

## Cuprins

<b>Gândirea algoritmică</b> .....	<b>1-3</b>
Problema căutării.....	1-2
Problema intrării într-un cabinet medical .....	2-3
<b>Structura unui program și a unei funcții C</b> .....	<b>3-5</b>
Reprezentarea numărului în binar.....	4-5
Transmiterea parametrilor în cadrul unei funcții .....	5
<b>Construcțiile de bază ale limbajului C</b> .....	<b>6-13</b>
Caracterele.....	6
Identificatorii.....	6
Cuvinte cheie.....	6
Tipuri de date .....	6-7
Tipuri aritmetice.....	7
Tipul întreg.....	7-8
Tipul flotant.....	8
Tipul caracter.....	8
Constante și variabile .....	9-10
Comentarii.....	10
Operatori .....	10
Operatori aritmetici .....	10-11
Operatorul de conversie explicită (cast) .....	11
Operatorul dimensiune (sizeof) .....	12
Operatorul condițional.....	12-13
<b>Structuri de date</b> .....	<b>13-23</b>
Lista liniară simplu înlănțuită.....	13-18
Structura de tip stivă (LIFO Last In First Out) .....	18-21
Structura de tip coadă .....	21-23
<b>Instrucțiuni C</b> .....	<b>23-32</b>
Instrucțiunea vidă.....	23
Instrucțiunea de atribuire.....	24
Instrucțiunea compusă.....	24
Instrucțiunea if.....	24-25
Instrucțiunea while .....	25
Cel mai mare divizor comun.....	25
Instrucțiunea do while .....	26
Instrucțiunea for .....	26-27
Programul C care calculează $n!$ .....	26-27
Instrucțiunea return.....	27-29
O funcție care returnează maximul a 2 numere.....	27
Numărul de elemente ale unei șir.....	27-28
Comparația celor 2 șiruri .....	28
Concatenare.....	28-29
Instrucțiunea break .....	29
Funcția exit.....	30
Instrucțiunea continue .....	30-31
Numerarea cifrelor existente într-un șir .....	30-31
Instrucțiunea goto.....	31
Instrucțiunea switch.....	31-32
<b>Pointeri C</b> .....	<b>32-39</b>
Operatori specifici pointerilor .....	33-34
Pointeri și tablouri .....	34-35
Pointeri la funcții .....	35-36

Tipuri structurate de date.....	36–39
Structuri .....	36–37
Câmpuri de biți .....	37–38
Uniuni .....	38
Enumerări .....	38–39
<b>Funcții de bibliotecă .....</b>	<b>39–43</b>
Funcția printf .....	39–40
Funcția scanf .....	40–42
Funcții de conversie.....	42–43
Funcții generatoare de numere aleatoare.....	42
Funcții pentru operații cu caractere și șiruri de caractere.....	43
<b>Operații cu fișiere .....</b>	<b>43–45</b>
<b>Calcul Matriceal .....</b>	<b>45–51</b>
Produsul a două matrici.....	45–46
Inversa unei matrici.....	46–49
Metoda lui Gauss.....	49–51
<b>Metode de sortare.....</b>	<b>51–64</b>
Sortare ordinară.....	51–53
Sortare prin selecție (Selection Sort) .....	53–54
Sortarea prin inserție directă (Direct Insertion Sort).....	54–56
Sortare prin inserție binară (Binary Insertion Sort) .....	56–58
Sortare prin inserție directă folosind o santinelă.....	58–59
Sortarea prin metoda bulelor (Bubble Sort).....	59–60
Sortare rapidă (Quick Sort).....	61–62
Sortare prin interclasare (Merge Sort) .....	62–64
<b>Recursivitate .....</b>	<b>64–65</b>
Calculul valorii $n!$ .....	64
Algoritmul lui Euclid recursiv .....	64–65
Să se genereze primele $n$ numere din șirul lui Fibonacci.....	65
<b>Metoda backtracking .....</b>	<b>66–76</b>
Permutările .....	66–67
Problema aranjamentelor.....	67
Problema combinărilor .....	67–69
Problema regiunilor .....	69–70
Problema labirintului .....	70–72
Problema calului .....	72–74
Problema mingii .....	74–76
<b>Metoda Divide et Impera .....</b>	<b>77–80</b>
Suma elementelor unui șir .....	77
Problema “Turnurilor din Hanoi” .....	77–79
Elementul maxim într-un șir.....	79
Problema căutării binare.....	79–80
<b>Grafuri neorientate .....</b>	<b>80–92</b>
Implementarea parcurgerii în lățime pentru un graf neo.....	80–82
Implementarea parcurgerii în adâncime pentru un graf neo.....	82–83
Drumuri într-un graf.....	83–84
Grafuri ponderate .....	84–86
Graful hamiltonian .....	86–88
Grafuri euleriene.....	88–90
Implementarea unui graf utilizând matricea de adiacență.....	90–92
Implementarea unui graf utilizând pointeri.....	92
Drumul optim într-un graf.....	92

