Funcții

CURS NR. 2/1

Funcții

Elementele de bază ale programelor

Reprezintă o colecție de instrucțiuni grupate sub un singur nume

- care pot fi executate prin apelul funcției

Structurarea programelor în funcții permite gestiunea mult mai ușoară a codului sursă

- dezvoltarea programelor mari și complexe
  - Scrierea, înțelegerea, modificarea, testarea, întreținerea

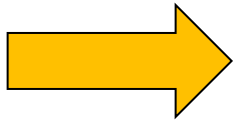
Până acum am scris cod folosind

- Funcția principală main – care nu poate lipsi din nici un program C
- Funcții din biblioteca standard C – apelate din main
  - Prin includerea fișierelor header corespunzătoare

Structura unui program

- Un program conţine una sau mai multe funcţii

```c
#include <stdio.h>

int main()
{
   int a, b, max, min, sum;

   printf("Introduceti doua numere intregi: ");
   scanf("%d %d", &a, &b);

   max = a > b ? a : b;
   min = a < b ? a : b;
   sum = a + b;

   printf("Max(%d, %d) = %d \n", a, b, max);
   printf("Min(%d, %d) = %d \n", a, b, min);
   printf("Sum(%d, %d) = %d \n", a, b, sum);

   return 0;
}
```



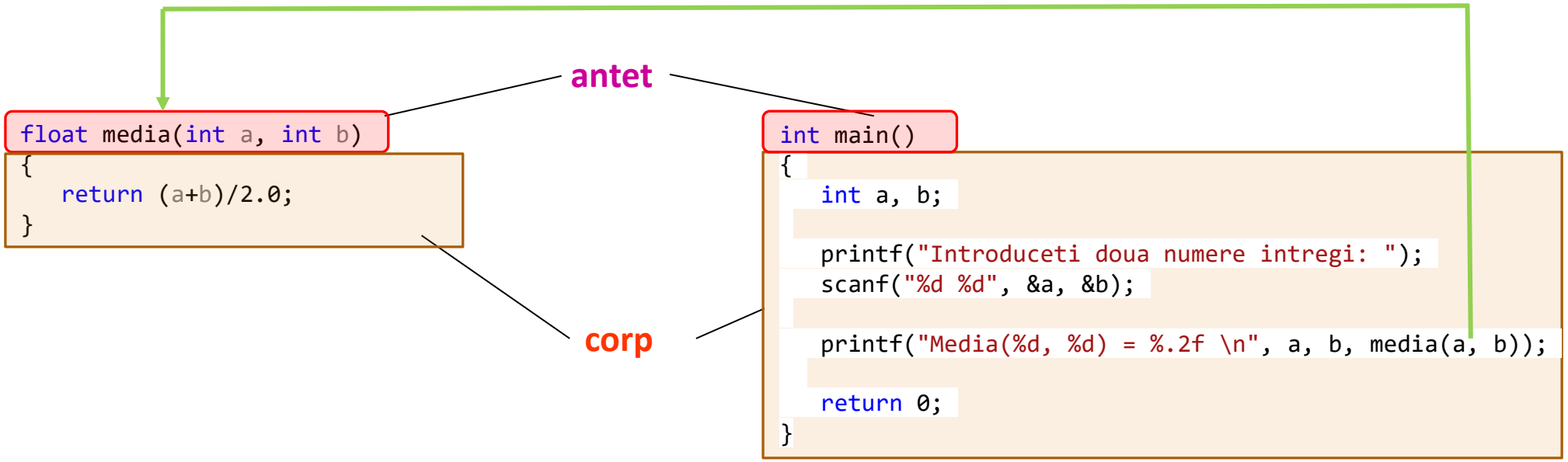
```c
#include <stdio.h>

int maxim(int a, int b)
{
   return a > b ? a : b;
}

int minim(int a, int b)
{
   return a < b ? a : b;
}

int suma(int a, int b)
{
   return a + b;
}

int main() {
   int a, b;

   printf("Introduceti doua numere intregi: ");
   scanf("%d %d", &a, &b);

   printf("Max(%d, %d) = %d \n", a, b, maxim(a, b));
   printf("Min(%d, %d) = %d \n", a, b, minim(a, b));
   printf("Sum(%d, %d) = %d \n", a, b, suma(a, b));

   return 0;
}
```

Definirea și apelul funcțiilor

Structura unei funcții

```
tip  nume_functie( lista_param_formali )
{
        corpul_functiei
}
```

Revenirea din funţie: la return sau
la terminarea instrucţiunilor



Antetul funcției


```
tip_returnat nume_funcție (lista_parametrilor_formali)
{
        corpul_funcției
}
```

Tipul valorii returnate de funcție

- ex. int, float, char, etc.
- dacă funcția nu returează nici o valoare atunci scriem **void**

Restricție

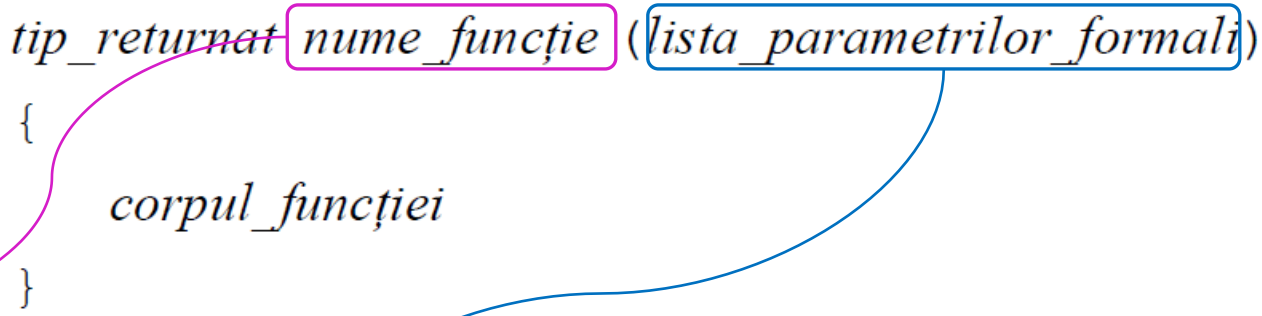
- nu se pot returna tablouri sau alte funcții

Dacă lipsește tipul returnat

- C89 interpretează implicit că funcția returnează un întreg
- C99 și C11 semnalează eroare

Returnarea unei valori și/sau revenirea din funcție se poate face prin instrucțiunea return

Antetul funcției



Numele unic al funcției

- analogic cu identificatorul unei variabile

- Specificată între paranteze rotunde și conține **tipul de date** și **identificatorul** fiecărui parametru al funcției, parametrii fiind separați prin virgulă

```
int suma_elemente(int v[], int n)
```

- Tipul datei trebuie trecut la fiecare parametru separat, chiar dacă mai mulți parametri au același tip

```
double media(int a, int b, int, c)
```

- Dacă lista este vidă, adică funcția nu primește parametri, atunci se trece tipul void, sau se lasă lista vidă

```
int main(void)
```

```
int main()
```

Apelul funcției

!

Apelul se realizează prin specificarea numelui și între paranteze rotunde a valorilor pentru fiecare dintre parametri - adică lista parametrilor actuali

nume_funcție (lista_parametrilor_actuali)

Apelul unei funcții este o expresie

- Poate participa ca operand în cadrul unor expresii, sau ca parametru actual la apelul altor funcții
- Poate forma o instrucțiune independentă sub forma unei instrucțiuni expresie, adică apelul este sufixat cu caracterul ; (punct și virgulă)

Antet:

```
int factorial(int n)
```

Apel:

```
a = factorial(10);
```

Exemplu

```c
/*  Nume: max_functii.c
*  Scop: determina maximul dintre doi intregi
*/

#include <stdio.h>

int maxim(int a, int b)
{
// folosim operatorul conditional
return a > b ? a : b;
}

int main() {
int a, b;

printf("Introduceti doua numere intregi: ");
scanf("%d %d", &a, &b);

printf("Maximul dintre %d si %d este: %d \n", a, b, maxim(a, b));

return 0;
}
```

Rezultatul unei rulări a acestui program este:

```
Introduceti doua numere intregi: 5 7
Maximul dintre 5 si 7 este: 7
```

Observații

Parametrii cu care se definește funcția sunt numesc *parametri formali*, sau pur și simplu *parametri*

```
int maxim(int a, int b) {
    return a > b ? a : b;
}
```

Valorile cu care se apelează funcția se numesc *parametri actuali* sau *argumente*

```
int main() {
   int a, b;

   printf("Introduceti doua numere intregi: "); scanf("%d %d", &a, &b);
   printf("Maximul dintre %d si %d este: %d \n", a, b, maxim(a, b));
   return 0;
}
```

- parametrii formali și cei actuali corespunzători pot avea nume identice (ca în exemplul de mai sus) sau nume diferite
- ordinea în care parametrii formali sunt definiți în cadrul antetului este ordinea în care trebuie specificate argumentele funcției

C89 permite ca apelul funcției să preceadă definirea sau declararea acesteia

C99 și C11 impune ca definiția sau declararea funcției să preceadă apelul funcției

Apelul funcțiilor

!

Când o funcţie este apelată, atunci se efectuează următorii pași:

- **Evaluarea** expresiilor din lista **parametrilor actuali** (lista argumentelor). ATENȚIE: evaluare <u>într-o ordine oarecare</u>
  - Rezultatul se convertește la tipul indicat de parametrii formali corespunzători
- Valorile argumentelor sunt **copiate în variabile locale** funcțiilor
- Se **execută corpul** funcției
- Când se ajunge la `return` sau se termină instrucțiunile din corpul funcției se **revine în funcția apelantă** (ex. înapoi în main)
  - Dacă return are parametru, atunci valoarea acestuia este trimisă înapoi funcției apelante

```
int suma_1_n(int n)
{
   int i, suma = 0;
   for (i = 1; i <= n; suma += i, i++)
      ;
   return suma;
}
```

```
int main()
{
   int n;

   printf("Introduceti un numar intreg: ");
   scanf("%d", &n);

   printf("Suma primelor %d numere pozitive este: %d \n",
                                                  n, suma_1_n(n));

   return 0;
}
```

Declararea funcției

!

În momentul apelului unei funcții compilatorul trebuie să cunoască *tipul returnat* și *lista parametrilor formali* - adică **prototipul**

tip_returnat nume_funcție (lista_parametrilor_formali);

Deci: funcția trebuie să fie definită sau declarată înainte de apel

Declararea unei funcții presupune declararea prototipului funcției care constă din antetul funcției urmat de ; (punct și virgulă)

Prototipul trebuie să corespundă cu antetul folosit la definire

- Numele parametrilor poate fi omis

```
void functie(int, float, char);

void functie(int i, float x, char c);
```

← Prototipuri echivalente

Domeniul de vizibilitate

Un identificator (variabilă, constantă etc.) este accesibil **doar** în cadrul blocului în care a fost declarat.

```
int c = 30;
int main(){
   printf("%d\n", c);    // afiseaza 30
   {
      int a = 4;
      printf("%d\n", a);// afiseaza 4
      {
         int a = 5;
         char c = 'A';
         printf("%d\n", a);// afiseaza 5
         printf("%d\n", c);// afiseaza 65
                           // codul ASCII al lui 'A'
      }
   }
   printf("%d\n", c);    // afiseaza 30
   // a nu mai este definit  aici
}
```

Variabila **a** din blocul exterior și variabila globală **c** sunt mascate

O variabilă declarată într-un bloc exterior este disponibilă şi în blocul interior cu excepţia cazului în care este redeclarată – caz în care variabila exterioară este “mascată” temporar.

Atenție: Mascarea poate îngreuna înțelegerea codului și depanarea.

Domeniul de vizibilitate

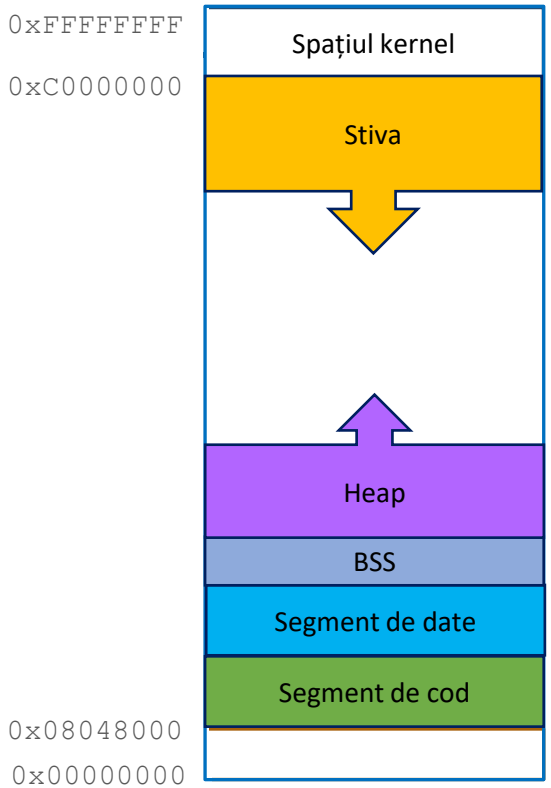
- Variablele definite în cadrul unei funcţii (incluzând main) sunt **locale** acestei funcţii şi nici o altă funcţie nu are acces direct la ele

- Funcția poate primi informații din exterior prin intermediul valorilor globale si al parametrilor

- Funcția poate transmitere informație înapoi la funcţia apelantă prin intermediul valorilor globale, prin parametri de tip pointer sau folosind instrucţiunea **`return`**

Spaţiul de adrese al unui program încărcat în memorie

Locaţiile pot fi diferite pentru alte modele de memorie



Variabile locale,
argumentele funcţiilor,
Adrese de retur

Date alocate dinamic

Date globale și statice neinițializate

Date globale și statice inițializate, constante

Codul programului (imagine binară)

