Metoda programării dinamice

- **Problemă.** Se consideră vectorul w = (w1,..., wn). Să se determine un subvector al lui w care nu conține elemente consecutive în w și are suma elementelor maximă

Exemplu: Pentru w = (1, 4, 7, 5) soluția este (4, 5)

Problemă echivalentă: Se consideră un graf de tip **lanț** cu V = {v_1,..., v_n}.
Vârfurile grafului au asociate ponderile w_1,..., respectiv w_n. Să se determine o mulțime independentă de vârfuri de pondere maximă - **Maximum Weighted Independent Set** (Aplicații – probleme de alocare de resurse cu evitarea interferenței, indicată prin muchii ⇒ graf de conflicte)

- **Abordare cu metoda Greedy** nu furnizează soluție optimă
- **Abordare cu metoda Divide et Impera** - soluția nu este corectă
- Generarea tuturor soluțiilor posibile și determinarea celei optime – algoritm exponențial

Analizăm structura unei soluții optime, evidențiind un element (primul/ultimul) al acesteia, pentru a determina subprobleme utile și relații de recurență

Fie S ⊆ V = {v_1,..., v_n} o soluție optimă

- Dacă $v_n \in S \Rightarrow$ S-{v_n} este soluție optimă pentru G-{v_n, v_{n-1}}
- Dacă $v_n \notin S \Rightarrow$ S este soluție optimă pentru G-{v_n}

Dacă am ști deja soluțiile pentru grafurile G-{v_n, v_{n-1}} și G-{v_n} , am putea determina S alegând dintre cele două situații cazul în care se obține soluția optimă

Notăm

S(i) = ponderea maximă a unei mulțimi independete în graful indus de vârfurile {v_1,..., v_i}

S(n) = max{ S(n-2)+w_n, S(n-1)}

S(1) = w_1, S(0) = 0

- Implementăm recursiv relațiile de recurență – trebuie să memorăm într-un vector rezultatele subproblemelor deja rezolvate (memoizare), altfel aceeaşi subproblemă va fi rezolvată de mai multe ori ⇒ algoritm O(n)

  sau

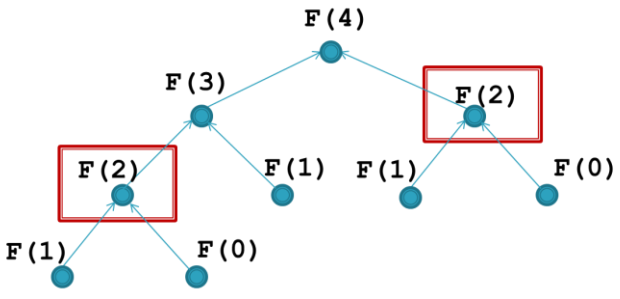
- Implementăm iterativ recurența (bottom-up)

Din relația de recurență putem deduce ce vârfuri au fost selectate în soluție.

Implementare: MIS.java

Concluzii.

1. Metoda Divide et Impera nu este eficientă dacă subproblemele în care se împarte o problemă nu sunt distincte/ se repetă (exemplu- Fibonacci `F(n)= F(n-1)+F(n-2)`)



2. Prin metoda Greedy nu obținem mereu o soluție optimă (exemplu - problema rucsacului cazul discret, problema monedelor)

- **Metoda programării dinamice**

Se poate aplica pentru probleme care presupun rezolvarea de relații de recurență. De obicei aceste relații se obțin din respectarea unui principiu de optimalitate (subprobleme optime)

Cadru general

Fie A şi B două mulţimi oarecare (B este de obicei **N**, **Z**, **R**, $\{0,1\}$ sau un produs cartezian).

Fiecărui element $x \in A$ **urmează să i se asocieze** o valoare $v(x) \in B$.

Inițial v este cunoscută doar pe submulţimea $X \subset A$, $X \neq \emptyset$.

