RECURSIVITATEA - METODĂ PRINCIPALĂ DE PROIECTARE A ALGORITMILOR

CURS 2- 09.03.2021

Titular: Şef. Lucr. Dr. Mat. Cărbureanu Mădălina

Universitatea Petrol-Gaze din Ploieşti

CURS 2- RECURSIVITATEA - METODĂ PRINCIPALĂ DE PROIECTARE A ALGORITMILOR

- Noțiunea de recursivitate;
- Funcții;
- Funcții recursive directe;
- Funcții recursive indirecte;
- Mecanismul recursivității;
- Oportunitatea utilizării recursivității;
- Aplicații.

Noțiunea de recursivitate

- **Recursivitatea** → metodă principală de proiectare a algoritmilor;
- **Recursivitatea** → mecanism care permite scrierea unor funcții care se **autoapelează.**

- **Autoapel** → ce se întâmplă la un nivel, are loc și la nivelele următoare, dar cu alte valori ale parametrilor;
- **Recursivitatea** → mecanism utilizat în dezvoltarea programelor destinate **algoritmilor recursivi;**
- **Recursivitatea** → procesul prin care o funcție se apelează pe sine;
- **Recursivitatea** → procesul prin care o entitate de un anumit tip se definește, descrie sau prelucrează prin intermediul entităților de același tip;
- **Recursivitatea** → mecanism general de elaborare, respectiv de proiectare a algoritmilor apărut din necesitatea transcrierii directe a formulelor matematice recursive.

Funcții

- **Funcții** → permit împărțirea unui program în mai multe părți mai mici;
- **Funcții recursive** → au la bază înțelegerea modului în care se realizează apelul funcțiilor;
- **Declararea unei funcții**: *tip nume_funcție ([listă_parametrii_formali]);*
- **Definirea unei funcții:**
- *tip nume_funcție([listă_parametrii_formali])*
- { //*corpul funcției;*
  }
- **Apel funcție**: *nume_funcție([listă_parametrii_efectivi]);*

Funcții recursive directe

- **Funcție recursivă directă** → funcție care se autoapelează direct, adică în corpul ei apar apeluri la ea însăși;

- **Formă funcție recursivă directă** *f()*:

```
tip f (listă_parametrii)
{ ..............
    apel f ( );
}
```

- **! funcția apelantă și funcția apelată coincid.**

Exemple de funcţii recursive directe

- **factorial**(n) $= \begin{cases} 1, & \text{dacă } n=0; \\ n \times \text{factorial}(n-1), & \text{dacă } n>0; \end{cases}$

- **fibonacci**(n) $= \begin{cases} 1, & \text{dacă } n=0 \text{ sau } n=1; \\ \text{fibonacci}(n-1)+\text{fibonacci}(n-2), & \text{dacă } n>1; \end{cases}$

- **suma**(n, a) $= \begin{cases} 0, & \text{dacă } n=0; \\ a[n]+\text{suma}(n-1, a), & \text{dacă } n>0; \end{cases}$

- **cmmdc**(a, b) $= \begin{cases} a, & \text{dacă } b=0; \\ \text{cmmdc}(b, a\%b), & \text{dacă } b>0; \end{cases}$

- $\mathbf{h(n,a,b,c)}=\begin{cases} a,b, & \text{dacă } n=1; \\ h(n-1, a,c,b), ab, h(n-1, c,b,a), & \text{dacă } n>1; \end{cases}$

- $\mathbf{s(}n\mathbf{)}=\begin{cases} 0 & \text{, dacă } n=0; \\ n\%10+s(n/10), & \text{dacă } n>0; \end{cases}$

- $\mathbf{baza2(}n\mathbf{)}=\begin{cases} 1, & \text{dacă } n=1; \\ baza2(n/2), \text{ scrie } n\%2, & \text{dacă } n>1; \end{cases}$

Funcții recursive indirecte

- **Funcție recursivă indirectă (mutual recursivă)** → funcție *f()* ce conține o referință la o altă funcție *g()*, funcție ce conține la rândul ei o referință la funcția *f()*;
- **Sintaxă:**

```
tip f (listă_parametrii)
{……
 apel g();
 ……
}
```

```
tip g (listă_parametrii)
{……
 apel f();
 ……
}
```

Exemple de funcții recursive indirecte

- Fie șirurile $a_0, b_0, a_1, b_1, a_2, b_2 \ldots, a_n, b_n$; să se calculeze $a_n = (a_{n-1} \times b_{n-1})/2$ și $b_n = a_{n-1} + b_{n-1}$;