Transmiterea parametrilor: prin valoare

În C, transmiterea parametrilor se face doar prin **valoare**

- Valoarea parametrului actual este copiată în parametrul formal corespunzător din funcţia apelată

```c
#include <stdio.h>

int suma_1_n(int n);

int main() {
   int n, sum;

   printf("Introduceti un numar intreg: ");
   scanf("%d", &n);

   sum = suma_1_n(n);
   printf("Suma primelor %d numere \
           pozitive este: %d \n", n, sum);

   return 0;
}

int suma_1_n(int n)
{
   int suma = 0;
   for ( ; n >=0; suma += n, n--)
   ;
   return suma;
}
```

prototip

```
Introduceti un numar intreg: 10
Suma primelor 10 numere pozitive este: 55
```

n rămâne neschimbat

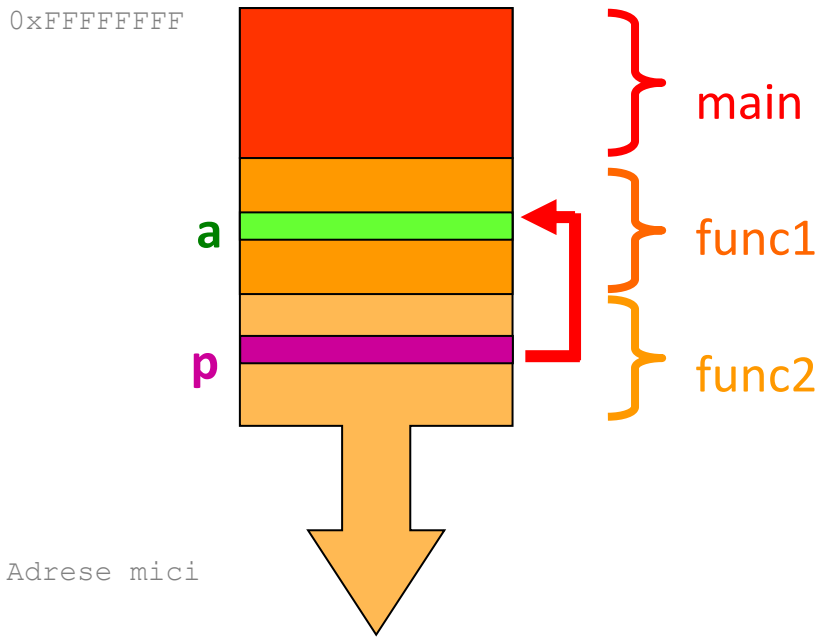
Copia locală a lui n, independentă de n din funcţia apelantă

- Avantaj: protejează variabilele din funcţia apelantă
- Dezavantaj: copierea poate fi ineficientă, ex. tablouri mari → pointeri

Transmiterea parametrilor

Cum ar putea o funcţie să modifice valoarea argumentului primit, astfel încât modificarea să fie vizibilă în funcția apelantă?

→ pointeri




```
int func1(void)
{
   int a;
   func2(&a);
}
```

Trebuie să corespundă

```
int func2(int *p)
{
   ....
}
```

func2 poate modifica a din func1, dacă are un pointer către a (cunoaște adresa lui a)

→ transmitem &a (adresa variabilei a) ca argument funcţiei func2
(care la rândul său nu poate fi modificat de func2 – adresa se transmite prin valoare)

Transmiterea parametrilor

Valoarea lui a este copiată
într-o variabila locală din f1

```
int f1(int a);

int main (void)
{
    int a = 1, b;
    b = f1( a );

    return 0;
}
```

a din main nu se modifică!

Transmitere prin valoare

Şi dacă vrem ca **a** să se modifice?

```
#include<stdio.h>

int main (void)
{
    int a;
    scanf("%d",&a);

    return 0;
}
```

indicăm funcţiei unde găseşte
variabila a: la adresa **&a**

Transmitere folosind pointeri

Exemplu

```c
#include <stdio.h>
void swap(int *, int *);

int main(void)
{
int a = 2, b = 5;

printf("Initial  : a: %d  b: %d\n", a, b);
swap(&a, &b);
printf("Dupa swap: a: %d  b: %d\n", a, b);
return 0;
}


void swap(int *p, int *q)
{
int tmp;
tmp = *p;
*p = *q;
*q = tmp;
}
```

Parametrii formali sunt pointeri la int

La apelul funcţiei transmitem adresa variabilelor

În cadrul funcţiei, folosim operatorul de dereferenţiere * pentru a accesa valoarea stocată la adresa transmisă

*p este echivalent cu a din main

*q este echivalent cu b din main

```
Initial  : a: 2  b: 5
Dupa swap: a: 5  b: 2
```

Exemplu

```c
/*  Nume: cifra_max_min_v2.c
*  Scop: afiseaza cifrea cea mai mare si cea mai mica dintr-un numar intreg
*/
#include <stdio.h>
void afisare_cifra_max_min(int nr, int *cifra_max, int *cifra_min) {
   int rest;
   if (nr == 0) printf("Cifra maxima = cifra minima = 0");
   else    {
      *cifra_max = nr % 10;   // initializarea cu prima cifra
      *cifra_min = *cifra_max;
      nr = nr / 10;
      while (nr != 0) {
          rest = nr % 10;
          if (rest > *cifra_max) *cifra_max = rest;
          else if (rest < *cifra_min) *cifra_min = rest;
          nr = nr / 10;
      }
   }
}

int main() {
   int nr, cifra_max, cifra_min;

   printf("Introduceti numarul intreg: ");
   scanf("%d", &nr);

   afisare_cifra_max_min(nr, &cifra_max, &cifra_min);

   printf("Cifra maxima = %d \
               \nCifra minima = %d\n", cifra_max, cifra_min);
   return 0;
}
```

Argumente de tip vector

Există două moduri echivalente pentru a transmite un tablou unei funcţii:

- Accentuăm faptul că parametrul este un tablou **tip v[ ]**
- Accentuăm faptul că parametrul este un pointer către primul element **tip *v**
- În ambele moduri se transmite adresa primului element din tablou

- Exemple echivalente

```
#include <stdio.h>

double media(int [], int );

int main(void) {
   int v[] = { 5, 3, 11, 7, 6, 2, 8, 1, 9 };
   printf("Media aritmetica a elementelor = %.3f\n",
           media(v, sizeof(v) / sizeof(int)));
   return 0;
}


double media(int v[], int n) {
   int i, suma = 0;
   for (i = 0; i<n; i++)
      suma += v[i];

   return  (double)suma/n;
}
```

```
#include <stdio.h>

double media(int *, int);

int main(void) {
   int v[] = { 5, 3, 11, 7, 6, 2, 8, 1, 9 };
   printf("Media aritmetica a elementelor = %.3f\n",
           media(v, sizeof(v) / sizeof(int)));
   return 0;
}


double media(int *v, int n) {
   int i, suma = 0;
   for (i = 0; i<n; i++)
      suma += *(v+i);

   return  (double)suma / n;
}
```

Instrucțiunea return și funcția exit


```
return expresie ;
```

Instrucțiunea `return` provoacă terminarea funcției și implicit revenirea din funcție în puncul imediat următor de unde s-a efectuat apelul funcției

```
// determinarea semnului
// + pt pozitiv, - pt negativ, 0 pt 0
char semn(int n)
{
   if (n > 0) return '+';
   else if (n < 0) return '-';
   else return '0';
}
```

Funcția predefinită `exit` în schimb are ca și efect terminarea execuției întregului program

```
exit ( expresie ) ;
```

- Pentru exit includem fișierul header `stdlib.h`

Funcții predefinite

Câteva dintre funcțiile predefinite în bibliotecile standard ale limbajului C

Fișiere header	Descriere	Exemple de funcții / macrouri
ctype.h	Gestiunea caracterelor individuale: funcții care clasifică și transformă caractere individuale	tolower, toupper, islower, isupper, isalpha, isdigit, isspace, etc.
float.h	Caracteristicile tipurilor reale: macrouri cu valori specifice sistemului și compilatorului	FLT_MAX, DBL_MAX, FLT_MAX_EXP, DBL_MAX_EXP, etc.
limits.h	Dimensiunile tipurilor întregi: macrouri cu valori specifice sistemului și compilatorului	INT_MAX, INT_MIN, CHAR_BIT, CHAR_MAX, SHRT_MIN, etc.
math.h	Operații și transformări matematice: funții trigonometrice, logaritmice, exponențiale, etc.	sin, cos, tan, exp, mod, log, modf, pow, ceil, floor, fmod, abs, sqrt, etc.
stdio.h	Operații de intrare/ieșire: funcții și macrouri pentru citirea și scrierea din/în fișiere (inclusiv tastatură și ecran)	getchar, putchar, gets, puts, printf, scanf, fopen, fseek, fgetc, fputc, fgets, fputs, fscanf, fprintf, fread, fwrite, fseek, EOF, NULL, FILE, etc.
stdlib.h	Funcții utilitare: conversii, alocarea memoriei, generare de numere aleatoare, sortare, căutare, etc.	atof, atoi, strtol, calloc, malloc, realloc, free, srand, rand, system, qsort, abs, etc.
string.h	Gestiunea șirurilor de caractere și a blocurilor de memorie: funcții de copiere, concatenare, căutare, comparare, etc.	strcpy, strcat, strcmp, strchr, strstr, strtok, memset, memmove, memcpy, etc.
time.h	Gestiunea datei și a timpului: funcții de determinare, setare, formatare a timpului	clock, time, difftime, localtime, CLOCKS_PER_SEC, etc.

Trebuie să includem fișierul header asociat bibliotecii

Clase de stocare

Clase de stocare

Variabilele au o serie de atribute, printre care

- Identificator
- Tip
- Dimensiune
- Valoare

În C fiecare identificator (variabilă / funcție) are trei caracteristici:

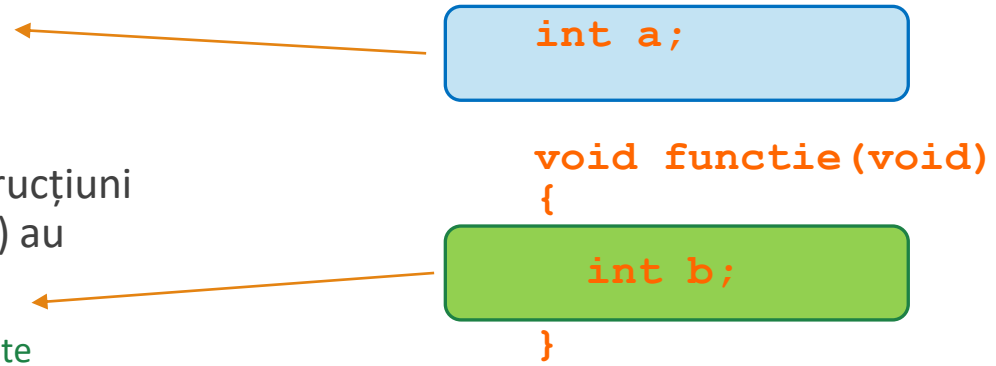
- **Durata de stocare**
  - Perioada în care identificatorul există în memorie
    - Durata de stocare poate varia de la foarte scurtă până la întreaga durată de execuție a programului
- **Vizibilitatea**
  - Locația unde identificatorul poate fi referit (accesat)
    - Vizibilitatea poate fi restricționată la anumite părți (blocuri de instrucțiuni) ale programului sau poate fi globală la nivelul întregului program
- **Legarea**
  - Posibilitatea de accesare directă a identificatorului din afara locației unde este definită
    - De regulă se referă la accesare din alt fișier sursă al programului (vezi programare modulară)

Clase de stocare

Valorile implicite ale acestor caracteristici depind de locația unde este definită variabila

- Variabilele declarate în exteriorul oricărui bloc (exteriorul funcțiilor) au
  - Durata de stocare **statică**
  - Vizibilitate în **întregul fișier** sursă
  - Legare de tip **extern**

- Variabilele declarate în interiorul unui bloc de instrucțiuni (inclusiv corpul unei funcții sau al unei instrucțiuni) au
  - Durata de stocare **automatică**
  - Vizibilitatea doar **în cadrul blocului** unde au fost declarate
  - **Nu** au legare



Caracteristicile implicite pot fi modificate folosind **clase de stocare** explicite

C furnizează 4 clase de stocare:

- **auto** **extern** **register** **static**

Clase de stocare

auto – este clasa de stocare implicită pentru **variabilele locale**

- Cea mai frecvent utilizată clasă de stocare
- De regulă nu se folosește explicit cuvântul cheie auto, deoarece toate variabilele locale sunt automatice, dacă nu se specifică altfel
- Doar variabilele locale pot fi automatice

```
{
    auto int a;
    .....
}
```

- **Durata de stocare**
  - Memoria se alocă în momentul definirii și este implicit dealocată la ieșirea din bloc
- **Vizibilitatea**
  - Din punctul unde sunt definite până la sfârșitul blocului de instrucțiuni unde au fost definite
- **Legarea**
  - Fără legare: nu pot fi accesate în mod direct din afara blocului unde au fost definite

register – categorie specială de variabilă automatică

- Sugerează compilatorului să aloce memorie pentru variabila locală într-un registru al procesorului (în loc de memoria internă)
  - Pentru acces mai rapid la variabilă (de ex. contorul dintr-un ciclu)

```
{
    register int i;
    for (i=0; i < 100; i++) {
        .....
    }
}
```

- **Durata de stocare, vizibilitate și legarea**
  - identică cu cele ale variabilelor automatice

Nota: nu putem obține adresa variabilei register ! (în legătură cu `restrict` (C99))

Clase de stocare

static

- Dacă se aplică asupra unei **varibile locale**
  - **Durata de stocare**: statică
    - Variabila locală va avea durata de stocare extinsă la toată durata programului (durată de stocare statică)
  - **Vizibilitatea**
    - doar în cadrul blocului unde a fost definită variabila
  - **Legarea**: fără legare

Notă: Își păstrează valoarea între aplurile succesive ale funcției. Se inițializează o singură dată, înainte de execuția programului.