Pentru fiecare $x \in A \setminus X$ știm că:

$$v(x) = f_x(v(a_1), \ldots, v(a_k))$$

unde $A_x = \{a_1, \ldots, a_k\} \subset A$ este mulţimea elementelor din A de a căror valoare depinde $v(x)$, iar f_x este o funcţie care specifică această dependență (poate fi un minim/maxim, o sumă etc).

Se mai dă $z \in A$.

Se cere să se calculeze (!eficient), **dacă este posibil**, valoarea $v(z)$.

Lucrurile devin mai clare dacă reprezentăm problema pe un *graf de dependenţe*. Vârfurile corespund elementelor din A, iar descendenţii unui vârf x sunt vârfurile din A_x. Vârfurile din X apar subliniate.

Exemple.

1. Fibonacci
2. Considerăm: $A = \{1,2,\ldots,5,6\}$;

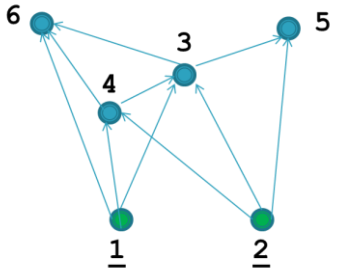
$$v(1) = v(2) = 1$$
$$v(3) = v(1) + v(2) + v(4)$$
$$v(4) = v(1) + v(2)$$
$$v(5) = v(2) + v(3)$$
$$v(6) = v(1) + v(3) + v(4)$$

Atunci $X = \{1,2\}$; $A_3 = \{1,2,4\}$; $A_4 = \{1,2\}$; $A_5 = \{2,3\}$; $A_6 = \{1,3,4\}$.

Fiecare funcţie f_x calculează $v(x)$ ca fiind suma valorilor elementelor din A_x. Alegem $z=6$. O ordine posibilă de a considera elementele lui $A \setminus X$ astfel încât să putem calcula valoarea asociată lor este: $4, 3, 6$. Astfel:

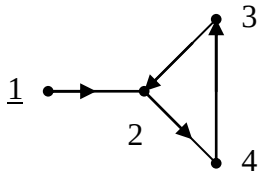
$$v(4) = v(1) + v(2) = 2,$$
$$v(3) = v(4) + v(1) + v(2) = 4,$$
$$v(6) = v(1) + v(3) + v(4) = 1 + 4 + 2 = 7$$

Să observăm că dacă dorim să calculăm v(6) este inutil să calculăm v(5).



Ne interesează doar valorile vârfurilor y de a căror valoare depinde v(z) (adică mulţimea vârfurilor `y` pentru care există un drum de la `y` la z în graful de dependenţe), numite *vârfurilor observabile* din z.

Problema enunţată nu are totdeauna soluţie, aşa cum se vede pe graful de dependenţe de mai jos, în care există un circuit care nu permite calculul lui `v` în `z=3`.



Ar fi bine dacă, atunci când vrem să calculăm v(z) :

- am cunoaşte de la început graful indus de mulţimea vârfurilor *observabile* din z ; vom numi acest graf *graful vârfurilor observabile* din z şi îl notăm G_z
- forma acestui graf ar permite o parcurgere mai simplă, care să conducă la calcularea valorii v(z).

Problema enunţată **are soluţie dacă şi numai dacă**:

1) G_z nu are circuite;
2) vârfurile din G_z în care nu sosesc arce fac parte din `X`.

