

Emanuela Cerchez, Marinel Șerban

PROGRAMAREA ÎN LIMBAJUL

C/C++

PENTRU LICEU

••

Metode și tehnici de programare

Ediția a II-a revăzută și adăugită

POLIROM
2022

Cuprins

<i>Cuvânt înainte</i>	9
1. Recursivitate	11
1.1. Conceptul de recursivitate	11
1.2. Mecanismul de realizare a recursivității	12
<i>Inversarea unui cuvânt</i>	12
1.3. Utilitatea funcțiilor recursive	14
<i>Șirul Fibonacci</i>	15
1.4. Aplicații	16
<i>Calculul celui mai mare divizor comun al două numere naturale</i>	16
<i>Algoritmul lui Euclid extins</i>	16
<i>Invers modular</i>	17
<i>Funcția Ackermann</i>	18
<i>Conversie</i>	18
<i>Numărare</i>	19
<i>Suma puterilor rădăcinilor</i>	19
<i>Anagrame</i>	21
<i>Codul Gray</i>	21
<i>Deșert</i>	23
<i>Windows</i>	24
<i>Exponențiere</i>	27
<i>Generare program</i>	28
<i>Compress</i>	29
<i>Restaurarea unui apel de funcție</i>	31
1.5. Recursivitate indirectă	33
1.6. Aplicații	34
<i>Șirul mediilor aritmetico-geometrice al lui Gauss</i>	34
<i>Verificarea corectitudinii sintactice a unei expresii aritmetice</i>	35
<i>Transformarea unei expresii aritmetice în formă poloneză</i>	37
<i>Evaluarea unei expresii aritmetice</i>	38
<i>Reziston</i>	40
1.7. Probleme propuse	43

2. Metoda <i>Divide et Impera</i>	51
2.1. Descrierea generală a metodei	51
2.2. Aplicații	52
<i>Cel mai mare divizor comun</i>	52
<i>Problema Turnurilor din Hanoi</i>	53
<i>Problema plierilor</i>	54
<i>Imagine</i>	56
<i>Compresie</i>	59
<i>Fractali</i>	63
<i>Problema tăieturilor</i>	65
<i>Descompunere</i>	67
2.3. Compararea performanțelor unor algoritmi de sortare	68
<i>Sortarea prin interclasare (MergeSort)</i>	71
<i>Sortarea rapidă (QuickSort)</i>	73
<i>Selecția rapidă (QuickSelect)</i>	75
2.4. Probleme propuse	77
3. Metoda <i>backtracking</i>	82
3.1. Descrierea generală a metodei	82
3.2. Exemplificarea modului de funcționare a metodei <i>backtracking</i>. <i>Problema regiilor</i>	84
3.3. Aplicații	90
<i>Plata unei sume cu monede de valori date</i>	90
<i>Generare șir</i>	92
<i>Generare numere</i>	94
<i>Comis-voiajorul</i>	96
<i>Medii</i>	98
<i>Domino</i>	100
<i>Immortal</i>	102
<i>Virgule</i>	106
<i>Scara</i>	108
<i>Stație</i>	112
3.4. Backtracking în plan	115
3.5. Aplicații	116
<i>Labirint</i>	116
<i>Fotografie</i>	118
<i>Cel mai lung prefix</i>	120
<i>Albina</i>	121
<i>Collapse</i>	125
3.6. Considerații finale asupra metodei <i>backtracking</i>	130
3.7. Probleme propuse	131