Variabila a este inițializată o singură dată: la compilare

```
#include <stdio.h>

void f() {
   static int a = 4;
   int b = 2;
   printf("%d %d \n", ++a, ++b);
}

int main() {
   f();    f();
   return 0;
}
```

```
5 3
6 3
```

- Dacă se aplică asupra unei **varibile globale**
  - **Durata de stocare**: statică
    - Variabila globală va avea durata de stocare egală cu durata programului
  - **Vizibilitatea**
    - doar în cadrul fișierului sursă unde a fost definită
      - Tehnică de ascundere a informației
  - **Legarea**: internă

Clase de stocare

extern – este clasa de stocare implicită pentru toate **variabilele globale**

- Toate variabilele declarate în afara funcțiilor sunt variabile globale și implicit au clasa de stocare extern
- Și variabile locale pot fi declarate cu clasa de stocare extern
- Se folosește pentru partajarea variabilei între mai multe fișiere sursă

- Declarația `extern int a;`
  - informează compilatorul că se dorește accesarea variabilei întregi `a` definită altundeva
    - mai târziu în același fișier sursă, sau mai frecvent într-un alt fișier sursă
  - la declarare nu se alocă spațiu de memorie pentru variabilă

Nota: declarație ≠ definiție

```
extern int a;


void functie(void)
{
  extern int b;
}
```

- Excepție `extern int a = 5;`
  - Declarația `extern` care inițializează variabila este în același timp și definiție

- **Durata de stocare**: statică
  - Variabila globală / locală va avea durata de stocare egală cu durata programului
- **Vizibilitatea**
  - Variabila globală este vizibilă în tot fișierul sursă
  - Variabila locală doar în cadrul blocului unde a fost declarată
- **Legarea**
  - De tip extern - implicit
  - De tip intern – dacă variabila a fost declarată înainte ca fiind statică (în afara funcțiilor)

Nota: o variabilă poate fi **declarată de mai multe ori** în cadrul unui program, dar **definită doar o singură dată**!

Clase de stocare – pentru funcții

C furnizează 2 clase de stocare pentru funcții

- **extern** **și** **static**

```
extern int f(int n);

static int g(int n);
```

extern

- Legare externă
  - Funcția poate fi apelată din alte fișiere sursă
- Clasa de stocare implicită pentru funcții
  - De regulă nu se folosește explicit cuvântul cheie extern, deoarece toate funcțiile sunt de clasa extern, dacă nu se specifică altfel

static

- Legare internă
  - Funcția nu poate fi apelată direct din alte fișiere sursă, este accesibilă doar în cadrul fișierului sursă unde a fost definită
- Beneficiile specificării clasei static pentru funcții
  - Ușurință în întreținerea codului
    - Modificări ulterioare asupra funcției statice nu afectează alte funcții din alte fișiere sursă
  - Reduce poluarea spațiului de nume
    - În diferitele fișiere sursă ale programului pot exista funcții statice cu nume identice – fără a intra în conflict

Parametrii formali ai funcțiilor au implicit aceleași caracteristici ca și variabilele `auto` (pot fi și de clasa `register`)

Clase de stocare - rezumat

Exemple de specificare explicită / implicită a claselor de stocare și tabel cu caracteristicile variabilelor

```
int a;
extern int b;
static int c;

void f(int d, register int e)
{
    auto int g;
    int h;
    static int i;
    extern int j;
    register int k;
}
```

Identi-ficator	Durata de stocare	Vizibilitate	Legare
a	static	Fișier	externă
b	static	Fișier	(de regulă externă, definite într-un alt fișier)
c	static	Fișier	internă
d	automatic	Bloc	fără
e	automatic	Bloc	fără
g	automatic	Bloc	fără
h	automatic	Bloc	fără
i	static	Bloc	fără
j	static	Bloc	(de regulă externă, definite într-un alt fișier)
k	automatic	Bloc	fără

Funcții inline (C99)

!

Apelul funcției inline este înlocuită de compilator cu corpul de instrucțiuni al funcției

- Inline sugerează compilatorului că apelul funcției trebuie eficientizat
  - Compilatorul poate lua în considerare sugestia, sau o poate ignora

Pentru evitarea încărcării specifice apelului de funcție

- Utilizat mai ales în situații când funcția este apelată de foarte multe ori

```
inline int suma(int a, int b)
{
   return (a + b);
}
```

Alternativa C89: macrodefiniții parametrizate

Problema

- Funcțiile au implicit legare externă – pot fi apelate din alte fișiere sursă, DAR când compilatorul întâlnește `inline`, nu va permite accesul la funcție din afara fișierului sursă unde a fost definită

Soluții

- Definirea funcției ca fiind **static**

```
static inline int suma(int a, int b) {
   return (a + b);
}
```

- Definirea funcției într-un fișier header și declararea funcției ca fiind externă în fișierul unde se folosește (vezi exemplu la programare modulară)

Funcții cu număr variabil de argumente

C permite defeinirea funcțiilor cu număr variabil de argumente

- Funcțiile printf() și scanf() sunt exemple foarte populare
- Uneltele pentru scrierea funcțiilor proprii cu număr variabil de argumente sunt disponibile prin includerea fișierului header <stdarg.h>
  - Declară tipul: `va_list` pentru declararea variabilei cu care se parcurg argumentele
  - Definește o serie de macrouri
    - În C89
      - va_start, va_arg, va_end

```
void va_start (va_list ap, paramN);
```

Indică unde încep argumentele cu număr variabil:
după parametrul `paramN` și inițializează variabila `ap`
- Funcția trebuie să aibă cel puțin un parametru ”normal”

```
type va_arg (va_list ap, type);
```

Argumentele pot fi accesate prin apeluri succesive `va_arg`
- Tipul parametrului este determinat de al doilea argument

```
void va_end (va_list ap);
```

Pentru a “curăța” variabila `ap` – o nouă parcurgere poate începe doar după reinițializare folosind va_start

```c
#include <stdarg.h>
#include <stdio.h>

/* this function will take the number of values to average
   followed by all of the numbers to average */
double average(int num, ...) {
   va_list arguments;
   double sum = 0;

   /* Initializing arguments to store all values after num */
   va_start(arguments, num);
   /* Sum all the inputs; we still rely on the function caller
      to tell us how many there are */
   for (int x = 0; x < num; x++) {
      sum += va_arg(arguments, double);
   }
   va_end(arguments);                  // Cleans up the list

   return sum / num;
}
```

num trebuie să fie cel puțin 1

Indică număr variabil de parametrii adiționali

```c
int main() {

   /* this computes the average of 13.2, 22.3 and 4.5
      (3 indicates the number of values to average) */
   printf("%f\n", average(3, 12.2, 22.3, 4.5));

   /* here it computes the average of the 5 values 3.3, 2.2, 1.1, 5.5 and 3.3 */
   printf( "%f\n", average ( 5, 3.3, 2.2, 1.1, 5.5, 3.3 ) );

   return 0;
}
```

La apelul unei funcții cu număr variabil de arg compilatorul efectuează promovarea implicită a argumentelor: **char, short → int,** **float → double**

Reguli și recomandări

- Ordinea de evaluare a argumentelor transmise funcțiilor nu este specificată
  - Atenție la argumentele care în urma evaluării modifică valoarea altor argumente

Valorile returnate de funcțiile `a()` și `b()` depind de valoarea variabilei globale `glob`

```
extern void c(int i, int j);
int glob;

int a(void) {
    return glob + 10;
}

int b(void) {
    glob = 42;
    return glob;
}
```

De evitat:

```
void func(void) {
    c( a(), b() );
}
```

Nu știm ordinea în care se evaluează argumentele funcției `c`

Corect: ordinea este fixată

```
void func(void) {
    int a_val, b_val;

    a_val = a();
    b_val = b();

    c(a_val, b_val);
}
```


Exemplu preluat de pe www.securecoding.cert.org

Reguli și recomandări

O funcție nu poate avea ca tip returnat un tablou

```
int[] funcție(/*…*/) {/* … */}
```

Dar o funcție poate returna un pointer

```
int *funcție(/*… */) {/* … */}
```

Unul dintre argumentele funcției: ex. pointer la un vector primit ca argument

```
int *cifre_nr(int v[], int n) {
   int i = 0;
   while (n != 0) {  v[i] = n % 10;  n = n / 10; i++; }
   return v;
}
```

Sau, mai frecvent:

```
int cifre_nr(int v[], int n) {
   int i = 0;
   while (n != 0) {  v[i] = n % 10;  n = n / 10; i++; }
   return i;
}
```

Elementele vectorului primit ca argument se modifică și în funcția apelantă. Returnăm numărul de cifre

Pointer către zonă alocată dinamic în funcție: ex. pointer la un vector alocat dinamic

```
int *cifre_nr(int n) {
   int i = 0;
   int *v = (int *)malloc(100 *sizeof(int));
   while (n != 0) {
      v[i] = n % 10;
      n = n / 10;
      i++;
   }
   return v;
}
```

Pointer către o variabilă locală statică: ex. pointer la un vector alocat static

```
int *cifre_nr(int n) {
   int i = 0;
   static int v[100];
   while (n != 0) {
      v[i] = n % 10;
      n = n / 10;
      i++;
   }
   return v;
}
```

Dar nu returnăm niciodată pointer către variabilă automatică!!

```
int *cifre_nr(int n) {
   int i = 0;
   int v[20];
   while (n != 0) {   v[i] = n % 10;   n = n / 10;  i++;  }
   return v;
}
```

Surse bibliografice

- K. N. King, C Programming – A Modern Approach, 2nd edition, W. W. Norton & Co., 2008
  - Capitolele 9 și 10
- Deitel & Deitel, C How to Program, 6th edition, Pearson, 2009
  - Capitolul 5

Recursivitate

Recursivitate

!

O funcţie este recursivă dacă se autoapelează

- **direct** – funcţia conţine un apel la ea însăşi sau
- **indirect** – funcţia conţine un apel al altei funcţii care o apelează pe prima

Ideea fundamentală

- Se apică recursivitatea când soluția unei probleme poate fi exprimată pe baza soluțiilor instanțelor mai mici ale aceleiași probleme
  - Când structura de date sau algoritmul de rezolvare sunt inerent recursive

- Problema mai mare / complexă este descompusă în subprobleme mai simple
  - Se continuă descompunerea (**cazul general**) până când ajungem la cea mai mică subproblemă pentru care soluția este ușor de calculat (**cazul de bază**) – poate fi exprimată nerecursiv
  - După care se revine în ordinea inversă și se finalizează rezolvarea fiecărei subprobleme, ducând la rezolvarea problemei originale

Exemplu:

```
int sum(int n)
{
        if ( n <= 1 )
                return n;
        else
                return (n + sum(n - 1));
}
```

Condiție de stopare a autoapelului

Cazul de bază

Cazul general

Iterație vs Recursivitate

Iterația

- Execuția repetată a unui bloc de instrucțiuni atâta timp cât o condiție este îndeplinită – for, while, do-while

Factorial iterativ:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

Recursivitatea

Factorial recursiv:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1 \end{cases}$$

```
factorial (5) = 5 x factorial (4)
              = 5 x (4 x factorial (3))
              = 5 x (4 x (3 x factorial (2)))
              = 5 x (4 x (3 x (2 x factorial (1))))
              = 5 x (4 x (3 x (2 x (1 x factorial (0)))))
              = 5 x (4 x (3 x (2 x (1 x 1))))
              = 5 x (4 x (3 x (2 x 1)))
              = 5 x (4 x (3 x 2))
              = 5 x (4 x 6)
              = 5 x 24
              = 120
```

- Execuția repetată a unei funcții
  - La fiecare execuție a funcției se verifică **condiția de stopare a recursivității**
    - Dacă nu este îndeplinită condiția de stopare, atunci funcția se reapelează (**cazul general**)
      - Se reia execuția funcției de la început – fără ca execuția curentă să se fi terminat
    - Dacă este îndeplinită condiția de terminare a reapelării (**cazul de bază - nerecursiv**), se revine – în ordine inversă ordinii de apelare – în lanțul de apeluri
      - Revenire în punctul imediat următor reapelului și continuarea executării instrucțiunilor rămase suspendate

Funcții recursive

!