# Gândirea algoritmică

## Problema căutării

Există și cazul care, pentru o aceeași problemă putem prezenta două soluții, în care una este mai rapidă ca alta. De pildă, fie un șir de numere naturale oarecare, de exemplu 4,2,10,1,8,15,7. Vrem să testăm dacă un număr dat (să zicem numărul 8) se află în această secvență sau nu.

O astfel de căutare, prin parcurgerea de la stânga la dreapta a întregii secvențe de numere, până se găsește numărul dorit sau se epuiează toate elementele din secvență, se numește căutarea secvențială. Dacă avem un șir  $S$  de  $n$  elemente (notat  $S[1..n]$ ), atunci căutarea secvențială a lui  $x$  în  $S$  se descrie prin:

Căutare\_secvențială( $x, S[1..n]$ ) înseamnă

Început

Fie  $elemental\_curent = primul\_element$ ;

Atât timp cât ( $elemental\_curent \neq x$ ) și (poziția elementului  $current \leq$  poziția ultimului element ( $n$ )) execută

Început

Dacă  $elemental\_curent = x$  atunci mesaj('găsit')

Altfel treci la următorul element

Sfârșit

Sfârșit.

În continuare să presupunem că numerele erau deja ordonate crescător 1, 2, 4, 7, 8, 10, 15. În acest caz particular, căutarea secvențială a unui număr cum este 8 nu este prea eficientă, deoarece 8 se află în a doua jumătate a secvenței, deci ar fi de preferat să nu-l căutăm în prima jumătate. Acest procedeu este mai rapid:

- Dacă numărul din mijloc este mai mic decât numărul căutat, atunci căutăm a doua jumătate;
- Dacă numărul din mijloc este mai mare ca numărul căutat, atunci căutăm în prima jumătate;
- Dacă numărul din mijloc este egal cu numărul căutat, înseamnă că am găsit numărul în cauză și trebuie să oprim căutarea.

Căutarea în jumătatea aleasă se face tot la fel, deci se va înjumătăți și această zonă...

Procedeu anterior se numește căutare binară, deoarece, de fiecare dată, o secvență de numere este divizată în două jumătăți. Putem descrie căutarea binară a unui număr  $x$  într-o secvență  $S[1..n]$  astfel.

Căutarea\_binară( $x, S[1..n]$ ) înseamnă

Început

Stabilește elemental\_curent ca fiind elemental din mijlocul secvenței;

Dacă  $n=1$  atunci

Dacă  $x = \text{elemental\_curent}$  atunci mesaj('găsit')

Altfel mesaj('negăsit')

Altfel

Început

Împarte secvența în cele două jumătăți ( $S[1..mijloc]$  și  $S[mijloc+1..n]$ );

Dacă  $\text{elemental\_curent} = x$  atunci mesaj('găsit')

Altfel

Dacă  $\text{elemental\_curent} < x$  atunci

Căutare\_binară( $x, S[mijloc+1..n]$ )

Altfel căutarea\_binară( $x, S[1..mijloc]$ )

Sfârșit

Sfârșit.

## Problema intrării într-un cabinet medical

Dacă un om vrea să intre pentru consult la un medic, atunci, mai întâi va bate la ușă: dacă medicul răspunde prin „poftim!”, atunci va intra, altfel va aștepta până când pacientul dinăuntru va ieși; abia după ce cabinetul va fi liber, va intra.

Intrare\_la\_medic înseamnă

Început

Bate la\_ușă;

Dacă răspunsul = „poftim!” atunci

Întră\_în\_cabinet;

Altfel

Început

Atât timp cât cabinetul\_este\_ocupat\_de\_alt\_pacient execută

Citeste\_un\_ziar;

Sfârșit

Sfârșit.

