

Programarea calculatoarelor

funcții; sintaxă; transmiterea argumentelor prin valoare și prin adresă; transmiterea vectorilor și a matricilor; recursivitate

Pe măsură ce aplicațiile devin mai complexe, este necesar ca sarcinile de rezolvat să poată fi împărțite în subsarcini pentru ca acestea să poată fi implementate mai ușor. Pentru aceasta, limbajul C oferă posibilitatea de a implementa diverse sarcini în **funcții** separate, fiecare funcție realizând o anumită operație. O funcție are următoarea sintaxă:

```
tip_returnat  nume_funcție(param1, param2, ..., paramN)
{
    instrucțiuni
}
```

Cu următoarele componente:

- **tip_returnat** – tipul returnat de funcție (ex: *int*, *float*). Dacă funcția nu returnează nicio valoare, tipul returnat va fi *void* („nimic”).
- **nume_funcție** – numele funcției, analog cu numele unei variabile
- **param₁, ..., param_N** – parametrii funcției, fiecare dat ca o definiție separată de variabilă (ex: *int a*, *float c*). La parametrii unei funcții este nevoie ca fiecare parametru să aibă propriul său tip, chiar dacă e identic cu cel anterior (nu se acceptă „*int a,b*”). În C nu se pot da valori inițiale parametrilor (nu se acceptă „*int a=3*”). Parametrii se pun între paranteze. Chiar dacă funcția nu are niciun parametru, tot trebuie să existe parantezele de parametri, scrise ca „()”.
- **{...instrucțiuni...}** – corpul funcției. Constă dintr-o serie de instrucțiuni puse între acolade „{}”.

O definiție de funcție corespunde notației matematice:

$r = \text{nume_funcție}(p_1, p_2, \dots, p_N)$, cu $r \in \text{tip_returnat}$, $p_1 \in T_1$, $p_2 \in T_2$, ..., $p_N \in T_N$ unde T_1, T_2, \dots, T_N sunt tipurile lui p_1, p_2, \dots, p_N

Exemplu: Se cere un număr impar $n > 4$. Să se deseneze litera „E” în așa fel încât pe verticală și pe orizontală să fie câte n steluțe.

Varianta 1: Rezolvarea acestei probleme **fără funcții** arată ca în programul următor:

```
#include <stdio.h>
int main()
{
    int n, i;
    printf("n=");
    scanf("%d", &n);
    for (i = 0; i < n; i++)           // @1 - linia orizontala de sus
        printf("*");
    printf("\n");
    for (i = 0; i < (n - 3) / 2; i++) // @2 - linia verticala superioara
        printf("*\n");
    for (i = 0; i < n; i++)           // @3 - linia orizontala de la mijloc
        printf("*");
```

```

printf("\n");
for (i = 0; i < (n - 3) / 2; i++)    // @4 - linia verticala inferioara
    printf("*\n");
for (i = 0; i < n; i++)              // @5 - linia orizontala de jos
    printf("*");
printf("\n");
return 0;
}

```

Se poate constata faptul că există două secvențe de cod care se repetă: trasarea unei linii orizontale se repetă la liniile @1,3,5 și trasarea unei linii verticale se repetă la liniile @2,4. Avem astfel două secvențe de cod duplicate, una de 3 ori, alta de 2 ori.

Duplicarea de cod într-o aplicație este în mod ferm interzisă de normele de practică în programare.

Următoarele două motive sunt evidente:

- dacă avem de modificat ceva, atunci acea modificare trebuie realizată în toate secvențele duplicate.
- dacă într-o asemenea secvență există o eroare, atunci eroarea va fi prezentă în toate duplicatele. Când se dorește remedierea erorii, remedierea trebuie și ea propagată în toate duplicatele.

Când programele sunt mici, aceste aspecte nu sunt foarte importante. Când dimensiunea programelor crește și codul este scris în mai multe fișiere, este foarte posibil să uităm să modificăm în toate locurile unde apare duplicată acea secvență, sau este posibil ca la unul dintre duplicate să se strecoare o greșeală la modificare. În această situație în aplicație vor apărea erori (bugs) care în general sunt greu de localizat și de reparat.