Scrierea funcțiilor recursive necesită

- Identificarea **dimensiunii** problemei – argumentele funcției recursive
- Alegerea **condiției de stopare** a recursivității
  - Stoparea autoapelului este esenţială ! Altfel se generează ciclu infinit
- Determinarea **cazului de bază** – ramura nerecursivă
  - Returnarea rezultatului corect / realizează acțiunea corectă (pentru funcții care nu returnează nimic)
- Determinarea **cazului general** – ramura recursivă
  - Ne asigurăm că formula de reapelare este corectă
  - Ne asigurăm că lanțul de autoapeluri va ajunge în final să îndeplinească condiția de stopare
- **Validarea** soluției recursive
  - Prin demonstrație formală sau testând toate cazurile posibile
    - Verificarea pentru cazuri care îndeplinesc condiția de stopare a recursivității + validare formală pentru restul cazurilor

Observații

- În rezolvarea unei probleme alegem soluții recursive dacă
  - Numărul de autoapelurilor este relativ mic comparativ cu dimensiunea problemei
  - Soluția recursivă este comparabilă în complexitate cu soluția iterativă
  - Soluția recursivă este mult mai clară, concisă și naturală decât soluția iterativă

Funcții recursive

Calculul factorialului - exemplu comparativ: implementare recursivă vs. iterativă

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Varianta recursivă reflectă mai compact și clar modul uman de gândire a problemei

Factorial - recursiv

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

Factorial - iterativ

```
int fact_i(int n)
{
    for (produs=1; n>1; --n)
        produs *= n;
    return produs;
}
```

Recursivitatea liniară poate fi transformată simplu în iterație

Funcții recursive

Generarea numerelor Fibonacci

- Sunt necesare 1.4 miliarde de apeluri de funcţie pentru determinarea celui de al 43 număr Fibonacci ! (care are valoarea 433494437)

Recursivitatea poate deveni foarte costisitoare

unsigned long long

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n >= 2 \end{cases}$$

Varianta recursivă: mult mai compactă și clară

Fibonacci - recursiv

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

Ramura nerecursivă

Fibonacci - iterativ

```
int fibonacci(int n)
{
    int i;
    int f1 = 0;
    int f2 = 1;
    int fi;

    if(n == 0)    return 0;
    if(n == 1)    return 1;

    for(i = 2 ; i <= n ; i++ )
    {
        fi = f1 + f2;
        f1 = f2;
        f2 = fi;
    }
    return fi;
}
```

Varianta iterativă: mult mai economică

Revenirea din funcţie se face în punctul următor celui din care s-a făcut apelul

Funcții recursive

Recursivitatea este adesea **ineficientă** (în timp și spațiu) deoarece implică mai multe apeluri de funcţii

- La fiecare reapel necesită alocarea pe stivă a memoriei pentru
  - parametrii funcţiei
  - variabilele locale funcţiei
  - adresa de revenire

Stiva poate crește foarte mult în timp scurt
și se poate umple, ducând la eșuarea programului

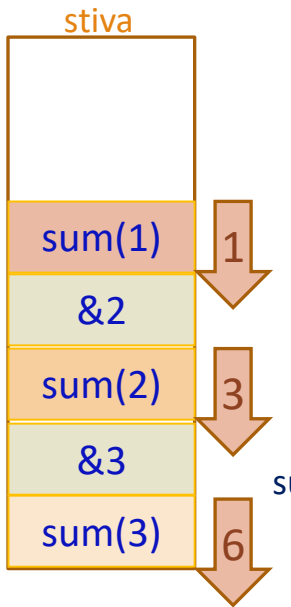
=> Stack overflow

Stiva este o structură LIFO

- Adăugare în / extragere din vârful stivei

Trasarea execuției:
sum(3) apelat din main

```
int sum(int n)
{
    int rez;
    if (n <= 1)
        return n;
    else
        rez = n + sum(n - 1);
    return rez;
}
```



sum(3)
n=3
rez = 3 + sum(2)

Salvează starea funcției apelante și adresa de revenire. Apelează sum(2)

sum(2)
n=2
rez = 2 + sum(1)

Salvează starea funcției apelante și adresa de revenire. Apelează sum(1)

sum(1)
n=1
rez = 1
returnează sum(1) = 1

Extrage adresa de revenire. Se revine în funcția apelantă: sum(2)

sum(2)
rez = 2 + 1
returnează sum(2) = 3

Extrage adresa de revenire. Se revine în funcția apelantă : sum(3)

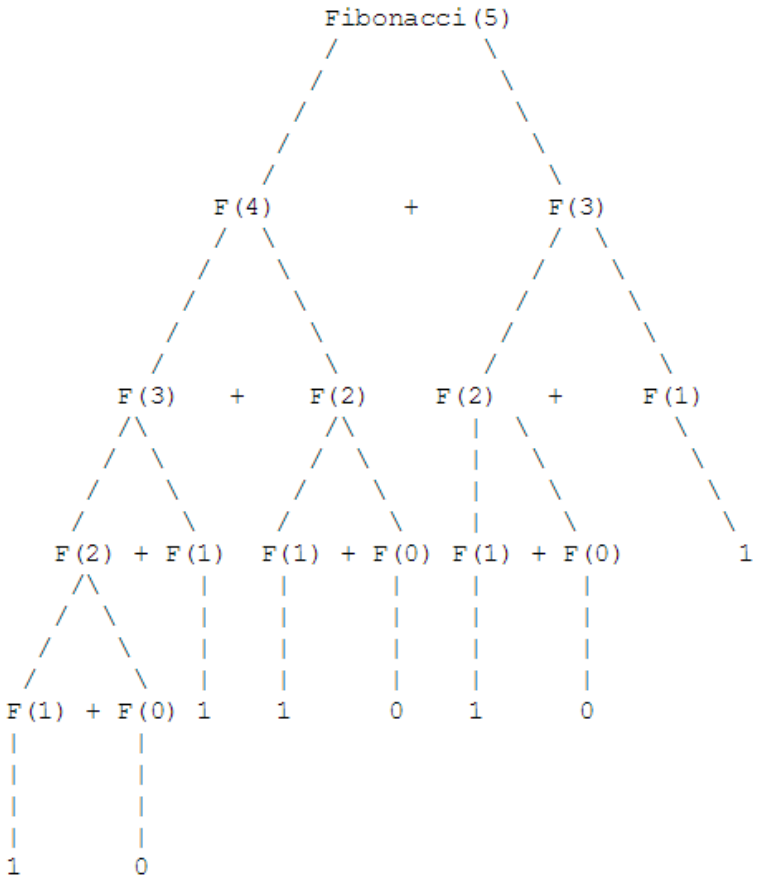
sum(3)
rez = 3 + 3
returnează sum(3) = 6

Extrage adresa de revenire. Se revine în funcția apelantă: main()

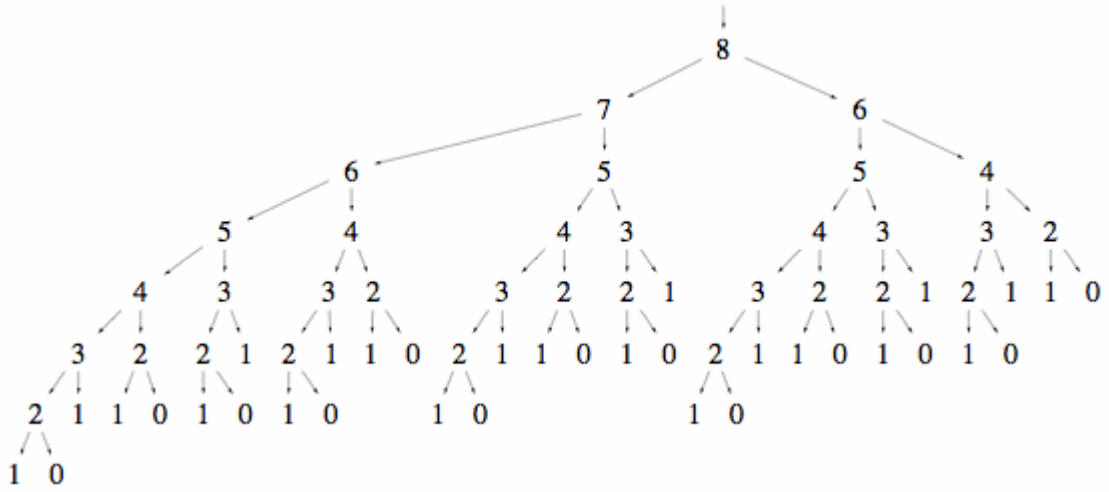
Funcții recursive

Seria Fibonacci – arborele de alpeluri recursive

Fibonacci(5)



Fibonacci(8)



Recursivitate

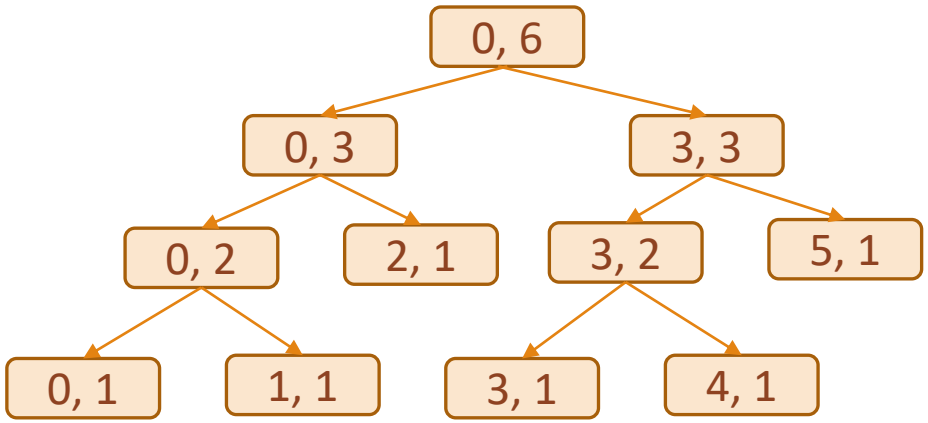
Recursivitate liniară

- presupune un singur autoapel recursiv la fiecare invocare a functiei
  - pe fiecare ramură din cazul general

```
void inversareElemente(int v[], int poz_i, int poz_j)
{
    int aux;
    if (poz_i < poz_j) {
        aux = v[poz_i];
        v[poz_i] = v[poz_j];
        v[poz_j] = aux;
        inversareElemente(v, poz_i + 1, poz_j - 1);
    }
}
```

Recursivitate binară

- presupune două autoapeluri recursive la fiecare invocare a functiei
  - pe fiecare ramură din cazul general



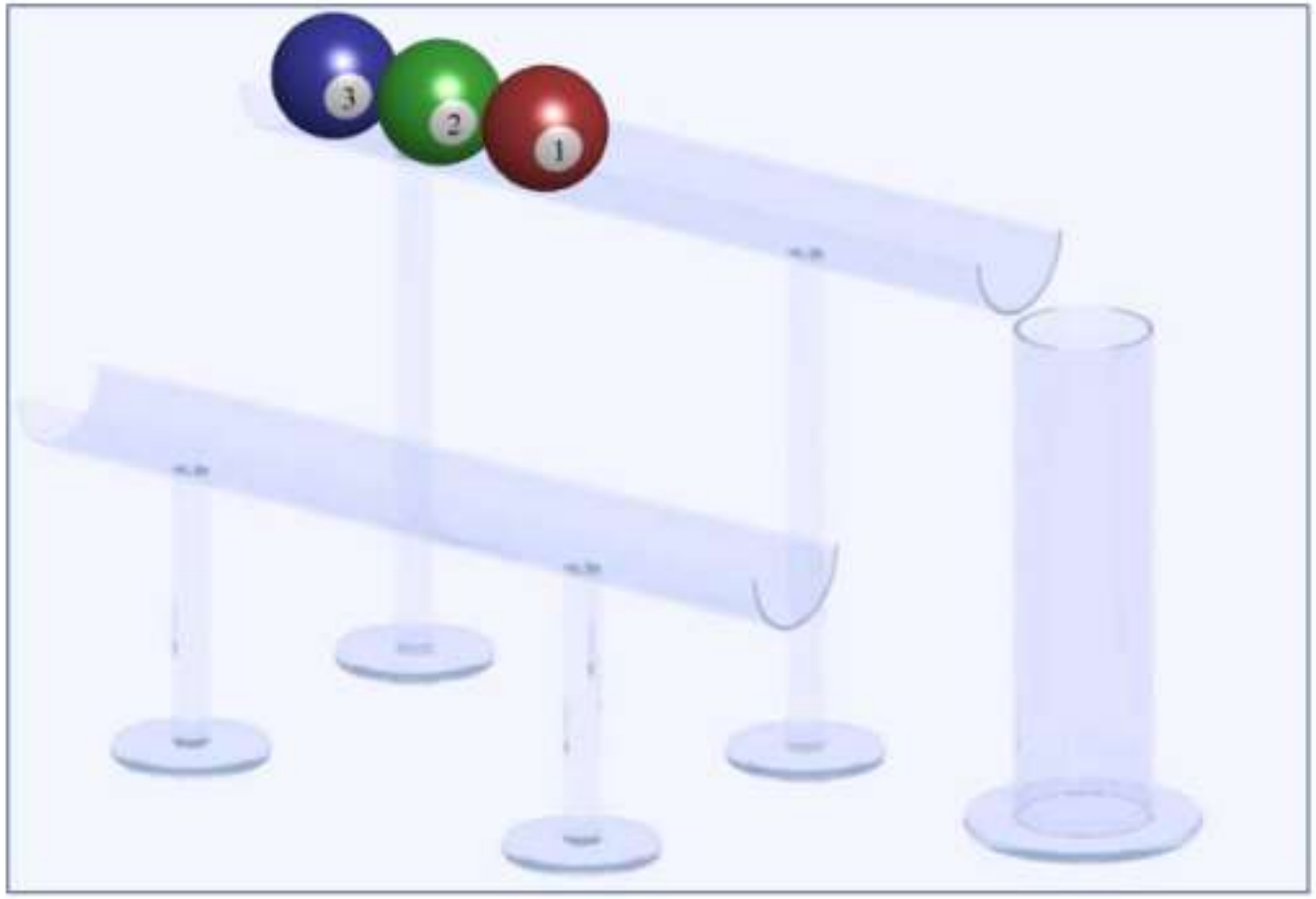

```
int max_binar(int v[], int poz, int nr) {
    printf("%d %d\n", poz, nr);
    if (nr == 1)
        return v[poz];
    else
        return max_binar(v, poz, ceil(nr / 2.0)) +
               max_binar(v, poz + ceil(nr / 2.0), floor(nr / 2.0));
}
```

Funcţii recursive

Precizați șirul de numere afișat de programul următor :

```
#include <stdio.h>

int n = 4;

void ex(int k)
{
  if (k < n)
  {
    ex(k+1);
    printf("%d", k);
  }
}

void main()
{
  ex(1);
}
```

n = ? k = ?



