Facultatea de Matematica si Informatica, Universitatea din Bucuresti

---

Recursivitate

1. *Tipuri de subprograme recursive*
2. Verificarea corectitudinii subprogramelor recursive
3. Complexitatea timp
4. Criterii de alegere; avantaje si dezavantaje
5. Greseli tipice in elaborarea subprogramelor recursive
6. Culegere de probleme

1. *Tipuri de subprograme recursive*
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

(1) bazate pe funcţii (primitiv) recursive unare, (funcţia se apelează pe ea însăşi în mod direct, ca în cazul calculării factorialului);

(2) bazate pe funcţii (primitiv) recursive de mai multe argumente (ca în cazul calculării cmmdc pentru două numere naturale);

acestea sunt cunoscute sub numele de recursivitate liniară directă:

```
int cmmdc1 (int x, int y)
{ if (x==y) return x;
     else
       if (x>y) return cmmdc1(x-y, y);
          else  return cmmdc1(x, y-x);
}
int cmmdc2 (int x, int y)
{ if  (0==y) return x;
     else
        return cmmdc2(y, x%y);
}
```

(3) bazate pe funcţii care se apelează una pe alta (aşa numita recursivitate indirectă),

```
int a (int n)
{
    if  (0==n) return 1;
    return a(n-1)+b(n-1);
}
int b (int n)
{
    if  (0==n) return 1;
    return a(n-1)*b(n-1);
}
```

1. *Tipuri de subprograme recursive*
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Criterii de alegere**
4. **Avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