- Fie șirurile definite recurent $a_0 = a$, $b_0 = b$, $a, b > 0$; să se calculeze $a_n = (a_{n-1} + b_{n-1})/2$ și $b_n = \sqrt{a_{n-1} \times b_{n-1}}$;
- Calculul valorii a două funcții *f(x)* și *g(x)*, pentru o valoare citită de la tastatură a argumentului *x*, unde:

$f(x)=\begin{cases} 3\times g(x), & \text{pt. } x<2; \\ 2\times x, & \text{pt. } x\geq 2; \end{cases}$	$g(x)=\begin{cases} 3\times x+f(x+2), & \text{pt. } x<0; \\ 2, & \text{pt. } x\geq 0; \end{cases}$

Mecanismul recursivității

- **Recursivitatea** → utilizează structura de date numită **stivă;**
- **Funcția recursivă** → utilizează implicit **stiva program**, adică în acest caz stiva este gestionată de limbaj și nu de către programator;
- La apelul recursiv al unei funcții se depun în stivă următoarele elemente:
  - valorile parametrilor transmiși prin valoare;
  - adresele parametrilor transmiși prin referință;
  - valorile variabilelor locale.

- **! La fiecare autoapel al unei funcții se salvează în stivă starea curentă a execuției sale**, adică:
- adresa de revenire în program (adresa instrucțiunii cu care va continua execuția);
- valorile variabilelor locale;
- valorile parametrilor actuali (efectivi).
- **! Orice funcție recursivă trebuie să conțină o condiție (respectiv o instrucțiune if) prin a cărei evaluare, să se realizeze, după un timp, ieșirea din recursivitate,** adică oprirea autoapelului și execuția instrucțiunilor amânate prin autoapel. **În caz contrar, funcția se va autoapela la nesfarșit;**
- ! Oprirea autoapelului presupune **coborârea pas cu pas în stivă și preluarea valorilor corespunzătoare de pe fiecare nivel al stivei.**

Exemplu: calculul sumei elementelor unui şir de numere întregi *a*

- $\text{suma}(n, a) = \begin{cases} 0, & \text{dacă } n=0; \\ a[n]+\text{suma}(n-1, a), & \text{dacă } n>0; \end{cases}$

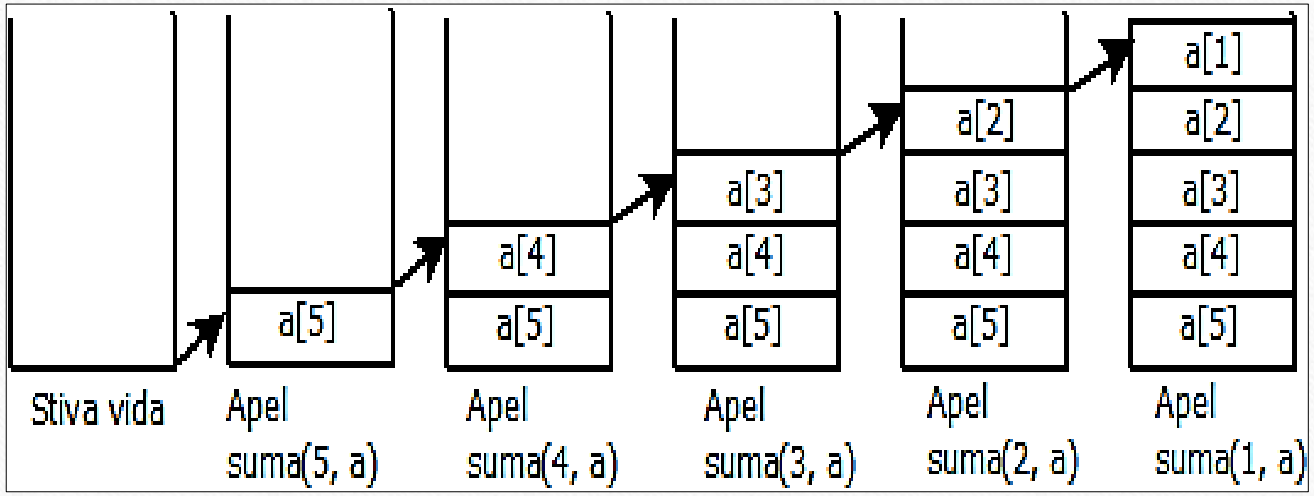
- transcrierea funcţiei dată de relaţia de mai sus, în **pseudocod**:

```
(1) funcţie suma(n, a[10]);
(2) început
(3) dacă (n==0) atunci suma←0;
(4)    altfel suma←a[n]+suma(n-1, a);
(5) sfârşit dacă;
(6) sfârşit;
```

