

# Cuprins

<i>Cuvânt înainte</i> .....	9
<b>Capitolul 1. Recursivitate</b> .....	II
1. Conceptul de recursivitate .....	II
2. Mecanismul de realizare a recursivității .....	12
<i>Inversarea unui curvînt</i> .....	12
3. Utilitatea funcțiilor recursive .....	14
<i>Asociativitate</i> .....	16
4. Aplicații .....	17
<i>Calculul celui mai mare divizor comun a două numere naturale</i> .....	17
<i>Funcția Ackermann</i> .....	17
<i>Numărare</i> .....	18
<i>Suma puterilor rădăcinilor</i> .....	19
<i>Anagrame</i> .....	20
<i>Generare</i> .....	21
<i>Conversie</i> .....	22
<i>Codul Gray</i> .....	23
<i>Desert</i> .....	24
<i>Windows</i> .....	26
<i>Restaurarea unui apel de funcție</i> .....	28
<i>Imagine</i> .....	30
5. Recursivitate indirecță .....	34
6. Aplicații .....	34
<i>Șirul mediilor aritmetico-geometrice al lui Gauss</i> .....	34
<i>Deplasarea pe ecran a unei bare</i> .....	35
<i>Verificarea corectitudinii sintactice a unei expresii aritmetice</i> .....	36
<i>Transformarea unei expresii aritmetice în formă poloneză</i> .....	38
<i>Evaluarea unei expresii aritmetice</i> .....	40
<i>Rezistor</i> .....	42
7. Probleme propuse .....	45
<b>Capitolul 2. Metoda <i>Divide et impera</i></b> .....	52
1. Descrierea generală a metodei .....	52
2. Aplicații .....	53
<i>Cel mai mare divizor comun</i> .....	53

<i>Problema Turnurilor din Hanoi</i>	54
<i>Problema pliерilor</i>	55
<i>Fractali</i>	57
<i>Problema săcăturilor</i>	58
<i>Descompunere</i>	60
3. Compararea performanțelor unor algoritmi de căutare și sortare .....	61
<i>Algoritmi de căutare. Căutarea binară</i> .....	61
<i>Algoritmi de sortare</i> .....	63
<i>Sortarea prin interclasare (Merge Sort)</i> .....	65
<i>Sortarea rapidă (Quick Sort)</i> .....	67
Probleme propuse .....	69
<b>Capitolul 3. Metoda Backtracking</b> .....	74
1. Descrierea generală a metodei .....	74
<i>Problema reginelor</i> .....	76
2. Aplicații .....	82
<i>Plata unei sume cu monede de valori date</i> .....	82
<i>Generarea sirului</i> .....	84
<i>Generare de numere</i> .....	86
<i>Comis-viajor</i> .....	88
<i>Medii</i> .....	90
<i>Domino</i> .....	93
<i>Scara</i> .....	95
<i>Stafie</i> .....	99
3. Backtracking în plan .....	102
<i>Labirint</i> .....	103
<i>Fotografie</i> .....	105
<i>Cel mai lung prefix</i> .....	107
<i>Albină</i> .....	108
<i>Collapse</i> .....	112
4. Considerații finale asupra metodei Backtracking .....	118
5. Probleme propuse .....	118
<b>Capitolul 4. Elemente de combinatorică</b> .....	126
Generarea produsului cartezian .....	126
Generarea permutărilor .....	128
Numărul de ordine al unei permutări .....	129
Generarea aranjamentelor .....	131
Generarea combinațiilor .....	132
Numărul de ordine al unei combinații .....	134
Partițiile unei mulțimi .....	136
Numărul lui Stirling de speță a II-a .....	137
Numărul lui Bell .....	138
Generarea funcțiilor surjective .....	138
Partițiile unui număr natural .....	140
Generare de paranteze .....	142