4. Metoda programării dinamice	140
4.1. Prezentare generală	140
4.2. Aplicații	141
<i>Pachete</i>	141
<i>Tren</i>	144
<i>Evaluare optimală</i>	147
<i>Pietre – cel mai lung subșir crescător</i>	150
<i>Problema rucsacului – varianta discretă</i>	155
<i>Transformare de cuvinte</i>	157
<i>Palindrom</i>	161
<i>Suma</i>	163
<i>Falling</i>	167
<i>Lăcusta</i>	171
<i>Paragrafare optimală</i>	174
<i>Cod</i>	177
<i>Scara</i>	179
<i>Recycle Bin</i>	183
<i>Poliție</i>	185
4.3. Probleme propuse	191
5. Elemente de combinatorică	209
5.1. Generări recursive de elemente combinatoriale	209
<i>Generarea codurilor Morse</i>	209
<i>Generare parantezări</i>	210
<i>Generarea produsului cartezian</i>	211
<i>Generarea permutărilor</i>	213
<i>Generarea aranjamentelor</i>	214
<i>Generarea combinărilor</i>	215
<i>Partițiile unui număr natural</i>	217
<i>Partițiile unei mulțimi</i>	219
<i>Generarea funcțiilor surjective</i>	220
<i>Permutări cu repetiție</i>	222
5.2. Numere combinatoriale	223
<i>Numărul de combinări – triunghiul lui Pascal</i>	223
<i>Numărul de combinări – numere mari</i>	225
<i>Numărul de combinări – modulo MOD</i>	226
<i>Numărul lui Stirling de speța a II-a</i>	227
<i>Numărul lui Bell</i>	228
<i>Numărare șiruri</i>	228
<i>Numărul lui Catalan</i>	229
<i>Numărul lui Stirling de speța I</i>	230
<i>Stars & Bars</i>	231
5.3. Numărul de ordine al unei configurații	232
<i>Numărul de ordine al unei permutări</i>	232
<i>Numărul de ordine al unei combinări</i>	233

5.4. Aplicații	235
<i>Permutări fără puncte fixe</i>	235
<i>Inversiune</i>	236
<i>Soldați</i>	236
<i>Numere</i>	238
<i>Aniversare</i>	239
<i>Ture</i>	241
<i>PM</i>	241
<i>Permutări cu k inversiuni</i>	242
<i>Potrivire</i>	243
<i>Vizibil</i>	245
<i>Bonus</i>	250
<i>Expoziție</i>	253
5.5. Probleme propuse	254
Soluții și indicații	265
Bibliografie	311

3.4. Backtracking în plan

În variantă elementară aplicam metoda *backtracking* pentru rezolvarea problemelor în care soluția era reprezentată ca vector. Putem generaliza ideea căutării cu revenire și pentru probleme în care căutarea se face „în plan”. Pentru noi planul va fi reprezentat ca un tablou bidimensional.

Pentru a intui modul de funcționare a metodei *backtracking* în plan să ne imaginăm explorarea unei peșteri. Speologul pornește de la intrarea în peșteră și trebuie să exploreze în mod sistematic toate culoarele peșterii. Ce înseamnă „în mod sistematic”? În primul rând, își stabilește o ordine pentru toate direcțiile posibile de mișcare (de exemplu, N, NE, E, SE, S, SV, V, NV) și, întotdeauna când se găsește într-un punct din care are mai multe culoare de explorat, alege direcțiile de deplasare în ordinea prestabilită. În al doilea rând, speologul va plasa marcaje pe culoarele pe care le-a explorat, pentru ca nu cumva să se rătăcească și să parcurgă de mai multe ori același culoar (ceea ce ar conduce la determinarea eronată a lungimii peșterii).

În ce constă explorarea? Speologul explorează un culoar până când întâlnește o intersecție sau până când culoarul se înfundă. Dacă a ajuns la o intersecție, explorează succesiv toate culoarele care pornesc din intersecția respectivă, în ordinea prestabilită a direcțiilor. Când un culoar se înfundă, revine la intersecția precedentă și alege un alt culoar, de pe următoarea direcție (dacă există; dacă nu există, revine la intersecția precedentă ș.a.m.d.).

Să descriem într-o formă mai generală această metodă.

Vom nota prin `NrDirectii` o constantă care reprezintă numărul de direcții de deplasare, iar `dx`, respectiv `dy` sunt doi vectori constanți care reprezintă deplasările relative pe direcția Ox , respectiv pe direcția Oy , urmând în ordine cele `NrDirectii` de deplasare.

```
void Bkt_Plan(int x, int y)
    //x, y reprezinta coordonatele pozitiei curente
{
    Explorare(x,y); //exploram pozitia curenta
    if (EFinal(x,y)) //pozitia x,y este un punct final
        Prelucrare_Solutie();
    else //continuum cautarea
        for (i=0; i<NrDirectii; i++)
            //ma deplasez succesiv pe directiile posibile de miscare
            if (Nvizitat(x+dx[i], y+dy[i]))
                //nu am mai trecut prin aceasta pozitie
                Bkt_Plan(x+dx[i], y+dy[i]);
}
```

3.5. Aplicații

Labirint

Într-un labirint, reprezentat ca o matrice L , cu n linii și m coloane, cu componente 0 sau 1 (1 semnificând perete, 0 culoar), se găsesc o bucată de brânză, pe poziția (x_b, y_b) și un șoricel pe poziția (x_s, y_s) . Afișați toate posibilitățile șoricelului de a ajunge la brânză, știind că el se poate deplasa numai pe culoar, iar direcțiile posibile de mișcare sunt N, NE, E, SE, S, SV, V, NV.

Fișierul de intrare `labirint.in` conține pe prima linie n, m, x_s, y_s, x_b, y_b cu semnificația din enunț, iar pe următoarele n linii câte m valori binare reprezentând labirintul; valorile scrise pe aceeași linie sunt separate prin câte un spațiu. Soluțiile vor fi scrise în fișierul `labirint.out`, în formatul din exemplul următor.

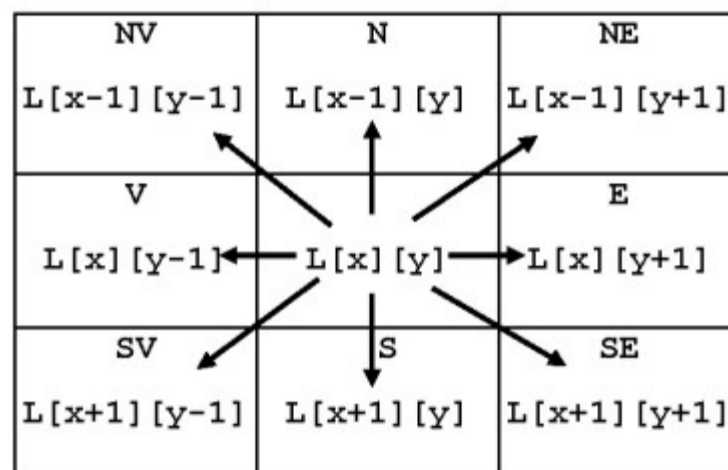
Exemplu

labirint.in	labirint.out
4 4 1 1 4 4	Solutia nr. 1
0 1 1 1	*111
0 1 1 1	*111
0 1 0 0	*1**
1 0 1 0	1*1*
	Solutia nr. 2
	*111
	*111
	*1*0
	1*1*

Reprezentarea informațiilor

Labirintul este reprezentat ca o matrice L , cu $n \times m$ elemente. Elementele labirintului sunt inițial 0 (semnificând culoar) și 1 (semnificând perete). Pentru ca șoricelul să nu treacă de mai multe ori prin aceeași poziție, existând riscul de a intra în buclă, vom marca în labirint pozițiile prin care trece șoricelul cu 2.

Pentru a determina pozițiile în care se poate deplasa șoricelul, vom utiliza doi vectori $D_x[8]$ și $D_y[8]$, pe care îi inițializăm cu deplasările pe linie, respectiv pe coloană pentru toate cele 8 direcții posibile de mișcare.



Pentru a nu verifica permanent dacă șoricelul a ajuns la marginea labirintului, bordăm labirintul cu perete (două linii și două coloane cu valoarea 1).

Condiții interne

Din poziția (x, y) șoricelul se poate deplasa pe direcția dir , deci în poziția $(x+Dx[dir], y+Dy[dir])$ dacă și numai dacă

$$L[x+Dx[dir]][y+Dy[dir]]=0$$

(această poziție reprezintă un culoar prin care șoricelul nu a mai trecut).

```
#include <fstream>
#define DIMMAX 20
using namespace std;
ifstream f("labirint.in");
ofstream fout("labirint.out");
int Dx[8]={-1,-1,0,1,1, 1, 0,-1};
int Dy[8]={ 0, 1,1,1,0,-1,-1,-1};
int L[DIMMAX][DIMMAX];
int n, m, xs, ys, xb, yb, NrSol;
void Citire();
void Cauta(int x, int y);

int main()
{ Citire();
  Cauta(xs, ys);
  if (!NrSol) fout<<"Nu exista solutii!\n";
  fout.close();
  return 0;
}

void Citire()
{int i, j;
  f>>n>>m>>xs>>ys>>xb>>yb;
  for (i=1; i<=n; i++)
    for (j=1; j<=m; j++) f>>L[i][j];
  f.close();
  for (i=0; i<=n+1; i++) L[i][0]=L[i][m+1]=1;
  for (i=0; i<=m+1; i++) L[0][i]=L[n+1][i]=1;
}

void Afisare()
{int i, j;
  fout<<"Solutia nr. "<<NrSol<<"\n";
  for (i=1; i<=n; i++)
    {for (j=1; j<=m; j++)
      if (L[i][j] == 2) fout<<'*';
      else fout<<L[i][j];
    }
  fout<<"\n"; }
}
```

```

void Cauta(int x, int y)
{ int dir;
  L[x][y]=2; //marchez pozitia x y
  if (x == xb && y == yb) Afisare();
  else
    for (dir=0; dir<8; dir++)
      if (!L[x+Dx[dir]][y+Dy[dir]]) //culoar nevizitat
        Cauta(x+Dx[dir], y+Dy[dir]);
  L[x][y]=0;
  /*la intoarcere sterg marcajul, pentru a putea explora
  acest culoar si in alta varianta*/
}

```

Exercițiu

Executați programul pas cu pas și urmăriți valorile variabilelor globale și locale.

Fotografie

Fotografia alb-negru a unui obiect este reprezentată sub forma unei matrice cu n linii și m coloane ale cărei elemente sunt 0 sau 1. Elementele egale cu 1 reprezintă punctele ce aparțin unui obiect. Două elemente de valoare 1 fac parte din același obiect dacă sunt adiacente pe linie sau pe coloană. Se cere să se determine numărul obiectelor din fotografie.

De exemplu, pentru matricea:

```

3 6
1 0 0 0 0 0
0 1 0 0 1 1
0 1 0 0 1 0

```

programul va afișa:

```
Nr. obiecte = 3
```

Soluție

Pentru a număra obiectele din fotografie, vom parcurge matricea care reprezintă fotografia, căutând un element cu valoarea 1, deci care aparține unui obiect. Vom număra noul obiect depistat, apoi vom „șterge” obiectul din fotografie, colorându-l în culoarea de fond (valoarea 0 în matrice). Pentru a „șterge” un obiect vom folosi funcția recursivă `Sterge_Obiect(x, y)`. Această funcție va verifica dacă punctul de coordonate (x, y) aparține obiectului ($a[x][y]=1$) și în caz afirmativ îl șterge ($a[x][y]=0$), apoi se autoapelează pentru toate punctele adiacente cu (x, y) (pe linie sau pe coloană).

Pentru a nu testa dacă în timpul căutării am depășit marginile fotografiei, am bordat matricea care reprezintă fotografia cu câte o linie și o coloană sus, jos, stânga, dreapta inițializate cu valoarea 0 (am „înramat” fotografia).