(4) bazate pe funcţii care au nevoie de mai multe valori anterioare pentru a calcula valoarea curentă (aşa numita recursivitatea neliniară sau în cascadă, ca în cazul determinării unui element al şirului lui Fibonacci după formula:

```
int fibonacci (int n)
{
  if (n<=1) return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

- se face intr-un mod similar verificării corectitudinii subprogramelor nerecursive.
- este mult simplificată de forma subprogramului (care permite utilizarea comodă a metodei inducţiei matematice complete):
  - se verifică mai întâi dacă toate cazurile particulare (de terminare a apelului recursiv) funcţionează corect;
  - se trece apoi la o verificare formală, prin inducţie, a funcţiei recursive corespunzătoare, pentru restul cazurilor.

1. *Tipuri de subprograme recursive*
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

Exemplificare: calculul factorialului unui număr

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

Demonstrarea corectitudinii: doi pasi:

- pentru n = 0, valoarea 1 returnată de program este corectă;
- dacă n>1 atunci, presupunând corectă valoarea returnată pentru (n-1), prin înmulţirea acesteia cu n se obţine valoarea corectă a factorialului numărului natural n, valoare returnată de subprogram.

  În ambele situaţii este satisfăcută condiţia de oprire.

1. *Tipuri de subprograme recursive*
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

Exemplul 1

Fie $f : N \to N$, $f(n) = 5n^3 + 2n^2 + 22n + 6$;

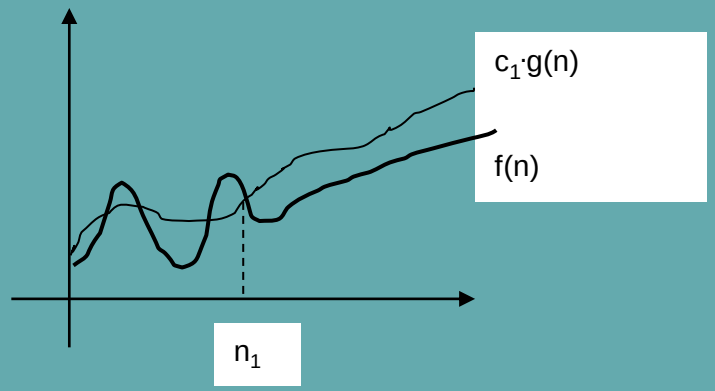
spunem că f tinde asimptotic către n^3 şi notăm acest lucru cu $\boldsymbol{O}(n^3)$

Definiţie 3

Fie $f, g : N \to R_+$

(i) $f(n) = O(g(n))$ şi citim "f(n) este de ordin cel mult g(n)" sau "f(n) este O mare de g(n)" $\Leftrightarrow$

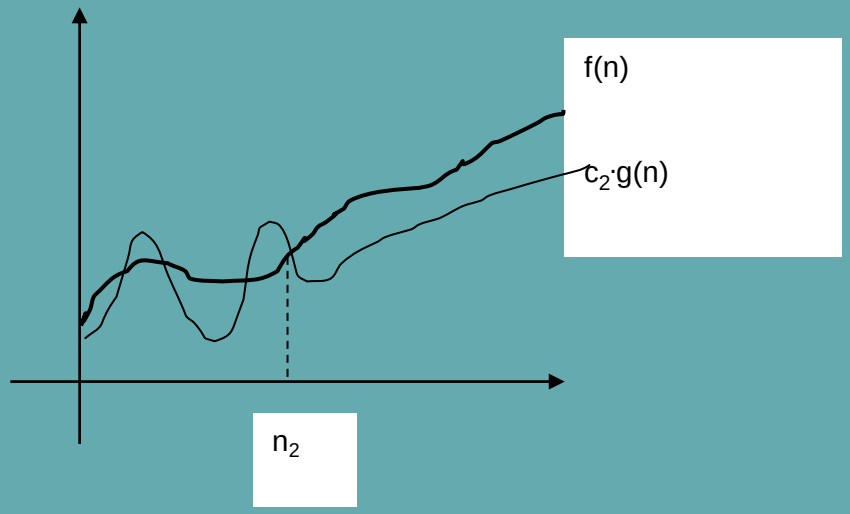
($\exists$) constantele $c_1 > 0$ şi $n_1 \in N$ astfel încât $f(n) \leq c_1.g(n)$, ($\forall$) $n \geq n_1$.



1. Tipuri de subprograme recursive
2. Verificarea corectitudinii subprogramelor recursive
3. Complexitatea timp
4. Criterii de alegere; avantaje si dezavantaje
5. Greseli tipice in elaborarea subprogramelor recursive
6. Culegere de probleme

(ii) $f(n) = \Omega(g(n))$ şi citim "f(n) este de ordin cel puţin g(n)" sau "f(n) este omega mare de g(n)" $\Leftrightarrow$

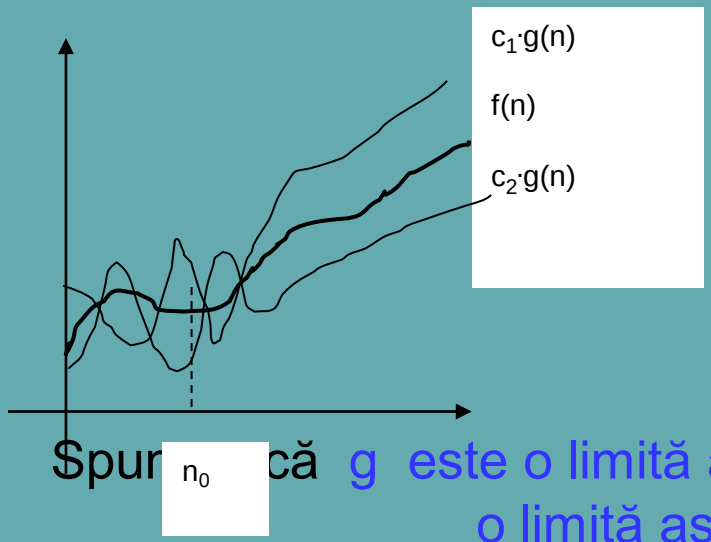
($\exists$) constantele $c_2 > 0$ şi $n_2 \in N$ astfel încât $f(n) \geq c_2.g(n)$, ($\forall$) $n \geq n_2$.



1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

(iii) $f(n) = \Theta(g(n))$ şi citim "f(n) este de ordin g(n)" sau "f(n) este theta de g(n)" $\Leftrightarrow$

$f(n) = O(g(n))$ şi $f(n) = \Omega(g(n))$.



Spun că g este o limită asimptotică superioară, o limită asimptotică inferioară, respectiv o limită asimptotică pentru f.

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

Exemplul 2

Revenim la notaţia ***O*** şi la funcţia polinomială

$$f(n) = 5n^3 + 2n^2 + 22n + 6 \Rightarrow$$

$f(n) = \boldsymbol{O}(n^3)$, de exemplu, pentru $c_1 = 6$ şi $n_1 = 10$;

$f(n) = \boldsymbol{O}(n^4)$, de exemplu, pentru $c_1 = 1$ şi $n_1 = 6$ sau pentru $c_1 = 36$ şi $n_1 = 1$;

$f(n) \neq \boldsymbol{O}(n^2)$,

presupunem prin absurd că există $c_1 > 0$ şi $n_1 \in N$ astfel încât

$5n^3 + 2n^2 + 22n + 6 \leq c_1.n^2, (\forall)\ n \geq n_1 \Leftrightarrow$

$5n^3 + (2- c_1).n^2 + 22n + 6 \leq 0$ etc.

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

Exemplul 3

Fie $f_1 : \mathbf{N} \rightarrow \mathbf{N}$, $f_1(n) = 3n.\log_2 n + 5n.\log_2(\log_2 n) + 2 \Rightarrow$

$f_1(n) = \boldsymbol{O}(n.\log n)$ pentru că log n domină log(log n).

Analog, $f_2(n) = \boldsymbol{O}(n^2) + \boldsymbol{O}(n) \rightarrow f_2(n) = \boldsymbol{O}(n^2)$

pentru că $\boldsymbol{O}(n^2)$ domină $\boldsymbol{O}(n)$.

Observaţia 1

A) Specificarea bazei logaritmilor nu este necesară ea intervenind cu cel mult un coeficient constant, conform formulei: $\log_a x = \frac{\log_b x}{\log_b a}$

B) Analog, nici specificarea bazei exponenţialei nu este necesară pentru că:

$$\forall x > 0 : \; x = 2^{\log_2 x} \rightarrow n^c = 2^{c \cdot \log_2 n} \Rightarrow$$

$2^{\boldsymbol{O}(\log n)}$ este o limită superioară pentru n^c, unde c este o constantă oarecare.

Evident, şi $2^{\boldsymbol{O}(n)}$ este o limită superioară pentru n^c.

Observaţia 2

Limitele asimptotice de tipul n^c se numesc limite polinomiale.

Limitele asimptotice de tipul 2^{n^δ} se numesc limite exponenţiale.