O instrucțiune compusă este formată dintr-o secvență de instrucțiuni, încadrate de cuvinte început și sfârșit, care pot conține, la rândul lor, blocuri de alternativă și repetiție.

Programarea este tehnica realizării de algoritmi descriși prin proceduri și programe. Ea devine o artă, atunci când se folosesc cele trei elemente de structurate și se numește programare structurată. Pentru a vedea ce ar însemna programare nestructurată, să reconsiderăm procedura de intrare în cabinetul medical:

```

Intare_la_medic înseamnă
Început
  Bate_la_ușă;
  Dacă răspunsul = ,poftim!' atunci intră_în_cabinet;
  Altfel Început
    Atât timp cât cabinetul_este_ocupat_de_alt_pacient execută
      Citeste_un_ziar;
    Intră_în_cabinet
  Sfârșit
Sfârșit.

```

## Structura unui program și a unei funcții C

Limbajul C este un *limbaj de programare procedural*, operațiile fiind grupate în blocuri de instrucțiuni delimitate de o pereche de acolade { }. La rândul lor, blocurile sunt grupate în una sau mai multe funcții. *O funcție C* este un modul care grupează în interiorul unei perechi de acolade un set de operații codificate sub forma unor instrucțiuni.

Deci conceptul de bază în C este cel de *funcție*. Fiecare funcție poate accepta parametri de intrare la apel și poate returna o valoare la revenire. Fiecare funcție are un nume precedat de un cuvânt cheie care desemnează tipul funcției (tipul valorii returnate de funcție). Numele funcției este urmat de o pereche de paranteze rotunde între care se specifică tipul și numele parametrilor funcției. Parantezele sunt necesare chiar dacă nu există parametri.

În limbajul C nu este permisă definirea unei funcții în interiorul altei funcții, lucru care este permis în limbajul Pascal. În limbajul C, toate funcțiile trebuie definite în mod independent.

Un program C se compune din una sau mai multe funcții, dintre care una este funcția principală. Fiecare funcție are un nume propriu, cu excepția funcției principale care se numește *main*. Orice program trebuie să aibă o funcție *main* iar execuția programului începe cu prima instrucțiune din această funcție.

*Structura generală* a unei funcții C este următoarea:

```

tip nume(lista_parametrii_formali)
declarare parametri
{
  instrucțiuni specifice funcției: declarare variabile locale,
  instrucțiuni executabile
}

```

Comandă versiunea completă, tipărită de  
[www.fituici-bacalaureat.ro](http://www.fituici-bacalaureat.ro)

[www.fituici-bacalaureat.ro](http://www.fituici-bacalaureat.ro)

## Operatorul dimensiune (sizeof)

Operatorul dimensiune returnează numărul de octeți ai reprezentării interne a unei date. Acest operator prelucrează tipuri, spre deosebire de ceilalți operatori care prelucrează valori. Operatorul dimensiune se utilizează în construcții de forma:

```
| sizeof(data)
```

unde *data* poate să fie variabilă simplă, nume de tablou, tip, element de tablou sau element de structură. Deoarece tipul operanzilor este determinat încă din faza de compilare, `sizeof` este un operator cu efect la compilare, adică operandul asupra căruia se aplică `sizeof` nu este evaluat, chiar dacă este reprezentat de o expresie.

Exemplu:

```
| char a;  
| int b;  
| float c;  
| int e[10];  
| printf("%d\n", sizeof(a)); //va fi afișată valoarea 1  
| printf("%d\n", sizeof(char)); //va fi afișată valoarea 1  
| printf("%d\n", sizeof(float)); //va fi afișată valoarea 4  
| printf("%d\n", sizeof(c)); //va fi afișată valoarea 4  
| printf("%d\n", sizeof(e)); //va fi afișată valoarea 20
```

Utilizarea operatorului `sizeof` permite creșterea portabilității programului.

## Operatorul condițional

Operatorul condițional se utilizează în expresii de forma:

```
| exp1 ? exp2 : exp3;
```

Această construcție are următorul *efect*: Se evaluează valoarea expresiei `exp1`. Dacă ea are valoarea adevărat, atunci se evaluează `exp2`, valoarea acesteia fiind și valoarea întregii expresii. Dacă `exp1` are valoarea fals, atunci se evaluează `exp3` iar valoarea lui `exp3` va fi și valoarea întregii expresii.