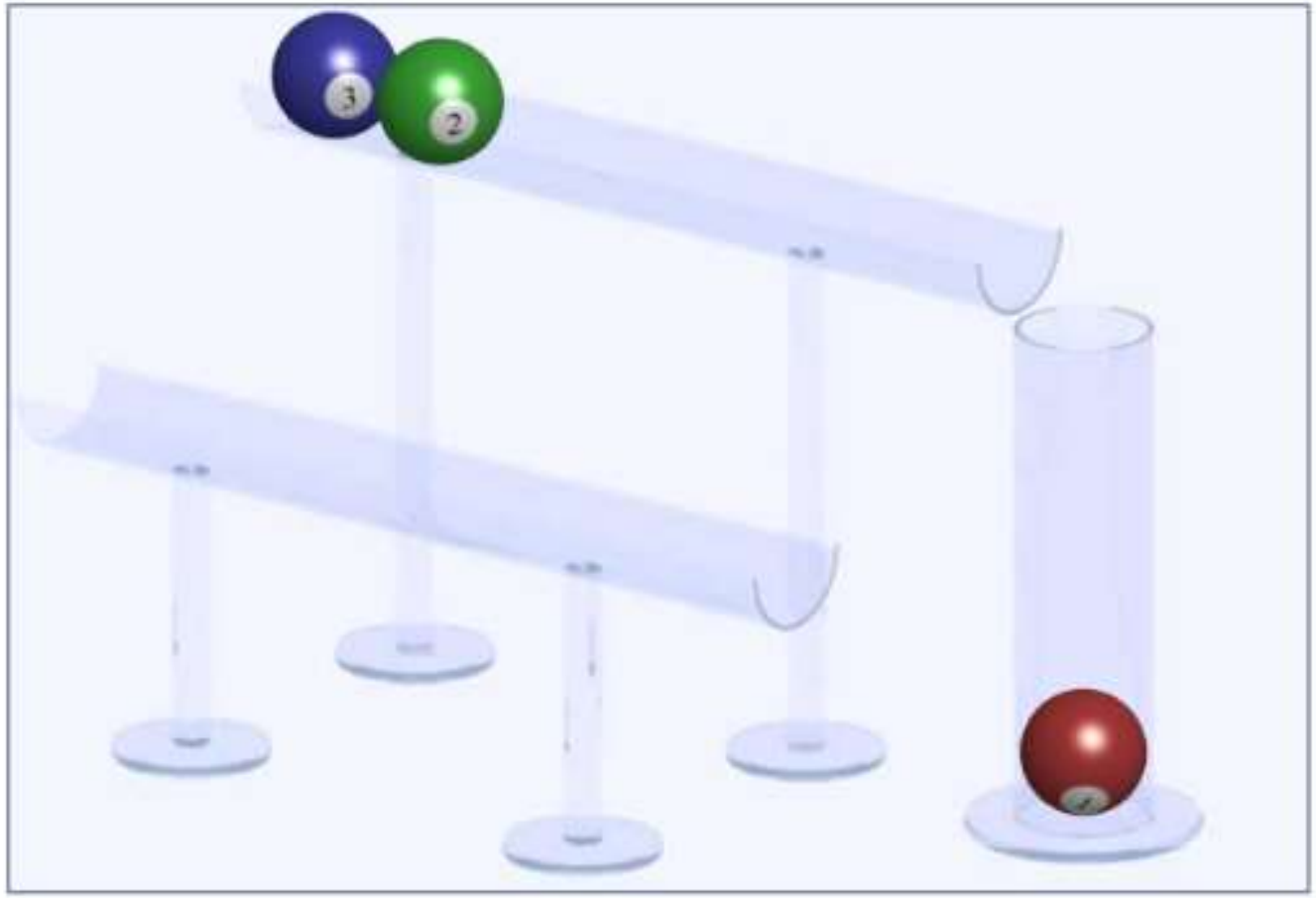
Exemple și figuri preluate de pe [Recursivitatea prin exemple]

Funcţii recursive

Precizați șirul de numere afișat de programul următor :

```
#include <stdio.h>

int n = 4;

void ex(int k)
{
  if (k < n)
  {
    ex(k+1);
    printf("%d", k);
  }
}

void main()
{
  ex(1);
}
```

n = 4 k = 1

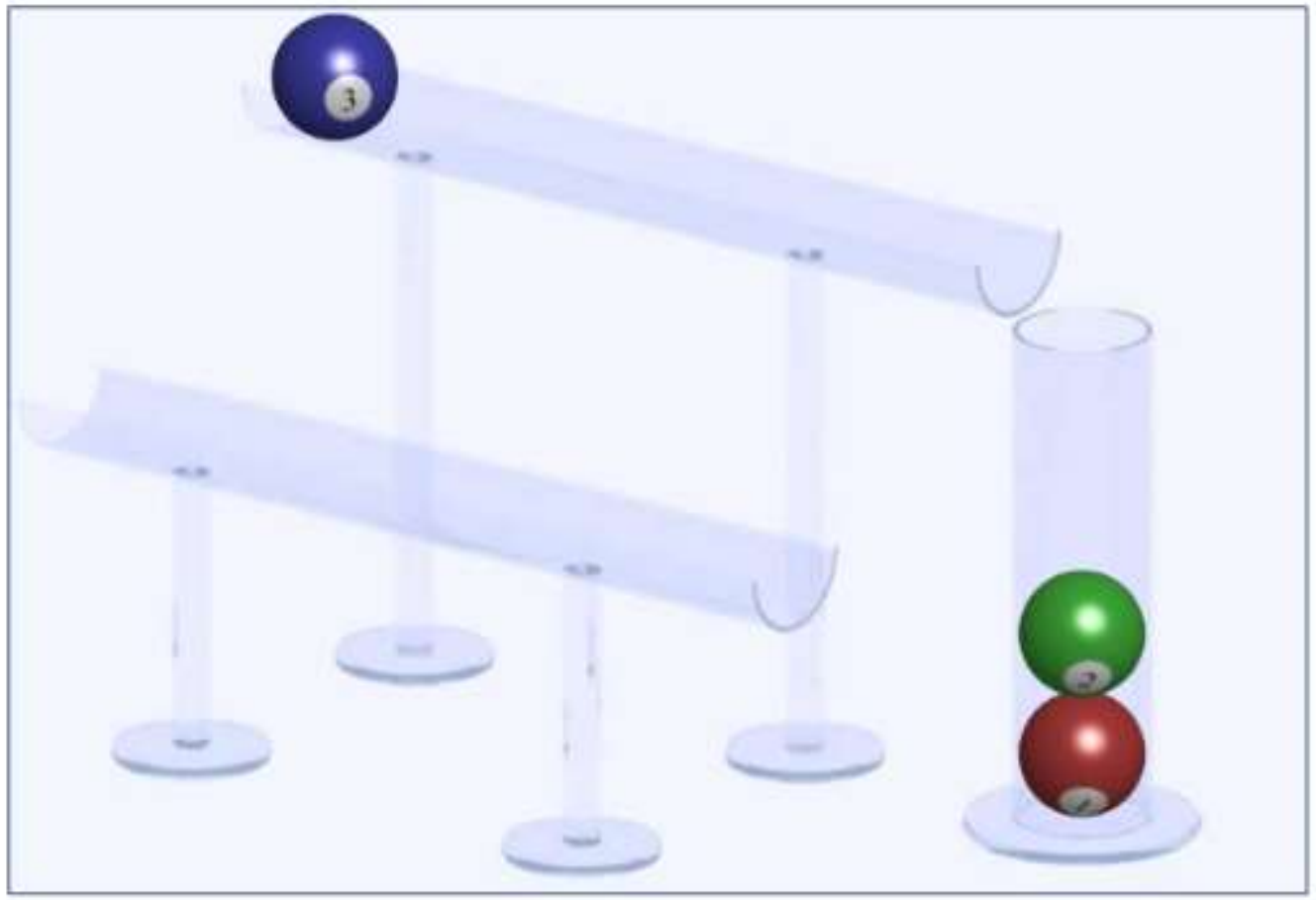


Funcţii recursive

Precizați șirul de numere afișat de programul următor :

```
#include <stdio.h>

int n = 4;

void ex(int k)
{
  if (k < n)
  {
    ex(k+1);
    printf("%d", k);
  }
}

void main()
{
  ex(1);
}
```

n = 4 k = 2

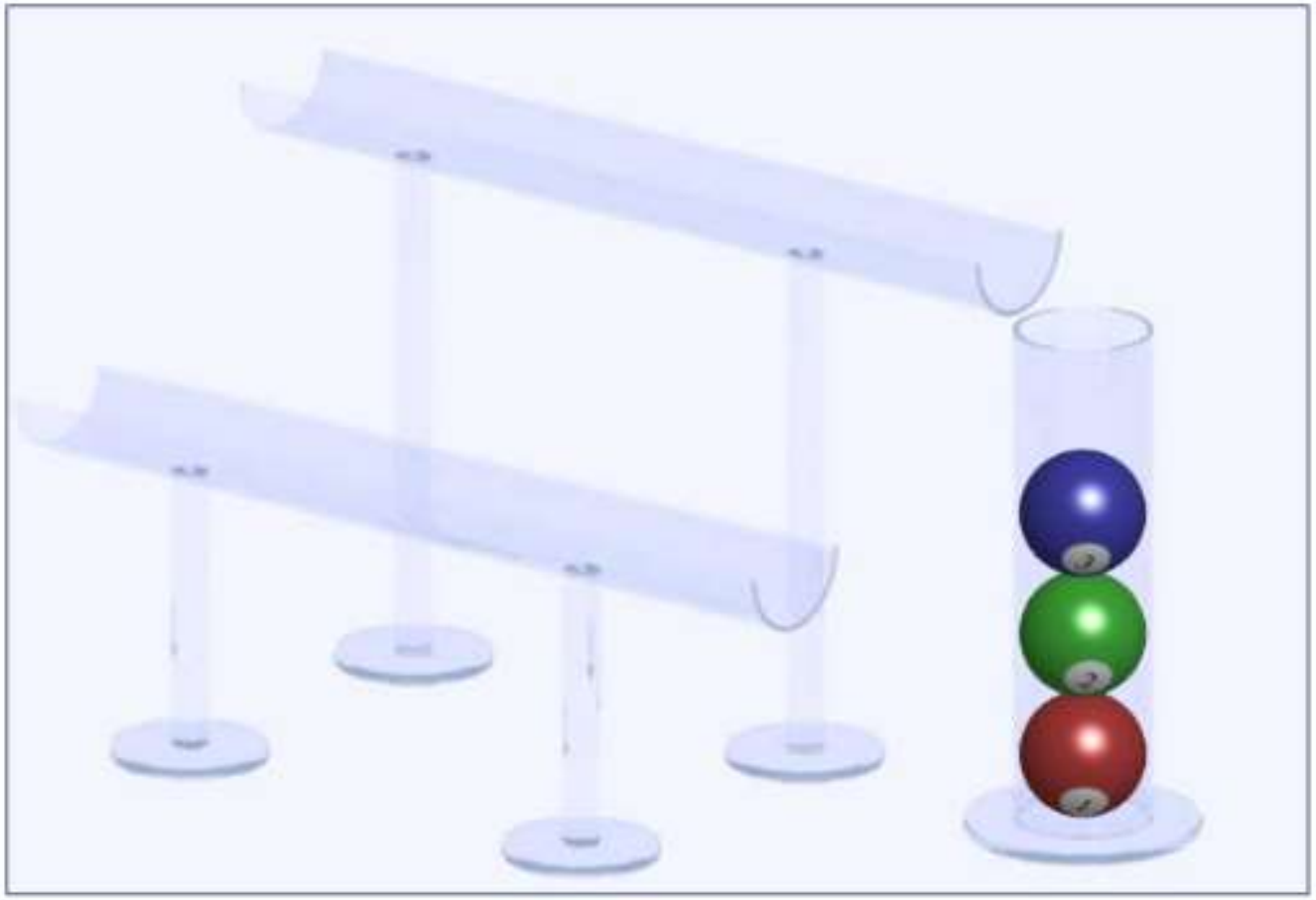


Funcţii recursive

Precizați șirul de numere afișat de programul următor :

```
#include <stdio.h>

int n = 4;

void ex(int k)
{
  if (k < n)
  {
    ex(k+1);
    printf("%d", k);
  }
}

void main()
{
  ex(1);
}
```

n = 4 k = 3



Funcţii recursive

Precizați șirul de numere afișat de programul următor :

```
#include <stdio.h>

int n = 4;

void ex(int k)
{
  if (k < n)
  {
    ex(k+1);
    printf("%d", k);
  }
}

void main()
{
  ex(1);
}
```

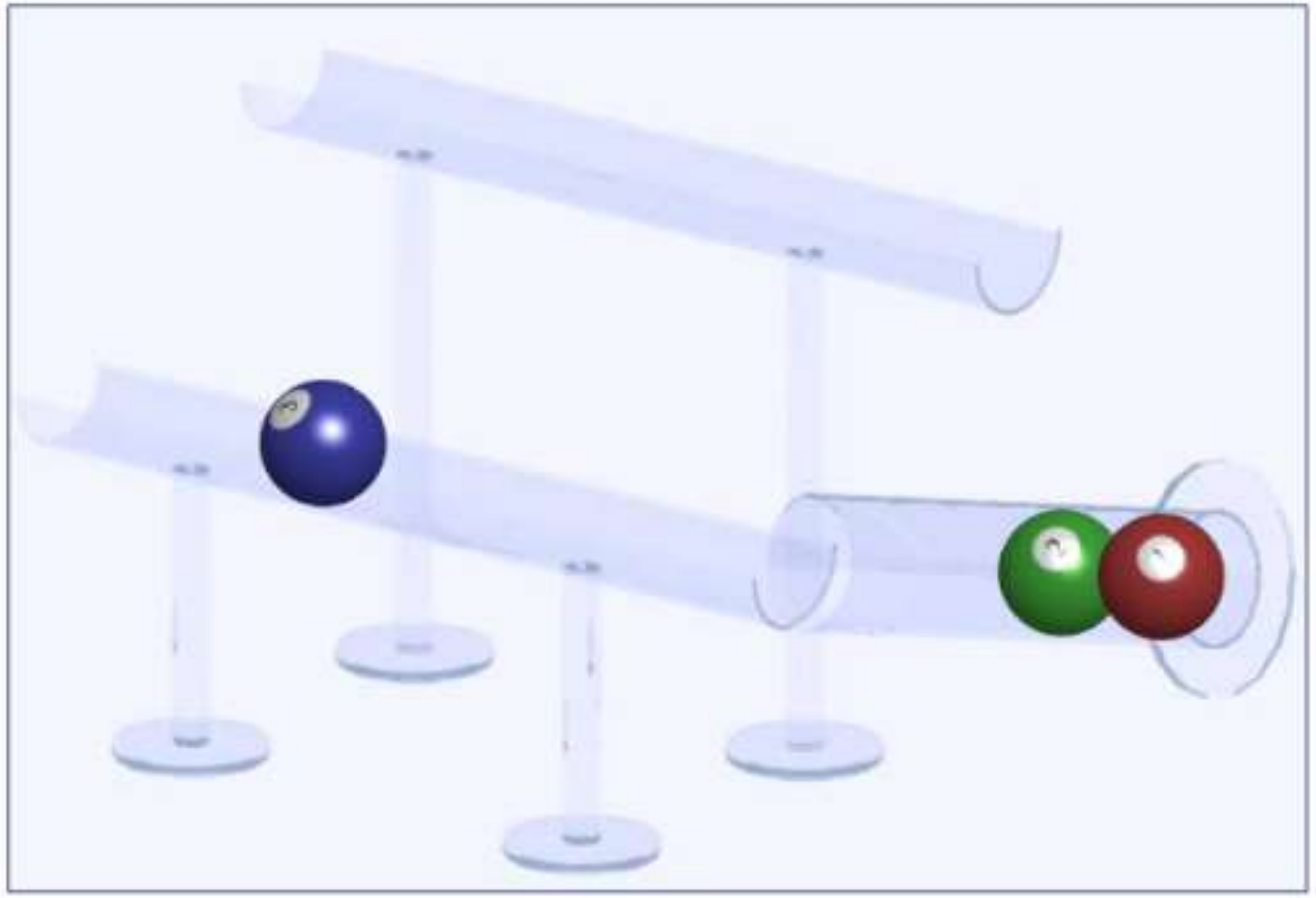
n = 4 k = 4

Funcţii recursive

Precizați șirul de numere afișat de programul următor :

```
#include <stdio.h>

int n = 4;

void ex(int k)
{
  if (k < n)
  {
    ex(k+1);
    printf("%d", k);
  }
}

void main()
{
  ex(1);
}
```