Limitele asimptotice de tipul $k.n$ se numesc limite lineare.

Limitele asimptotice de tipul $\sqrt{n}$ se numesc limite sublineare.

Observaţia 3

Pe lângă notaţiile **O** şi **Ω** mai există şi notaţiile **o** şi **ω**, obţinute din Definiţia 2 prin înlocuirea inegalităţii ≤ cu inegalitatea strictă <, sau

$$f(n) = o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

1. Tipuri de subprograme recursive
2. Verificarea corectitudinii subprogramelor recursive
3. Complexitatea timp
4. Criterii de alegere; avantaje si dezavantaje
5. Greseli tipice in elaborarea subprogramelor recursive
6. Culegere de probleme

Exemplul 4

$\sqrt{n} = o(n)$

n = ***o***(n.log log n)

n.log log n = ***o***(n.log n)

n.log n = ***o*** (n^2)

n^2 = ***o***(n^3)

Propozitia 1

(i) Notaţiile O, Ω, Θ, o, ω sunt tranzitive:

$f(n) = O(g(n))$ & $g(n) = O(h(n))$ $\rightarrow$ $f(n) = O(h(n))$

etc.;

(ii) Notaţiile O, Ω, Θ, sunt reflexive, dar nu şi o, ω

$f(n) = O(f(n))$ dar $f(n) \neq o(f(n))$ etc.;

(iii) Notaţia Θ este simetrică dar nu şi celelalte notaţii:

$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$.

Propozitia 2

Notaţiile O, Ω etc. pot fi manipulate algebric, dar cu precautie (de ex. egalitatea din formula (5) are loc intr-un singur sens: de la stanga la dreapta):

1. $c.O(f(n)) = O(f(n))$;
2. $O(f(n)) + O(f(m)) = O(f(n))$;
3. $O(O(f(n))) = O(f(n))$;
4. $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$;
5. $O(f(n) \cdot g(n)) = f(n) \cdot O(g(n))$.

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

Pentru a calcula complexitatea timp a unui algoritm (nr de pasi executati) vom proceda astfel:

- pentru fiecare etapa calculam:
  - nr de pasi necesari executarii ei,
  - de cate ori se executa etapa respectiva,
  - inmultim cele 2 valori;
- adunam valorile obtinute pentru etapele algoritmului si aplicam regulile calculului asimptotic.

	Nr de pasi	*Nr de executii*	*Total*
Etapa nr. 1	n^2	k	$k.n^2$
Etapa nr. 2	n^k	n	n^{k+1}
……			

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

Alegerea variantei recursive / iterative pentru scrierea unui program presupune:

- cunoasterea fiecarei metode in parte si a particularitatilor sale;
- cunoasterea tehnicii de transformare a recursivitatii in iteratie;
- studiu profund al structurilor de date optime reprezentarii datelor problemei;
- stapanirea tuturor facilitatilor oferite de limbajul de programare in care va fi implementat algoritmul.

1. Tipuri de subprograme recursive
2. Verificarea corectitudinii subprogramelor recursive
3. Complexitatea timp
4. Criterii de alegere; avantaje si dezavantaje
5. Greseli tipice in elaborarea subprogramelor recursive
6. Culegere de probleme

In alegerea intre metoda recursiva si iterativa in elaborarea unui program trebuie sa se tina seama de :

- eficienta oferita programului de catre fiecare dintre variante,
- relatia dintre timpului de rulare si spatiului de memorie necesar
- nevoia de compactizare a programului.

1. Tipuri de subprograme recursive
2. Verificarea corectitudinii subprogramelor recursive
3. Complexitatea timp
4. Criterii de alegere; avantaje si dezavantaje
5. Greseli tipice in elaborarea subprogramelor recursive
6. Culegere de probleme

- Exista o legatura stransa intre recursivitate si structurile de date de tip stiva, arbore, etc folosite in limbajele Borland Pascal si C++ pentru reprezentarea functiilor / procedurilor recursive (insasi definitia stivelor, arborilor, listelor realizandu-se recursiv).
- Iterativitatea minimizeaza complexitatea unor algoritmi
- Deseori, variantele iterative necesita folosirea explicita a structurii de tip stiva, generand astfel un cod extrem de laborios. In aceste situatii se considera solutia recursiva mai eleganta, datorita simplitatii sale.

- Recursivitatea poate fi inlocuita prin iteratie atunci cand recursivitatea este prea adanca sau cand limbajul de programare nu permite implementarea de apeluri recursive.
- Din punctul de vedere al memoriei solicitate, o varianta recursiva necesita un spatiu de stiva suplimentar pentru fiecare apel fata de varianta iterativa.
- Dimensiunea stivei trebuie aleasa astfel incat sa poata permite memorarea elementelor pentru toate iteratiile.

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

- Exemple de probleme pentru care solutia recursiva este mai putin intuitiva decat cea iterativa:
- Sa se calculeze suma cifrelor unui numar natural *n*.

