Funcţii recursive.

O definiţie recursivă foloseşte termenul de definit în corpul definiţiei.
O funcţie poate fi recursivă, adică se poate apela pe ea însăşi, în mod direct sau indirect, printr-un lanţ de apeluri.

```
/* apel recursiv direct */
void f(int x) {
  . . .
  f(10);
  . . .
}
```

```
/* apel recursiv indirect */
void f1(int x) {
  . . .
  f2(10);
  . . .
}
```

```
void f2(int y) {
  . . .
  f1(20);
  . . .
}
```

În cazul recursivităţii indirecte, compilatorul trebuie să verifice corectitudinea apelurilor recursive din fiecare funcţie. O declarare anticipată a prototipurilor funcţiilor oferă informaţii suficiente compilatorului pentru a face verificările cerute.

Exemple de definiţii recursive.

Multe obiecte matematice au definiri recursive, care pot fi exploatate algoritmic. De exemplu:

1^0. numărul natural

- **1 ∈ N**
- dacă **x ∈ N** atunci **succ(x) ∈ N**

2^0. factorialul **n!=**

- **1** dacă **n=0**
- **n*(n-1)!** dacă **n > 0**

```
long fact(int n)
{ return (n? n*fact(n-1) : 1);}
```

3^0. cel mai mare divizor comun **cmmdc(a, b)=**

- **a** dacă **b=0**
- **cmmdc(b, a%b)** dacă **b > 0**

```
int cmmdc(int a, int b)
{ return (b? cmmdc(b, a%b) : a); }
```

4^0. arborele binar

- arborele vid este arbore binar
- 

este arbore binar, cu **SS** şi **SD** arbori binari

Orice definiţie recursivă are două părţi:

- baza sau partea nerecursivă
- partea recursivă

Codul recursiv:

- rezolvă explicit cazul de bază
- apelurile recursive conţin valori mai mici ale argumentelor

Funcţiile definite recursiv sunt simple şi elegante. Corectitudinea lor poate fi uşor verificată.

Definirea unor funcţii recursive conduce la ineficienţă, motiv pentru care se evită, putând fi înlocuită întotdeauna prin iteraţie.

Definirea unor funcţii recursive este justificată dacă se lucrează cu structuri de date definite tot recursiv.

Recursivitate liniară.

Este forma cea mai simplă de recursivitate şi constă dintr-un singur apel recursiv.

De exemplu pentru calculul factorialului avem:

`f(0)=1` cazul de bază

`f(n)=n*f(n-1)` cazul general

Se vede că dacă timpul necesar execuţiei cazului de bază este **a** şi a cazului general este **b**, atunci timpul total de calcul este: `a+b*(n-1) = O(n)`, adică avem complexitate liniară.

```
typedef unsigned long UL;

UL facrec(int n){
  if(n==0 || n==1) return 1UL;
  return n * facrec(n-1);
}

UL faciter(int n){
  UL fac = 1UL;
  int k;
  for(k=2; k<=n; k++)
    fac *= k;
  return fac;
}

UL cmmdc_rec(UL a, UL b){
   if(b==0) return a;
   return cmmdc_rec(b, a%b);
}

UL cmmdc_it(UL a, UL b){
   UL rest;
   if(b==0) return a; //e necesar pentru ca testul e la sfarsit
   do{rest = a % b;
      a = b;
      b = rest;
   } while(rest);
   return a;
}
```

Recursivitate binară.