n = 4 k = 3

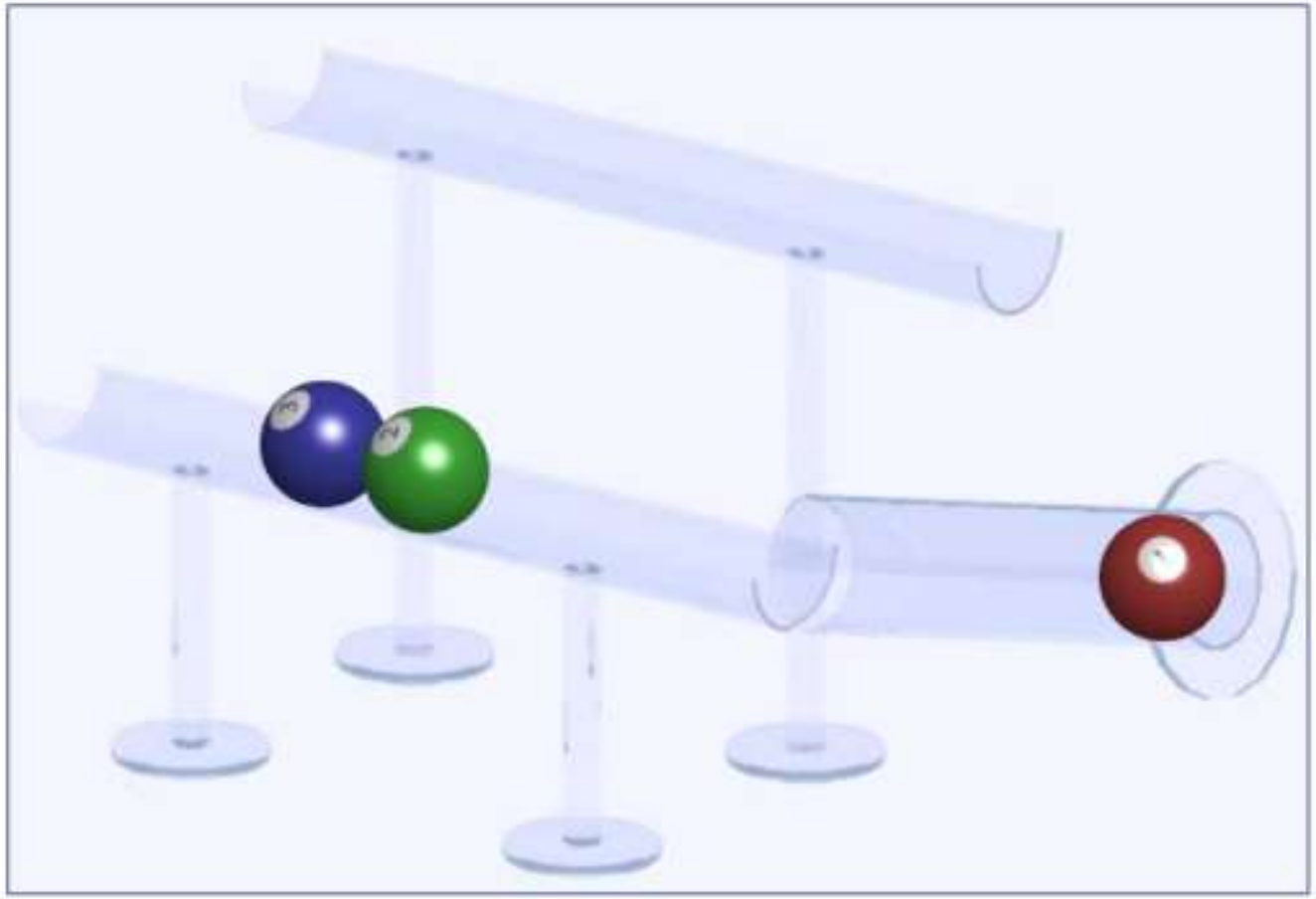


Funcţii recursive

Precizați șirul de numere afișat de programul următor :

```
#include <stdio.h>

int n = 4;

void ex(int k)
{
  if (k < n)
  {
    ex(k+1);
    printf("%d", k);
  }
}

void main()
{
  ex(1);
}
```

n = 4 k = 2

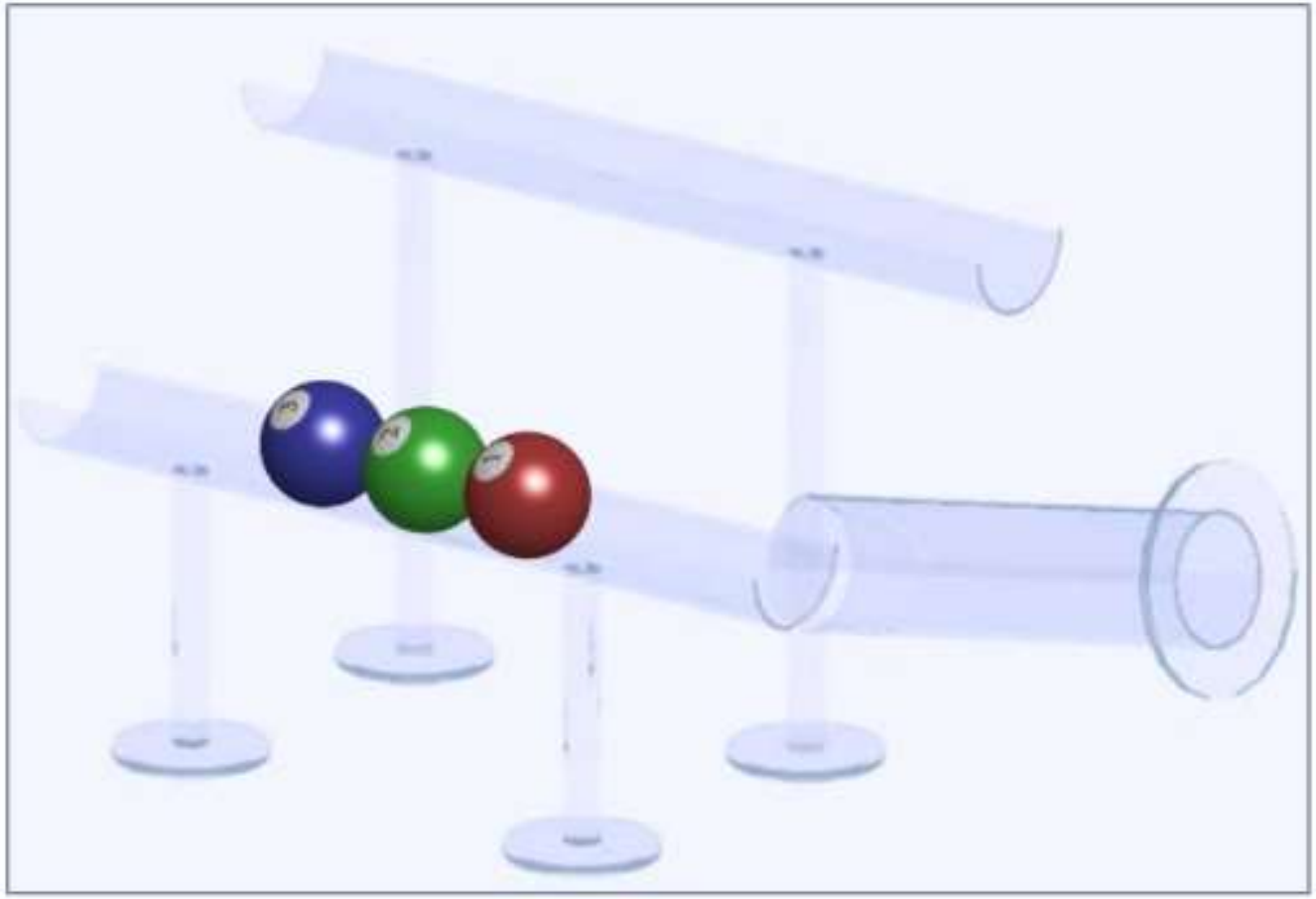


Funcţii recursive

Precizați șirul de numere afișat de programul următor :

```
#include <stdio.h>

int n = 4;

void ex(int k)
{
  if (k < n)
  {
    ex(k+1);
    printf("%d", k);
  }
}

void main()
{
  ex(1);
}
```

n = 4 k = 1



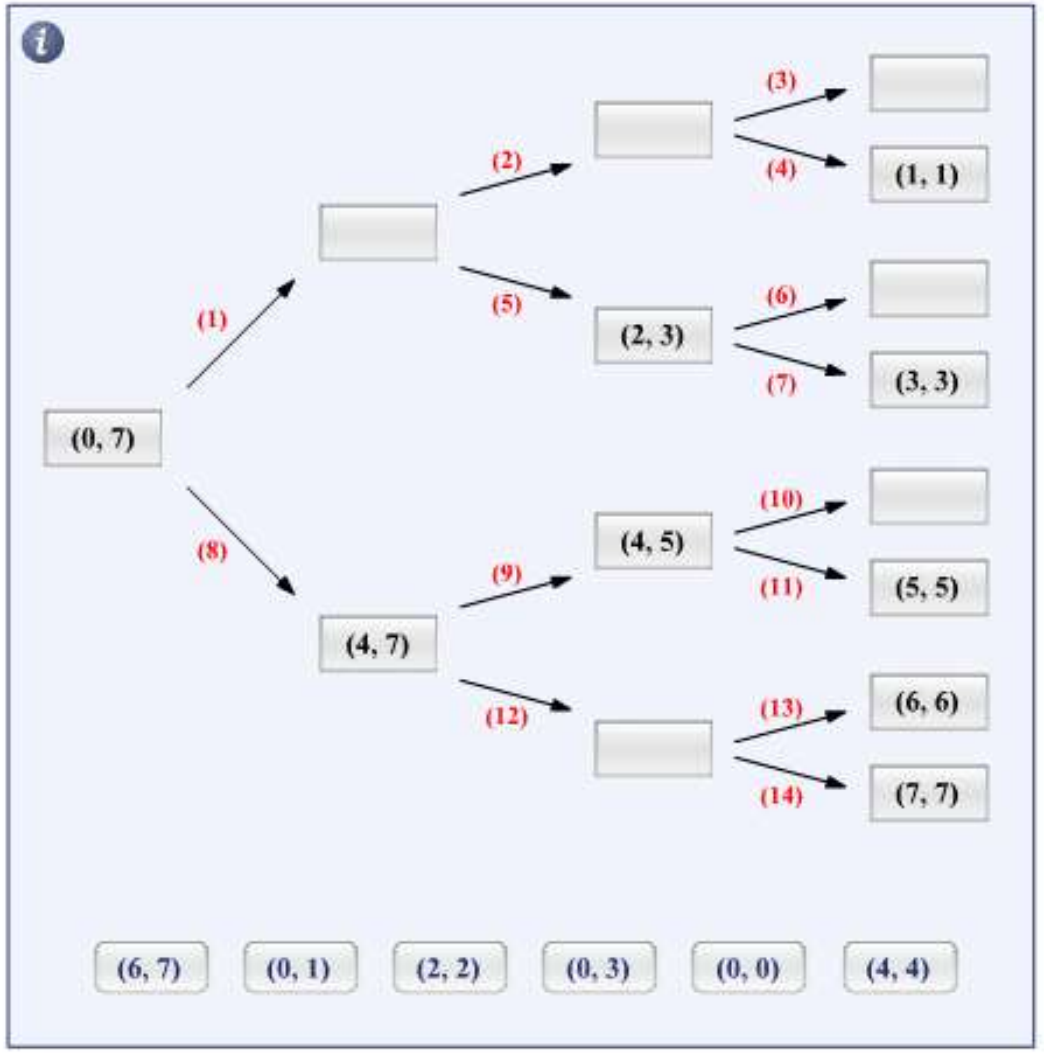
Funcţii recursive

Pentru următorul program, completați schema din dreapta corespunzător ordinii de apel a funcțiilor și indicați rezultatul afișat.

Indiciu :
Ordinea de parcurgere a schemei este indicată de cifrele de culoare roșie situate pe săgeți.

