Programarea dinamică

Sau cum determinăm soluția optimă

Introducere

- Programarea dinamică este o tehnică ce poate fi aplicată pentru rezolvarea acelor probleme pentru care se cere o soluţie optimă.
- Programarea dinamică se aplică în general problemelor de optimizare, atunci când dorim să determinăm rapid soluţia optimă pentru o problemă. Aplicând această tehnică determinăm **una din soluţiile optime,** problema putând avea mai multe soluţii optime.
- Nu există un criteriu pe baza căruia să identificăm cu siguranţă o problemă pentru rezolvarea căreia trebuie să utilizăm metoda programării dinamice.
- Se crește consumul de memorie pentru o scădere a timpului de execuție.

Principii de funcționare

- Programarea dinamică are la bază principiul optimalităţii: problema poate fi descompusă în subprobleme asemănătoare de dimensiuni mai mici iar soluţiile optime ale acestor subprobleme contribuie la obţinerea soluţiei optime pentru problema iniţială.
- Programarea dinamică presupune rezolvarea unei probleme prin descompunerea ei în subprobleme și rezolvarea acestora.
- Spre deosebire de divide-et-impera, subproblemele nu sunt disjuncte, ci se suprapun.
- Pentru optimizare vom rezolva subproblemele o singură dată, reţinând rezultatele într-o structură de date suplimentară (vector, matrice bidimensională).

Principii de funcționare

- Rezolvarea unei probleme prin programare dinamică presupune următorii paşi:
- Se identifică subproblemele problemei date.
- Se alege o structură de date suplimentară, capabilă să reţină soluţiile subproblemelor.
- Se caracterizează substructura optimală a problemei printr-o relaţie de recurenţă.
- Pentru a determina soluţia optimă, se rezolvă relaţia de recurenţă în mod *bottom-up* (se rezolvă subproblemele în ordinea crescătoare a dimensiunii lor).

Exemple de probleme

- Trepte
- Cel mai lung subșir crescător
- Subșir comun maximal
- Sumă maximă/minimă în triunghi
- Drum minim în labirint

Trepte

- O persoana are de urcat **n** trepte. Ştiind că de pe treapta i poate trece pe treapta i + 1, i + 2, ..., i + (**k** - 1) sau i + **k**, aflați în câte moduri poate urca cele n trepte. (inițial este pe treapta 1)
- Exemplu: pentru **n**=4 și **k**=2 se afișează 3

  (1 -> 2 -> 3 -> 4, 1 -> 2 -> 4, 1 -> 3 -> 4)
- Exemplu: pentru **n**=10 și **k**=3

1	2	3	4	5	6	7	8	9	10
1	1	2	4	7	13	24	44	81	149

Implementare Trepte

```
#include <iostream>

using namespace std;
const int mo=9001, nm=100002;
int ar[nm*2+1], *a=ar+nm, n, k, i, j;
int main()
{
    cin >> n >> k;
    a[1]=1;a[2]=1;
    for(i=3; i<=n; i++)
        for(j=1;i>j&&j<=k;j++)
          a[i]=(a[i]+a[i-j])%mo;
    cout <<a[n]<<endl;

    return 0;
}
```

Subșir crescător

- Enunț: *Fiind dat un șir de **n** numere să se determine cel mai lung subșir strict crescător*.
- Exemplu: **n**=9, și elementele : 2 7 5 3 4 1 6 7 5
- Cel mai lung subșir crescător: 2 3 4 6 7
- Descriere soluție: Pentru a rezolva problema vom calcula pentru fiecare element de pe poziţia **i** în şirul de numere întregi **a** lungimea celei mai lungi subsecvenţe crescătoare care poate fi formată începând cu elementul **a**[i]. Memorarea se poate face într-un vector **lu**, unde **lu**[i] memorează lungimea șirului ce începe cu **a**[i].
- Dacă ne interesează doar o singură soluție putem memora pentru fiecare element **i** poziția succesorului său, sau 0 dacă acesta nu există.

Subșir crescător

- Funcționare algoritm:
- Pornim de la ultimul element **a**[n], pentru care **lu**[n]=1(lungimea șirului ce începe cu el e 1) iar **p**[n]=0 (nu are succesor)

	1	2	3	4	5	6	7	8	9
a	2	7	5	3	4	1	6	7	5
lu	5	1	3	4	3	3	2	1	1
p	4	0	7	5	7	7	8	0	0

- după completarea vectorilor se determină maximul din vectorul **lu** iar acesta e lungimea celui mai lung subșir crescător.