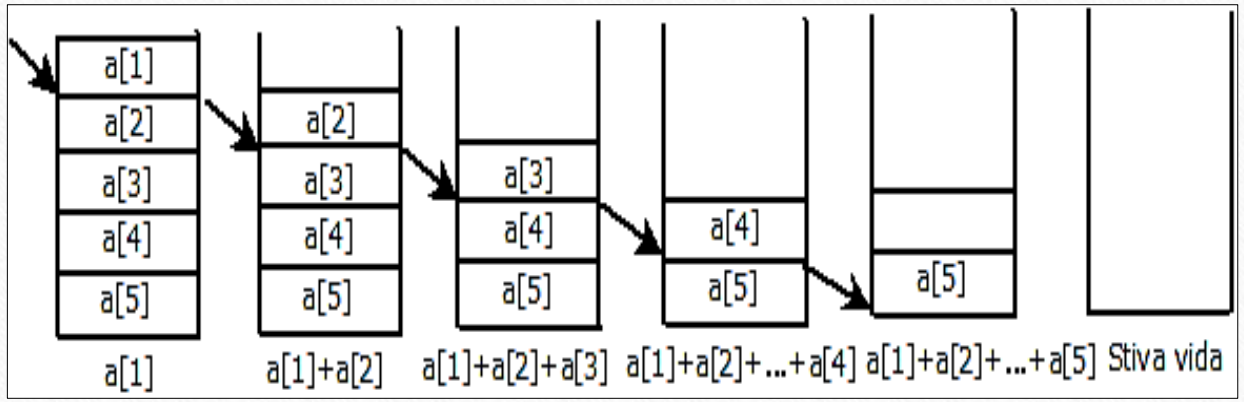
- Se consideră apelul *suma(5, a)*, pentru *n*=5.
- Acest apel va activa succesiv următoarele instrucțiuni:

- *a*[5]+suma(4, *a*); în stivă se adaugă elementul *a*[5];
- *a*[4]+suma(3, *a*); în stivă se adaugă elementul *a*[4];
- *a*[3]+suma(2, *a*); în stivă se adaugă elementul *a*[3];
- *a*[2]+suma(1, *a*); în stivă se adaugă elementul *a*[2];
- *a*[1]+suma(0, *a*); în stivă se adaugă elementul *a*[1];
- suma(0, *a*)=0; oprire autoapel, respectiv ieșire din recursivitate.

- Reprezentarea schematică a stivei, la fiecare apel al funcției *suma(n, a)*.
- Practic, apelul recursiv al funcției presupune deplasarea în stivă pe un nivel superior și depunerea parametrilor actuali (efectivi), în cazul de față *a*[5], *a*[4], *a*[3], *a*[3] și *a*[1].



- **Rezolvarea fiecărui autoapel**, presupune deplasarea pe nivelele inferioare ale stivei și parcurgerea (preluarea) în ordine inversă a parametrilor actuali.
- În fig. de mai jos este prezentat modul în care se realizează rezolvarea apelurilor.



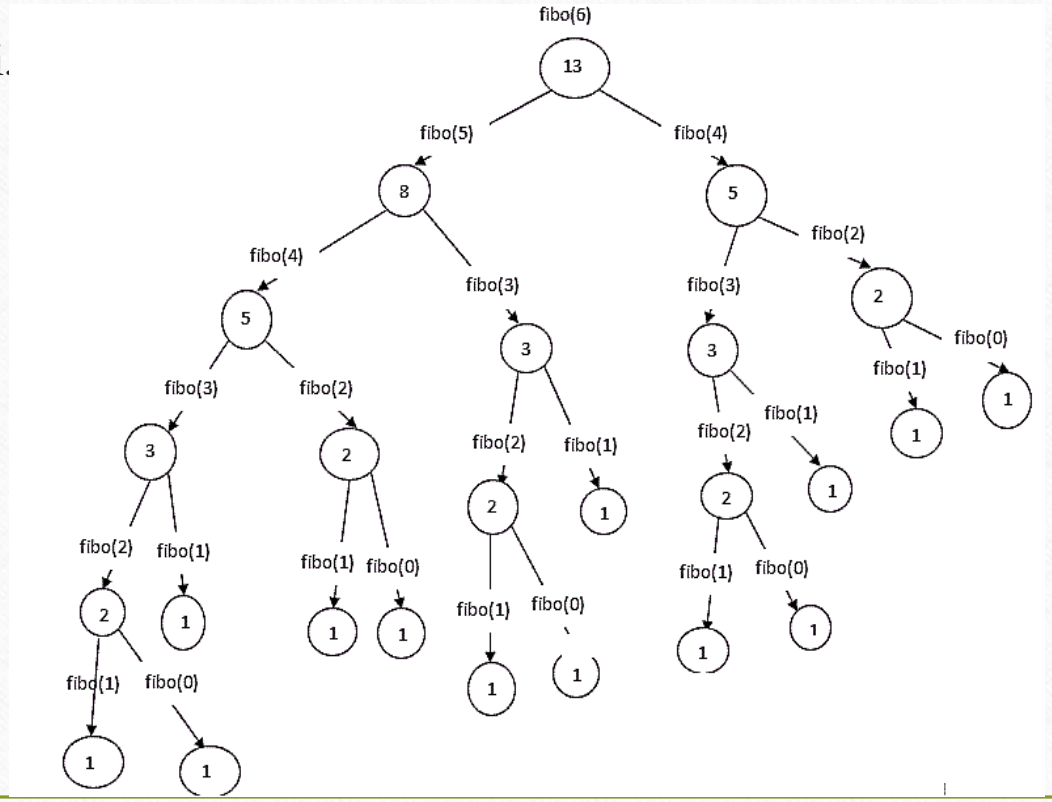
- ! Un **algoritm recursiv corect** trebuie să se termine, adică trebuie să conțină o condiție pentru ieșirea din autoapel, condiție stabilită de către programator; în caz contrar, autoapelul se va realiza la nesfârșit.