Exemplu:

```
| int a=5, b=3,c;
```

Comandă versiunea completă, tipărită de  
[www.fituici-bacalaureat.ro](http://www.fituici-bacalaureat.ro)

[www.fituici-bacalaureat.ro](http://www.fituici-bacalaureat.ro)

## Sortare rapidă (Quick Sort)

Principiul acestei metode este următorul: Se alege un element pivot din tabloul ce trebuie sortat. Tabloul este astfel partiționat în 2 subtablouri, alcătuite de o parte și de alta a acestui pivot, astfel: elementele mai mari decât pivotul sunt mutate în dreapta pivotului iar elementele mai mici în stânga pivotului. Cele 2 subtablouri sunt sortate în mod independent prin apeluri recursive ale algoritmului.

```
#include<stdio.h>
#include<conio.h>
#include<values.h>

//procedura recursivă de sortare a secvenței a[l]...a[r]
void qs(int a[], int l, int r)
{
    int v,i,j,elem;
    if(r>l)
    {
        v=a[r];
        //v este elementul pivot ales ca cel mai din dreapta element
        i=l-1;
        //indicele i pornește din stânga (-1 pentru că în ciclul
        //while de mai jos se face întâi incrementarea și apoi testul)
        j=r;
        //indicele j pornește din dreapta (va fi întâi decrementat
        //în ciclul while pornind de la r-1 deoarece a[r] este pivotul)
        for(;;)
        {
            while(a[++i]<v);
            //ieșim din ciclu la primul element a[i] mai mare sau
            //egal cu pivotul
            while(a[--j]>v);
            //ieșim din ciclu la primul element a[j] mai mic sau egal
            //cu pivotul
            if (i>=j) break;
            //dacă i>=j părăsim ciclul for
            elem=a[i];a[i]=a[j];a[j]=elem;
            //în cazul în care i<j inversăm a[i] cu a[j]
            //pentru a duce în stânga pivotului un element mai mic
            //și în dreapta lui un element mai mare decât pivotul
        }
        elem=a[i];a[i]=a[r];a[r]=elem;
        //la ieșirea din ciclul for (când i>=j) inversăm a[i] cu pivotul
        qs(a,l,i-1);
        //apelările recursive ale funcției de sortare conform tehnicii
```

```

    qs(a,i+1,r); //de programare "Divide et Impera"
}
}

void main(void)
{
    int n,a[100],i;
    printf("Introd dim sirului:");
    scanf("%d",&n);
    printf("Introd elem sirului:\n");
    //elementele șirului încep de la indicele 1
    for(i=1;i<=n;i++)
    {
        printf("a[%d]=",i);
        scanf("%d",a+i);
    }
    a[0]=-MAXINT;
    //apelul funcției de sortare rapidă
    qs(a,1,n);
    printf("Sirul sortat este: ");
    for(i=1;i<=n;i++) printf("%d ",a[i]);
    putchar('\n');
    getch();
}

```

## **Sortare prin interclasare (Merge Sort)**

Algoritmul de sortare prin inserție este eficient pentru valori mici ale lui  $n$  ( $n \leq 16$ ). De aceea, sortarea prin interclasare propune o sortare bazată pe principiul Divide et Impera, care să utilizeze sortarea prin inserție pentru valori mici ale lui  $n$ , rezultate prin descompunerea șirului inițial în subșiruri.

Algoritmul Merge Sort se bazează pe 3 pași esențiali:

- separarea tabloului în 2 părți de mărime cât mai apropiate
- sortarea acestor părți prin apeluri recursive, până la atingerea unor valori mici ale lui  $n$  pentru care se aplică sortarea prin inserție sau până la atingerea unor subșiruri de 1 element (cazul banal)
- interclasarea părților sortate obținându-se direct un șir ordonat crescător

```

#include<stdio.h>
#include<conio.h>

```

```

//funcția care interclasează 2 șiruri ordonate crescător
void merge(int a[], int l, int m, int r)
{