Implementare subșir crescător

```
#include <iostream>
#include <fstream>
using namespace std;
const int nmax=100;
int a[nmax],lu[nmax],p[nmax],n,i,x,mx,k,pmx,j;
void citire()
{
    ifstream fin("date.in");
    n=1;
    while(fin>>a[n])n++;
    n--;
    fin.close();
}
void afis(int *a, int n)
{
    int i;
    for(i=1;i<=n;i++)cout<<a[i]<<" ";
    cout<<endl;
}
```

Implementare subșir crescător

```
int main()
{
    citire();
    afis(a,n);
    mx=1;pmx=n;
    lu[n]=1;p[n]=0;//sirul care incepe cu al n-lea element are lungi
    for(i=n-1;i>0;i--)
    {   //caut un element mai mare ca a[i] care sa aiba cei mai mult
        k=0;//presupunem ca nu exista
        for(j=n;j>i;j--)//caut primtre elementele de dupa a[i]
         if((a[i]<a[j])&&(lu[j]>lu[k]))k=j;
        p[i]=k;//succesorul lui
        lu[i]=lu[k]+1;
        if(lu[i]>mx){mx=lu[i];pmx=i;}
    }
    i=pmx;
    while(i>0)
    {
        cout<<a[i]<<" ";
        i=p[i];
    }
    return 0;
}
```

Subşir comun maximal

- Enunţ: *Fie $X=(x_1, x_2, ..., x_n)$ şi $Y=(y_1, y_2, ..., y_m)$ două şiruri de n, respectiv m numere întregi. Determinaţi un subşir comun de lungime maximă.*
- Exemplu: pentru X=(2, 5, 5, 6, 2, 8, 4, 0, 1, 3, 5, 8) şi Y=(6, 2, 5, 6, 5, 5, 4, 3, 5, 8) o soluţie posibilă este: Z=(2,5,5,4,3,5,8)
- Descriere soluție: Notăm cu $X_k=(x_1, x_2, ..., x_k)$ (prefixul lui X de lungime k) şi cu $Y_h=(y_1, y_2, ..., y_h)$ prefixul lui Y de lungime h.
- O subproblemă a problemei date constă în determinarea celui mai lung subşir comun al lui X_k, Y_h.
- Notăm cu $LCS(X_k,Y_h)$ lungimea celui mai lung subşir comun al lui X_k, Y_h.
- Utilizând aceste notaţii, problema cere determinarea $LCS(X_n,Y_m)$, precum şi un astfel de subşir.

Subşir comun maximal

- *Observaţie*
- Dacă $X_k=Y_h$ atunci $LCS(X_k,Y_h)=1+LCS(X_{k-1},Y_{h-1})$.
- Dacă $X_k \neq Y_h$ atunci $LCS(X_k,Y_h)=max(LCS(X_{k-1},Y_h), LCS(X_k,Y_{h-1}))$.
- Pentru a reţine soluţiile subproblemelor vom utiliza o matrice cu n+1 linii şi m+1 coloane, denumită lcs. Linia şi coloana 0 sunt utilizate pentru iniţializare cu 0, iar elementul lcs[k][h] va fi lungimea celui mai lung subşir comun al şirurilor X_k şi Y_h.

Subsir comun maximal

- Exemplu pentru 2 șiruri de caractere: **cadastru** și **calendar**.
- Se creează matricea din dreapta

	c	a	l	e	n	d	a	r
c	1	1	1	1	1	1	1	1
a	1	2	2	2	2	2	2	2
d	1	2	2	2	2	3	3	3
a	1	2	2	2	2	3	4	4
s	1	2	2	2	2	3	4	4
t	1	2	2	2	2	3	4	4
r	1	2	2	2	2	3	4	5
u	1	2	2	2	2	3	4	5

Subsir comun maximal

- Exemplu pentru 2 șiruri de caractere: **cadastru** și **calendar**.
- Construcție șir comun.

	c	a	l	e	n	d	a	r
c	1	1	1	1	1	1	1	1
a	1	2	2	2	2	2	2	2
d	1	2	2	2	2	3	3	3
a	1	2	2	2	2	3	4	4
s	1	2	2	2	2	3	4	4
t	1	2	2	2	2	3	4	4
r	1	2	2	2	2	3	4	5
u	1	2	2	2	2	3	4	5

c	a	d	a	r

Implementare – subșir comun

```
#include <iostream>
#include <cstring>
using namespace std;
int i,j,n,m,a[257][257],l;
char s1[256],s2[256],rez[256];
int main()
{
    cin.getline(s1+1,256);
    cin.getline(s2+1,256);
    n=strlen(s1+1);
    m=strlen(s2+1);
```

Implementare – subșir comun

```
for(i=1;i<=n;i++)
    for(j=1;j<=m;j++)
        if(s1[i]==s2[j])a[i][j]=a[i-1][j-1]+1;
        else a[i][j]=max(a[i][j-1],a[i-1][j]);
cout<<a[n][m]<<endl;
```

Implementare – subșir comun

```
l=a[n][m];
i=n;j=m;
while(l>0)
{
    while(a[i][j]==a[i][j-1])j--;
    while(a[i][j]==a[i-1][j])i--;
    rez[--l]=s1[i]; i--; j--;
}
cout<<rez;
return 0;
```

Sumă maximă în triunghi

- Enunț: *Să considerăm un triunghi format din n linii ($1<n\leq 100$), fiecare linie conţinând numere întregi din domeniul [1,99], ca în exemplul următor:*