Oportunitatea utilizării recursivității

- Recursivitatea este utilă atunci când rezolvarea unei anumite probleme permite o abordare recursivă;

- **! Orice rezolvare recursivă a unei aplicații are asociată și o anumită rezolvare iterativă. Reciproca (orice rezolvare iterativă are asociată și o posibilă rezolvare recursivă) însă nu este valabilă, adică nu orice rezolvare iterativă are asociată și o posibilă rezolvare recursivă;**
- Abordarea recursivă a rezolvării unei aplicații nu este întotdeauna cea mai bună variantă de lucru, din cauza timpului crescut de calcul generat de adâncimea mare a recursivității și a memoriei consumate;
- Abordarea iterativă este mai eficientă decât aceea recursivă, însă abordarea recursivă conduce de cele mai multe ori la un cod sursă redus și clar (anumite operații dintr-un program sunt mai ușor de înțeles), ce exprimă mult mai bine esența problemei de rezolvat;
  - **Exemplu**: implementarea recursivă a șirului lui Fibonacci.

- **Rezolvarea recursivă este mult mai simplă însă decât cea iterativă;**
  - **Exemple:** problema Turnurilor din Hanoi, sortarea rapidă (Quick Sort).
- Varianta iterativă de rezolvare a unei probleme este preferată uneori în detrimentul variantei recursive din cauza așa-numitei redundanțe, adică executarea inutilă, de mai multe ori a anumitor instrucțiuni;
  - **Exemplu**: șirul lui Fibonacci.



Aplicații

- **Implementarea recursivă a calculului $[\sqrt{x}]$;**

- $\text{parte}(l_i, l_s) = \begin{cases} l_i, & \text{dacă } l_s - l_i \leq 1; \\ \dfrac{l_i + l_s}{2}, & \text{dacă } x = (l_i+l_s)/2 \times (l_i+l_s)/2; \\ \text{parte}\left(l_i, \dfrac{l_i+l_s}{2}\right), & \text{dacă } x < (l_i+l_s)/2 \times (l_i+l_s)/2; \\ \text{parte}\left(\dfrac{l_i+l_s}{2}, l_s\right), & \text{dacă } x > (l_i+l_s)/2 \times (l_i+l_s)/2; \end{cases}$

```
#include<stdio.h>
#include<conio.h>
int x;
int parte(int li, int ls)
{if(ls-li<=1) return li;
 else if(x==((li+ls)/2*(li+ls)/2))
            return (li+ls)/2;
       else if(x<(li+ls)/2*((li+ls)/2))
          return parte(li, (li+ls)/2);
          else return parte((li+ls)/2, ls);}
 void main()
 {clrscr();
  printf("x=");scanf("%d",&x);
  printf("%d", parte(1, x));
  getch(); }
```

- **Implementarea recursivă a Turnurilor din Hanoi;**

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
void hanoi(int n, char a, char b, char c)
{if(n==1) printf("\n %c,%c",a,b);
 else { hanoi(n-1,a,c,b);
        printf("\n %c,%c",a,b);
        hanoi(n-1,c,b,a);
            }}
void main()
{int n;
clrscr();
cout<<"numar de discuri n=";
cin>>n;
cout<<"succesiunea de mutari este:";
hanoi(n, 'a','b', 'c');
getch();}
```

Să vă fie de folos!