Program sursă :

```
#include <stdio.h>
int v[8] = {2, 4, 1, 3, 11, 5, 1, 3};

int impar(int n, int m)
{
    // printf("%d \t %d \n", n, m);
    if (n == m) return v[n] % 2;
    else return impar(n, (n+m)/2) +
                impar((n+m)/2 + 1, m);
}

void main()
{
    printf("%d", impar(0, 7));
}
```



Funcţii recursive

Definirea recursivă a produsului a două numere *a* şi *b*

$$a \cdot b = (a-1) \cdot b + b \Rightarrow pr(a,b) = \begin{cases} 0, \text{ dacă } a = 0 \\ pr(a-1, b) + b, \text{ dacă } a > 0 \end{cases}$$

Varianta 1 :

```
long pr(int n, int m)
{
     if (n == 0) return 0;
     else return m+pr(n-1, m);
}
```

n	m	m + pr (n-1, m)	lanțul de revenire

Varianta 1 :

```
long pr(int n, int m)
{
     if (n == 0) return 0;
     else return m+pr(n-1, m);
}
```

n	m	m + pr (n-1, m)	lanțul de revenire
3	2		

Varianta 1 :

```
long pr(int n, int m)
{
     if (n == 0) return 0;
     else return m+pr(n-1, m);
}
```

n	m	m + pr (n-1, m)	lanțul de revenire
3	2	2 + pr (2,2)	

Varianta 1 :

```
long pr(int n, int m)
{
    if (n == 0) return 0;
    else return m+pr(n-1, m);
}
```

n	m	m + pr (n-1, m)	lanțul de revenire
3	2	2 + pr (2,2)	
2	2	2 + pr (1,2)	

Varianta 1 :

```
long pr(int n, int m)
{
    if (n == 0) return 0;
    else return m+pr(n-1, m);
}
```

n	m	m + pr (n-1, m)	lanțul de revenire
3	2	2 + pr (2,2)	
2	2	2 + pr (1,2)	
1	2	2 + pr (0,2)	

Varianta 1 :

```
long pr(int n, int m)
{
    if (n == 0) return 0;
    else return m+pr(n-1, m);
}
```

n	m	m + pr (n-1, m)	lanțul de revenire
3	2	2 + pr (2,2)	
2	2	2 + pr (1,2)	
1	2	2 + pr (0,2)	
0	2		0

Varianta 1 :

```
long pr(int n, int m)
{
    if (n == 0) return 0;
    else return m+pr(n-1, m);
}
```

n	m	m + pr (n-1, m)	lanțul de revenire
3	2	2 + pr (2,2)	
2	2	2 + pr (1,2)	
1	2	2 + pr (0,2)	2 + 0 = 2
0	2		0

Varianta 1 :

```
long pr(int n, int m)
{
    if (n == 0) return 0;
    else return m+pr(n-1, m);
}
```

n	m	m + pr (n-1, m)	lanțul de revenire
3	2	2 + pr (2,2)	
2	2	2 + pr (1,2)	2 + 2 = 4
1	2	2 + pr (0,2)	2 + 0 = 2
0	2		0

Varianta 1 :

```
long pr(int n, int m)
{
    if (n == 0) return 0;
    else return m+pr(n-1, m);
}
```

n	m	m + pr (n-1, m)	lanțul de revenire
3	2	2 + pr (2,2)	2 + 4 = 6
2	2	2 + pr (1,2)	2 + 2 = 4
1	2	2 + pr (0,2)	2 + 0 = 2
0	2		0

Funcţii recursive

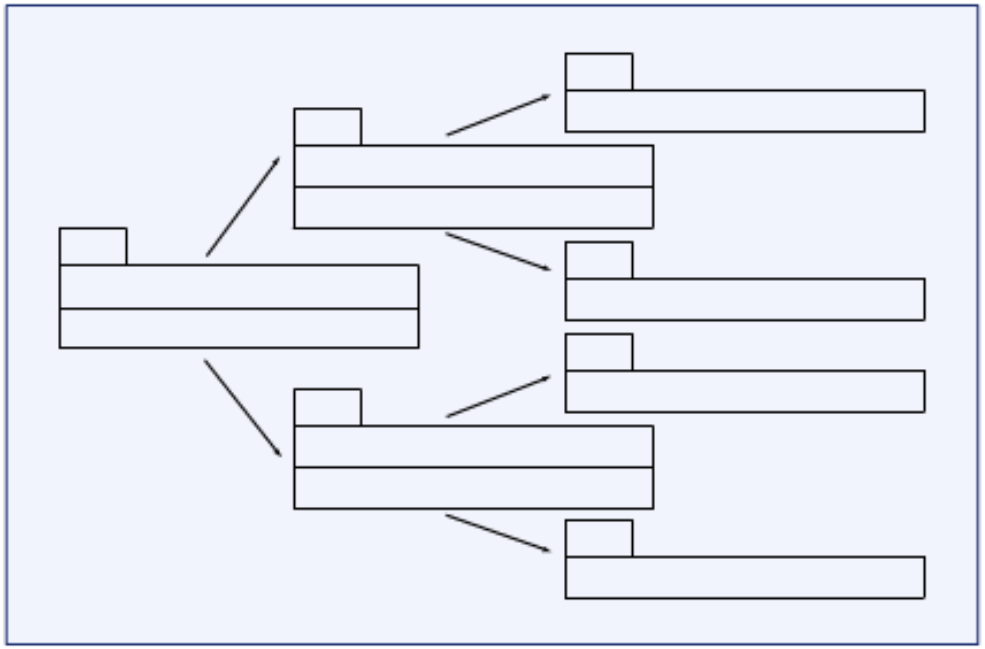
Se consideră şirurile : $a_n = \frac{a_{n-1} + b_{n-1}}{2}$ $\quad b_n = \sqrt{a_{n-1}\, b_{n-1}}$

$a_n , b_n > 0$

$a_0 = a \; ; \; b_0 = b$

Să se calculeze a_n , b_n pentru a, b, n date.

Exemplificăm funcționarea algoritmului pentru calculul a_n pentru $a = 4, b = 9, n = 2$.



```
#include <stdio.h>
#include <math.h>

double a, b;
int n;

double bb(int n);

double aa(int n)
{
    if (!n) return a;
    else return (aa(n-1)+
                 bb(n-1))/2;
}

double bb(int n)
{
    if (!n) return b;
    else return sqrt(aa(n-1)*
                     bb(n-1));
}


void main()
{
    scanf("%lf %lf %d",&a,&b,&n);
    printf("%lf",aa(n));
}
```

Funcţii recursive

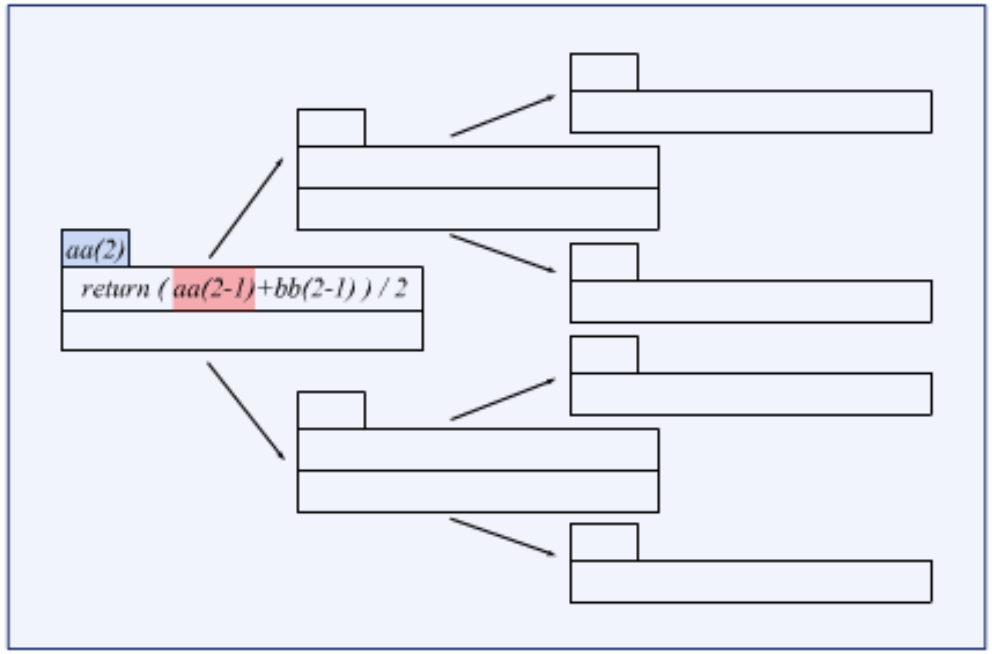
Se consideră şirurile : $a_n = \frac{a_{n-1} + b_{n-1}}{2}$ $\quad b_n = \sqrt{a_{n-1}\, b_{n-1}}$

$a_n , b_n > 0$

$a_0 = a \; ; \; b_0 = b$

Să se calculeze a_n , b_n pentru a, b, n date.

Exemplificăm funcționarea algoritmului pentru calculul a_n pentru $a = 4$, $b = 9$, $n = 2$.




```
#include <stdio.h>
#include <math.h>

double a, b;
int n;

double bb(int n);

double aa(int n)
{
    if (!n) return a;
    else return (aa(n-1)+
                 bb(n-1))/2;
}

double bb(int n)
{
    if (!n) return b;
    else return sqrt(aa(n-1)*
                     bb(n-1));
}


void main()
{
    scanf("%lf %lf %d",&a,&b,&n);
    printf("%lf",aa(n));
}
```

Funcţii recursive

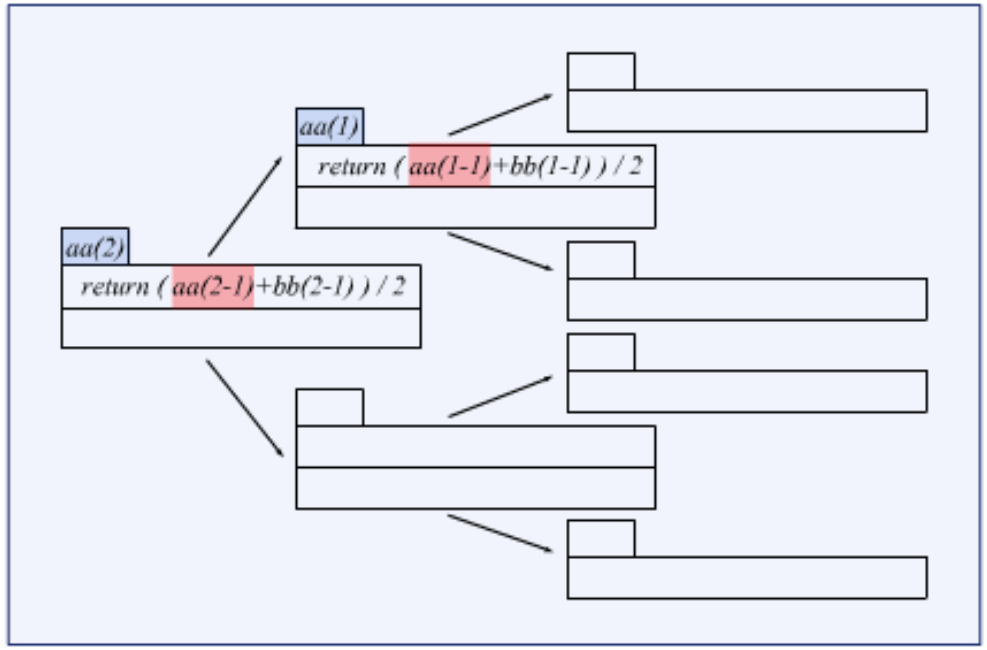
Se consideră şirurile : $a_n = \frac{a_{n-1} + b_{n-1}}{2}$ $\quad b_n = \sqrt{a_{n-1}\, b_{n-1}}$

$a_n , b_n > 0$

$a_0 = a \; ; \; b_0 = b$

Să se calculeze a_n , b_n pentru a, b, n date.

Exemplificăm funcționarea algoritmului pentru calculul a_n pentru $a = 4$, $b = 9$, $n = 2$.




```
#include <stdio.h>
#include <math.h>

double a, b;
int n;

double bb(int n);

double aa(int n)
{
    if (!n) return a;
    else return (aa(n-1)+
                 bb(n-1))/2;
}

double bb(int n)
{
    if (!n) return b;
    else return sqrt(aa(n-1)*
                     bb(n-1));
}


void main()
{
    scanf("%lf %lf %d",&a,&b,&n);
    printf("%lf",aa(n));
}
```

Funcţii recursive

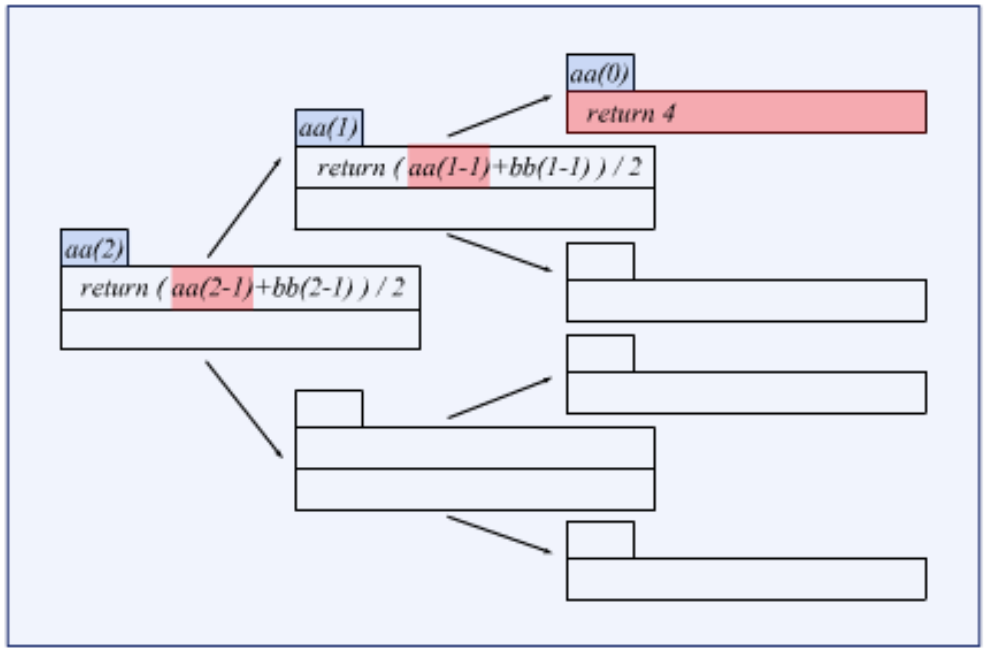
Se consideră şirurile : $a_n = \frac{a_{n-1} + b_{n-1}}{2}$ $\quad b_n = \sqrt{a_{n-1}\, b_{n-1}}$

$a_n , b_n > 0$

$a_0 = a \; ; \; b_0 = b$

Să se calculeze a_n , b_n pentru a, b, n date.