```
// suma cifrelor unui numar - varianta iterativa
#include<iostream.h>
 #include<conio.h>
 int suma(int n)
  { int s=0;
    while(n!=0)
     { s=s+n%10;
       n=n/10; }
     return s;  }
  void main()
  { int n;
    cout<<" n = "; cin>>n;
    int n1=n;
    cout<<" Suma cifrelor numarului "<<n1<<" = "<< suma(n); }
```

```
// suma cifrelor unui numar - varianta recursiva
 #include<iostream.h>
 #include<conio.h>
 int suma(int n)
  {  if(n==0) return 0;
     else return n%10 + suma(n/10);  }
  void main()
  { int n;
     clrscr();
     cout<<" n = "; cin>>n;
     int n1=n;
     cout<<" Suma cifrelor numarului "<<n1<<" = "<< suma(n); }
```

1. Tipuri de subprograme recursive
2. Verificarea corectitudinii subprogramelor recursive
3. Complexitatea timp
4. Criterii de alegere; avantaje si dezavantaje
5. Greseli tipice in elaborarea subprogramelor recursive
6. Culegere de probleme

- Sa se verifice egalitatea a doua siruri de caractere introduse de la tastatura .

```
// egalitatea a 2 stringuri - varianta iterativa
var a,b:string;
    egal:boolean;
    i:byte;
begin
writeln('introduceti sirurile :');
readln(a); readln(b);
egal:=true;
if length(a)<>length(b) then egal:=false
   else
        for i:=1 to length(a) do
             if a[i]<>b[i] then egal:=false;
if egal then writeln('sunt egale')
          else writeln('nu sunt egale')
end.
```

```
// egalitatea a 2 stringuri - varianta recursiva
var a,b:string;
function egal(a,b:string):boolean;
begin
if length(a)<>length(b) then egal:=false
   else
      if a[1]<>b[1]   then egal:=false
        else
           if length(a)=1  then egal:=true
              else
egal:=egal(copy(a,2,length(a)-1),
                 copy(b,2,length(b)-1))
end;
begin
writeln('introduceti sirurile :');
readln(a);  readln(b);
if egal(a,b) then writeln('sunt egale')
                 else writeln('nu sunt egale')
end.
```

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

- ***Motivul:*** Dificultatea descoperirii formulei de recurenta nu de iteratie
- Să se scrie un program care calculează suma

$$S_n = 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \ldots + \frac{1}{2^n}$$

- Indicatie: In algoritmul recursiv se vor folosi 2 funcţii definite astfel:

$$2^n = \begin{cases} 1 & ,n = 0 \\ 2 \cdot 2^{n-1} & ,n \geq 1 \end{cases} \qquad S_n = \begin{cases} 1 & ,n = 0 \\ \frac{1}{2^n} + S_{n-1} & ,n \geq 1 \end{cases}$$

1. Tipuri de subprograme recursive
2. Verificarea corectitudinii subprogramelor recursive
3. Complexitatea timp
4. Criterii de alegere; avantaje si dezavantaje
5. Greseli tipice in elaborarea subprogramelor recursive
6. Culegere de probleme

// calculul sumei - varianta iterativa

```
#include<iostream.h>
#include<conio.h>
unsigned n;
void citire()
{cout<<"n=";cin>>n;}
float suma(unsigned n)
{unsigned i;
 long p=1;
 float s=1;
 for(i=1;i<=n;i++)
        {p*=2;
         s+=(float)1/p;}
 return s;}
void main()
{clrscr();
 citire();
 cout<<"suma pentru n="<<n<<" este="<<suma(n);
 getch();}
```

// calculul sumei - varianta recursiva