Recursivitatea binară presupune existenţa a două apeluri recursive. Exemplul tipic îl constituie şirul lui Fibonacci:

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

Funcţia este foarte ineficientă, având complexitate exponenţială, cu multe apeluri recursive repetate.

```
// termenul n din sirul Fibonacci (recursiv si iterativ)
typedef unsigned long UL;

UL fib_rec(int n){
  // program simplu dar foarte ineficient (complexitate exponentiala)
  if(n==0 || n==1) return 1;
  return fib_rec(n-1)+fib_rec(n-2);
}

UL fib_iter(int n){
  // program simplu si eficient (complexitate liniara)
  UL a=1, b=1, c;
  int k;
  for(k=2; k<=n; k++){
     c = a + b;
     a = b;
     b = c;
  }
  return b; //functioneaza corect si pt. n=0 sau n=1
}

// observatii: la n=40 varianta recursiva dureaza foarte mult
// la n=45 varianta iterativa da inca rezultat corect
// pentru n>45 executia iterativa este rapida dar rezultatul depaseste
UL
```

Se poate înlocui dublul apel recursiv printr-un singur apel recursiv. Pentru aceasta ţinem seama că dacă `a`, `b` şi `c` sunt primii 3 termeni din şirul Fibonacci, atunci calculul termenul situat la distanţia `n` în raport cu `a`, este situat la distanţa `n-1` faţă de termenul următor `b`. Necesitatea cunoaşterii termenului următor impune prezenţa a doi termeni în lista de parametri.

Algoritmul rezultat are complexitate liniară.

```
long f(long a, long b, int n){
  if(n<=1) return b;
  return f(b, a+b, n-1);
}

long fiblin(int n){
  f(0, 1, n);
}
```

Inversarea unui şir de caractere preesupune în mod natural prezenţa unui tablou care să păstreze şirul de caractere. O soluţie recursivă, în care după apelul recursiv se scrie

caracterul citit, permite evitarea folosirii tablourilor. Elementele şirului de caractere vor fi păstrate ca variabile locale (caractere) în înregistrările de activare generate lanţul de apeluri recursive.

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

// inversarea recursiva a unui sir de caractere, introdus de la
// tastatura si terminat prin Enter, fara a folosi un tablou

void invsir(){
  char c;
  scanf("%c", &c);
  if(c=='\n') return;
  // printf("%c", c);  vezi comentariul din final
  invsir();
  printf("%c", c);
}

int main(){
    invsir();
    printf("\n");
    return 0;
}

// Ce rezultat se obtine daca se transforma comentariul
// din main() in instructiune
```

Un algoritm recursiv obţine soluţia problemei prin rezolvarea unor instanţe mai mici ale aceleeaşi probleme.

- se împarte problema în subprobleme de aceeaşi natură, dar de dimensiune mai scăzută
- se rezolvă subproblemele obţinându-se soluţiile acestora
- se combină soluţiile subproblemelor obţinându-se soluţia problemei
- ultima etapă poate lipsi (în anumite cazuri), soluţia problemei rezultând direct din soluţiile subproblemelor.
- procesul de divizare continuă şi pentru subprobleme, până când se ajunge la o dimensiune suficient de redusă a problemei care să permită rezolvarea printr-o metodă directă.
- metoda divizării se exprimă natural în mod recursiv: rezolvarea problemei constă în rezolvarea unor instanţe ale problemei de dimensiuni mai reduse

Exemple de probleme rezolvate recursiv.

1. Ridicarea unui număr la o putere întreagă.

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{dacă } x \text{ este par} \\ x \cdot x^{n/2} \cdot x^{n/2} & \text{dacă } x \text{ este impar} \end{cases}$$

```
double putere(double x, int n){
  if (n==0) return 1;
  double y = putere(x, n/2);
  if (n%2==0)
    return y*y;
  else
    return x*y*y;
}
```

Se constată că în urma separării se rezolvă o singură subproblemă de dimensiune **n/2.** Aşadar: **T(n)= T(n/2)+1,** care are soluţia **T(n)=O(log$_2$n).**

2. Căutarea binară – presupune căutarea unei valori într-un tablou sortat. În acest scop

- se compară valoarea căutată `y` cu elementul din mijlocul tabloului `x[m]`
- dacă sunt egale, elementul `y` a fost găsit în poziţia `m`
- în caz contrar se continuă căutarea într-una din jumătăţile tabloului (în prima jumătate, dacă `y<x[m]` sau în a doua jumătate dacă `y>x[m]`).
- Funcţia întoarce poziţia `m` a valorii `y` în `x` sau `-1`, dacă `y` nu se află în `x`. Complexitatea este `O(log`$_2$`n)`

```
 int CB(int i, int j, double y, double x[]) {
  if(i > j) return -1;
  double m = (i+j)/2;
  if(y == x[m]) return m;
  if(y < x[m])
    return CB(i, m-1, y, x);
  else
    return CB(m+1, j, y, x);
}
int CautBin(int n, double x[], double y){
  if(n==0) return -1;
  return CB(0, n-1, y, x);
}
```

3. Localizarea unei rădăcini prin bisecţie

Localizarea unei rădăcini a ecuaţiei `f(x)=0`, separată într-un interval `[a, b]` prin bisecţie este un caz particular de căutare binară. Deoarece se lucrează cu valori reale, comparaţia de egalitate se evită.

```
double Bis(double a, double b, double (*f)(double)){
  double c = (a+b)/2;
  if(fabs(f(c))< EPS || b-a < EPS) return c;
  if(f(a)*f(c) < 0)
    return Bis(a, c, f);
  else
    return Bis(c, b, f);
}
```

4. Elementul maxim dintr-un vector

Aplicarea metodei divizării se bazează pe observaţia că maximul este cea mai mare valoare dintre maximele din cele două jumătăţi ale tabloului.

```
double max(int i, int j, double *x){
  double m1, m2;
  if(i==j) return x[i];
  if(i==j-1) return ( x[i]>x[j]? x[i]:x[j];);
  int k = (i+j)/2;
  m1 = max(i, k, x);
  m2 = max(k+1, j, x);
  return (m1>m2)? m1: m2;
}
```

5. Turnurile din Hanoi

Se dau 3 tije:stânga, centru, dreapta; pe cea din stânga sunt plasate **n** discuri cu diametre descrescătoare, formând un turn. Generaţi mutările care deplasează turnul de pe tija din stânga pe cea din dreapta. Se impun următoarele restricţii:

- la un moment dat se poate muta un singur disc
- nu se permite plasarea unui disc de diametru mai mare peste unul cu diametrul mai mic
- una din tije serveşte ca zonă de manevră.

Problema mutării celor **n** discuri din zona sursă în zona destinaţie poate fi redusă la două subprobleme:

- mutarea a **n−1** discuri din zona sursă în zona de manevră, folosind zona destinaţie ca zonă de manevră
- mutarea a **n−1** discuri din zona de manevră în zona destinaţie, folosind zona sursă ca zonă de manevră
- Între cele două subprobleme se mută discul rămas în zona sursă în zona destinaţie

Avem satisfăcută ecuaţia recurentă: **T(n) = 2.T(n−1) + 1,** care se rescrie:

$$T(n)=2^2T(n-2)+1)+1=2^2T(n-2)+2+1=2^3T(n-3)+2^2+2+1=$$
$$2^{n-1}\ T(1)\ +2^{n-2}\ +\ldots+2+1$$
$$T(n)=2^{n-1}+...+2+1=2^n-1=O(2^n)$$

```
#include <stdio.h>
#include <stdlib.h>
// Turnurile din Hanoi - mutarile pentru deplasarea a n discuri
// de pe tija din stanga pe cea din dreapta, folosind tija din
// mijloc ca zona de manevra
void hanoi(int n, int sursa, int man, int dest);
int main(){
  int n;
  printf("n="); scanf("%d", &n);
  hanoi(n, 1, 2, 3);
  return 0;
}
```

```
void hanoi(int n, int sursa, int man, int dest){
   if(n){
      hanoi(n-1, sursa, dest, man);
      printf("%2d - %2d\n", sursa, dest);
      hanoi(n-1, man, sursa, dest);
   }
}
```