Emanuela Cerchez, Marinel Şerban

PROGRAMAREA ÎN LIMBAJUL C/C++ PENTRU LICEU

• •

Metode și tehnici de programare

Ediția a II-a revăzută și adăugită

POLIROM
2022

Cuprins

Cuvânt înainte 9

1. Recursivitate 11

1.1. *Conceptul de recursivitate* 11

1.2. *Mecanismul de realizare a recursivității* 12
Inversarea unui cuvânt 12

1.3. *Utilitatea funcțiilor recursive* 14
Șirul Fibonacci 15

1.4. *Aplicații* 16
Calculul celui mai mare divizor comun al două numere naturale 16
Algoritmul lui Euclid extins 16
Invers modular 17
Funcția Ackermann 18
Conversie 18
Numărare 19
Suma puterilor rădăcinilor 19
Anagrame 21
Codul Gray 21
Deșert 23
Windows 24
Exponențiere 27
Generare program 28
Compress 29
Restaurarea unui apel de funcție 31

1.5. *Recursivitate indirectă* 33

1.6. *Aplicații* 34
Șirul mediilor aritmetico-geometrice al lui Gauss 34
Verificarea corectitudinii sintactice a unei expresii aritmetice 35
Transformarea unei expresii aritmetice în formă poloneză 37
Evaluarea unei expresii aritmetice 38
Reziston 40

1.7. *Probleme propuse* 43

2. Metoda *Divide et Impera* 51

2.1. Descrierea generală a metodei 51

2.2. Aplicaţii 52
Cel mai mare divizor comun 52
Problema Turnurilor din Hanoi 53
Problema plierilor 54
Imagine 56
Compresie 59
Fractali 63
Problema tăieturilor 65
Descompunere 67

2.3. Compararea performanţelor unor algoritmi de sortare 68
Sortarea prin interclasare (MergeSort) 71
Sortarea rapidă (QuickSort) 73
Selecţia rapidă (QuickSelect) 75

2.4. Probleme propuse 77

3. Metoda *backtracking* 82

3.1. Descrierea generală a metodei 82

3.2. Exemplificarea modului de funcţionare a metodei* backtracking. *Problema reginelor 84

3.3. Aplicaţii 90
Plata unei sume cu monede de valori date 90
Generare şir 92
Generare numere 94
Comis-voiajorul 96
Medii 98
Domino 100
Immortal 102
Virgule 106
Scara 108
Staţie 112

3.4. Backtracking *în plan* 115

3.5. Aplicaţii 116
Labirint 116
Fotografie 118
Cel mai lung prefix 120
Albina 121
Collapse 125

***3.6. Consideraţii finale asupra metodei* backtracking** 130

3.7. Probleme propuse 131

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

5.4. ***Aplicații*** 235
Permutări fără puncte fixe 235
Inversiune 236
Soldați 236
Numere 238
Aniversare 239
Ture 241
PM 241
Permutări cu k inversiuni 242
Potrivire 243
Vizibil 245
Bonus 250
Expoziție 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 soluţia era reprezentată ca vector. Putem generaliza ideea căutării cu revenire şi pentru probleme în care căutarea se face „în plan". Pentru noi planul va fi reprezentat ca un tablou bidimensional.

Pentru a intui modul de funcţionare a metodei *backtracking* în plan să ne imaginăm explorarea unei peşteri. Speologul porneşte de la intrarea în peşteră şi trebuie să exploreze în mod sistematic toate culoarele peşterii. Ce înseamnă „în mod sistematic"? În primul rând, îşi stabileşte o ordine pentru toate direcţiile posibile de mişcare (de exemplu, N, NE, E, SE, S, SV, V, NV) şi, întotdeauna când se găseşte într-un punct din care are mai multe culoare de explorat, alege direcţiile 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 şi să parcurgă de mai multe ori acelaşi culoar (ceea ce ar conduce la determinarea eronată a lungimii peşterii).

În ce constă explorarea? Speologul explorează un culoar până când întâlneşte o intersecţie sau până când culoarul se înfundă. Dacă a ajuns la o intersecţie, explorează succesiv toate culoarele care pornesc din intersecţia respectivă, în ordinea prestabilită a direcţiilor. Când un culoar se înfundă, revine la intersecţia precedentă şi alege un alt culoar, de pe următoarea direcţie (dacă există; dacă nu există, revine la intersecţia precedentă ş.a.m.d.).

Să descriem într-o formă mai generală această metodă.

Vom nota prin `NrDirectii` o constantă care reprezintă numărul de direcţii de deplasare, iar `dx`, respectiv `dy` sunt doi vectori constanţi care reprezintă deplasările relative pe direcţia `Ox`, respectiv pe direcţia `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                                       //continuam cautarea
   for (i=0; i<NrDirectii; i++)
    //ma deplasez succesiv pe directiile posibile de miscare
    if (Nevizitat(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 (`xb`, `yb`) și un șoricel pe poziția (`xs`, `ys`). 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`, `xs`, `ys`, `xb`, `yb` 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.

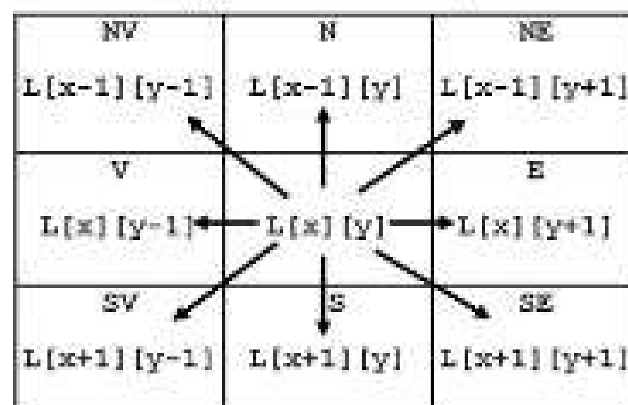
Exemplu

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

Reprezentarea informațiilor

Labirintul este reprezentat ca o matrice `L`, cu `nxm` 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 `Dx[8]` și `Dy[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 şoricelul a ajuns la marginea labirintului, bordăm labirintul cu perete (două linii şi două coloane cu valoarea `1`).

Condiţii interne

Din poziţia `(x,y)` şoricelul se poate deplasa pe direcţia `dir`, deci în poziţia `(x+Dx[dir], y+Dy[dir])` dacă şi numai dacă

`L[x+Dx[dir]][y+Dx[dir]]=0`

(această poziţie reprezintă un culoar prin care şoricelul 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 „înrămat" 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 şi m coloane, care conţine 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 poziţii adiacente (pe linie, coloană sau diagonală). Nu se va face distincţie între literele mari şi cele mici. De exemplu, pentru cuvântul s='elefantel' şi matricea:

```
elean
ghanr
ttnde
elfen
aeatt
```

prefixul maximal obţinut va fi: elefa.

Soluţie

Vom utiliza funcţia recursivă Cauta(x,y), care verifică dacă următoarea literă din cuvânt (indicată de variabila globală Lg, care reţine lungimea prefixului curent) se află sau nu pe poziţia (x,y) în matricea C. În caz afirmativ, am mai găsit o literă în matrice, deci lungimea prefixului curent creşte şi continuăm căutarea în poziţiile adiacente poziţiei (x,y). În caz contrar, am obţinut un prefix maximal şi 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 depăşit lungimea cuvântului sau limitele indicilor din matrice, vom borda matricea cu spaţii şi vom marca sfârşitul 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 aceeaşi literă de mai multe ori;
- nu există riscul de a apela recursiv funcţia Cauta la infinit (sau mai exact până se umple stiva) deoarece la fiecare apel mărim Lg, lungimea prefixului curent, şi după strlen(s) paşi vom ajunge la '.', marcajul de sfârşit 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);
```