➢ **Încercare de rezolvare cu metoda Divide et Impera.**

Este folosită o procedură `DivImp`, apelată prin `DivImp(z)`.

```
procedure DivImp(x)
   if x∈X then v(x) este cunoscut
   else
       for toţi y∈Ax\X    {valorile y de care x depinde direct}
             DivImp(y)
       calculează v(x) conform funcţiei fx
end;
```

Dezavantaje:

- algoritmul nu se termină pentru grafuri ciclice;
- **valoarea unui vârf poate fi calculată de mai multe ori**

➢ **Soluţie: Sortarea topologică pentru G_z** (care va da ordinea în care se calculează valorile v ale vârfurilor)

Ar fi totuşi mai bine dacă **forma grafului ar permite o parcurgere mai simplă.**

Metoda programării dinamice se aplică problemelor care urmăresc calcularea unei valori v(z) şi constă în următoarele:

1) Se asociază problemei un graf de dependenţe;
2) În graf este pus în evidenţă un graf de vârfuri observabile din z (numit **PD-arbore de rădăcină z)**; acest graf are proprietăţile:
   - nu conţine cicluri
   - mulţimea vârfurilor cu gradul intern 0 (i.e. ale căror valori nu depind de nicio altă valoare) este inclusă în mulţimea X

- se poate așeza pe niveluri (nivelul lui y = lungimea maximă a unui drum de la y la z)

Problema se reduce la determinarea valorii asociate lui `z` (rădăcina PD-arborelui);

3) Se parcurge în postordine PD-arborele (cu mici modificări, marcând vârfurile vizitate, adică cele pentru care s-a calculat deja valoarea ataşată, pentru a nu calcula o valoare de două ori). Mai exact, este iniţializat vectorul `vizitat` şi se începe parcurgerea prin apelul `postord(z)`:

```
for toate vârfurile x∈A
    if x∈X
       vizitat[x] ← true
    else
       vizitat[x] ← false
postord(z)
```

unde procedura `postord` cu argumentul `x` calculează `v(x)`:

```
procedure postord(x)
     for toţi j∈A_x
        if vizitat[j]=false    {diferența față de Divide et Impera}
           postord(j)
     calculează v(x) conform funcţiei f_x;
     vizitat[x]←true
end
```

Timpul de executare a algoritmului este evident liniar.

Prin parcurgerea în postordine, vârfurile vor fi sortate topologic.

Mai pe scurt, putem afirma că: ***Metoda programării dinamice constă în identificarea unui PD-arbore şi parcurgerea sa în postordine.***

Observaţie. Programarea dinamică generalizează metoda Divide et Impera în sensul că dependenţele nu au forma unui arbore, ci a unui PD-arbore.

În multe probleme este util să căutăm în PD-arbore regularităţi care să evite memorarea valorilor tuturor vârfurilor şi/sau să simplifice parcurgerea în postordine.

Un tip de probleme pentru care se poate utiliza metoda programării dinamice sunt **problemele de optim**, în care soluția este rezultatul unui șir finit de decizii d_1, ..., d_n (fiecare decizie depinzând de cele anterioare), care satisfac un **principiu de optimalitate** (din care se obţin relaţiile de calcul) sub una din următoarele forme. Fie secvența de stări S_0 (starea inițială), S_1, …, S_n (starea finală) prin care se ajunge după fiecare decizie:

$$S_0 \xrightarrow{d_1} S_1 \xrightarrow{d_2} S_2 \xrightarrow{d_3} .... \xrightarrow{d_n} S_n$$

(1) Dacă șirul d_1, ..., d_n duce sistemul în mod optim din S_0 în S_n, atunci pentru orice $1 \le k \le n$, șirul d_k, ..., d_n duce sistemul în mod optim din S_{k-1} în S_n.

$$(1)\ S_0 \xrightarrow{d_1,d_2,....,d_n} S_n \ \Rightarrow\ S_{k-1} \xrightarrow{d_k,....,d_n} S_n\ ,\ \forall\ 1 \le k \le n$$

(2) Dacă șirul d_1, ..., d_n duce sistemul în mod optim din S_0 în S_n, atunci pentru orice $1 \le k \le n$, șirul d_1, ..., d_k duce sistemul în mod optim din S_0 în S_k

$$(2)\ S_0 \xrightarrow{d_1,d_2,....,d_n} S_n \ \Rightarrow\ S_0 \xrightarrow{d_1,....,d_k} S_k\ ,\ \forall\ 1 \le k \le n$$

(3) Dacă şirul d_1, ..., d_n duce sistemul în mod optim din S_0 în S_n, atunci pentru orice $1 \le k \le n$, şirul d_1, ..., d_k duce sistemul în mod optim din S_0 în S_k, iar şirul d_{k+1}, ..., d_n duce sistemul în mod optim din S_k în S_n

$$(3)\ S_0 \xrightarrow{d_1,d_2,\ldots,d_n} S_n \ \Rightarrow\ S_0 \xrightarrow{d_1,\ldots,d_k} S_k\ ,\ \forall\ 1 \le k \le n$$
$$S_{k-1} \xrightarrow{d_k,\ldots,d_n} S_n\ ,\ \forall\ 1 \le k \le n$$

După ce principiul optimalității a fost verificat, **se scriu relațiile de recurență corespunzătoare** (în care d_i se exprimă în funcție de d_{i+1},…, d_n sau d_1, … ,d_{i-1}, după forma în care a fost verificat principiul de optimalitate).

- **Exemplul 1. Şirul lui Fibonacci**

Ştim că acest şir este definit astfel:

$F_0=0;\ F_1=1;\ F_n = F_{n-1} + F_{n-2}\ ,\ \forall n\ge 2$

Dorim să calculăm F_n pentru un n oarecare.

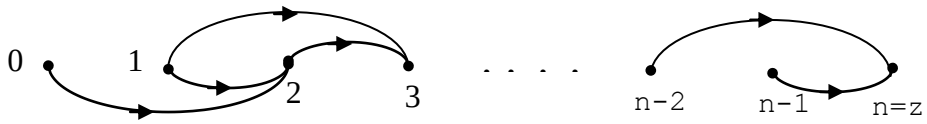
Aici $A=\{0,\ldots,n\}$, $X=\{0,1\}$, $B=\mathbf{N}$, iar

$A_k=\{k-1,k-2\},\ \forall k\ge 2$

$v(k)=F_k\ ;\ f_k(a,b)=a+b,\ \forall k\ge 2$

Avem deci $v(k)= v(k-1)+ v(k-2)$

Un prim graf de dependenţe este următorul:



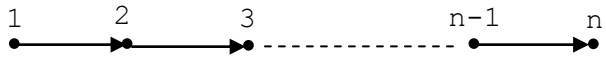
Să observăm că **o mai bună alegere a mulţimii B simplifică structura PD-arborelui.**

$A=\{1,2,\ldots,n\};\ B=\mathbf{N}\times\mathbf{N};$

$v(k)=(F_{k-1},\ F_k);\ \ f_k(a,b)=(b,a+b)$

$v(1)=(0,1).$

Avem atunci $A_k=\{k-1\},\ \forall k\ge 2$ şi $v(k)=f_k(v(k-1))$



şi obţinem algoritmul binecunoscut:

```
a←0; b←1
for i=2,n
   (a,b)←(b,a+b)
write(b)
```

- **Exemplul 2.** ***Determinarea subşirului crescător de lungime maximă (Longest Increasing Subsequence)***

Se consideră vectorul $a=(a_1,\ldots,a_n)$. Se cere să se determine lungimea celui mai lung subşir crescător şi un astfel de subşir (crescător de lungime maximă)

Exemplu. Pentru **a = (8, 1, 7, 4, 6, 5, 11)** lungimea maximă este **4**, un subşir fiind **1, 4, 6, 11**

- Înrudită cu problema determinării celui mai lung subşir comun a două şiruri (Longest Common Subsequence)

- **Aplicații:** cautarea de tiparuri (patterns): baze de date mari, bioinformatică -similitudini în genetică (ADN), alinierea secvențelor (sequence alignment)

Lavanya, B., Murugan, A.: *Discovery of longest increasing subsequences and its variants using DNA operations*. International Journal of Engineering and Technology 5(2), 1169-1177 (2013)

Varianta 1.

Observație (principiu de optimalitate): Dacă a_{i1}, a_{i2}, …, a_{ip}, este un subșir optim (crescător de lungime maximă) care începe pe poziția i1, atunci a_{ik}, …, a_{ip} este un subșir optim care începe pe poziția ik.

Subproblemă. Calculăm pentru fiecare poziție i lungimea maximă a subșirului crescător ce **începe** pe poziția i (cu elementul a_i)

Introducem notațiile:

nr = lungimea maximă căutată;

lung[i]= lungimea maximă a subșirului crescător ce începe pe poziția i (cu a_i)

Atunci **nr = max{lung[i] | i=1,2,…,n}**

Știm direct: lung[n] = 1.

Are loc relația de recurență (formula de calcul):

lung[i]= 1+max{lung[j] | j>i , $a_i<a_j$}

Convenție: max ∅ = 0

Avem atunci o problemă de tipul celei generale, prezentate la începutul cursului:

A={1,2,...,n}; X={n};

A_i={i+1,...,n} și f_i=lung[i], ∀i<n;

Evident, suntem în prezența unui PD-arbore de rădăcină 1.

Ordinea de parcurgere a grafului de dependențe (ordinea de calcul):

i = n, n - 1, … , 1.

Pentru a determina și **un subșir optim** (!doar un subșir), memorăm în plus

succ[i]= indicele următorului element dintr-un subșir optim care începe pe poziția i

Exemplu pas cu pas - v. slideuri

Implementare Java – SubsirCrescator.java (pentru o înțelegere mai ușoară vectorul are primul element pe poziția 1)

Complexitate $O(n^2)$

Observații.

1. Determinarea tuturor subșirurilor crescătoare de lungime maximă se face printr-un backtracking (vom reveni)
2. Există o implementare O(n log n) a acestei probleme

Varianta 2: Calculăm pentru fiecare poziție i lungimea maximă a subșirului crescător ce **se termină** pe poziția i.

Soluția este similară celei pentru varianta 1. PD-arborele va avea rădăcina n, iar ordinea de parcurgere va fi : i = 1, 2, … , n. Pentru a **memora o soluție se va folosi un vector de predecesori:** pred[i]= indicele elementului anterior lui i dintr-un subșir optim care se **termină** pe poziția i

Exemplu pas cu pas - v. slideuri

Varianta 3 O(n log n) - Indicații

Pentru fiecare lungime j=1..,n reținem

m[j] = poziția **celui mai mic element din șir** cu proprietatea că există un **subșir de lungime j care se termină cu el**

- **a[m[1]] ≤ a[m[2]] ≤ … ≤ a[m[n]]**
- La pasul i – căutăm binar cea mai mare lungime j cu **a[m[j]] ≤ a[i]**

Varianta 4 O(n log n) – v. laborator - Patience Sort (Varianta 1 - problema 3)

Exemplu pas cu pas - v. slideuri

- **Exemplul 3.** ***Triunghi***

Se consideră un triunghi de numere naturale cu n linii. Să se determine cea mai mare sumă pe care o putem forma dacă ne deplasăm în triunghi și adunăm numerele din celulele de pe traseu, regulile de deplasare fiind următoarele: pornim de la numărul de pe prima linie; la un pas ne putem deplasa pe linia următoare, dedesubt sau imediat în dreapta poziției anterioare (din (i,j) putem merge doar în (i+1,j) sau (i+1,j+1)).

Observații.

1. Se pot construi în total 2^{n-1} astfel de trasee (pentru fiecare linie de la 2 la n avem câte două opțiuni).
2. O strategie de tip Greedy (ne deplasăm mereu în celula cu cel mai mare număr) nu este optimă. Exemplu:

```
1
6 2
1 2 10
```

Principiu de optimalitate: Dacă $(i_1, j_1), (i_2, j_2), \dots, (i_n, j_n)$ este un traseu optim care începe din celula $(i_1, j_1) = (1,1)$, atunci subșirul $(i_k, j_k), \dots, (i_n, j_n)$ este un traseu optim dacă pornim din celula (i_k, j_k).

Astfel, vom calcula pentru o poziție (i, j) suma maximă pe care o putem obține dacă **pornim din celula (i, j).** Soluția va fi valoarea obținută pentru poziția (1, 1) **(!!!ideea este asemănătoare cu cea de la subșirul crescător de lungime maximă)**

Notăm `s[i][j]`= suma maximă pe care o putem obține dacă **pornim din celula (i, j)**

Știm direct: `s[n][j] = t[n][j]` , pentru j=1,2,…,n (mulțimea elementelor cunoscute X este formată din elementele de pe ultima linie - (n, j), de pe care nu ne mai putem deplasa)

Formulele de calcul (relația de recurență):

```
s[i][j] = t[i][j]+max{s[i+1][j],s[i+1][j+1]}
```

((i,j) depinde de (i+1,j), (i+1,j+1), deci $A_{(i,j)}$ = {(i+1,j), (i+1,j+1)})

Soluția: `s[1][1]`

Ordinea de parcurgere (de calcul): de la ultima linie către prima; fiecare linie de la prima coloană la ultima

Reconstituirea unei soluții: memorăm coloana pe care ne deplasăm într-o nouă matrice u, cu semnificația

`u[i][j]` = coloana pe care ne deplasăm din celula (i, j) pe linia i+1 într-un traseu optim

sau reconstituim traseul folosind relația de recurență (ne deplasăm mereu în celula de pe linia următoare cu s maxim)

Exemplu pas cu pas - v. slideuri

Implementare Java – Triunghi.java.

Complexitate $O(n^2)$

Varianta 2: **Principiu de optimalitate**: Dacă (i_1, j_1), (i_2, j_2), … ,(i_n, j_n) este un traseu optim care începe din celula $(i_1, j_1) = (1,1)$, atunci subșirul (i_1, j_1), …,(i_k, j_k) este un traseu optim care începe din celula $(i_1, j_1) = (1,1)$ şi se termina cu celula (i_k, j_k)

Subproblemă: `s[i][j]`= suma maximă pe care o putem obține dacă **pornim din (1,1) şi ajungem în celula (i, j)**

Soluția: max{`s[n][j] | j=1,…,n`}

- **Exemplul 4.** ***Înmulţirea optimă a unui şir de matrice***

Avem de calculat produsul de matrice $A_1 \times A_2 \times \ldots \times A_n$, unde dimensiunile matricelor sunt respectiv $(d_1,d_2),(d_2,d_3),\ldots,(d_n,d_{n+1})$. Ştiind că înmulţirea matricelor este asociativă, se pune problema ordinii în care trebuie înmulţite matricele astfel încât numărul de înmulţiri elementare să fie minim.

Presupunem că înmulţirea a două matrice se face în modul uzual, adică produsul matricelor `A(m,n)` şi `B(n,p)` necesită **`m×n×p` înmulţiri elementare**.

Pentru a pune în evidenţă importanţa ordinii de înmulţire, să considerăm produsul de matrice $A_1 \times A_2 \times A_3 \times A_4$ unde $A_1(100,1)$, $A_2(1,100)$, $A_3(100,1)$, $A_4(1,100)$. Pentru ordinea de înmulţire $(A_1 \times A_2) \times (A_3 \times A_4)$ sunt necesare 1.020.000 de înmulţiri elementare. În schimb, pentru ordinea de înmulţire $(A_1 \times (A_2 \times A_3)) \times A_4$ sunt necesare doar 10.200 de înmulţiri elementare.

Principiu de optimalitate $(A_i \times \ldots \times A_k) \times (A_{k+1} \times \ldots \times A_j)$ optim $\Rightarrow$ $A_i \times \ldots \times A_k$ şi $A_{k+1} \times \ldots \times A_j$ optim

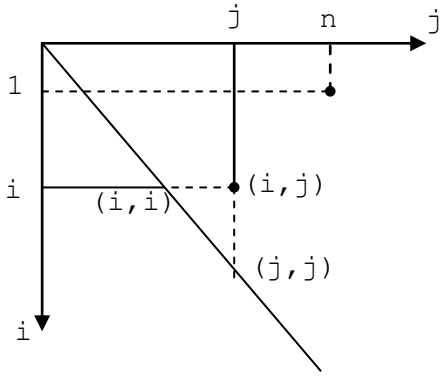
Subproblemă Fie `cost[i][j]` numărul minim de înmulţiri elementare pentru calculul produsului $A_i \times \ldots \times A_j$. Punând în evidenţă ultima înmulţire de matrice, obţinem relaţiile:

`cost[i][i] = 0, ∀i=1,2,...,n`

`cost[i][j] = min {cost[i][k]+cost[k+1][j]+`$\mathbf{d_i \times d_{k+1} \times d_{j+1}}$` | i≤k<j}.`

Valoarea cerută este `cost[1][n]`.

Vârfurile grafului de dependenţă sunt perechile `(i,j)` cu `i≤j`. Valoarea `cost[i][j]` depinde de valorile vârfurilor din stânga şi de cele ale vârfurilor de dedesubt. Se observă uşor că suntem în prezenţa unui PD-arbore.



Din forma particulară a PD-arborelui se observă că trebuie să parcurgem matricea astfel încât valorile `cost[i][k]`,`cost[k+1][j]`să fie calculate înainte de `cost[i][j]`.

Varianta 1. Parcurgem în ordine coloanele 2,...,n, iar pe fiecare coloană j să mergem în sus de la diagonală până la prima linie.

```
for (j=2; j<=n; j++)
  for (i=j-1; i>=1; i--) {
     calculeaza cost[i][j] dupa relatia de recurenta
     fie k valoarea pentru care se realizează minimul
     cost[j][i]=k
  }
scrie cost[1][n]
```

(**am folosit partea inferior triunghiulară a matricei pentru a memora indicii pentru care se realizează minimul**).

Dacă dorim să afişăm o ordine de înmulţire optimă, vom apela `sol(1,n)`, unde procedura `sol` are forma:

```
void sol(int p,int u){
   if (p==u)
       System.out.print(p);
   else { int k = cost[u][p];
          System.out.print ('(');
          sol(p,k);
          System.out.print (',');
          sol(k+1,u);
         System.out.print(')');
        }
}
```

Varianta 2. Parcurgem indicii în ordinea modulului diferenței `(i - j)` (paralel cu diagonala principală).

```
for (dif=1; dif<=n-1; dif++)
  for (i=1; i<=n-dif; i++){
       j = i+dif
       calculeaza cost[i][j] dupa relatia de recurenta
       fie k valoarea pentru care se realizează minimul
       cost[j][i]=k
   }
scrie cost[1][n]
```

Implementare Java – InmultireMatr.java

Numărul de comparări efectuate este:

$$\sum_{j=2}^{n}\sum_{i=1}^{j-1}(j-i+1)=\sum_{j=2}^{n}\left[j(j-1)-\frac{(j-1)(j-2)}{2}\right]=O(n^3)$$

- **Exemplul 5.** ***Problema discretă a rucsacului***

Se consideră un rucsac de capacitate (greutate) maximă G şi n obiecte caracterizate prin:

- greutăţile lor **(!!!numere naturale)** $g_1, \ldots, g_n$;
- câştigurile $v_1, \ldots, v_n$ obţinute la încărcarea lor în totalitate în rucsac.

Un obiect nu poate fi fracționat.

Se cere o modalitate de încărcare de obiecte în rucsac, astfel încât câştigul total să fie maxim.

Observaţie: Aplicarea metodei Greedy eşuează în acest caz. Într-adevăr, aplicarea ei pentru: `G=5, n=3 şi g=(4,3,2), v=(6,4,2.5)`
are ca rezultat încărcarea primul obiect; câştigul obţinut este 6. Dar încărcarea ultimelor două obiecte conduce la câştigul superior 6.5.

Principiu de optimalitate: Fie O o soluţie optimă pentru obiectele `{1,...,n}` şi greutatea G. Avem situaţiile

- dacă $n \in O$, atunci $O - \{n\}$ este soluţie optimă pentru obiectele `{1,...,n-1}` şi greutatea $G - g_n$
- dacă $n \notin O$, atunci O este soluţie optimă pentru obiectele `{1,...,n-1}` şi greutatea G

Subprobleme: Notăm `c[i][g]` câştigul maxim pentru greutatea `g` şi obiectele `{1,...,i}`

Avem **relaţiile de recurenţă**:

$$c[i][g]=\begin{cases} 0, \text{ daca } i = 0 \text{ sau } g = 0 \\ c[i-1][g], \text{ daca } g_i > g \\ \max\{v_i + c[i-1][g-g_i], c[i-1][g]\}, \text{ altfel} \end{cases}$$

pentru `i = 0,...,n, g = 0,...,G`

Suntem evident în prezența unui PD-arbore cu rădăcina (n, G)

Valorile cunoscute inițial sunt: $X=\{ (i, 0), 0 \le i \le n\} \{ (0, g) \mid 0 \le g \le G \}$

(i,g) depinde de (i-1, g - g_i) și (i-1,g), deci de perechi cu prima componentă i – 1 și cealaltă componentă mai mică sau egală cu g

Soluția: `c[n][G]`

Ordinea de calcul: `i=1,…,n, g=1,…,G`

Complexitate: `O(nG)`

Afișarea obiectelor care conduc la costul optim: din relația de recurență, ca la triunghi sau cu o matrice `luat[i][g]` cu valori 1 sau 0, după cum obiectul i aparține soluției optime pentru greutatea rucsacului g

Implementarea în Java – Rucsac.java

Alte aplicații

- **Distanțe de editare. Alinierea secvențelor (Bioinformatică)**
  Putem măsura similaritatea între secvențe (ADN) prin
  - **Elemente comune –** cel mai lung subșir comun pentru două secvențe
  - **Distanțe de editare –** numărul minim de inserări și modificări (eventual și stergeri) de caractere necesar pentru transforma prima secvență în cea de a doua

  (aplicații și în procese de căutare de cuvinte – sugestii de cuvinte similare)
- **Testarea apartenenţei unui cuvânt la un limbaj generat de o gramatică independentă context în formă Chomsky** - v. H.Georgescu – Tehnici de programare

Alinierea secvențelor – Exemplu

Aliniere = punerea poziţiilor (caracterelor) din cele două secvenţe a şi b în corespondenţă 1 la 1, cu posibilitatea de a insera spaţii.

Date două secvențe, aliniem secvențele inserând în ele caracterul “ ” astfel încât secvențele să devină de aceeași lungime și penalizând pozițiile pe care diferă secvențele obținute. Scorul (penalizarea) alinierii obţinute este dat de suma penalizărilor alinierilor de caractere diferite şi alinierilor caracter-spaţiu (scorul Needleman-Wunsch).

Exemplu: ADN – alfabet A,C,G,T şi cuvintele AGGGCT, AGGCA

```
AGGGCT
AGG-CA
```

penalizarea totala = penalizare spatiu + penalizare pentru diferenta T/A

Indicaţie: Subproblemă c[i][j] = scorul alinierii prefixului format din primele i litere din a cu primele j litere din b