Metoda programării dinamice

Observații.

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

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 $v(x) \in B$.

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

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

$$\mathbf{v(x)=f_x(v(a_1),...,v(a_k))}$$

unde $A_x=\{a_1,...,a_k\} \subset A$ 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, un maxim, o sumă etc).

Se mai dă $z \in A$.

Se cere să se calculeze, **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:

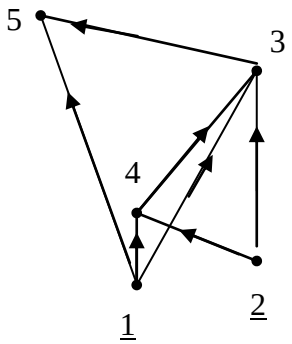
```
A = {1,2,...,5};
v(1) = v(2)
v(3) = v(1)+v(2)+v(4)
v(4) = v(1) + v(2)
v(5)  = v(1) + v(3)
```

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

Fiecare funcţie f_x calculează $v(x)$ ca fiind suma valorilor elementelor din A_x. Alegem $z=5$. O ordine posibilă de a considera elementele lui $A \setminus X$ astfel încât să putem calcula valoarea asociată lor este: $4,3,5$. Astfel:

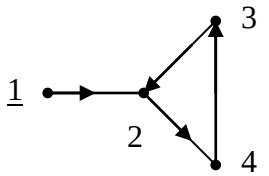
```
v(4) = v(1) + v(2) = 2,
v(3) = v(4) + v(1) + v(2) = 4,
v(5) = v(1) + v(3) = 5
```

Să observăm că dacă dorim să calculăm v(4) 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.**

Dezavantaje:

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

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

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

➢ **Soluție: 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

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 subgraful G_z al vârfurilor accesibile din z (numit **PD-arbore de rădăcină z)**; acest graf are proprietățile:
   - nu conţine cicluri
   - mulțimea vârfurilor cu gradul intern 0 (i.e. ale căror valori nu depind de nicio altă valoare) este inclusă în mulțimea 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∈Ax
      if vizitat[j]=false    {diferența față de Divide et Impera}
           postord(j)
   calculează v(x) conform funcţiei fx;
   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** 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 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, iar șirul d_{k+1}, ..., d_n duce sistemul în mod optim din S_k în S_n

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

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:

$F_0=0;\ F_1=1;$

$F_n = F_{n-1} + F_{n-2}\ ,\ \forall n\geq 2$

Dorim să calculăm F_n pentru un n oarecare.

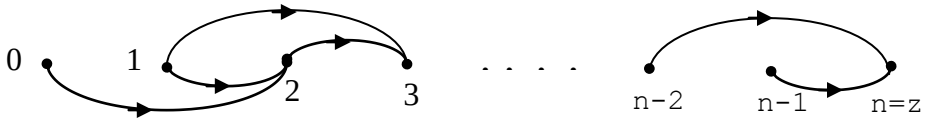
Aici $A=\{0,...,n\}$, $X=\{0,1\}$, $B=\mathbf{N}$, iar

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

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

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

Un prim graf de dependenţe este următorul:



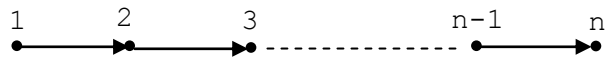
Să observăm că **o mai bună alegere a mulţimii B simplifică structura PD-arborelui.**

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

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

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

Avem atunci $A_k=\{k-1\},\ \forall k\geq 2$ 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***

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

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.

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

Introducem notaţiile:

nr = lungimea maximă căutată;

lung[i]= lungimea maximă a subşirului 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):

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

Convenție: max ∅ = 0

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

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

$\mathbf{A_i}$**={i+1,...,n}** şi f_i=lung[i], ∀i<n;

Evident, suntem în prezenţa 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

```
nr = 1;
poz = n; //poz de inceput a sirului maxim
lung[n] = 1;     succ[n] = n+1; //stim
for (int i=n-1;i>=1;i--){//ordine de calcul
   succ[i]= n+1; lung[i]=1;
   //formula de recurenta
      for(int j=i+1;j<=n; j++){
        if((a[i]<a[j]) && (1+lung[j]>lung[i])){
             lung[i]= 1 + lung[j];
              succ[i] = j;
        }
      }
      if(lung[i]> nr){nr = lung[i]; poz = i;}
}
```

Afișare unei soluții optime se face astfel

```
for (int i=1;i<=nr;i++){
      System.out.print(a[poz]+" ");
     poz = succ[poz];
}
```

Complexitate $O(n^2)$

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

Observaţii.

1. Determinarea tuturor subşirurilor crescătoare de lungime maximă se face printr-un backtracking (vom reveni)
2. Există o implementare O(n log n) a acestei probleme – **Cătălin Frâncu, Psihologia concursurilor de informatică** http://www.infobits.ro/psihologia-concursurilor-de-informatica.php

Varianta 2: Calculăm pentru fiecare poziție i lungimea maximă a subşirului 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

- **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 şi 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 $(i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n)$ este un traseu optim care începe din celula $(i_1, j_1) = (1,1)$, atunci subșirul $(i_k, j_k), \ldots, (i_n, j_n)$ 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)

Implementare Java – Triunghi.java

Complexitate $O(n^2)$

- **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 **$m \times n \times p$ 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

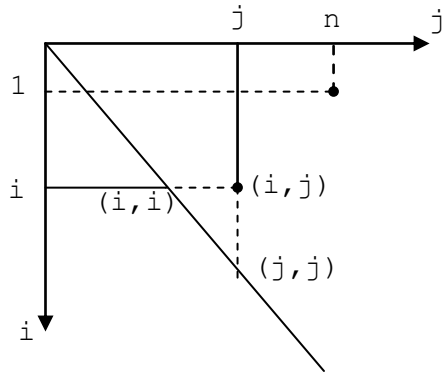
Subproblemă Fie `cost[i][j]` numărul minim de înmulţiri elementare pentru calculul produsului $A_i \times \ldots \times A_j$. Punând în evidenţă ultima înmulţire de matrice, obţinem relaţiile:

`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.



Forma particulară a PD-arborelui nu face necesară aplicarea algoritmului general de parcurgere în postordine, este suficient să ne asigurăm că valorile `cost[i][k],cost[k+1][j]` sunt 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

Pentru evaluarea timpului de lucru, vom calcula numărul de comparări efectuate. Aceste 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.** ***Descompunerea unui dreptunghi în pătrate***

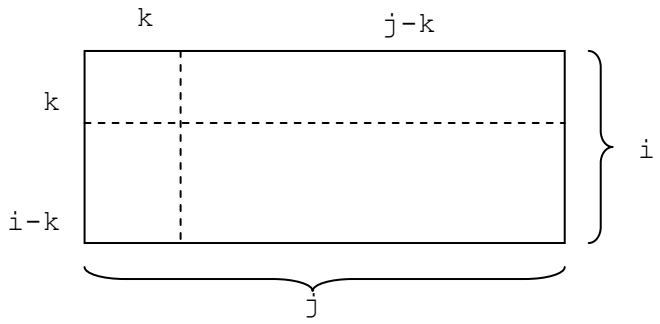
Se consideră un dreptunghi cu laturile de `m`, respectiv `n` unităţi (`m<n`). Asupra sa se pot face tăieturi *complete* pe orizontală sau verticală. Se cere numărul minim de pătrate în care poate fi descompus dreptunghiul.

Observaţie: O soluţie de tip Greedy (să tăiem la fiecare pas un pătrat de latură maximă) nu este optimă. De exemplu, dacă avem un dreptunghi 5 × 6, soluţia Greedy va fi: un pătrat de latură 5 şi 6 pătrate de latură 1. Există însă o soluţie mai bună: două pătrare de latură 3 şi 3 pătrate de latură 2.

Fie `a[i][j]` = numărul minim de pătrate în care poate fi descompus un dreptunghi de laturi `i` şi `j`. Evident `a[i][j]=a[j][i]`. Rezultatul căutat este `a[m][n]`.
Vârfurile grafului de dependenţe sunt `(i,j)`, iar valorile asociate sunt `a[i][j]`.

Ştim: Dacă i = 1 sau j = 1 sau i = j, avem:

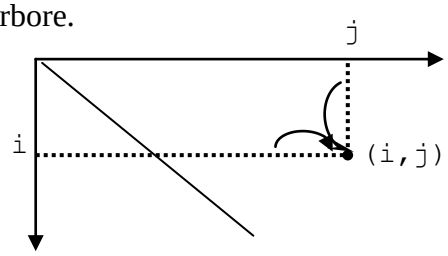
```
a[i][1]=i, ∀i=1,...,m
a[1][j]=j, ∀j=1,...,n
a[i][i]=1, ∀=1,...,m
```



Formula de calcul: Pentru calculul lui `a[i][j]` avem de ales între a face:

- o tăietură pe verticală; costurile sunt: `a[i][k]+a[i][j-k]`, `k≤ ⌊j/2⌋` ;
- o tăietură pe orizontală; costurile sunt: `a[k][j]+a[i-k][j]`, `k≤ ⌊i/2⌋`.

Rezultă că valoarea `a[i][j]` a unui vârf `(i,j)` depinde de valorile vârfurilor din stânga sa şi de cele aflate deasupra sa. Se observă că graful de dependenţe este un PD-arbore.



Dependenţele pot fi exprimate astfel:

`a[i][j]` = min{α, β}, unde

α=min{`a[i][k]+a[i][j-k]` | k≤ ⌊`j/2`⌋} ,

iar β=min{`a[k][j]+a[i-k][j]` | k≤ ⌊`i/2`⌋ }.

Ordinea de parcurgere: Forma particulară a PD-arborelui permite o parcurgere mai uşoară decât aplicarea algoritmului general de postordine. De exemplu putem coborî pe linii, iar pe fiecare linie mergem de la stânga la dreapta.

După iniţializările date de primele trei dependenţe de mai sus, efectuăm calculele:

```
for (i=2; i<=m; i++)
   for (j=i+1; j<=n;j++)
      calculul lui a[i][j] conform formulei de mai sus
      if (j<=m)
            a[j][i] = a[i][j]
```

Observaţie. Am lucrat numai pe partea superior triunghiulară, cu actualizări dedesubt.

- **Exemplul 6.** ***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 fracţionat.

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: Dacă $\{i_1, i_2, \ldots, i_k\}$ este soluție optimă pentru greutatea g și obiectele $\{1,2,\ldots,i_k\}$, atunci $\{i_1, i_2, \ldots, i_{k-1}\}$ este soluție optimă pentru greutatea $g - g_{ik}$ și obiectele $\{1,2,\ldots,i_k-1\}$

Notăm `c[i][g]` câştigul maxim pentru greutatea `g` şi obiectele $\{1, \ldots, i\}$

Avem **relațiile**:

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

pentru $i = 0,\ldots,n, g = 0,\ldots,G$

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

Valorile cunoscute inițial sunt: X={ (i, 0), 0 ≤ i ≤ n} { (0, g) | 0 ≤ g ≤ 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

- **Exemplul 7.** ***Problema monedelor***

Având la dispoziție un număr nelimitat de monede de valori $\{v_1, v_2, \ldots, v_n\}$, dorim să plătim o sumă de bani S folosind un număr minim de monede (știind că este posibil).

O *soluţie* este un vector $x=(x_1,\ldots,x_n)$ cu $\begin{cases}x_i \geq 0, \forall i\\ \sum_{i=1}^{n} v_i x_i = S\end{cases}$

O *soluţie optimă* este soluţie care minimizează funcţia $f(x)=\sum_{i=1}^{n} x_i$.

O **strategie Greedy** de rezolvare a acestei probleme este următoarea: considerăm moneda cu cea mai mare valoare și plătim o sumă cât mai mare din S cu ajutorul acesteia. Aplicăm aceeași strategie pentru suma rămasă și restul tipurilor de monede rămase, până se acoperă toată suma.

Evident, **strategia nu conduce la o soluție optimă pentru orice instanță a problemei**. De exemplu, pentru monedele `{15, 8, 6, 4, 1}` și `S= 18`, algoritmul va afișa soluția `1 x 15 + 3 x 1`, care folosește 4 monede, în timp ce soluția optimă necesită doar 3 monede: `1 x 8 + 1 x 6 + 1 x 4`.

Există însă cazuri particulare ale acestei probleme, în care proprietățile pe care le au valorile monedelor asigură faptul că strategia Greedy furnizează soluția optimă.

Spre exemplu, să presupunem că valorile satisfac proprietățile:

$v_1 \geq v_2 \geq ... \geq v_n = 1$ **și**

v_{i-1} **este multiplu al lui** v_i**, pentru orice** $i \in \{2,3,...,n\}$

Corectitudinea algoritmului în acest caz poate fi demonstrată prin inducţie, ţinând cont de următoarele observații.

Fie $(x_1, x_2, \ldots, x_n)$ soluția construită cu strategia Greedy și $O = (o_1, o_2, \ldots, o_n)$ o soluție optimă

a) $o_1 = x_1$. (se demonstrează prin reducere a absurd)

b) $(o_2, \ldots, o_n)$ este soluție optimă pentru problema acoperirii sumei

$S - x_1v_1 = S - o_1v_1$.

Pentru valori ale sumei S suficient de mici (cât să permită crearea unui vector de dimensiune S), o soluție pentru problemă folosind tehnica programării dinamice este următoarea.

Principiul de optimalitate: Dacă ultima monedă pe care o folosim pentru plata optimă a unei sume s este v_i, atunci restul monedelor folosite pentru această plată optimă constituie o soluție optimă pentru $s - v_i$.

Subproblemă: Calculăm nr[s] = numărul minim de monede necesare pentru a plăti o sumă $s \leq S$.

Au loc relațiile:

$nr[0] = 0$

$nr[s] = \min\{1 + nr[s - v_i],\ 1 \leq i \leq n,\ v_i \leq s\}$

Soluția: nr[S]

PD-arbore de rădăcină S. Elementul s depinde de cele mai mici decât el.

Odinea de calcul: s = 1, …, S

Pentru a reține și o modalitate optimă de plată folosim un vector cu semnificația moneda[s] = indicele i pentru care se realizează minimul din formula pentru nr[s] (asemănător cu subșirul crescător de lungime maximă).

```
nr[0] = 0; moneda[0] = -1;
for (s = 1; s<=S; s++){
      nr[s] = ∞; moneda[s] = -1;
      for (i=1; i<=n; i++)
          if (v_i ≤ s && nr[s - v_i] + 1 < nr[s]){
                nr[s] = nr[s - v_i] + 1; moneda[s] = i;
          }
}
scrie nr[S]
```

Afișarea unei soluții

```
if (nr[S] < ∞){
    s = S
    for (i = 1; i<= nr[S]; i++){
         scrie v[moneda[s]]
         s = s – v[ moneda[s]]
    }
}
```

Complexitate: O(nS)

Implementarea în Java – Monede.java

Exemplu: {1,4,5} și S= 12

```
nr pentru s de la 0 la 12 (indicii in nr de la 0 la 12)
0 0 0 0 0 0 0 0 0 0 0 0 0 (s=0)
0 1 0 0 0 0 0 0 0 0 0 0 0 (s=1) moneda[1]=1
0 1 2 0 0 0 0 0 0 0 0 0 0 (s=2) moneda[2]=1
0 1 2 3 0 0 0 0 0 0 0 0 0 (s=3) moneda[3]=1
0 1 2 3 1 0 0 0 0 0 0 0 0 (s=4) moneda[4]=4
0 1 2 3 1 1 0 0 0 0 0 0 0 (s=5) moneda[5]=1
0 1 2 3 1 1 2 0 0 0 0 0 0 (s=6) moneda[6]=1
0 1 2 3 1 1 2 3 0 0 0 0 0 (s=7) moneda[7]=1
0 1 2 3 1 1 2 3 2 0 0 0 0 (s=8) moneda[8]=4
0 1 2 3 1 1 2 3 2 2 0 0 0 (s=9) moneda[9]=4
0 1 2 3 1 1 2 3 2 2 2 0 0 (s=10) moneda[10]=5
0 1 2 3 1 1 2 3 2 2 2 3 0 (s=11) moneda[11]=1
0 1 2 3 1 1 2 3 2 2 2 3 3 (s=12) moneda[12]=4
```