Metoda Programării dinamice

Sumar

Competenţe generale

- *elaborarea algoritmilor de rezolvare a problemelor*
- *implementarea algoritmilor într-un limbaj de programare*

Competenţe specifice

- *analiza problemei în scopul identificării metodei de programare adecvate pentru rezolvarea problemei*
- *aplicarea creativă a metodelor de programare pentru rezolvarea unor probleme intradisciplinare sau interdisciplinare, sau a unor probleme cu aplicabilitate practică*
- *analiza comparativă a eficienţei diferitelor metode de rezolvare a aceleiaşi probleme şi alegerea unui algoritm eficient de rezolvare a unei probleme*
- elaborarea unui algoritm de rezolvare a unor probleme din aria curriculară a specializării
- utilizarea tehnicilor moderne în implementarea aplicaţiilor

Generalități

- metoda *programării dinamice* se aplică problemelor de optimizare a căror soluție se poate construi dinamic în timp
- metodă dezvoltată de Richard Bellman, care publică în anul 1957 o carte cu titlul ”*Dynamic programming*” în care enunță principiul optimalității
- *programare* – planificare
- *dinamic* – maniera în care se construiesc soluțiile parțiale
- este cea mai flexibilă metodă de programare, nu este o metodă standardizată
- metoda este adecvată problemelor care solicită determinarea unui optim (minim/maxim)
- metoda determină una dintre soluțiile optime ale problemei, chiar dacă problema are mai multe soluții optime
- pentru a determina toate soluțiile optime, algoritmul trebuie combinat cu unul de tip backtracking

- metodă de rezolvare a unor probleme complexe prin descompunerea lor în subprobleme mai mici și rezolvarea acestora
- se aplică problemelor cu următoarele proprietăți:
  - *suprapunere a subproblemelor*
  - *structură optimală*

Suprapunerea subproblemelor	**Structură optimală**
- subproblemele nu sunt independente - subproblemele se suprapun (sunt imbricate) - soluția unei subprobleme se utilizează în construirea soluției altei subprobleme	- problema poate fi descompusă în subprobleme - soluția unei probleme de optimizare poate fi obținută prin combinarea unor soluții optimale ale subproblemelor

Caracteristici

- soluția optimă se alege dintr-o mulțime de soluții, fiecărei soluții putând să i se asocieze o valoare
- problema poate fi descompusă în subprobleme similare cu problema inițială care respectă principiul optimalității
- subproblemele în care se descompune problema nu sunt independente
- soluția problemei se memorează într-un tablou pentru a economisi timp în cazul în care această soluție ar trebui folosită și în alte subprobleme (operație numită *memoizare*)

Abordări

- înainte (în rezolvarea problemei se pleacă de la starea finală)
- înapoi (în rezolvarea problemei se pleacă de la starea inițială)
- mixtă (combinare înainte-înapoi)

Etape

- împărțirea problemei în subprobleme de dimensiuni mai mici, numite subprobleme unitare
- rezolvarea subproblemelor în ordinea crescătoare a dimensiunilor subproblemelor
- combinarea soluțiilor subproblemelor pentru a obține soluția optimă a problemei date

Subșir crescător maximal

Fie **A** un șir de numere întregi de lungime **n**, având forma:

$$A=(a_1,a_2,a_3,\ldots,a_n).$$

Să se determine lungimea celui mai lung subșir crescător al șirului **A**.

sir.in	sir.out
9 6 3 1 2 7 8 4 9 5	5

Terminologie:

- se numește *subșir* al șirului **A** o succesiune de elemente din **A**, în ordinea în care acestea apar în **A**, nu neapărat în poziții consecutive
- un *subșir* se numește *crescător* dacă elementele sale sunt în ordine crescătoare
- *lungimea* unui subșir reprezintă numărul de elemente care alcătuiesc subșirul respectiv

Date:

Intrări: n

$a_1, a_2, a_3, \ldots, a_n$

Ieșiri: l_{max}

unde $L=(l_1, l_2, l_3, \ldots, l_n)$

$L[i]$ memorează lungimea celui mai lung subșir crescător care începe la poziția i, $1 \le i \le n$

$l_{max} = max(L[i]),\ 1 \le i \le n$

Analiză:

- se utilizează metoda ”înainte” (deplasare de la sfârșitul tabloului);
- se folosesc doi vectori:
  - **A=($a_1, a_2, a_3, \ldots, a_n$)**, care memorează elementele șirului dat
  - **L=($l_1, l_2, l_3, \ldots, l_n$)**, care memorează lungimile subșirurilor maxime;
- pentru fiecare element a_i se calculează lungimea celui mai lung subșir care se formează cu el
- l_i memorează lungimea celui mai lung subșir crescător care începe la poziția **i**

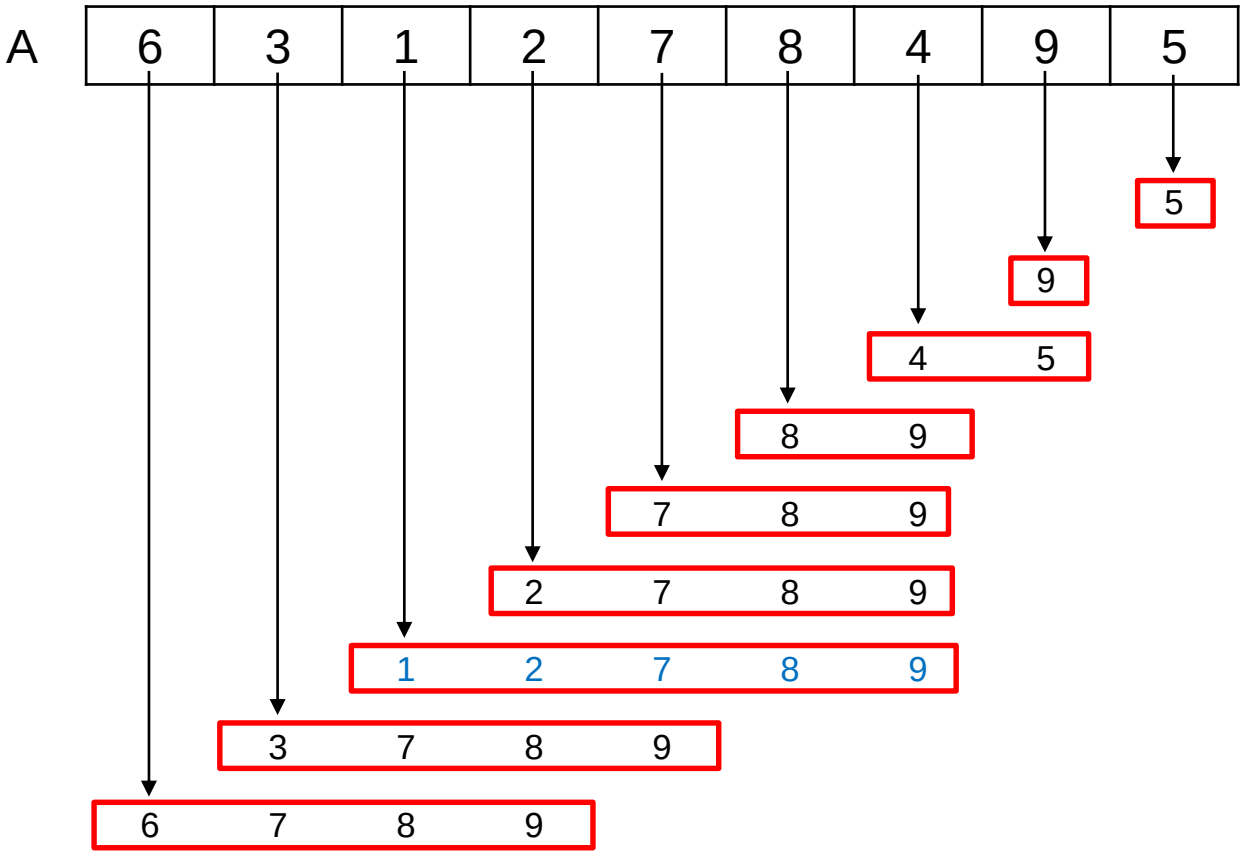
L[i]=max(L[j]+1)