7

3 *8*

8 *1* *0*

2 ***7*** *4* *4*

4 ***5*** *2* *6* *5*

- *Problema constă în scrierea unui program care să determine cea mai mare sumă de numere aflate pe un drum între numărul de pe prima linie şi un număr de pe ultima linie. Fiecare număr din acest drum este situat sub precedentul, la stânga sau la dreapta acestuia.* (IOI, Suedia 1994)

Sumă maximă în triunghi

- Soluție: Vom reţine triunghiul într-o matrice pătratică T, de ordin n, sub diagonala principală.
- Pentru a reţine soluţiile subproblemelor, vom utiliza o matrice suplimentară S, pătratică de ordin n, cu semnificaţia S[i][j]= suma maximă ce se poate obţine pe un drum de la T[i][j] la un element de pe ultima linie, respectând condiţiile problemei.
- Evident, soluţia problemei va fi S[1][1].
- Observație:
- $S[n][i]=T[n][i], \forall i \in \{1,2,...,n\}$
- $S[i][j]=T[i][j]+\max(S[i+1][j], S[i+1][j+1])$

Implementare – suma maximă

Numere de suma cifrelor S

- Să se determine câte numere de **n** cifre au suma cifrelor **s.**

S

n

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
2	1	2	3	4	5	6	7	8	9	9	8	7	6	5	4
3	1	3	6	10	15	21	28	36	45	54	61	66	69	70	69
4	1	4	10	20	35	46	74								
5	1	5	15	35											

Se observă că: **a[i][j]=a[i-1][j]+a[i][j-1]-a[i-1][max(0,j-10)];**

Numere de suma cifrelor S

- Pentru a calcula câte numere cu i cifre au suma cifrelor j, ne folosim de câte numere de i-1 cifre au suma cifrelor j,j-1…

Drum minim în labirint

- Soluție: Se pornește din P și toți vecinii se notează cu 1, apoi vecinii valorilor de 1 se notează cu 2 ș.a.m.d. până când se ajunge la sosire.

2	1				
1	P		A		
2	1		9	A	S
3			8		
4	5	6	7		
5	6	7			
6	7	8	9	A	
7	8	9	A		

Drum de cost minim

- Matrice în care intrarea într-o celulă te costă. Trebuie găsit drumul de cost minim între 2 puncte.
- Folosesc o coadă ordonată în care pun punctul de pornire (1, 1) cu costul 1.

j

i

	1	2	3	4	5	
1	1	9	1	1	1	
2	1	9	1	9	1	
3	1	9	1	9	1	
4	1	1	1	9	1	

I	1							
J	1							
cost	1							

Drum de cost minim

- Din celula (1, 1) calculăm distanța până la celula din dreapta și o introducem în coadă.

I	1	1						
J	1	2						
cost	1	10						

j

i

	1	2	3	4	5	
1	1	9	1	1	1	
2	1	9	1	9	1	
3	1	9	1	9	1	
4	1	1	1	9	1	

Drum de cost minim

- Din celula 1, 1 calculăm distanța până la celula de mai jos și o introducem în coadă.

I	1	2	1					
J	1	1	2					
cost	1	2	10					

j

i

	1	2	3	4	5	
1	1	9	1	1	1	
2	1	9	1	9	1	
3	1	9	1	9	1	
4	1	1	1	9	1	

Drum de cost minim

- Ștergem primul element din coadă și continuăm cu următorul: celula (2,1)

I	2	1						
J	1	2						
cost	2	10						

j

i

	1	2	3	4	5	
1	1	9	1	1	1	
2	1	9	1	9	1	
3	1	9	1	9	1	
4	1	1	1	9	1	

Drum de cost minim

- Din celula (2, 1) calculăm distanța până la celula din dreapta și o introducem în coadă.

j

	1	2	3	4	5	
1	1	9	1	1	1	
2	1	9	1	9	1	
3	1	9	1	9	1	
4	1	1	1	9	1	

i

I	2	1	2					
J	1	2	2					
cost	2	10	11					

Drum de cost minim

- Din celula (2, 1) calculăm distanța până la celula de jos și o introducem în coadă.

j

i

	1	2	3	4	5	
1	1	9	1	1	1	
2	1	9	1	9	1	
3	1	9	1	9	1	
4	1	1	1	9	1	

I	2	3	1	2				
J	1	1	2	2				
cost	2	3	10	11				

Drum de cost minim

- Am terminat cu vecinii celulei (2, 1), o scoatem din coadă și continuăm cu vecinii celulei (3, 1).

j

	1	2	3	4	5	
1	1	9	1	1	1	
2	1	9	1	9	1	
3	1	9	1	9	1	
4	1	1	1	9	1	

i

I	3	1	2					
J	1	2	2					
cost	3	10	11					

Drum de cost minim

- Din celula (3, 1) calculăm distanța până la celula din dreapta și o introducem în coadă.

I	3	1	2	3				
J	1	2	2	2				
cost	3	10	11	12				

j

i

	1	2	3	4	5	
1	1	9	1	1	1	
2	1	9	1	9	1	
3	1	9	1	9	1	
4	1	1	1	9	1	