Exemplificăm funcționarea algoritmului pentru calculul a_n pentru $a = 4, b = 9, n = 2$.




```
#include <stdio.h>
#include <math.h>

double a, b;
int n;

double bb(int n);

double aa(int n)
{
    if (!n) return a;
    else return (aa(n-1)+
                 bb(n-1))/2;
}

double bb(int n)
{
    if (!n) return b;
    else return sqrt(aa(n-1)*
                     bb(n-1));
}


void main()
{
    scanf("%lf %lf %d",&a,&b,&n);
    printf("%lf",aa(n));
}
```

Funcţii recursive

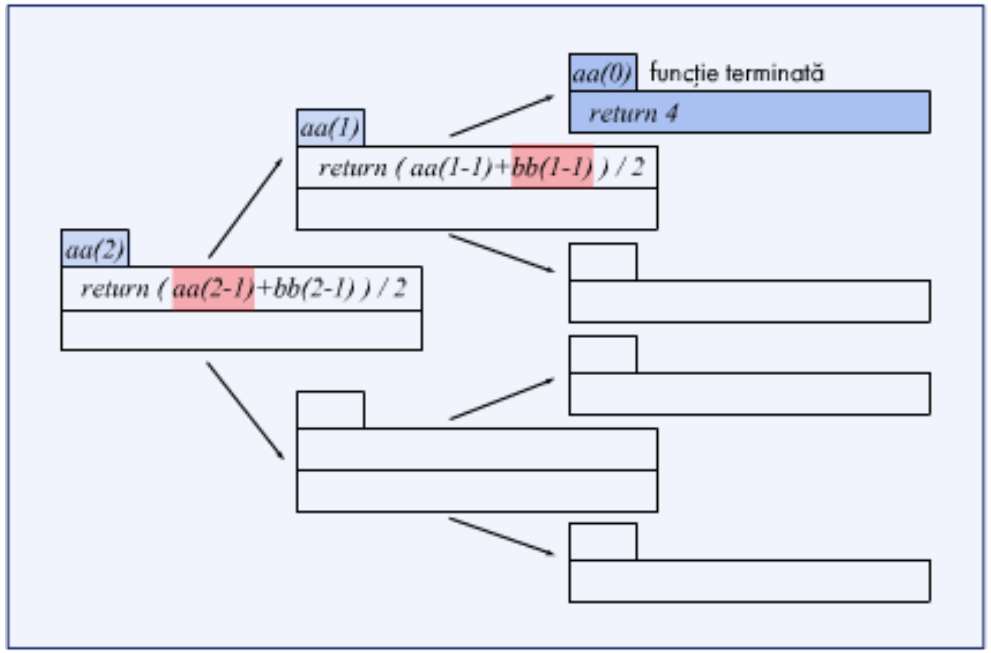
Se consideră şirurile : $a_n = \frac{a_{n-1} + b_{n-1}}{2}$ $\quad b_n = \sqrt{a_{n-1}\, b_{n-1}}$

$a_n , b_n > 0$

$a_0 = a \; ; \; b_0 = b$

Să se calculeze a_n , b_n pentru a, b, n date.

Exemplificăm funcționarea algoritmului pentru calculul a_n pentru $a = 4, b = 9, n = 2$.




```
#include <stdio.h>
#include <math.h>

double a, b;
int n;

double bb(int n);

double aa(int n)
{
    if (!n) return a;
    else return (aa(n-1)+
                 bb(n-1))/2;
}

double bb(int n)
{
    if (!n) return b;
    else return sqrt(aa(n-1)*
                     bb(n-1));
}


void main()
{
    scanf("%lf %lf %d",&a,&b,&n);
    printf("%lf",aa(n));
}
```

Funcţii recursive

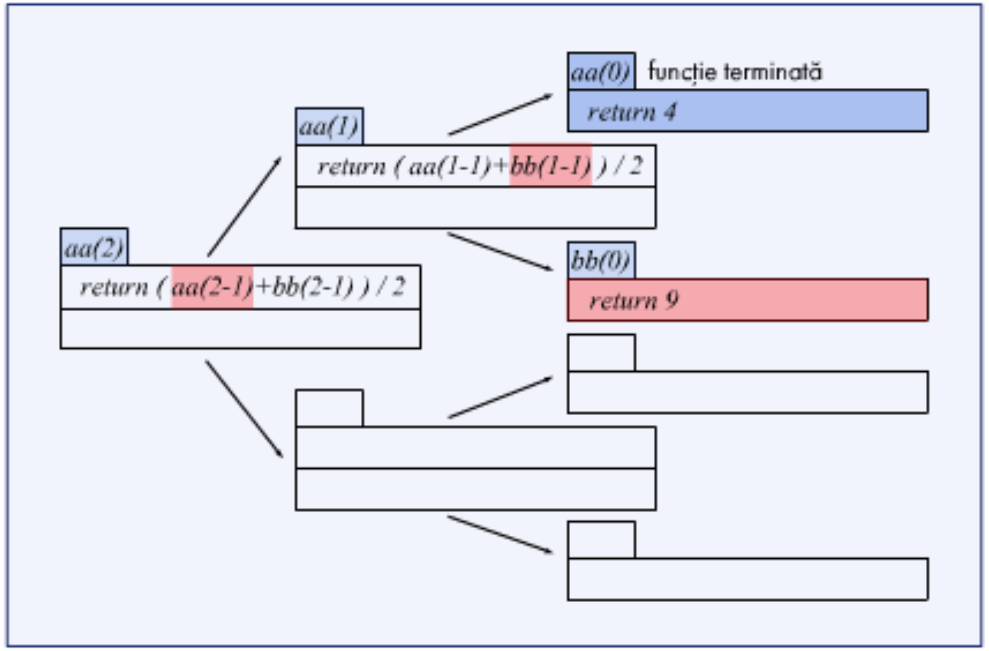
Se consideră şirurile : $a_n = \frac{a_{n-1} + b_{n-1}}{2}$ $\quad b_n = \sqrt{a_{n-1}\, b_{n-1}}$

$a_n , b_n > 0$

$a_0 = a \; ; \; b_0 = b$

Să se calculeze a_n , b_n pentru a, b, n date.

Exemplificăm funcționarea algoritmului pentru calculul a_n pentru $a = 4, b = 9, n = 2$.




```
#include <stdio.h>
#include <math.h>

double a, b;
int n;

double bb(int n);

double aa(int n)
{
    if (!n) return a;
    else return (aa(n-1)+
                 bb(n-1))/2;
}

double bb(int n)
{
    if (!n) return b;
    else return sqrt(aa(n-1)*
                     bb(n-1));
}


void main()
{
    scanf("%lf %lf %d",&a,&b,&n);
    printf("%lf",aa(n));
}
```

Funcţii recursive

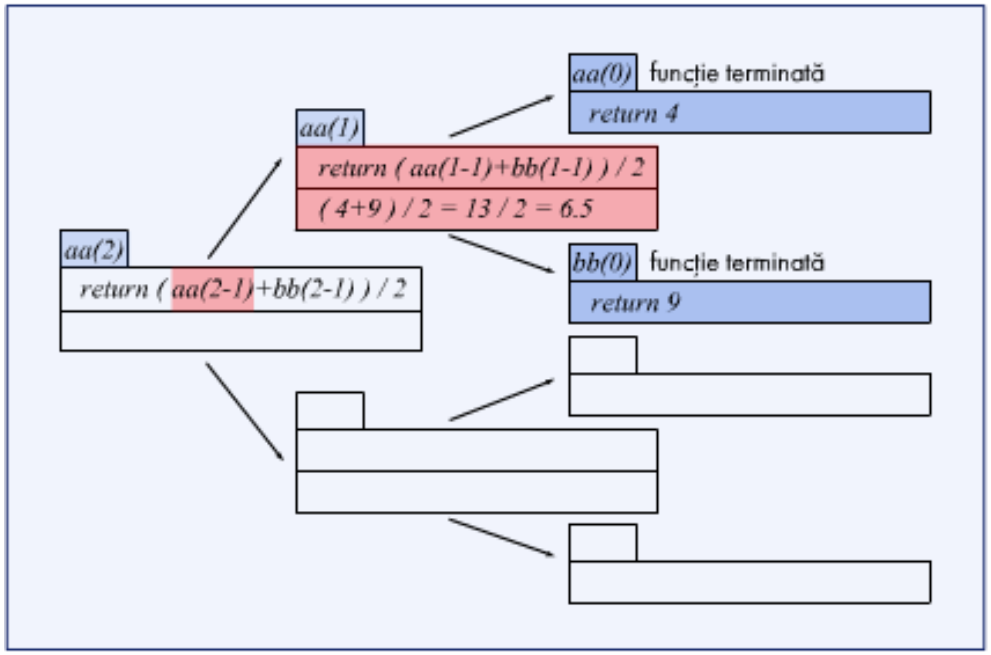
Se consideră şirurile : $a_n = \frac{a_{n-1} + b_{n-1}}{2}$ $\quad b_n = \sqrt{a_{n-1}\, b_{n-1}}$

$a_n , b_n > 0$

$a_0 = a \; ; \; b_0 = b$

Să se calculeze a_n , b_n pentru a, b, n date.

Exemplificăm funcționarea algoritmului pentru calculul a_n pentru $a = 4, b = 9, n = 2$.




```
#include <stdio.h>
#include <math.h>

double a, b;
int n;

double bb(int n);

double aa(int n)
{
    if (!n) return a;
    else return (aa(n-1)+
                 bb(n-1))/2;
}

double bb(int n)
{
    if (!n) return b;
    else return sqrt(aa(n-1)*
                     bb(n-1));
}


void main()
{
    scanf("%lf %lf %d",&a,&b,&n);
    printf("%lf",aa(n));
}
```

Funcţii recursive

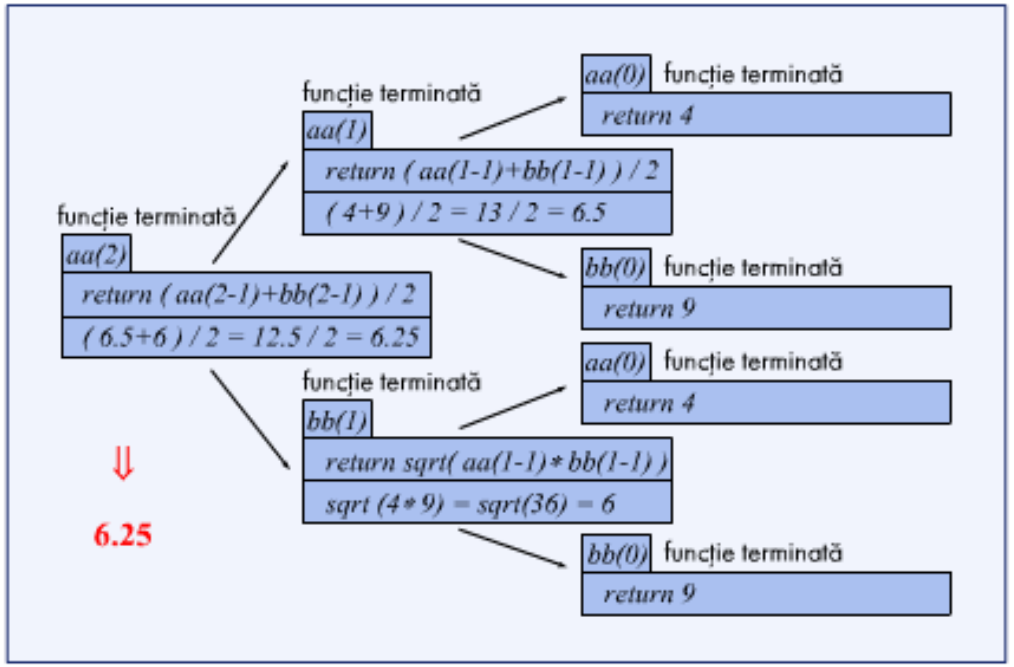
Se consideră şirurile : $a_n = \frac{a_{n-1} + b_{n-1}}{2}$ $\quad b_n = \sqrt{a_{n-1}\, b_{n-1}}$

$a_n , b_n > 0$

$a_0 = a \; ; \; b_0 = b$

Să se calculeze a_n , b_n pentru a, b, n date.

Exemplificăm funcționarea algoritmului pentru calculul a_n pentru $a = 4, b = 9, n = 2$.




```
#include <stdio.h>
#include <math.h>

double a, b;
int n;

double bb(int n);

double aa(int n)
{
    if (!n) return a;
    else return (aa(n-1)+
                 bb(n-1))/2;
}

double bb(int n)
{
    if (!n) return b;
    else return sqrt(aa(n-1)*
                     bb(n-1));
}


void main()
{
    scanf("%lf %lf %d",&a,&b,&n);
    printf("%lf",aa(n));
}
```

Surse bibliografice

- K. N. King, C Programming – A Modern Approach, 2nd edition, W. W. Norton & Co., 2008
  - Capitolul 9.6
- Deitel & Deitel, C How to Program, 6th edition, Pearson, 2009
  - Capitolul 5.14
- Recursivitatea prin exemple, Disponibil online: http://www.advancedelearning.com/index.php/articles/214