Duplicarea de cod se rezolvă prin **factorizare**, extrăgând codul comun și implementându-l cu ajutorul unei funcții, funcție care ulterior va putea fi apelată din toate punctele din program de unde este nevoie. Prin această metodă, cele două deficiențe de mai sus dispar:

- dacă avem de modificat ceva, vom modifica într-un singur loc (în corpul funcției)
- depanarea erorilor se face și ea mai simplu, testând și modificând doar codul funcției

Un alt avantaj major al funcțiilor este faptul că în timp, pe măsură ce implementăm diverși algoritmi, vom avea o bibliotecă de funcții bine puse la punct. Vom putea refolosi aceste funcții și în proiecte viitoare, ceea ce va reduce considerabil timpul de dezvoltare al unei aplicații.

Limbajul C dispune de o colecție bogată de funcții standard (C standard library). Acestea sunt grupate pe categorii și sunt accesibile prin intermediul includerii unor fișiere antet (header, .h). Printre funcțiile standard se găsesc funcții pentru fișiere, pentru caractere sau șiruri de caractere, matematică, formatare, gestiunea memoriei, etc. Este bine să se parcurgă aceste funcții pentru a ne forma o idee despre ce ni se pune la dispoziție și deci ce putem folosi fără a mai fi nevoie ca noi să implementăm de la zero.

Pe lângă funcțiile C standard, mai putem găsi biblioteci de funcții pentru cele mai variate aplicații: grafică, rețele de calculatoare, simulare, baze de date, etc. Multe dintre aceste biblioteci au fost dezvoltate ani de zile de către echipe de programatori și implementează funcționalități complexe, astfel încât folosirea lor ne poate scuti de foarte mult efort la scrierea aplicației noastre.

Varianta 2: rezolvarea problemei anterioare folosind funcții:

```

#include <stdio.h>

void linieOrizontala(int n)
{
    int i;
    for (i = 0; i < n; i++)

```

```

    printf("***");
    printf("\n");
}

void linieVerticala(int n)
{
    int i;
    for (i = 0; i < (n - 3) / 2; i++)
        printf("*\n");
}

int main()
{
    int n;
    printf("n=");
    scanf("%d", &n);
    linieOrizontala(n);           // linia orizontala de sus
    linieVerticala(n);           // linia verticala superioara
    linieOrizontala(n);           // linia orizontala de la mijloc
    linieVerticala(n);           // linia verticala inferioara
    linieOrizontala(n);           // linia orizontala de jos
    return 0;
}

```

În această variantă s-au folosit două funcții:

- *linieOrizontala* - pentru trasarea unei linii orizontale
- *linieVerticala* - pentru trasarea unei linii verticale

Fiecare funcție primește un singur parametru „*int n*” și nu returnează nicio valoare (sunt de tip „**void**”). Se poate constata că fiecare funcție este destul de simplă și ușor de citit, deoarece conține doar o secvență cu funcționalitate specifică de cod. În același timp și programul principal, *main*, este mult mai simplu și mai lizibil, deoarece dacă funcțiile au nume bine alese, programatorul are o imagine intuitivă asupra rolului lor.

Și în această a doua variantă apare o ușoară duplicare de cod: secvența *linieOrizontala/linieVerticala* se repetă de două ori. Putem elimina și această duplicare, implementând o funcție „*linieOV(int lungime, int inaltime)*” ceea ce ne-ar ajuta și la scrierea altor litere, gen L, F, C.

În C, fiecare funcție trebuie scrisă în afara altei funcții, deci nu am putut să scriem cele două funcții auxiliare în interiorul lui *main*. La fel ca și în cazul variabilelor, **o funcție trebuie mai întâi definită și abia apoi folosită**. Din acest motiv, dacă am fi implementat cele două funcții după *main*, am fi avut eroare de compilare în *main*, fiindcă nu s-ar fi știut ce înseamnă *linieOrizontala* și *linieVerticala*. De aceea, funcțiile au trebuit implementate **înainte** de a fi folosite.