```
#include<iostream.h>
#include<conio.h>
unsigned n;
void citire()
{cout<<"n=";cin>>n;}
long putere2(unsigned n)
{if(n==0) return 1;
        else return 2*putere2(n-1);}
float suma(unsigned n)
{if(n==0) return 1;
        else
  return (float)1/putere2(n)+suma(n-1);}
void main()
{clrscr();
 citire();
 cout<<"suma pentru n="<<n<<" este="<<suma(n);
 getch();}
```

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. **Culegere de probleme**

Greseli tipice in realizarea subprogramelor recursive:

1. Declararea globala a unor variabile care controleaza adresa de revenire (cazul cand apelurile se fac din interiorul unei structuri repetitive).
2. Declararea ca parametri transmisi prin valoare sau ca variabile locale a unor date structurate (de exemplu de tip tablou) micsoreaza semnificativ adancimea acceptata a recursivitatii, deoarece memorarea lor necesita un spatiu mare de memorie.

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. ***Greseli tipice in elaborarea subprogramelor recursive***
6. **Culegere de probleme**

---

4. Absenta unei conditii de oprire.
5. Exprimarea unei conditii de oprire in care nu intervin nici variabile locale si nici parametrii transmisi prin valoare sau prin referinta ai subprogramului.
6. Marirea dimensiunii problemei prin transmiterea in cadrul autoapelurilor a unor parametri actuali care nu tind sa se aproprie de valorile impuse prin conditia de oprire.
7. Neutilizarea directivei forward (limbajul Borland Pascal) in cazul declararii unor subprograme indirect recursive

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. ***Culegere de probleme re rezolvate si propuse***

Probleme propuse:

1. Calculul recursiv al mediei aritmetice într-un vector.
2. Calculul recursiv al maximului şi al minimului dintr-un vector.
3. Calculul recurent / inductiv al funcţiei:

$$f_n(x)=\begin{cases}\dfrac{1}{1+x}, n=0\\ f_{n-1}(x)+\dfrac{2^n}{1+x^{2^n}}, n>0\end{cases}$$

4. Calculul recurent / inductiv al funcţiei:

$$f_{n,k}=\begin{cases}\dfrac{1}{k(k+1)}, n=1\\ f_{n-1,k}+\dfrac{1}{(k+n-1)(k+n)}, n>1\end{cases}$$

unde $k \geq 1$ este un număr natural.

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. ***Culegere de probleme re rezolvate si propuse***

---

5. Valoarea funcţiei Ackerman în variantă recursivă şi nerecursivă pentru perechea (*m*, *n*), unde funcţia este definită astfel:

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

6. Să se realizeze parcurgerea arborilor binari (preordine, postordine, inordine), recursiv şi iterativ.
7. Se consideră şirul a_1, a_2, …, a_n în progresie aritmetică (i.e. $\forall n\geq 2$: $a_n=(a_{n-1}+a_{n+1})/2$) . Să se calculeze recursiv suma dată prin formula de recurenţă:

$$S_n:\begin{cases} S_1=\dfrac{1}{\sqrt{a_1}+\sqrt{a_2}}, n=1 \\ S_n=S_{n-1}+\dfrac{1}{\sqrt{a_n}+\sqrt{a_{n+1}}}, n\geq 2 \end{cases}$$

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. ***Culegere de probleme re rezolvate si propuse***

---

8. Se consideră şirul $a_1, a_2, \ldots, a_n$ in progresie geometrica (i.e. $\forall n \geq 2$: $a_n = \sqrt{a_{n-1} \cdot a_{n+1}}$). Să se calculeze recursiv suma dată prin formula de recurenţă:

$$S_n : \begin{cases} S_1 = \dfrac{\sqrt{a_1}}{\sqrt{a_2} - \sqrt{a_1}}, n = 1 \\ S_n = S_{n-1} + \dfrac{\sqrt{a_n}}{\sqrt{a_{n+1}} - \sqrt{a_n}}, n \geq 2 \end{cases}$$

9. Să se calculeze recursiv suma: $\dfrac{1}{a_1 a_2 \ldots a_k} + \dfrac{1}{a_2 a_3 \ldots a_{k+1}} + \ldots + \dfrac{1}{a_n a_{n+1} \ldots a_{n+k-1}}$

unde şirul $(a_n)_{n \geq 1}$ este reprezentat printr-o progresie aritmetică.

10. Să se calculeze recursiv suma

$a_1 a_2 \ldots a_k + a_2 a_3 \ldots a_{k+1} + \ldots + a_{n+1} a_{n+2} + \ldots + a_{n+k}$ unde şirul $(a_n)_{n \geq 1}$ este reprezentat printr-o progresie aritmetică.

1.Tipuri de subprograme recursive
2. Verificarea corectitudinii subprogramelor recursive
3. Complexitatea timp
4. Criterii de alegere; avantaje si dezavantaje
5. Greseli tipice in elaborarea subprogramelor recursive