Observație

Acest tip de algoritm, prin care plecând de la un element sunt „atinse” succesiv toate elementele care au o legătură (directă sau indirectă) cu elementul respectiv poartă numele de algoritm de *fill* (umplere). În volumul I al acestei cărți am prezentat o variantă iterativă de implementare a acestui algoritm folosind fie o stivă, fie o coadă. Practic implementarea recursivă gestionează stiva în mod automat într-un mod similar cu varianta iterativă.

```
#include <fstream>
#include <iostream>
#define DMAX 102
using namespace std;

ifstream fin("foto.in");
int Dx[4]={-1, 0, 1, 0};
int Dy[4]={ 0, 1, 0,-1};
int a[DMAX][DMAX], m, n, NrObiecte;
void Citire();
void Sterge_Obiect(int x , int y);

int main()
{ int i, j;
  Citire();
  for (i=1; i<=n; i++)
    for (j=1; j<=m; j++)
      if (a[i][j]) //am depistat un obiect
        {NrObiecte++;
         Sterge_Obiect(i, j);}
  cout<<"Nr. obiecte = " << NrObiecte << '\n';
  return 0;
}

void Citire()
{int i, j;
  fin>>n>>m;
  for (i=1; i<=n; i++)
    for (j=1; j<=m; j++) fin>>a[i][j];
  fin.close();
}

void Sterge_Obiect(int x , int y)
{if (a[x][y])
  { a[x][y]=0; //sterg acest element de imagine
  //cautarea continua in cele 4 directii posibile
  for (int dir=0; dir<4; dir++)
    Sterge_Obiect(x+Dx[dir], y+Dy[dir]); }
}
```

Cel mai lung prefix

Fie C , o matrice cu n linii și m coloane, care conține numai litere. Să se determine cel mai lung prefix al unui cuvânt dat s , care să se găsească în matricea C , astfel încât oricare două litere succesive ale prefixului să se afle în matrice pe poziții adiacente (pe linie, coloană sau diagonală). Nu se va face distincție între literele mari și cele mici. De exemplu, pentru cuvântul $s = \text{'elefantel'}$ și matricea:

```
elean
ghanr
ttnde
elfen
aeatt
```

prefixul maximal obținut va fi: *elefa*.

Soluție

Vom utiliza funcția recursivă $Cauta(x, y)$, care verifică dacă următoarea literă din cuvânt (indicată de variabila globală Lg , care reține lungimea prefixului curent) se află sau nu pe poziția (x, y) în matricea C . În caz afirmativ, am mai găsit o literă în matrice, deci lungimea prefixului curent crește și continuăm căutarea în pozițiile adiacente poziției (x, y) . În caz contrar, am obținut un prefix maximal și comparăm lungimea lui cu lungimea celui mai lung prefix determinat până la momentul curent.

Pentru a nu verifica permanent dacă prin deplasări succesive nu am depășit lungimea cuvântului sau limitele indicilor din matrice, vom borda matricea cu spații și vom marca sfârșitul cuvântului prin caracterul ' . '.

De data aceasta nu trebuie să marcăm literele din matrice pe care le-am utilizat pentru construirea prefixului din două motive:

- putem utiliza aceeași literă de mai multe ori;
- nu există riscul de a apela recursiv funcția $Cauta$ la infinit (sau mai exact până se umple stiva) deoarece la fiecare apel mărim Lg , lungimea prefixului curent, și după $strlen(s)$ pași vom ajunge la ' . ', marcajul de sfârșit de cuvânt.

```
#include <fstream>
#include <iostream>
#include <cstring>
#define DMAX 52
using namespace std;
ifstream fin("prefix.in");
int Dx[8]={-1,-1,0,1,1,1,0,-1};
int Dy[8]={0,1, 1,1,0,-1,-1,-1};
char s[200];
int Lg, LgMax, n, m;
char C[DMAX][DMAX];
void Citire();
void Bordare();
void Cauta(int i, int j);
```