Funcția **main** este o funcție obișnuită ca sintaxă, dar compilatorul o interpretează ca fiind **începutul** programului, chiar dacă înaintea ei mai sunt și alte funcții. Din acest motiv, întotdeauna un program C începe cu funcția *main*, oriunde ar fi aceasta. Valoarea returnată de funcția *main* este considerată de către sistemul de operare (Windows, Linux, etc) un indicator al felului în care s-a executat aplicația: cu succes (valoarea returnată 0) sau cu eroare (altă valoare).

Dacă o variabilă se declară într-o funcție (ex: *i* din funcția *linieOrizontala*), atunci acea variabilă este **locală** funcției respective și ea nu există decât în interiorul acelei funcții (este vizibilă doar în interiorul funcției). La fel și parametrii unei funcții sunt locali funcției în care au fost declarați. Mai multe funcții pot să aibă aceleași nume pentru parametri și pentru variabile locale (ex: *linieOrizontala* și *linieVerticala* au fiecare definită câte o variabilă *i*), fără ca cele două variabile să se suprapună sau să existe vreo problemă în selectarea lor de către compilator.

Dacă o variabilă este definită în afara unei funcții, ea este **globală**. Iată un tabel comparativ între variabilele locale și cele globale:

Variabile locale	Variabile globale
Nu sunt inițializate de compilator, deci inițial au valori nedeterminate.	Sunt inițializate chiar de la începutul programului cu valoarea 0.
Există (ocupă memorie) doar pe durata apelului funcției.	Există de la începutul până la sfârșitul programului,
Sunt vizibile (accesibile) doar în funcția în care sunt declarate.	Sunt vizibile din tot programul.

Aplicația 9.1: Se cere un număr impar $n > 4$. Să se deseneze cifra „8”, scris ca un pătrat cu o linie orizontală în mijloc, în așa fel încât pe verticală și pe orizontală să fie câte n steluțe. În program nu vor fi admise duplicări de cod.

O funcție poate returna o valoare sau poate fi întreruptă din execuție folosind instrucțiunea **return**:

```
return expresie;    //pentru funcții care returnează o valoare
return;             //pentru funcții void
```

return poate să apară oriunde în corpul unei funcții, nu doar la sfârșit. El are ca efect terminarea imediată a funcției respective și revenirea în programul apelant.

Aplicația 9.2: Să se scrie o funcție care returnează maximumul dintre 3 valori de tip *float*, primite ca parametri. Funcția nu va folosi nicio variabilă, cu excepția parametrilor săi. Se va testa funcția cu valori introduse de la tastatură.

Transmiterea argumentelor prin valoare și prin adresă

Când se apelează o funcție, valorile expresiilor date ca argumente se vor copia în parametrii funcției și apoi funcția va folosi aceste copii ale valorilor originare. Acest mod de apel se numește **transmitere prin valoare**. Din acest motiv, chiar dacă modificăm valoarea unui parametru în interiorul funcției, această modificare se efectuează doar pe copia valorii cu care a fost inițializat acel parametru, fără ca valoarea originală să fie afectată.

Exemplu: Să se scrie o funcție *swap(x,y)*, care să interschimbe valorile variabilelor date ca parametri.

Varianta 1: *swap1* nu implementează corect cerința din exemplu!!!

```
void swap1(int x,int y)
{
    int tmp=x;
    x=y;
    y=tmp;
}

int main()
{
    int a=5,b=7;
    swap1(a,b);
    printf("%d %d\n",a,b);    // 5 7
    return 0;
```

```
}
```

Într-o primă variantă încercăm să scriem o funcție *swap1*, care are 2 parametri pe care-i interschimbă. Apelăm această funcție cu două variabile *a* și *b*. Ne-am aștepta ca după apelul funcției, valorile lui *a* și *b* să fie interschimbate, iar pe ecran să se afișeze "7 5". Cu toate astea, se vor afișa tot valorile inițiale, deci funcția *swap1* nu le-a interschimbato.