6. Culegere de probleme re rezolvate si propuse

---

11. Se consideră matricea: $A=\begin{pmatrix}1 & 1 & a\\ 0 & 1 & 1\\ 0 & 0 & 1\end{pmatrix}$. Calculaţi recursiv A^n, $n\geq 2$.

12. Idem pentru matricele: .

$$A=\begin{pmatrix}0 & e^{x} & e^{-x}\\ e^{-x} & 0 & 0\\ 0 & 0 & 0\end{pmatrix}, \quad A=\begin{pmatrix}0 & e^{x}+1 & e^{-x}+1\\ e^{-x} & 0 & 1\\ e^{x} & 0 & 0\end{pmatrix}, \quad A=\begin{pmatrix}-a & a & a\\ a & -a & a\\ a & a & -a\end{pmatrix}.$$

13. Fiind dată matricea $A=\begin{pmatrix}x & 0 & x\\ 0 & y & 0\\ x & 0 & x\end{pmatrix}$, să se calculeze $\sum_{k=1}^{n} A^k$.

14. Să se calculeze în variantă recursivă şi iterativă primele n ($n\geq 5$ dat) elemente din şirul lui Fibonacci.
15. Problema turnurilor din Hanoi.
16. Să se calculeze recursiv a^n, $n\geq 2$.

19. Să se calculeze recursiv C_n^k, $n \geq 2$, $0 \leq k \leq n$.
20. Să se scrie funcţia recursivă pentru calculul sumei cifrelor unui număr natural.
21. Să se scrie funcţia recursivă care citeşte un număr oarecare de caractere şi le afişează in ordinea inversă citirii. Atenţie! nu se lucrează cu şiruri, nu se cunoaşte numărul de caractere, iar sfârşitul şirului este dat de citirea caracterului ‘0’.
22. Se cere calculul recursiv al sumei a *n* numere naturale citite.
23. Să se realizeze programele recursive pentru problema aflării celui mai mare divizor comun pentru calculul simplu, dar şi pentru calculul folosind algoritmul lui Euclid.

1. **Tipuri de subprograme recursive**
2. **Verificarea corectitudinii subprogramelor recursive**
3. **Complexitatea timp**
4. **Criterii de alegere; avantaje si dezavantaje**
5. **Greseli tipice in elaborarea subprogramelor recursive**
6. ***Culegere de probleme re rezolvate si propuse***

---

24. Să se caute rădăcina unei funcţii crescatoare, cunoscându-se următorul rezultat: Fie f o funcţie crescătoare. Dacă f(a) < 0 şi f(b) > 0, aunci f are rădăcină în intervalul [a, b].
25. Să se scrie programul *C/C++* pentru realizarea căutării binare (recursiv şi iterativ) (se caută cheia v intr-un tablou sortat şi se returnează indicele).
26. Drum minim. Între n oraşe există o reţea de drumuri care permite ca dintr-un oraş să se ajungă în oricare dintre celelalte. Între două oraşe, există cel mult un drum direct, de lungime cunoscută, iar timpul necesar parcurgerii unui drum este proporţional cu lungimea sa. Să se scrie programul recursiv pentru determinarea traseului pe care se poate merge între două oraşe date, într-un timp minim.

SUBPROGRAME

Un subprogram este un ansamblu ce poate conține tipuri de date, variabile şi instrucțiuni destinate unei anumite prelucrări (calcule, citiri, scrieri). Subprogramul poate fi executat doar dacă este apelat de către un program sau un alt subprogram.

În limbajul Pascal, subprogramele sunt de 2 tipuri: funcții şi proceduri. În C/C++, este utilizată doar noțiunea de funcție, o procedură din Pascal fiind asimilată în C/C++ unei funcții care returnează *void*.

O altă clasificare a subprogramelor le împarte pe acestea în următoarele categorii:

- predefinite
- definite de către utilizator.

Dintre subprogramele predefinite, amintim subprogramele:

- matematice: *sin*, *cos*, *exp*, *log*;
- de citire/scriere: *read*/*write* (Pascal), *scanf*/*printf* (C/C++)
- de prelucrare a şirurilor de caractere: *substr*, *strlen*, *strcpy* (C/C++).

Exemplu: Conjectura lui Goldbach afirmă că orice număr par > 2 este suma a două numere prime. Să se scrie un subprogram care verifică acest lucru pentru toate numerele pare mai mici decât *N* dat.

Vom rezolva această problemă cu ajutorul subprogramelor. Astfel, programul va conține:

- un subprogram *prim* care determină dacă un număr furnizat ca parametru este prim;
- un subprogram *verifica* prin care se obține dacă un număr furnizat ca parametru verifică proprietatea din enunț;
- programul principal, care apelează subprogramul *verifica* pentru toate numerele pare $2 < k < N$.

Soluția în C este următoarea:

```
#include "stdio.h"
#include "stdlib.h"

bool prim(int n)
{
      int i;
```

```
      for (i = 2; i<= n/2; i++)
            if(n%i==0) return false;
      return true;
}

bool verifica(int n)
{
      int i;
      for (i = 2; i<= n/2; i++)
            if (prim(i) && prim(n - i))
                  return true;
      return false;
}

void main(void)
{
      //declarare variabile
      int n, k;
      //citire valori de intrare
      printf(„n=”);
      scanf(„%d”, &n);
      //prelucrare: verificare daca este indeplinita conditia
      //pentru fiecare k par
      for(k=4; k<=n; k+=2)
            if(!verifica(k))
            {
                  printf("%d nu verifica!\n", k);
                  //ieşire fara eroare
                  exit(0);
            }
      //afisare rezultat
      printf("proprietatea este indeplinita pentru numere >2 si
             <=%d\n", n);
}
```

Observație: Problema a fost descompusă în altele mai mici. Rezolvarea unei probleme complexe este mai uşoară dacă descompunem problema în altele mai simple.

Avantaje ale utilizării subprogramelor:

- reutilizarea codului (un subprogram poate fi utilizat de mai multe subprograme);
- rezolvarea mai simplă a problemei, prin descompunerea ei în probleme mai simple, care conduc la elaborarea de algoritmi ce reprezintă soluții ale problemei inițiale;
- reducerea numărului de erori care pot apărea la scrierea programelor şi despistarea cu uşurință a acestora.

Elementele unui subprogram sunt prezentate în continuare, cu referire la noțiunea de funcție din C/C++. Pentru funcțiile şi procedurile din Pascal, aceste elemente se pot distinge în mod similar.

- Antetul funcției *prim* este: `bool prim(int n)`
- Numele funcției este `prim`
- Funcția *prim* are un parametru *n*.
- Funcția are un tip, care reprezintă tipul de date al rezultatului său. Tipul funcției *prim* este *bool*.
  În cazul programelor *C*/*C++*, dacă tipul funcției este *void* înseamnă că funcția nu returnează un rezultat prin nume.
- Funcția are variabila proprie (locală) *i*.
- Funcția întoarce un anumit rezultat (în cazul funcției *prim*, o valoare booleană), prin intermediul instrucțiunii *return*.

Parametrii unui subprogram sunt de două tipuri:
- Parametri formali – cei ce se găsesc în antetul subprogramului;
- Parametri actuali (efectivi) – cei care se utilizează la apel.

De exemplu, în linia:
```
if(!verifica(k))
```
parametrul *k* este un parametru efectiv.

Declararea variabilelor
- Sistemul de operare alocă fiecărui program trei zone distincte în memoria internă în care se găsesc memorate variabilele programului:
  - Segment de date
  - Segment de stivă
  - Heap.

  Există şi posibilitatea ca variabilele să fie memorate într-un anumit registru al microprocesorului, caz în care accesul la acestea este foarte rapid.
- O variabilă se caracterizează prin 4 atribute:
  - Clasa de memorare – locul unde este memorată variabila respectivă; o variabilă poate fi memorată în:
    - segmentul de date (variabilele globale)
    - segmentul de stivă (în mod implicit, variabilele locale)
    - heap
    - un registru al microprocesorului (în mod explicit, variabilele locale).
  - Vizibilitatea – precizează liniile textului sursă din care variabila respectivă poate fi accesată. Există următoarele tipuri de vizibilitate:
    - la nivel de bloc (instrucțiune compusă) (variabilele locale);
    - la nivel de fişier (în cazul în care programul ocupă un singur fişier sursă) (variabilele globale, dacă sunt declarate înaintea tuturor funcțiilor);
    - la nivel de clasă (în legătură cu programarea orientată pe obiecte).
  - Durata de viață – timpul în care variabila respectivă are alocat spațiu în memoria internă. Avem:
    - durată statică – variabila are alocat spațiu în tot timpul execuției programului (variabilele globale);

        - durată locală – variabila are alocat spațiu în timpul în care se execută instrucțiunile blocului respectiv (variabilele locale);
        - durată dinamică – alocarea şi dezalocarea spațiului necesar variabilei respective se face de către programator prin operatori şi funcții speciale.
    - Tipul variabilei.
- În C++ variabilele pot fi împărțite în 3 mari categorii: locale, globale şi dinamice.

Transmiterea parametrilor

Parametrii actuali trebuie să corespundă celor formali, ca număr şi tip de date. Tipul parametrilor actuali trebuie fie să coincidă cu tipul parametrilor formali, fie să poată fi convertit implicit la tipul parametrilor formali.

- Pentru memorarea parametrilor, subprogramele folosesc segmentul de stivă, la fel ca pentru variabilele locale.
- Memorarea se face în ordinea în care parametrii apar în antet.
- În cadrul subprogramului, parametrii transmişi şi memorați în stivă sunt variabile. Numele lor este cel din lista parametrilor formali.
- Variabilele obținute în urma memorării parametrilor transmişi sunt variabile locale.
- La revenirea în blocul apelant, conținutul variabilelor memorate în stivă se pierde.

Transmiterea parametrilor se poate realiza prin intermediul a două mecanisme:

- prin valoare
- prin referință.

Transmiterea prin valoare este utilizată atunci când dorim ca subprogramul să lucreze cu acea valoare, dar, în prelucrare, nu ne interesează ca parametrul actual (din blocul apelant) să rețină valoarea modificată în subprogram.

În toate apelurile din exemplul precedent, transmiterea parametrilor se realizează prin valoare.

Observație: Dacă nu ar exista decât acest tip de transmitere, ar fi totuşi posibil să modificăm valoarea anumitor variabile care sunt declarate în blocul apelant. Acest lucru se realizează în situația în care am lucra cu variabile de tip pointer.

Exemplu: Să se scrie o funcție care interschimbă valorile a două variabile. Transmiterea parametrilor se va face prin valoare.

```
#include <iostream.h>
void interschimba(int *a, int *b)
{
      int aux = *a;
      *a = *b;
      *b = aux;
}

main()
```

```
{
      int x = 2, y = 3;
      interschimba(&x, &y);
      cout<< x << " " << y;
}
```

Observație: În limbajul C, acesta este singurul mijloc de transmitere a parametrilor.

Transmiterea prin valoare a tablourilor permite ca funcțiile să returneze noile valori ale acestora (care au fost modificate în funcții). Explicația este dată de faptul că numele tabloului este un pointer către componentele sale. Acest nume se transmite prin valoare, iar cu ajutorul său accesăm componentele tabloului.

Exemplu: Să se scrie o funcție care inițializează un vector transmis ca parametru.