Numărarea sirurilor .....	143
Numărul lui Catalan .....	144
Permutări cu repetiție .....	145
Numărul lui Stirling de speță I .....	146
Aplicații .....	147
<i>Permutări fără puncte fixe</i> .....	147
<i>Soldați</i> .....	148
<i>Numere</i> .....	149
<i>Aniversare</i> .....	151
<i>Ture</i> .....	152
<i>Permutări cu și inversiuni</i> .....	154
<i>Potrivire</i> .....	155
<i>Vizibil</i> .....	157
<i>Bonus</i> .....	161
Probleme propuse .....	164
<b>Capitolul 5. Metoda programării dinamice</b> .....	173
1. Prezentare generală .....	173
2. Aplicații .....	174
<i>Pachete</i> .....	174
<i>Tren</i> .....	177
<i>Evaluare optimă</i> .....	180
<i>Pietre</i> .....	183
<i>Rucsac</i> .....	185
<i>Transformare de cuvinte</i> .....	188
<i>Palindrom</i> .....	192
<i>Suma</i> .....	194
<i>Lăcustă</i> .....	198
<i>Paragrafare optimă</i> .....	201
<i>Cod</i> .....	205
<i>Scara</i> .....	207
<i>Politie</i> .....	211
Probleme propuse .....	217
<b>Capitolul 6. Soluții și indicații</b> .....	233
1. Recursivitate .....	233
2. Metoda <i>Divide et impere</i> .....	240
3. Metoda <i>Backtracking</i> .....	244
4. Elemente de combinatorică .....	264
5. Metoda programării dinamice .....	271
<b>Bibliografie</b> .....	287

## Date de ieșire

Fișierul `desert.out` conține o singură linie pe care este scris un număr real reprezentând distanța maximă ce poate fi parcursă de Indiana Jones, exprimată în kilometri.

## Restricții

- $1 \leq n \leq 100$
- $2 \leq k \leq 50$
- $5 \leq p \leq 20$
- Distanța maximă parcursă va fi afișată cu trei zecimale, cu rotunjire.

## Exemple

<code>desert.in</code>	<code>desert.out</code>	<code>desert.in</code>	<code>desert.out</code>
2 3 10	60.000	4 3 10	76.000

Olimpiada Municipală de Informatică, Iași, 2003

## Soluție

Pentru a calcula distanța maximă care poate fi parcursă având la dispoziție  $n$  recipiente vom utiliza o funcție recursivă `dist()`.

Dacă la oază există un recipient, acesta va fi turnat în rezervor, iar distanța maximă ce poate fi parcursă cu el este  $k/p$ . Dacă la oază există două recipiente, unul va fi pus în rezervor, celălalt va fi pus în mașină și astfel se parcurge distanța  $2 * k/p$ .

Dacă numărul de recipiente este mai mare de două, turnăm conținutul unui recipient în rezervor, al doilea recipient îl punem în mașină și parcurgem o distanță  $x$ , lăsăm recipientul plin aici, după care ne întoarcem la oază. La oază punem în mașină alt recipient, parcurgem aceeași distanță  $x$ , lăsăm recipientul plin și revenim la oază. După  $2n-3$  astfel de drumuri, la distanța  $x$  de oază se află  $n-1$  recipiente pline. Vom calcula distanța  $x$  astfel încât după  $2n-3$  drumuri de lungime  $x$  rezervorul mașinii să fie gol ( $x = k / ((2 * n - 3) * p)$ ). Suntem într-o situație similară cu cea inițială: rezervorul mașinii este gol, avem  $n-1$  recipiente cu benzină, dar am parcurs deja o distanță  $x$ .

```
#include<stdio.h>
#define InFile "desert.in"
#define OutFile "desert.out"
int n, k;
long double p;

long double dist(int n)
{ if (n<=2)  return k*n/p;
  return k/((2*n-3)*p)+dist(n-1); }

int main()
{ FILE *fin, *fout;
  long double d;
```

```

fin=fopen(InFile,"rt");
fscanf(fin, "%d %d %Lf", &n, &k, &p);
fclose(fin);
p=p/100;
d=dist(n);
fout=fopen(OutFile, "wt");
fprintf(fout, "%.3Lf\n", d);
fclose(fout);
return 0; }

```

## **Windows**

Vasile folosește un sistem de operare care deschide pe ecran numeroase ferestre. Ecranul este împărțit în pătrate elementare (care au aria  $1 \times 1$ ), formând un caroaj în care liniile sunt numerotate de la 1 de sus în jos, iar coloanele sunt numerotate de la 1 de la stânga la dreapta. Astfel, fiecare pătrat elementar de pe ecran poate fi identificat specificând numărul liniei și numărul coloanei pe care se află. Fiecare fereastră este un dreptunghi format din unul sau mai multe pătrate elementare. O fereastră nou deschisă poate să se suprapună (parțial sau total) peste alte ferestre, deschise în prealabil. Putem închide o fereastră dacă executăm un click în pătratul elementar ce constituie colțul din dreapta-sus al ferestrei (dacă acesta este vizibil).