De ce *swap1* nu a interschimbato valorile lui *a* și *b*? Răspunsul constă chiar în transmiterea parametrilor prin valoare. În acest caz, la apelul lui *swap1(a,b)*, valorile lui *a* și *b* au fost copiate în *x* și *y*. Ulterior, în timpul execuției lui *swap1*, când *x* și *y* au fost interschimbate, de fapt s-au interschimbato valorile din aceste două variabile, fără ca originalele *a* și *b* să fie afectate în vreun fel.

Pentru a implementa funcții care modifică valorile originare ale variabilelor cu care au fost apelate, putem folosi **transmiterea prin adresă**. Pentru aceasta, vom transmite funcției adresele variabilelor, iar funcția va folosi pointeri pentru a opera cu valorile de la aceste adrese. În acest fel, orice modificare operată asupra valorii de la o anumită adresă, va modifica de fapt chiar variabila originală, deoarece acea variabilă se află amplasată la adresa respectivă din memorie.

Varianta 2: *swap* folosește transfer prin adresă și astfel reușește să interschimbe valorile variabilelor *a* și *b*

```
void swap(int *x,int *y)    // swap are ca parametri doi pointeri, deci două adrese de memorie
{
    int tmp=*x;             // se operează asupra valorilor pointate de cei doi pointeri
    *x=*y;                  // (valorile de la adresele stocate în x și y)
    *y=tmp;
}

int main()
{
    int a=5,b=7;
    swap(&a,&b);             // transmitere prin adresă: funcția swap primește adresele variabilelor a și b
    printf("%d %d\n",a,b);  // 7 5
    return 0;
}
```

Folosind transmiterea prin adresă, am reușit să implementăm funcția *swap*. Cu acest mod de transmitere, putem implementa orice funcție care trebuie să modifice valorile parametrilor săi. Putem astfel înțelege de ce funcția *scanf* are nevoie de adresele variabilelor în care va depune valorile citite (ex: *scanf("%d",&a);*): funcția *scanf* folosește adresele variabilelor pentru a ști unde anume în memorie să depună valorile citite. Altfel, dacă am fi încercat să scriem o funcție *scanf* fără pointeri, la apelul ei s-ar fi copiat valorile variabilelor în parametrii funcției, citirea s-ar fi făcut în aceste copii, iar variabilele originare ar fi rămas neschimbate.

Folosind transmiterea prin adresă, putem scrie funcții care returnează mai multe valori, nu doar una, ca în cazul folosirii lui *return*. Pentru aceasta, vom include în funcție câte un nou parametru pentru fiecare valoare returnată, parametru care va folosi transmitere prin adresă. La apelul funcției, acești parametri vor fi inițializați cu adresele variabilelor destinație pentru valorile returnate. În funcție se vor folosi parametrii respectivi pentru a seta la adresele variabilelor destinație valorile care trebuie returnate.

Aplicația 9.3: Să se scrie o funcție care returnează două valori pe care le citește de la tastatură, sortate în mod crescător. Se va verifica funcția.

Transmiterea vectorilor și a matricilor ca argumente

Când o funcție primește ca argument un vector, ea va primi de fapt adresa acelui vector. În acest caz, vectorul este interpretat ca pointer și se transmite funcției doar adresa lui, fără a se copia niciun element. Deci vectorii se transmit implicit prin adresă. Din acest motiv, orice modificare efectuată în funcție asupra valorilor unui vector, de fapt se va efectua chiar asupra vectorului original.

Deoarece pointerii nu știu numărul de elemente care se află în memorie începând de la adresa pointată de ei, rezultă că la transmiterea vectorilor va trebui să transmitem funcției încă un parametru suplimentar, în care să specificăm numărul de elemente din vector.