```

```

int b[100],i=l,j=m+1,k=l,ind;
//atât timp cât nici unul din șiruri nu s-a terminat
while((i<=m)&&(j<=r))
{
    if(a[i]<=a[j])
        //dacă elementul din primul subșir e mai mic sau egal îl
        //preluăm pe acesta în șirul interclasat
        b[k]=a[i++];
    else
        //dacă nu, preluăm din al doilea subșir
        b[k]=a[j++];
    k++; //k este indicele în șirul interclasat
}
if(i>m)
    //dacă primul subșir s-a terminat, preluăm restul elementelor
    //din cel de al doilea subșir
    for(ind=j;ind<=r;ind++,k++) b[k]=a[ind];
else
    //dacă al doilea subșir s-a terminat preluăm restul elementelor
    //din primul subșir
    for(ind=i;ind<=m;ind++,k++) b[k]=a[ind];
//în final mutăm elementele interclasate în șirul original
for(ind=l;ind<=r;ind++) a[ind]=b[ind];
}

//funcția recursivă de sortare prin interclasare
void merge_sort(int a[], int l, int r)
{
    int m;
    //condiția de oprire din recursivitate
    if(l<r)
    {
        //calculăm mediana pentru a permite împărțirea șirului în 2
        //subșiruri
        m=(l+r)/2;
        //aplicăm Divide et Impera pentru cele 2 subșiruri rezultate
        merge_sort(a,l,m);
        merge_sort(a,m+1,r);
        //interclasăm subșirurile sortate
        merge(a,l,m,r);
    }
}

void main(void)
{
    int n,a[100],i;

```

```

printf("Introd dim sirului:");
scanf("%d",&n);
printf("Introd elem sirului:\n");
for(i=0;i<n;i++)
{
printf("a[%d]=",i);
scanf("%d",a+i);
}
merge_sort(a,0,n-1);
printf("Sirul sortat este: ");
for(i=0;i<n;i++) printf("%d ",a[i]);
putchar('\n');
getch();
}

```

## **Recursivitate**

### **Calculul valorii n !**

```

#include<stdio.h>
#include<conio.h>

void main(void)
{
int n;
int fact(int);
clrscr();
scanf("%d",&n);
printf("%d !=%d",n,fact(n));
getch();
}

int fact(int n)
{
if (!n)
return 1;
else
return (n*fact(n-1));
}

```

### **Algoritmul lui Euclid recursiv**

```

#include<stdio.h>
#include<conio.h>

void main(void)

```

```

{
    int m,n;
    int cmmdc(int,int);
    clrscr();
    scanf("%d %d",&m,&n);
    printf("cmmdc dintre %d si %d este %d",m,n,cmmdc(m,n));
    getch();
}

int cmmdc(int m,int n)
{
    if (n==0)
        return (m);
    else
        return cmmdc(n,m%n);
}

```

## Să se genereze primele n numere din șirul lui Fibonacci

Să se genereze primele n numere din șirul lui Fibonacci. Acesta se definește recurent astfel:

$$\text{Fib}(0)=\text{Fib}(1)=1;$$

$$\text{Fib}(n)=\text{Fib}(n-1)+\text{Fib}(n-2), \text{ pentru } n \geq 2.$$

```

#include<stdio.h>
#include<conio.h>

void main(void)
{
    int i,n;
    int fib(int);
    //declarăm prototipul funcției recursive ce calculează fiecare
    //termen al șirului lui Fibonacci
    clrscr();
    scanf("%d",&n);
    for(i=0;i<n;i++) printf("fib[%d]=%d\n",i,fib(i));
    getch();
}

int fib(int n)
{
    if((n==0)||(n==1))
        return(1);
    else
        return(fib(n-1)+fib(n-2));
}

```

Comandă versiunea completă, tipărită de  
[www.fituici-bacalaureat.ro](http://www.fituici-bacalaureat.ro)

[www.fituici-bacalaureat.ro](http://www.fituici-bacalaureat.ro)

# Metoda Divide et Impera

## Suma elementelor unui șir

Să se calculeze suma elementelor unui șir folosind Divide et Impera.

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

int a[20];

int divide(int ls, int ld)
{
    int mijloc,d1,d2;
    if(ls!=ld)
    {
        mijloc=(ls+ld)/2;
        d1=divide(ls,mijloc);
        d2=divide(mijloc+1,ld);
        return(d1+d2);
    }
    else
        return(a[ls]);
}