```
#include <iostream.h>

void vector (int x[10])
{
      for (int i = 0; i < 10; i++)
            x[i] = i;
}

main()
{
      int a[10];
      vector(a);
      for (int i=0; i < 10; i++)
            cout << a[i] << " ";
}
```

Transmiterea prin referință este utilizată atunci când dorim ca la revenirea din subprogram variabila transmisă să rețină valoarea stabilită în timpul execuției programului.

În acest caz, parametrii actuali trebuie să fie referințe la variabile. La transmitere, subprogramul reține în stivă adresa variabilei.

La compilare, orice referința la o variabilă este tradusă în subprogram ca adresare indirectă. Acesta este motivul pentru care în subprogram putem adresa variabila normal (nu indirect), cu toate că, pentru o variabilă transmisă, se reține adresa ei.

Exemplu: Să se scrie o funcție care interschimbă valorile a două variabile. Transmiterea parametrilor se va face prin referință.

```
#include <iostream.h>
void interschimba(int &a, int &b)
{
      int aux = a;
```

```
      a = b;
      b = aux;
}

main()
{
      int x = 2, y = 3;
      interschimba(x, y);
      cout<< x << " " << y;
}
```

Acest program, scris în limbajul Pascal, are următoarea formă:

```
program transm_referinta;
var x, y: integer;
procedure interschimba(var a,b: integer);
var aux: integer;
begin
aux := a;
a := b;
b := aux;
end;

procedure nu_interschimba(a,b: integer);
var aux: integer;
begin
aux := a;
a := b;
b := aux;
end;

begin
x := 31;
y := 21;
interschimba(x, y);
writeln(x, ' ', y);
writeln('---------');
nu_interschimba(x, y);
writeln(x, ' ', y);
end.
```

Supraîncărcarea funcțiilor

În C++ există posibilitatea ca mai multe funcții să poarte acelaşi nume. În această situație, funcțiile trebuie să fie diferite fie ca număr de parametri, fie ca tip. În acest din urmă caz, este necesară o condiție suplimentară: parametrii efectivi să nu poată fi convertiți implicit către cei formali.

Exemplu: Să se scrie o funcție supraîncărcată care afişează aria pătratului, respectiv a dreptunghiului, în funcție de numărul de parametri.

```
#include <iostream.h>

void arie(double latura)
{
      cout << "aria patratului este "<< latura * latura;
}

void arie(double lung, double lat)
{
      cout << "aria dreptunghiului este "<< lung * lat;
}

main()
{
      arie(3);
      arie (4, 7);
      double d = 7;
      arie(&d);
}
```

Exerciții:

1) Se da un sir de numere intregi. Se cauta un subsir cu lungimea cuprinsa intre l si u, format din elemente consecutive ale sirului initial, cu suma elementelor maxima.
2) Se considera un numar natural n.Determinati cel mai mic numar m care se poate obtine din n, prin eliminarea unui numar de k cifre din acesta fara a modifica ordinea cifrelor ramase.
3) Se citesc de la tastatura un numar natural n si un sir de numere naturale. Sa se scrie un program care afiseaza indicii i si j care indeplinesc urmatoarele conditii:
   a) $1 \le i < j \le n$;
   b) $a_i > a_k, a_j > a_k, \forall k, i+1 \le k \le j-1$;
   c) *diferenta* $j-i$ *este* max *ima*.

Probleme propuse:

1) Se considera *N* numere intregi care trebuie repartizate in *p* grupuri. Grupurile sunt identificate prin numere naturale cuprinse intre *1* si *p*. Repartizarea in grupuri trebuie sa se realizeze astfel incat suma numerelor din oricare grup *i* sa fie divizibila cu numarul total de numere care fac parte din grupurile identificate prin valori cuprinse intre *i* si *p*.
2) Consideram ca toate punctele de coordonate intregi din plan (coordonate mai mici de 1000) sunt colorate in negru, cu exceptia a *n* puncte care sunt colorate in rosu. Doua puncte rosii aflate pe aceeasi linie verticala (au aceeasi ordonata sau aceeasi abscisa) pot fi unite printr-un segment. Coloram in rosu toate punctele de coordonate intregi de pe acest segment. Repetam operatia cat timp se obtin puncte rosii noi. Cunoscand coordonatele celor n puncte care erau initial rosii, aflati numarul maxim de puncte rosii care vor exista in final.
3) Se considera o matrice binara de dimensiune *m x n* (elementele matricei sunt 0 sau 1). Se cere sa se determine aria maxima care poate fi acoperita de doua dreptunghiuri care contin numai elemente cu valoare 0 (dreptunghiurile nu se pot suprapune).

in.txt	*out.txt*
6 8 1 **0 0 0 0 0 0 0** 1 **0 0 0 0 0 0 0** 1 1 1 **0 0 0** 1 1 0 0 1 **0 0 0** 1 1 0 0 1 **0 0 0** 1 1 0 0 1 1 1 1 1 1	23