Exemplu: Să se scrie o funcție care primește ca parametru un vector de tip *float* și numărul său de elemente și returnează 1 (true) dacă toate valorile sunt pozitive sau 0 (false) dacă există și valori negative în vector. Se va testa funcția folosind un program care cere un $n \leq 10$ și apoi n elemente de tip *float*.

```
#include <stdio.h>

int pozitive(float v[], int n)           // alternativ: float *v
{
    int i;
    for (i = 0; i < n; i++){
        if (v[i] < 0)                   // daca a gasit un element negativ
            return 0;                   // iese imediat din functie returnand 0
    }
    return 1;                           // daca nu a gasit niciun element negativ
}

int main()
{
    int n, i;
    float v[10];
    printf("n=");
    scanf("%d", &n);
    for (i = 0; i < n; i++){
        printf("v[%d]=", i);
        scanf("%g", &v[i]);
    }
    if (pozitive(v, n)){
        printf("toate elementele sunt pozitive");
    }else{
        printf("exista si elemente negative");
    }
    return 0;
}
```

Vectorii transmiși ca argumente se scriu fără să se specifice dimensiunea lor (ex: *float v[]*). Se iterează vectorul, și dacă se găsește un element negativ se revine imediat din funcție, folosind „*return 0;*”, pentru că în acest caz nu mai are rost să se continue execuția funcției. Dacă s-a terminat instrucțiunea *for*, înseamnă că nu s-a descoperit niciun element negativ și atunci în final se returnează 1. În programul principal, valoarea returnată de funcție se folosește direct în *if*, pe baza faptului că în C „0” înseamnă *false* și orice altă valoare înseamnă *true*.

Aplicația 9.4: Să se scrie un program care citește un număr $n < 10$ iar apoi citește 2 vectori $v1$ și $v2$ de câte n elemente de tip *int*. Pentru citirea elementelor unui vector se folosește o funcție *citire(v,n)*. Se va implementa o funcție *egal(v1,v2,n)*, care testează dacă toate elementele din $v1$ sunt egale cu cele din $v2$ la poziții corespondente.

În același mod ca vectorii, matricile, atunci când sunt transmise ca argumente, nu se copiază în funcție, ci este transmisă doar adresa lor. Spre deosebire de vectori, la matrici parametrul funcției trebuie să aibă specificat numărul de coloane de la declararea matricii (nu cel folosit efectiv, conform dimensiunilor introduse de la tastatură), deoarece acest număr intervine în formula de calcul a adresei unui element ($\&a[i][j] = \&a[0][0] + nr_col_max * i + j$), deci compilatorul are nevoie de el.

Exemplu: Se consideră o matrice care are maxim 10 linii și 10 coloane. Să se scrie două funcții, una care citește valori în matrice și alta care afișează matricea.

```
#include <stdio.h>

// citește elementele unei matrici a[m][n]
// numărul de coloane trebuie să coincidă cu cel de la declararea matricii
void citire(int a[][10], int m, int n)
{
    int i, j;
    for(i=0; i<m; i++){
        for(j=0; j<n; j++){
            printf("a[%d][%d]= ", i, j);
            scanf("%d", &a[i][j]);
        }
    }
}

void afisare(int a[][10], int m, int n)
{
    int i, j;
    for(i=0; i<m; i++){
        for(j=0; j<n; j++){
            printf("%d\t", a[i][j]);
        }
        printf("\n");
    }
}

int main()
{
    int a[10][10];
    int m, n;
    printf("m="); scanf("%d", &m);
    printf("n="); scanf("%d", &n);
    citire(a, m, n);
    afisare(a, m, n);
    return 0;
}
```

Ținând cont că elementele unei matrici sunt dispuse în memorie în mod consecutiv, linie după linie, putem să preluăm adresa unei linii și să considerăm că de la această adresă începe un vector de elemente, care sunt de fapt chiar elementele din linia respectivă. Putem astfel să folosim la matrici orice funcții care procesează vectori.

