+\frac{1}{1+2n}$

(1.50p)

4. a) Determină elementul maxim din şir

b) Determină numărul elementelor pozitive din şir

(3p)

BIBLIOGRAFIE

[1] Donald E. Knuth, ***Fundamental Algorithms***, volumul 1 din ***The Art of Computer Programming***, Addison-Wesley, 1968. Second edition, 1973.

[2] Cormen, Thomas H.; Leiserson, Charles, Rivest, Ronald R.: ***Introducere în algoritmi***, Cluj-Napoca, Editura Computer Libris Agora, 2000).

[3] Lupea, Mihaiela; Cioban, Vasile, ***POO şi structuri de date în C++, Teorie, exemple şi probleme***, Editura RISOPRINT, Cluj – Napoca, 2011.

[4] Prejmerean, Vasile, ***Algoritmică şi programare,*** Litografia Universităţii "Babeş - Bolyai", Cluj – Napoca, 1999.

[5] Lazăr, Ioan, ***Structuri de date***, Litografia Universităţii "Babeş - Bolyai", Cluj – Napoca, 1999.

[6] Motogna, Simona, ***Metode şi tehnici de proiectare a algoritmilor***, Litografia Universităţii "Babeş - Bolyai", Cluj – Napoca, 1999.

[7] Eugen, Popescu; Mihaela, Codres; Ecaterina, Boarna: ***Limbajul Pascal-Teorie şi Aplicaţii***, (***Partea a II-a***), Editura Else, Bucuresti, 2007.

[8] Eugenia, Kalisz; Irina, Athanasiu; Valentin, Cristea: ***Initiere in Turbo Pascal***, Editura Teora, Bucuresti, 2006.

DECLARAŢIE DE AUTENTICITATE PE PROPRIE RĂSPUNDERE

Subsemnatul (a) _____, înscris (ă) la examenul pentru obţinerea Gradului didactic I, seria 2010-2012, specializarea _____, prin prezenta, certific că lucrarea metodico-ştiinţifică cu titlul:

RECURSIVITATEA ÎN PROGRAMARE

conducător ştiinţific _____

este rezultatul propriilor mele activităţi de investigare teoretică şi aplicativă şi prezintă rezultatele personale obţinute în activitatea mea didactică.

În realizarea lucrării am studiat doar surse bibliografice consemnate în lista bibliografică, iar preluările din diferitele surse, inclusiv din alte lucrări personale, au fost citate în lucrare.

Prezenta lucrare nu a mai fost utilizată în alte contexte evaluative – examene sau concursuri.

Data: _____ Semnătura:

_____