Scrieți un program care să determine numărul minim de click-uri necesare pentru a închide prima fereastra pe care am deschis-o.

### *Date de intrare*

Fișierul de intrare `win.in` conține pe prima linie un număr natural  $N$ , reprezentând numărul de ferestre deschise pe ecran.

Fiecare dintre următoarele  $N$  linii conține câte patru numere naturale separate prin căte un spațiu,  $R_1$ ,  $S_1$ ,  $R_2$  și  $S_2$ , cu semnificația „am deschis o fereastră care are colțul din stânga-sus pe linia  $R_1$  și coloana  $S_1$ , respectiv colțul din dreapta-jos pe linia  $R_2$  și coloana  $S_2$ ”. Ferestrele se deschid în ordinea în care apar în fișierul de intrare.

### *Date de ieșire*

Fișierul de ieșire `win.out` va conține o singură linie pe care va fi scris numărul minim de click-uri necesare pentru a închide prima fereastră deschisă.

### *Restricții*

- $0 < N \leq 100$
- $1 \leq R_1 \leq R_2 \leq 10000$
- $1 \leq S_1 \leq S_2 \leq 10000$

*Exemple*

win.in	win.out	win.in	win.out	win.in	win.out
3	3	3	3	3	1
3 1 6 4		4 1 6 3		3 3 4 4	
1 2 4 6		2 2 5 5		1 1 2 2	
2 3 5 5		1 4 3 6		5 5 6 6	

Olimpiada Municipală de Informatică, Iași, 2005

*Soluție*

Vom reprezenta o fereastră ca o structură cu patru câmpuri: coordonatele colțului din stânga-sus și coordonatele colțului din dreapta-jos al ferestrei.

Vom aborda problema recursiv: pentru a inchide fereastra cu numărul *nr* trebuie să inchidem, în prealabil, toate ferestrele deschise ulterior, care acoperă colțul din dreapta-sus al ferestrei *nr*. Acest lucru este realizat de funcția recursivă *inchide()*, ce are ca parametru numărul ferestrei care trebuie să fie inchisă. Pentru a nu inchide de mai multe ori aceeași fereastră, vom reține mulțimea ferestrelor inchise în vectorul caracteristic *inchisa*.

```
#include <stdio.h>
#define INPUT_FILE "win.in"
#define OUTPUT_FILE "win.out"
#define NMAX 100

int N; //N reprezinta numarul de ferestre
struct { int ls, ld, cs, cd;} f[NMAX];
//f - vector cu N componente in care retinem ferestrele
int inchisa[NMAX];
/* inchisa[i]=0 daca fereastra i este deschisa,
   respectiv 1 daca fereastra i este inchisa */
int rez;
/* rez - variabila globala in care calculam numarul total
   de click-uri necesare */

void citire(void)
{ int i;
 FILE * fin = fopen(INPUT_FILE, "r");
 fscanf(fin, "%d", &N);
 for (i=0; i<N; ++i)
     fscanf(fin, "%d %d %d %d", &f[i].ls, &f[i].cs,
            &f[i].ld, &f[i].cd);
 fclose(file);
}
```

```

int peste(int a, int b)
/* functia returneaza valoarea 1 daca fereastra a este peste
coltul din dreapta-sus al ferestrei b si 0 in caz contrar */
{ if (f[a].ls <= f[b].ls && f[a].ld >= f[b].ls &&
     f[a].cs <= f[b].cd && f[a].cd >= f[b].cd)
    return 1;
  return 0;
}

void inchide(int nr)
/* functia inchide fereastra cu numarul nr */
for (int i=N-1; i>=nr+1; i--)
  if (!inchisa[i] && peste(i, nr))
    inchide(i);
inchisa[nr]=1;
++rez;
}

void rezolvare(void)
{ for (int i=0; i<N; i++) inchisa[i] = 0;
  rez = 0;
  inchide(0);
}
void afisare(void)
{ FILE * fout = fopen(OUTPUT_FILE,"w");
  fprintf(fout,"%d\n", rez);
  fclose(fout);
}

int main(void)
{ citire();
  rezolvare();
  afisare();  return 0;
}