unde **j>i** și **A[j]≥A[i]**

- relația de recurență:

$$L[i] = \begin{cases} 1, & pentru\ i = n \\ \max(L[j] + 1), & pentru\ a[i] \le a[j], \quad unde\ i < j \le n \end{cases}$$

Soluție:



A	6	3	1	2	7	8	4	9	5
L	4	4	**5**	4	3	2	2	1	1
	1	2	3	4	5	6	7	8	9

Programul C++:

```cpp
#include <fstream>

using namespace std;

ifstream fin("sir.in");
ofstream fout("sir.out");

void citire_sir(int A[], int &n)
{
    int i;
    fin>>n;
    for(i=1;i<=n;i++)
        fin>>A[i];
}

void construire_lungmi(int A[], int n, int L[])
{
    int i,j,maxl;
    L[n]=1;
    for(i=n-1;i>=1;i--)
    {
        maxl=0;
        for(j=i+1;j<=n;j++)
            if(A[i]<=A[j] && L[j]>maxl)
                maxl=L[j];
        L[i]=maxl+1;
    }
}

int determinare_maxim(int L[], int n)
{
    int i,Lmax;
    Lmax=L[1];
    for(i=2;i<=n;i++)
        if(L[i]>Lmax)
            Lmax=L[i];
    return Lmax;
}

void afisare(int Lmax)
{
    fout<<Lmax;
}

int main()
{
    int A[101],L[101],n,Lmax;
    citire_sir(A,n);
    construire_lungmi(A,n,L);
    Lmax=determinare_maxim(L,n);
    afisare(Lmax);
    fin.close();
    fout.close();
    return 0;
}
```

Temă:

- să se modifice funcția **`construire_lungimi(int A[], n, L[])`** astfel încât construirea vectorului **`L`** să se facă de la stânga la dreapta (metoda ”înapoi”);

Reconstruirea soluției:

Să se determine și să se afișeze un subșir de lungime maximă.

sir.in	sir.out
9 6 3 1 2 7 8 4 9 5	1 2 7 8 9

Date:

Intrări: n

$a_1, a_2, a_3, \ldots, a_n$

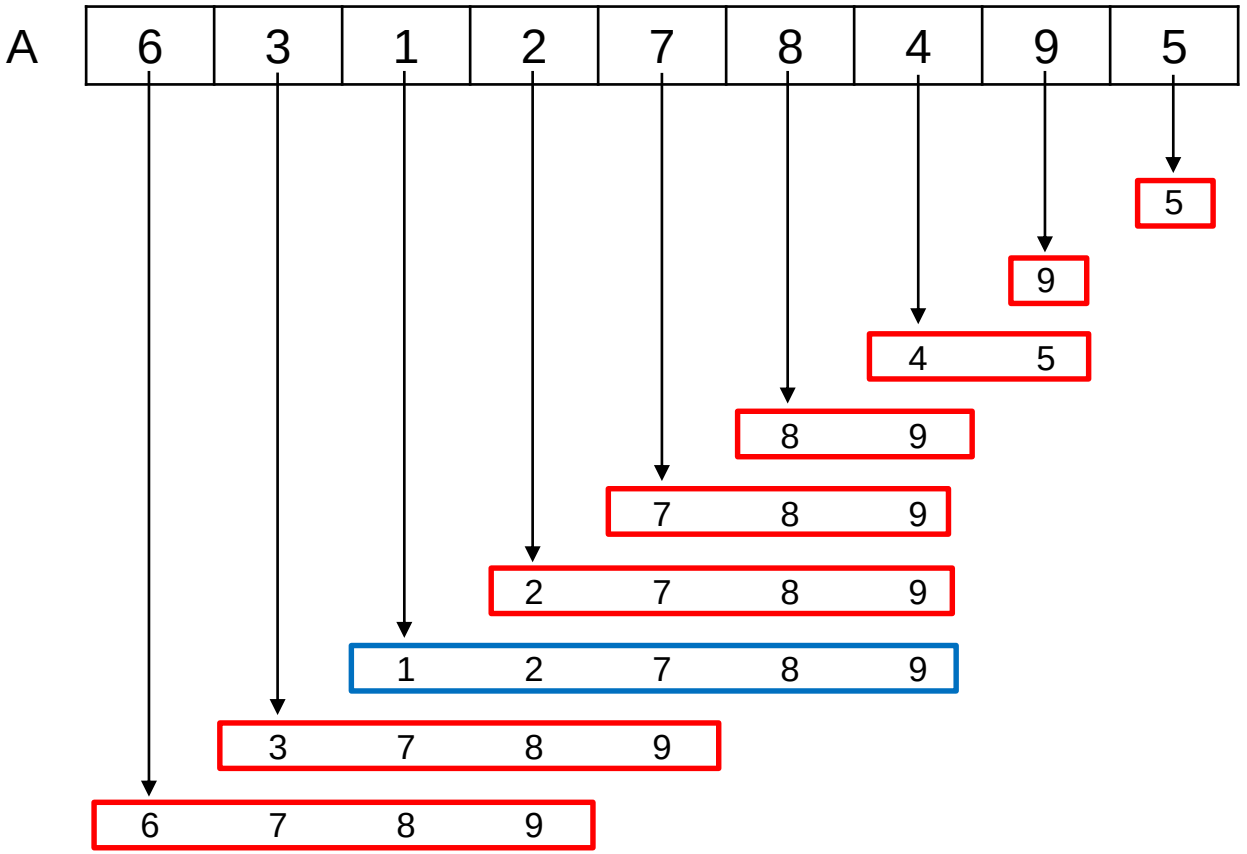
Ieșiri: $a_{j1}, a_{j2}, a_{j3}, \ldots, a_{jk}$

unde $a_{j1} \le a_{j2} \le a_{j3} \le \ldots \le a_{jk}$ are numărul maxim de elemente, $1 \le k \le n$

Analiza:

- se determină poziția **poz** a primului element din subșirul de lungime maximă și se afișează acest element;
- se caută următorul element din șirul dat care respectă condițiile:
  - are o valoare mai mare sau egală decât cea a ultimului element afișat
  - lungimea celui mai lung subșir care începe cu acest element este cu o unitate mai mică decât lungimea subșirului care începe cu ultimul element afișat
- se repetă procedeul pentru toate elementele rămase în șirul dat

Soluție:



A	6	3	1	2	7	8	4	9	5
L	4	4	**5**	4	3	2	2	1	1
	1	2	3	4	5	6	7	8	9

Programul C++:

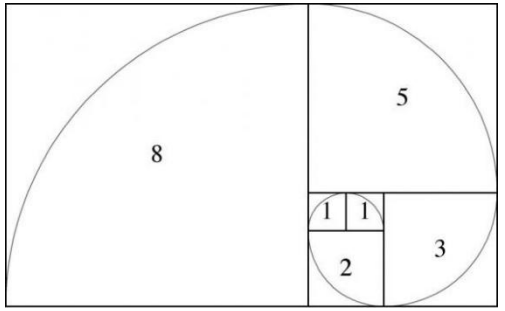
```cpp
#include <fstream>

using namespace std;

ifstream fin("sir.in");
ofstream fout("sir.out");

void citire_sir(int A[], int &n)
{
    int i;
    fin>>n;
    for(i=1;i<=n;i++)
        fin>>A[i];
}

void construire_lungmi(int A[], int n, int L[])
{
    int i,j,maxl;
    L[n]=1;
    for(i=n-1;i>=1;i--)
    {
        maxl=0;
        for(j=i+1;j<=n;j++)
            if(A[i]<=A[j] && L[j]>maxl)
                maxl=L[j];
        L[i]=maxl+1;

    }
}

void determinare_maxim(int L[],int n,int &Lmax,int &poz)
{
    int i;
    Lmax=L[1];
    poz=1;
    for(i=2;i<=n;i++)
        if(L[i]>Lmax)
        {
            Lmax=L[i];
            poz=i;
        }
}

void afisare(int A[],int n,int L[],int Lmax,int poz)
{
    int i;
    fout<<A[poz]<<' ';
    for(i=poz+1;i<=n;i++)
        if(L[i]==Lmax-1 && A[i]>=A[poz])
        {
            fout<<A[i]<<' ';
            poz=i;
            Lmax--;
        }
}

int main()
{
    int A[101],L[101],n,Lmax,poz;
    citire_sir(A,n);
    construire_lungmi(A,n,L);
    determinare_maxim(L,n,Lmax,poz);
    afisare(A,n,L,Lmax,poz);
    fin.close();
    fout.close();
    return 0;
}
```

Exemplul 1

Șirul lui Fibonacci

Numerele Fibonacci sunt definite prin următoarea relație de recurență:

$f_0 = 0,$
$f_1 = 1,$
$f_i = f_{i-1}+f_{i-2}$, pentru $i \geq 2$



Astfel, fiecare număr Fibonacci este suma celor două numere Fibonacci anterioare, rezultând secvența:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Să se determine al **n**-lea număr al șirului lui Fibonacci.

sir.in	**sir.out**
9	**34**

Șirul lui Fibonacci

- pentru a calcula termenul **`f(n)`** este nevoie de termenii **`f(n-1)`** și **`f(n-2)`**
- dacă acești termeni ar fi calculați recursiv ar trebui ca același termen să fie calculat de mai multe ori
- o soluție optimă ar fi salvarea termenilor șirului pentru a putea fi folosiți în calculul altor termeni
- o astfel de soluție optimă folosește principiul optimalității

Șirul lui Fibonacci

Calculul recursiv

```
1  #include <fstream>
2
3  using namespace std;
4
5  ifstream fin("sir.in");
6  ofstream fout("sir.out");
7
8  int f(int n)
9  {
10     if(n==0 || n==1)
11         return n;
12     else
13         return f(n-1)+f(n-2);
14 }
15
16 int main()
17 {
18     int n;
19     fin>>n;
20     fout<<f(n-1);
21     fin.close();
22     fout.close();
23     return 0;
24 }
```

- implementare ineficientă
- subproblemele se suprapun
- redundanță foarte mare a operațiilor

Șirul lui Fibonacci

Principiul optimalității (memoizare)

```
1  #include <fstream>
2
3  using namespace std;
4
5  ifstream fin("sir.in");
6  ofstream fout("sir.out");
7
8  int f(int n)
9  {
10     int a[100],i;
11     a[0]=0;
12     a[1]=1;
13     for(i=2;i<n;i++)
14         a[i]=a[i-1]+a[i-2];
15     return a[n-1];
16
17 }
18
19 int main()
20 {
21     int n;
22     fin>>n;
23     fout<<f(n);
24     fin.close();
25     fout.close();
26     return 0;
27 }
```

- valorile termenilor din șir sunt memorate în tablou
- utilizează spațiu adițional
- eficiență de timp

