Crețu Laurențiu Vasile
CTI, Grupa 161

Tema 4 – Programare Dinamică

Problema 12

Enunț:

Ali Baba și cei 40 de hoți stăpânesc un deșert de formă dreptunghiulară, împărțit în n linii și m coloane, care definesc n*m sectoare. În fiecare sector se află o comoară ascunsă de Ali Baba. Se cunoaște valoarea în galbeni a fiecărei comori.Un călător trebuie să traverseze deșertul de la Nord la Sud, trecând dintr-un sector în altul, astfel: din sectorul (i j) se poate ajunge în unul din sectoarele (i+1,j-1), (i+1,j) sau (i+1,j+1), dar fără a părăsi deșertul (ar fi omorât de oamenii lui Ali Baba). La trecerea printr-un sector, călătorul colectează comoara din acel sector.
Determinați valoarea totală maximă a comorilor pe care le poate colecta călătorul la traversarea deșertului, știind că pleacă din orice sector al liniei 1 și se oprește în orice sector al linei n, cu respectarea condițiilor de mai sus.

Date de intrare:
Fişierul de intrare **date.in** conţine pe prima linie numerele n și m. Următoarele n linii conțin câte m numere naturale, reprezentând valorile comorilor din fiecare sector.

Date de ieșire:
Fişierul de ieşire **date.out** va conţine pe prima linie numărul V, reprezentând valoarea maximă a comorilor care pot fi colectate.

Exemplu:

date.in	date.out
4 5 5 8 3 7 7 1 1 4 5 1 5 8 9 1 7 5 8 6 6 9	29

Explicație:
Un traseu prin care se colectează comori în valoare de *29* galbeni este:

5 8 3 **7** 7
1 1 4 **5** 1
5 8 **9** 1 7
5 **8** 6 6 9

1. Rezolvare cu metoda Backtracking

La o primă vedere, această problemă se poate rezolva ușor cu metoda Backtracking și anume: pornim o buclă *for* ce parcurge prima linie a matricei citite și începem un algoritm de tip Backtracking pentru fiecare indice de coloană `j` (0, 1, 2, .., m-1). În acest algoritm Backtracking aflăm suma maximă ce se poate obține pornind de pe elementul de coordonate `(0,j)` și încheind pe oricare element al ultimei linii. La fiecare apelare a subprogramului `void bkt(int i_curent,int j_curent,int suma)` verificăm dacă ne aflăm pe ultima linie a matricei (`i_curent == n-1`) și în caz afirmativ verificăm dacă această sumă obținută este mai mare decât suma maximă obținută până în acel moment, iar dacă acest lucru se verifică suma maximă ia valoarea sumei curente obținute. În cazul în care nu ne aflăm pe ultima linie, ne deplasăm în continuare în matrice pe una din direcțiile: `(i+1,j-1)`, `(i+1,j)` sau `(i+1,j+1),` verificând la fiecare pas dacă acești indici sunt valizi (`i>=0 && i<n && j>=0 && j<m`) și dacă acest lucru se confirmă mergem mai departe în recursivitate cu apelul `bkt(i_vecin,j_vecin,suma+matrice[i_vecin][j_vecin])`. Algoritmul se încheie când toate aceste sume au fost calculate și a fost afișată cea maximă. Acest tip de rezolvare aduce o complexitate maximă $\mathcal{O}(3^{n \cdot m})$, dată de numărul de vecini ce trebuiesc vizitați.

Datele de intrare:

```
int n,m;
int **matrice;
int di[] = {1,1,1};
int dj[] = {-1,0,1};
int maxim;
```

Citirea datelor din fișier:

```
void citire(){
    FILE *fin = fopen("date.in","r");
    fscanf(fin,"%d %d",&n,&m);
    matrice = (int**)malloc(n*sizeof(int*));
    for (int i=0; i<n; i++)
        matrice[i] = (int*)malloc(m*sizeof(int));
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
```

```
            fscanf(fin,"%d",&matrice[i][j]);
    fclose(fin);
}
```