```

### *Restaurarea unui apel de funcție*

Se citește din fișierul `f.in` un sir de maximum 100 de caractere care reprezintă o succesiune de nume de funcții și variabile, formate dintr-un singur caracter. Pentru fiecare funcție ce intervine în sir se citesc numele și aritatea (numărul de argumente) de pe o linie din fișier. Restaurați sirul de caractere, punând la locul lor parantezele și virgulele, astfel încât să reprezinte un apel de funcție corect.

De exemplu, pentru fișierul de intrare :

```
fxgxyzhx
f 3
g 3
h 1
```

funcția restaurată va fi :

```
f(x, g(x, y, z), h(x))
```

### Soluție

Vom memora datele de intrare în doi vectori : **NF**, în care reținem numele funcțiilor, în ordinea din fișierul de intrare, și **AF**, în care reținem aritatea fiecărei funcții (numărul de argumente). Pentru a restaura apelul de funcție din sirul de caractere, citit de pe prima linie a fișierului de intrare, vom utiliza o funcție recursivă denumită **Restaurare()**, care are un singur parametru (**i**), ce indică poziția la care am ajuns cu restaurarea în sirul de intrare.

În funcția **Restaurare()** verificăm dacă pe poziția **i** în sirul de intrare este un nume de funcție. Dacă da, se scrie numele funcției, urmat de „(”, după care se restaurează (cu ajutorul funcției recursive) fiecare argument al funcției, separând argumentele prin virgulă. Numărul de argumente se obține din vectorul **AF**. După ultimul argument restaurat se închide paranteza (se scrie pe ecran „)”).

Dacă pe poziția **i** în sirul de intrare nu este o funcție, se scrie pe ecran caracterul respectiv (este o variabilă).

### Observație

Soluția se bazează în mod esențial pe faptul că datele de intrare sunt corecte (deci apelul de funcție poate fi restaurat).

```
#include <fstream.h>
#define NrMaxF 100
char NF[NrMaxF]; //NF contine numele fiecarei functii
char s[100]; //sirul care trebuie restaurat
int AF[NrMaxF]; //AF contine aritatea functiilor
int NrF; //NrF - numarul de functii
void Citire();
int AR(char);
void Restaurare(int s);

int main()
{ int i=0;
  Citire();
  Restaurare(i);
  return 0;
}
```

```

void Citire()
{
    ifstream f("f.in");
    char c;
    int a;
    f.getline(s, 100);
    while (!f.eof())
        { f>>c>>a;
          NF[NrF]=c; AF[NrF++]=a; }
    f.close();
}

int AR(char c)
//functia intoarce 0 daca c nu este un nume de functie,
//respectiv aritatea functiei, in caz contrar
for (int j=0; j<NrF; j++)
    if (NF[j] == c) return AF[j];
return 0;

void Restaurare(int & i)
{
    int k, j;
    cout<<s[i];
    k=AR(s[i++]);
    if (k)           //s[i] reprezinta un nume de functie
        {cout<<'(';
         for (j=1; j<k; j++)
             {Restaurare(i); //restauram argumentele functiei
              cout<<','; } //separandu-le prin virgule
         Restaurare(i);
         cout<<')';}
    }
}

```

### *Exercițiu*

Modificați programul astfel încât să testeze dacă restaurarea este posibilă.

### *Imagine*

Să considerăm o imagine alb-negru de dimensiune  $L \times L$  pixeli. Un pixel poate fi alb (codificat cu valoarea 0) sau negru (codificat cu valoarea 1). Imaginele pot fi compresate în diverse moduri. Una dintre cele mai cunoscute scheme de compresie este următoarea:

1. Dacă imaginea este formată atât din pixeli 1, cât și din pixeli 0, se reține valoarea 1, care indică faptul că imaginea va fi partită în alte patru subimagini, așa cum este descris la pasul 2. Altfel codificăm întreaga imagine ca 00 sau 01, semnificând faptul că întreaga imagine este formată numai din pixeli 0, respectiv numai din pixeli 1.