Șirul lui Fibonacci

Eficiență

```
1  #include <fstream>
2
3  using namespace std;
4
5  ifstream fin("sir.in");
6  ofstream fout("sir.out");
7
8  int f(int n)
9  {
10     int x,y,i;
11     x=0;
12     y=1;
13     for(i=2;i<n;i++)
14     {
15         y=x+y;
16         x=y-x;
17     }
18     return y;
19
20 }
21
22 int main()
23 {
24     int n;
25     fin>>n;
26     fout<<f(n);
27     fin.close();
28     fout.close();
29     return 0;
30 }
```

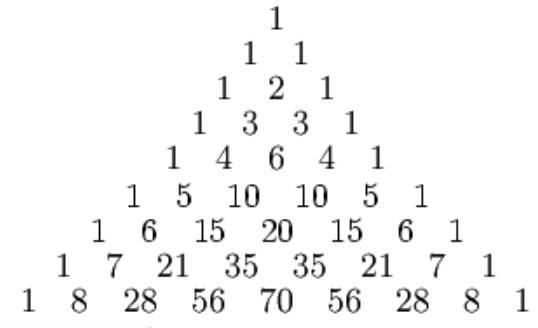
- eficiență d.p.d.v. al timpului de executare
- eficiență d.p.d.v. al memoriei utilizate

Exemplul 2

Triunghiul lui Pascal

Triunghiul lui Pascal este un aranjament geometric al coeficienților binomiali.
Înălțimea și laturile triunghiului conțin cifra **1**, iar fiecare număr de pe o linie **n** reprezintă suma celor două numere de pe linia superioară **n-1**.

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1} \text{ pt. } 0 < k < n,$$
$$C_n^0 = C_n^n = 1$$



Să se determine numărul ***k***-combinarilor dintr-o mulțime cu **n** elemente.

combinari.in	**combinari.out**
8 3	**56**

Triunghiul lui Pascal

Calculul recursiv

```
1  #include <fstream>
2
3  using namespace std;
4
5  ifstream fin("combinari.in");
6  ofstream fout("combinari.out");
7
8  int f(int n,int k)
9  {
10     if(k==0 || k==n)
11         return 1;
12     else
13         return f(n-1,k-1)+f(n-1,k);
14 }
15
16 int main()
17 {
18     int n,k;
19     fin>>n>>k;
20     fout<<f(n,k);
21     fin.close();
22     fout.close();
23     return 0;
24 }
```

- implementare ineficientă
- subproblemele se suprapun
- aceleași valori se calculează de mai multe ori

Triunghiul lui Pascal

Principiul optimalității (memoizare)

```cpp
#include <fstream>

using namespace std;

ifstream fin("combinari.in");
ofstream fout("combinari.out");

const int N = 100;
int c[N+1][N+1];

void f(int n)
{
    int i;
    for(int i=0;i<=n;i++)
        c[i][0]=c[i][i]=1;
    for(i=1;i<=n;i++)
        for(int j=1;j<i;j++)
            c[i][j]=c[i-1][j]+c[i-1][j-1];

}

```

- valorile combinărilor sunt memorate în tablou
- valorile se calculează progresiv

```cpp
void afisare_triunghi(int n)
{
    int i,j;
    for(i=0;i<=n;i++)
    {
        for(j=0;j<=i;j++)
            fout<<c[i][j]<<" ";
        fout<<endl;
    }
}

int main()
{
    int n,k;
    fin>>n>>k;
    f(n+1);
    afisare_triunghi(n);
    fout<<c[n][k];
    fin.close();
    fout.close();
    return 0;
}
```

Fişă de lucru

- Aplicaţii metoda programării dinamice

6. Bibliografie şi webografie

1. Miloşescu M., *Informatică. Manual pentru clasa a XI*, Editura Didactică şi Pedagogică, Bucureşti, 2006
2. https://en.wikipedia.org/wiki/Dynamic_programming
3. https://ro.wikipedia.org/wiki/Triunghiul_lui_Pascal
4. https://en.wikipedia.org/wiki/Fibonacci_number