void main(void)
{
    int l,n;
    printf("Introd nr de elem ale sirului:");
    scanf("%d",&n);
    for(l=0;l<n;l++)
    {
        printf("a[%d]=");
        scanf("%d",a+l);
    }
    printf("Suma elem este: %d\n",divide(0,n-1));
    getch();
}
```

## Problema “Turnurilor din Hanoi”

Legenda acestei probleme spune că Brahma a fixat pe Pământ trei tije de diamante și pe una din ele a pus în ordine crescătoare 64 de discuri de aur de dimensiuni diferite, cu discul cel mai mare jos. Singura operațiune permisă călugărilor era mutarea a câte unui singur disc de pe o tijă pe alta, astfel încât niciodată să nu se pună un disc mai mare peste unul mai mic.

Legenda spune că atunci când călugării vor muta toate cele 64 de discuri respectând regulile de mai sus, atunci va veni sfârșitul lumii. Presupunând că în fiecare secundă se mută un disc, lucrând fără întrerupere, cele 64 de discuri nu pot fi mutate nici în 500 de miliarde de ani de la începutul acțiunii!

Vom considera trei tije verticale notate Stânga, Mijloc, Dreapta. Pe tija Stânga se găsesc așezate  $n$  discuri de diametre diferite, în ordinea descrescătoare a diametrelor, privind de jos în sus. Inițial, tijele Mijloc și Dreapta sunt goale. Să se afișeze toate mutările prin care discurile de pe tija Stânga se mută pe tija Mijloc, în aceeași ordine, respectând următoarele reguli:

- la fiecare mișcare se mută un singur disc
- un disc mai mare nu poate fi plasat peste un disc cu diametrul mai mic
- un disc mai mare nu poate fi plasat peste un disc cu diametrul mai mic

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

int n;

void scrie(char a, char b)
{
    printf("Muta discul de pe %c pe %c\n",a,b);
}

void muta(int n, char a, char b, char c)
{
    if (n==1)
    {
        scrie(a,b);
        return;
    }
    else
    {
        muta(n-1,a,c,b);
        scrie(a,b);
        muta(n-1,c,b,a);
    }
}

void main(void)
{
    clrscr();
    printf("Introd nr de tumuri:");
    scanf("%d",&n);
```

Comandă versiunea completă, tipărită de  
[www.fituici-bacalaureat.ro](http://www.fituici-bacalaureat.ro)

[www.fituici-bacalaureat.ro](http://www.fituici-bacalaureat.ro)

```

(
printf("De la nodul:");
scanf("%d",&x);
printf("La nodul:");
scanf("%d",&y);
a[x][y]=a[y][x]=1;
)
parc_latime();
getch();
}

```

## Implementarea parcurgerii în adâncime pentru un graf neorientat

```

#include <stdio.h>
#include <conio.h>

int a[20][20],n,i,j;

void parc_adancime()
(
int s[40],varf,k,nod,urm[20],viz[20],in;
printf("Introduceti nodul initial de pornire=");
scanf("%d",&in);
for (nod=1;nod<=n;nod++)
(
urm[nod]=0;
viz[nod]=0;
)
varf=1;
//șirul s va păstra ordinea de traversare în adâncime;
//introducem nodul de start în acest șir
s[varf]=in;
//marcăm nodul de pornire ca vizitat
viz[in]=1;
//afișăm nodul de start ca prim nod în traversarea în adâncime
printf("%d ",in);
while (varf>=1)
(
//nod va fi nodul căruia îi mai căutăm succesori
nod=s[varf];
//k va conține numărul nodului de unde reîncepem să căutăm
//succesori pentru nodul nod
k=urm[nod]+1;
while ( (k<=n) &&

```

```

((a[nod][k]==0)||((a[nod][k]==1)&&(viz[k]==1))) )
{
    k++;
    urm[nod]=k;
}
if (k==n+1)
    varf--;
else
{
    varf++;
    s[varf]=k;
    viz[k]=1;
    printf("%d ",s[varf]);
}
}
putchar('\n');
}

void main()
{
    int m,i,j,x,y;
    clrscr();
    printf("Introduceti numarul de noduri n=");
    scanf("%d",&n);
    printf("Introduceti numarul de muchii m=");
    scanf("%d",&m);
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++) a[i][j]=0;
    for (i=1;i<=m;i++)
    {
        printf("De la nodul:");
        scanf("%d",&x);
        printf("La nodul:");
        scanf("%d",&y);
        a[x][y]=a[y][x]=1;
    }
    parc_adancime();
    getch();
}

```

### Drumuri într-un graf

Matricea de adiacență  $A(n,n)$  a unui graf  $G=(X,U)$  evidențiază drumurile de lungime 1 între oricare 2 noduri ale grafului.

Comandă versiunea completă, tipărită de  
[www.fituici-bacalaureat.ro](http://www.fituici-bacalaureat.ro)

[www.fituici-bacalaureat.ro](http://www.fituici-bacalaureat.ro)