Algoritmul de Backtracking:

```
int valid(int i,int j){
    if (i<0 || i>=n || j<0 || j>=m)
        return 0;
    return 1;
}

void bkt(int i_curent,int j_curent,int suma){
    if (i_curent==n-1){
        if (suma>maxim)
            maxim = suma;
    }
    else{
        for (int i=0; i<3; i++){
            int i_vecin = i_curent + di[i];
            int j_vecin = j_curent + dj[i];
            if (valid(i_vecin,j_vecin)==1){
                bkt(i_vecin,j_vecin,suma+matrice[i_vecin][j_vecin]);
            }
        }
    }
}
```

Analiza timpului de executare:

Nr. Crt.	Numarul de linii N	Numărul de coloane M	Timpul de executare T(s)
1	5	11	0.000976
2	12	19	0.037435
3	17	24	10.662481
4	21	7	107.087651

* Date calculate pe Linux 5.10.0-kali3-amd64, compilator gcc 10.2.1 20210110

2. Rezolvare cu metoda Programării Dinamice

O variantă optimă de rezolvare este cea a metodei Programării Dinamice, în care construim o nouă matrice `suma` după următoarea formulă de recurență:

$$\text{suma}[i][j] = \begin{cases} \text{matrice}[n-1][j], & i = n-1 \\ \text{matrice}[i][j] + \text{maxim}(\text{suma}[i+1][j], \text{suma}[i+1][j+1]), & j = 0 \\ \text{matrice}[i][j] + \text{maxim}(\text{suma}[i+1][j-1], \text{suma}[i+1][j]), & j = m-1 \\ \text{matrice}[i][j] + \text{maxim}(\text{suma}[i+1][j-1], \text{suma}[i+1][j], \text{suma}[i+1][j+1]), & j > 0 \text{ și } j < m-1 \end{cases}$$

După construirea matricei `suma`, aflăm maximul de pe prima linie a matricei, ce va reprezenta suma căutată. Complexitatea acestei implementări este $\mathcal{O}(n \cdot m)$.

Citirea datelor de intrare:

```
void citire(int ***matrice,int *n,int *m){
    FILE *fin = fopen("date.in","r");
    fscanf(fin,"%d %d",n,m);
    *matrice = (int**)malloc(*n*sizeof(int*));
    for (int i=0; i<*n; i++)
        (*matrice)[i] = (int*)malloc(*m*sizeof(int));
    for (int i=0; i<*n; i++)
        for (int j=0; j<*m; j++)
            fscanf(fin,"%d",&(*matrice)[i][j]);
    fclose(fin);
}
```

Algoritmul de Programare Dinamică:

```
int maxim(int a,int b){
    if (a>b)
        return a;
    return b;
}

int maxim2(int a,int b,int c){
    int maxim = a;
```

```
    if (b>maxim)
        maxim = b;
    if (c>maxim)
        maxim = c;
    return maxim;
}

void dezalocare(int **matrice,int n){
    for (int i=0; i<n; i++)
        free(matrice[i]);
    free(matrice);
}

void programare_dinamica(int **matrice,int n,int m){
    int **suma;
    suma = (int**)malloc(n*sizeof(int*));
    for (int i=0; i<n; i++)
        suma[i] = (int*)calloc(m,sizeof(int));

    for (int j=0; j<m; j++)
        suma[n-1][j] = matrice[n-1][j];

    for (int i = n-2; i>=0; i--)
        for (int j=0; j<m; j++){
            if (j==0)
                suma[i][j] = matrice[i][j] + maxim(suma[i+1][j],
suma[i+1][j+1]);
            if (j==m-1)
                suma[i][j] = matrice[i][j] + maxim(suma[i+1][j-1],
suma[i+1][j]);
            if (j>0 && j<m-1)
                suma[i][j] = matrice[i][j] + maxim2(suma[i+1][j-1],
suma[i+1][j], suma[i+1][j+1]);
        }

    int maxim = suma[0][0];
    for (int j=1; j<m; j++)
        if (maxim<suma[0][j])
            maxim = suma[0][j];

    FILE *fout = fopen("date.out","w");
    fprintf(fout,"%d",maxim);
```

```
    fclose(fout);
    dezalocare(matrice,n);
    dezalocare(suma,n);
}
```

Analiza timpului de executare:

Nr. Crt.	Numarul de linii N	Numărul de coloane M	Timpul de executare T(s)
1	2	4	0.000462
2	68	30	0.000685
3	459	419	0.027230
4	810	667	0.082520
5	3949	2667	1.423450
6	5054	9987	8.209006
7	16868	18376	67.511285