Exemplu: considerăm că avem o funcție $vmax(v,n)$, care returnează maximul unui vector v de n elemente:

```
// returneaza maximul unui vector v de n elemente
int vmax(int *v, int n){
    int m=*v;
    for(v++; --n ; v++){
        if (m < *v) {
            m = *v;
        }
    }
    return m;
}
```

Această funcție inițializează prima oară maximul cu valoarea primului element, iar apoi folosește parametrul v ca iterator, pentru a parcurge toate elementele. Chiar dacă în funcție valoarea lui v se modifică, această schimbare nu este vizibilă din exterior (ex: nu se modifică adresa unui vector care este transmis funcției), deoarece v în sine este transmis prin valoare (adresa unui vector se copiază în v).

Folosind $vmax$, putem afișa foarte simplu maximul unei matrici $a[m][n]$, considerând matricea ca fiind un vector cu $m*n$ elemente:

```
printf("%d\n", vmax(&a[0][0],m*n));    // matricea este vazuta ca un vector de m*n elemente
```

Tot folosind $vmax$, putem determina maximul elementelor de pe linia k , ținând cont că adresa de început a acestei linii este $\&a[k][0]$ și va trebui să testăm n elemente:

```
printf("%d\n", vmax(&a[k][0],n));
```

În general, dacă folosim funcții pe vectori, putem să operăm asupra oricăror elemente consecutive dintr-o matrice, inclusiv în interiorul unei linii (ex: să determinăm maximul ultimelor 3 elemente din linie). În schimb, cu funcțiile pe vectori, nu vom putea să operăm la nivel de coloane, deoarece elementele unei coloane nu sunt dispuse alăturat în memorie.

Deoarece operarea cu o singură linie dintr-o matrice este destul de des întâlnită în practică, limbajul C ne permite să izolăm direct adresa de început a unei linii, scriind $a[k]$, ceea ce este echivalent cu $\&a[k][0]$. Constatăm că se respectă și aici faptul că de fapt matricile sunt vectori de vectori, prin faptul că scriind doar $a[k]$, de fapt accesăm vectorul corespunzător liniei k din matrice. Cu această simplificare, afișarea maximului unei linii devine:

```
printf("%d\n", vmax(a[k],n));
```

Aplicația 9.5: Se citesc numerele $m < 10$, $3 < n < 10$ și apoi elementele de tip *float* ale unei matrici $a[m][n]$. Ulterior, se va citi un număr $0 < k < 5$ și apoi elementele de tip *float* ale unui vector $v[k]$. Să se afișeze toate pozițiile unde vectorul v se găsește în liniile matricii. De exemplu, dacă v începe în matrice la poziția $a[i][j]$, se va afișa (i,j) . Dacă vectorul nu a fost găsit în nicio linie, se va afișa un mesaj corespunzător. Pentru testarea egalității elementelor se va folosi funcția *egal*, definită la aplicația 9.4.

Recursivitate

În anumite cazuri, soluția unei probleme este definită în funcție de rezultatele anterioare ale aceleiași probleme. De exemplu „n!” (factorialul) se poate defini ca:

1 dacă $n=1$
 $n*(n-1)!$ dacă $n>1$

Adică factorialul unui număr $n>1$ se definește în funcție de factorialul numărului său anterior. Numim această modalitate de definire a unei probleme „**recursivă**” sau „**prin recurență**”. Recursivitatea este o modalitate foarte puternică de definire și de multe ori duce la soluții concise și clare. Ea nu este limitată doar la matematică ci se folosește în multe aplicații, de exemplu:

- *relație recursivă copil-parinte*: Un om are un tată și o mamă. Aceștia la rândul lor au fiecare un tată și o mamă. Să se determine dacă doi oameni sunt înrudiți genealogic.
- *definire recursivă a unei expresii în funcție de expresii mai simple*: O expresie matematică este un număr sau o aplicare a unuia dintre operatorii adunare, scădere, înmulțire, împărțire, având ca operanzi expresii. Să se evalueze o expresie dată.
- *definirea recursivă a unor figuri geometrice de tip fractal*: Un copac începe cu o singură ramură (trunchi) din care pot crește mai multe ramuri. La rândul lor, din acestea pot crește alte ramuri sau frunze. Când a crescut o frunză, aceasta se consideră terminal și din ea nu mai pot crește alte ramuri sau frunze.

Recursivitatea este ușor de implementat cu ajutorul unei funcții care se apelează pe ea însăși (este recursivă).

Exemplu: Să se scrie o funcție care primește ca parametru un întreg $n \geq 0$ și returnează Fibonacci(n), definit astfel: $F(0)=0$, $F(1)=1$, $F(n)=F(n-2)+F(n-1)$, pentru $n>1$.

```
#include <stdio.h>

int f(int n)
{
    if (n == 0)                // conditii de terminare a recursivitatii
        return 0;
    if (n == 1)
        return 1;
    return f(n - 2) + f(n - 1); // apel recursiv
}

int main()
{
    int n;
    printf("n=");
    scanf("%d", &n);
    printf("Fibonacci(%d): %d", n, f(n));
    return 0;
}
```

Când se implementează o funcție recursivă, în primul rând trebuie să fie definită clar condiția sau condițiile de terminare a recursivității. Dacă aceste condiții nu există sau nu acoperă toate cazurile posibile, funcția va continua să se apeleze pe ea însăși la infinit.

Pentru a se înțelege mai bine ce se petrece la un apel recursiv, trebuie să se țină cont de faptul că într-un apel recursiv, funcția apelantă (cea care a inițiat apelul) continuă să existe și pe lângă ea va apărea în memorie și funcția apelată. Vom avea astfel simultan în memorie două sau mai multe apeluri de funcții, fiecare cu proprii săi

parametri și variabile locale. De exemplu, putem avea simultan în memorie 1000 de apeluri ale funcției Fibonacci, fiecare cu propriul său parametru n . Funcția apelantă va aștepta ca funcția apelată să-și încheie execuția, și apoi va continua cu valoarea returnată de aceasta. De exemplu, $\text{Fibonacci}(4)$ va trece prin următoarea secvență de apeluri: $f(4) = f(2)+f(3) = (f(0)+f(1))+(f(1)+f(2)) = (0+1)+(1+(f(0)+f(1))) = 1+(1+(0+1)) = 1+(1+1) = 1+2 = 3$. Se poate reprezenta o asemenea secvență de apeluri sub forma unui arbore care pleacă de la funcția inițială și fiecare ramură corespunde unui apel recursiv.

Aplicația 9.6: Cel mai mare divizor comun (*cmmdc*) al două numere a și b este definit recursiv astfel:

a	dacă $a==b$
$\text{cmmdc}(a-b,b)$	dacă $a>b$
$\text{cmmdc}(a,b-a)$	dacă $b>a$

Să se scrie o funcție recursivă pentru calculul *cmmdc* și să se testeze aceasta pe două numere de la tastatură.

Aplicații propuse

Aplicația 9.7: Folosind funcții standard din biblioteca matematică C (*math.h*), să se scrie o funcție care primește ca parametri coordonatele reale a două puncte (x_0,y_0,x_1,y_1) și returnează unghiul în grade $[0^\circ, 360^\circ)$ dintre segmentul $(x_0,y_0)-(x_1,y_1)$ și axa OX.

Aplicația 9.8: Se cere să se scrie funcția lui *Ackermann* pentru două numere m,n date ca parametri. Să se testeze funcția cu valori citite de la tastatură. Funcția lui Ackermann se definește astfel:

$n+1$	dacă $m=0$
$A(m-1,1)$	dacă $m>0$ și $n=0$
$A(m-1,A(m,n-1))$	dacă $m>0$ și $n>0$

Aplicația 9.9: Scrieți o funcție care parcurge un vector și returnează indicii elementelor minim și maxim.