* Date calculate pe Linux 5.10.0-kali3-amd64, compilator gcc 10.2.1 20210110

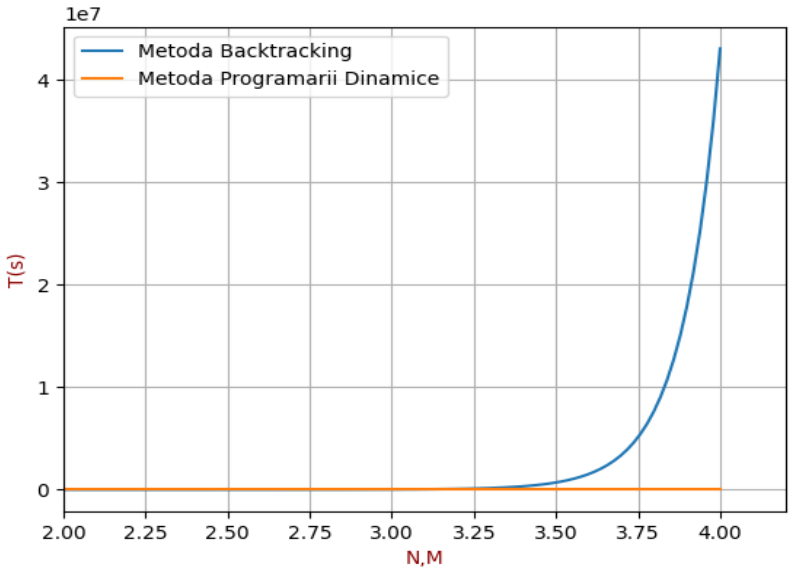
3. Generarea datelor de test

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define NMAX 1000
#define NMIN 2
#define MAX_VALUE 100
#define MIN_VALUE 1

void generare_date(){
    FILE *fin = fopen("date.in","w");
    srand(time(0)); rand();
    int n = (int)rand()/(RAND_MAX/NMAX)+NMIN; rand();
    int m = (int)rand()/(RAND_MAX/NMAX)+NMIN;
    fprintf(fin,"%d %d\n",n,m);

    for (int i=0; i<n; i++){
        for (int j=0; j<m; j++){
            int x = rand()/(RAND_MAX/MAX_VALUE)+MIN_VALUE; rand();
            fprintf(fin,"%d ",x);
        }
```

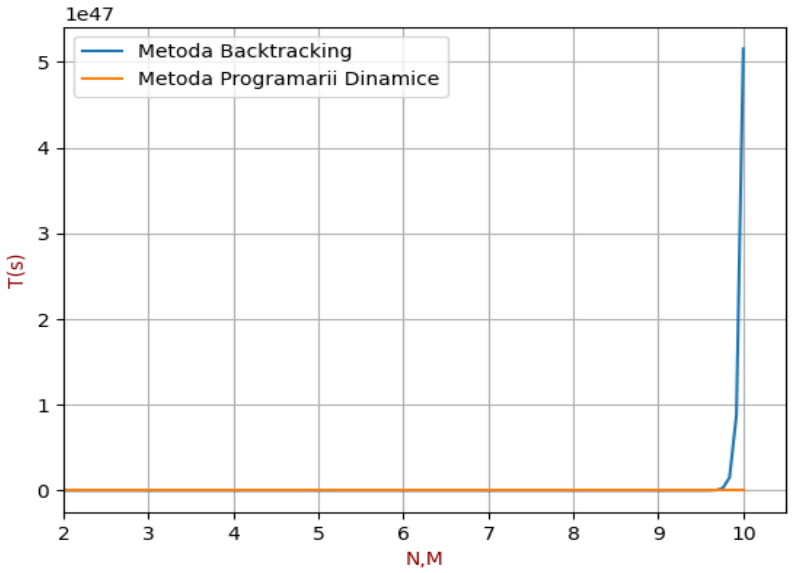
```
        fprintf(fin,"\n");
    }
    fclose(fin);
}
```

4. Compararea celor două abordări – reprezentare grafică

- Pentru N,M ∈ [2,4]:



- Pentru N,M ∈ [2,10]:



- Pentru N,M ∈ [2,100]: