Metoda programării dinamice

Vom începe prin a enunţa o problemă generală şi a trece în revistă mai mulţi algoritmi de rezolvare. Abia după aceea vom descrie metoda programării dinamice.

• O problemă generală

Fie A şi B două mulţimi oarecare.
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 $x \in A \setminus X$ sunt cunoscute:

$A_x \subset A$: mulţimea elementelor din A de a căror valoare depinde $v(x)$;

f_x : funcţie care specifică dependenţa de mai sus. Dacă $A_x=\{a_1,\ldots,a_k\}$, atunci $v(x)=f_x(v(a_1),\ldots,v(a_k))$.

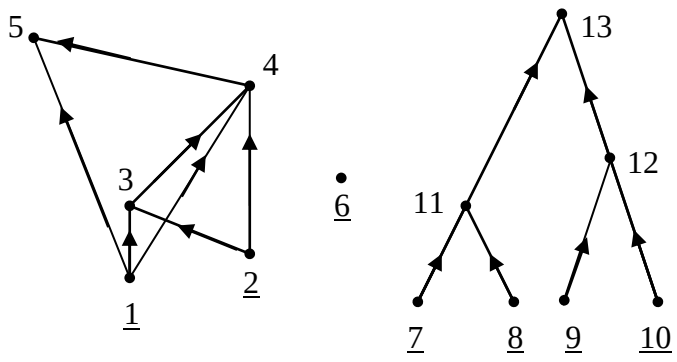
Se mai dă $z \in A$.
Se cere să se calculeze, dacă este posibil, valoarea $v(z)$.

Exemplu.

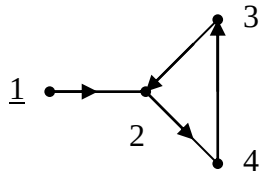
$A=\{1,2,\ldots,13\}$; $X=\{1,2,6,7,8,9,10\}$;
$A_3=\{1,2\}$; $A_4=\{1,2,3\}$; $A_5=\{1,4\}$;
$A_{11}=\{7,8\}$; $A_{12}=\{9,10\}$; $A_{13}=\{11,12\}$.

Elementele din X au asociată valoarea 1.
Fiecare funcţie f_x calculează $v(x)$ ca fiind suma valorilor elementelor din A_x. Alegem $z=5$.
Este evident că vom obţine $v=(1,1,2,4,5,1,1,1,1,1,2,2,4)$. O ordine posibilă de a considera elementele lui $A \setminus X$ astfel încât să putem calcula valoarea asociată lor este: $3,11,12,13,4,5$.

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.



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 $z=3$.



Observaţii:

- A poate fi chiar infinită;
- B este de obicei $\mathbf{N}$, $\mathbf{Z}$, $\mathbf{R}$, $\{0,1\}$ sau un produs cartezian;
- f_x poate fi un minim, un maxim, o sumă etc.

Pentru orice $x \in A$, spunem că x este *accesibil* dacă, plecând de la X, poate fi calculată valoarea $v(x)$. Evident, problema are soluţie dacă şi numai dacă z este accesibil.

Pentru orice $x \in A$, notăm prin O_x *mulţimea vârfurilor observabile din* x, adică mulţimea vârfurilor y pentru care există un drum de la y la x. Problema enunţată are soluţie dacă şi numai dacă:

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

Prezentăm în continuare mai multe metode/încercări de rezolvare a problemei enunţate.

• Metoda şirului crescător de mulţimi

Fie A o mulţime finită şi X o submulţime a sa. Definim următorul şir crescător de mulţimi:

$X_0 = X$

$X_{k+1} = X_k \cup \{x \in A \mid A_x \subset X_k\}, \forall k>0$

Evident, $X_0 \subset X_1 \subset \ldots \subset X_k \subset X_{k+1} \subset \ldots \subset A$.

Propoziţie. Dacă $X_{k+1}=X_k$, atunci $X_{k+i}=X_k$,$\forall i \in \mathbf{N}$.

Facem demonstraţia prin inducţie după i.

Pentru $i=1$ rezultatul este evident.

Presupunem $X_{k+i}=X_k$ şi demonstrăm că $X_{k+i+1}=X_k$:

X_{k+i+1} = cf. definiţiei şirului de mulţimi

$= X_{k+i} \cup \{x \in A \mid A_x \subset X_{k+i}\}$ = cf. ipotezei de inducţie

$= X_k \cup \{x \in A \mid A_x \subset X_k\}$ = cf. definiţiei şirului de mulţimi

$= X_{k+1}$ = cf. ipotezei

$= X_k$.

Consecinţe.

- ne oprim cu construcţia şirului crescător de mulţimi la primul k cu $X_k=X_{k+1}$ (A este finită!);
- dacă aplicăm cele de mai sus pentru problema generală enunţată, aceasta are soluţie dacă şi numai dacă $z \in X_k$.

Prezentăm în continuare algoritmul corespunzător acestei metode, adaptat la problema generală.

Vom lucra cu o partiţie $A=U \cup V$, unde U este mulţimea curentă de vârfuri a căror valoare asociată este cunoscută.

```
U ← X; V ← A\X
repeat
   b ← false
   for toţi x∈V
      if Ax⊂U
      then U ← U ∪{x}; V ←V\{x}; b ← true
           calculează v(x) conform funcţiei fx
           if x=z
           then write v(x); stop
until not b { nici un element din V nu a trecut în U }
write(z, 'nu este accesibil')
```

Metoda descrisă are două deficienţe majore:

- la fiecare reluare se parcurg toate elementele lui V;
- nu este precizată o ordine de considerare a elementelor lui V.

Aceste deficienţe fac ca această metodă să nu fie performantă.

Metoda şirului crescător de mulţimi este larg folosită în teoria limbajelor formale, unde de cele mai multe ori ne interesează existenţa unui algoritm şi nu performanţele sale.

• Sortarea topologică

Fie $A=\{1,...,n\}$ o mulţime finită. Pe A este dată o relaţie tranzitivă, notată prin "<". Relaţia este dată prin mulţimea perechilor (i,j) cu $i<j$.

Se cere să se listeze elementele $1,..,n$ ale mulţimii într-o ordine ce satisface cerinţa: dacă $i<j$, atunci i apare la ieşire înaintea lui j.

Problema enunţată apare, de exemplu, la înscrierea unor termeni într-un dicţionar astfel încât explicaţiile pentru orice termen să conţină numai termeni ce apar anterior.

Este evident că problema se transpune imediat la grafurile de dependenţă: se cere o parcurgere a vârfurilor grafului astfel încât dacă există un arc de la i la j, atunci i trebuie vizitat înaintea lui j.

Observaţii:
- problema are soluţie dacă şi numai dacă graful este aciclic;
- dacă există soluţie, ea nu este neapărat unică.

În esenţă, algoritmul care urmează repetă următorii paşi:
- determină `i` care nu are predecesori;
- îl scrie;
- elimină perechile pentru care sursa este `i`.

Fie `M` mulţimea curentă a vârfurilor care nu au predecesori. Iniţial `M=X`. Mulţimea `M` va fi reperezentată ca o coadă, notată cu *C*.

Pentru fiecare `i∈A`, considerăm:
S_i = lista succesorilor lui `i`;
$nrpred_i$ = numărul predecesorilor lui `i` din mulţimea `M` curentă.

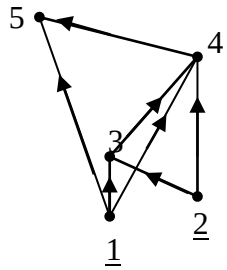
Etapa de iniţializare constă în următoarele:

```
Si ← Ø, nrpredi←0, ∀i
C ← Ø; nr ← 0     { nr este numărul elementelor produse la ieşire }
for k=1,m         { m este numărul perechilor din relaţia "<" }
   read(i,j)
   Si ⇐ j; nrpredj ← nrpredj+1
for i=1,n
   if nrpredi=0
   then i ⇒ C
```

Să observăm că timpul cerut de etapa de iniţializare este de ordinul `O(m+n)`.

Algoritmul propriu-zis, adaptat la problema generală, este următorul:

```
while C ≠ Ø
   i ⇐ C; write(i); nr ← nr+1
   calculează v(i) conform funcţiei fi
   if i=z
   then write(i,v(i)); stop
   for toţi j∈Si
      nrpredj ← nrpredj-1
      if nrpredj=0 then j ⇒ C
if nr<n then write('Nu')
```



i	S_i	$nrpred_i$
1	3 4 5	0
2	3 4	0
3	4	2
4	5	3
5	0	2

C	
out	

Fiecare executare a corpului lui `while` necesită un timp proporţional cu $|S_i|$. Dar $|S_1|+...+|S_n|=m$, ceea ce face ca timpul de executare să fie de ordinul $O(m)$. Ţinând cont şi de etapa de iniţializare, rezultă că timpul total este de ordinul $O(m+n)$, deci liniar.

Totuşi, sortarea topologică aplicată problemei generale prezintă un dezavantaj: sunt calculate şi valori ale unor vârfuri "neinteresante", adică neobservabile din z.

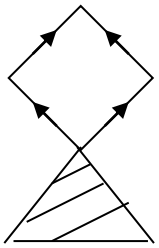
- **Încercare cu metoda Divide et Impera**

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

```
procedure DivImp(x)
    for toţi y∈Ax\X
        DivImp(y)
    calculează v(x) conform funcţiei fx
end;
```

Apare un avantaj: sunt parcurse doar vârfurile din O_z. Dezavantajele sunt însă decisive pentru renunţarea la această încercare:

- algoritmul nu se termină pentru grafuri ciclice;
- valoarea unui vârf poate fi calculată de mai multe ori, ca de exemplu pentru situaţia:



- **Soluţie finală**

Etapele sunt următoarele:

- identificăm G_z = subgraful asociat lui O_z ;
- aplicăm sortarea topologică.

Fie $G_z=(V_z,M_z)$. Iniţial $V_z=\emptyset$, $M_z=\emptyset$.

Pentru a obţine graful G_z executăm apelul `DF(z)`, unde procedura `DF` este:

```
procedure DF(x)
    x ⇒ Vz
    for toţi y∈Ax
        if y∉Vz then (y,x) ⇒ Mz; DF(y)
end;
```

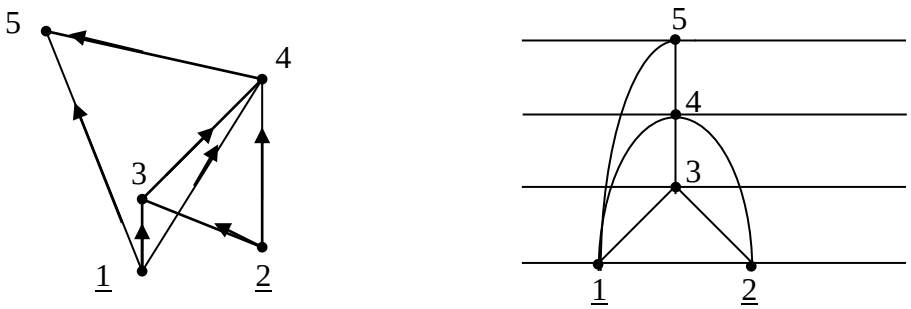
Timpul este liniar.

Observaţie. Ar fi totuşi mai bine dacă :
- am cunoaşte de la început G_z;
- forma grafului ar permite o parcurgere mai simplă.

• Metoda programării dinamice

Definim un *PD–arbore de rădăcină* z ca fiind un graf de dependenţe fără cicluri în care:
- $\forall x$, $x \in O_z$ (pentru orice vârf x există un drum de la x la z);
- $X=\{x \mid grad^-(x)=0\}$ (vârfurile în care nu sosesc arce sunt exact cele din submulţimea X).

Exemplu. Următorul graf este un PD-arbore de rădăcină $z=5$.



Un PD-arbore nu este neapărat un arbore, dar:
- poate fi pus pe niveluri: fiecare vârf x va fi pus pe nivelul egal cu lungimea celui mai lung drum de la x la z, iar sensul arcelor este de la nivelul inferior către cel superior;
- poate fi parcurs (cu mici modificări) în postordine;

Postordinea poate fi generalizată pentru PD-arbori.

Algoritmul de parcurgere în postordine foloseşte un vector `vizitat` pentru a ţine evidenţa vârfurilor vizitate, adică cele pentru care s-a calculat valoarea ataşată. Este iniţializat vectorul `vizitat` şi se începe parcurgerea prin apelul `postord(z)`:

```
for toate vârfurile x∈A
   vizitat(x) ← x∈X
postord(z)
```

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

```
procedure postord(x)
   for toţi j∈Ax cu vizitat(j)=false
      postord(j)
   calculează v(x) conform funcţiei fx; vizitat(x)←true
end
```

Prin parcurgerea în postordine, vârfurile vor fi sortate topologic.
Timpul de executare a algoritmului este evident liniar.

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

1) Se asociază problemei un graf de dependenţe;
2) În graf este pus în evidenţă un PD-arbore; problema se reduce la determinarea valorii asociate lui z (rădăcina arborelui);
3) Se parcurge în postordine PD-arborele.

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.
Vom începe cu câteva exemple, la început foarte simple, dar care pun în evidenţă anumite caracteristici ale metodei programării dinamice.

Exemplul 1. *Şirul lui Fibonacci*

Ştim că acest şir este definit astfel:

$F_0=0;\ F_1=1;$

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

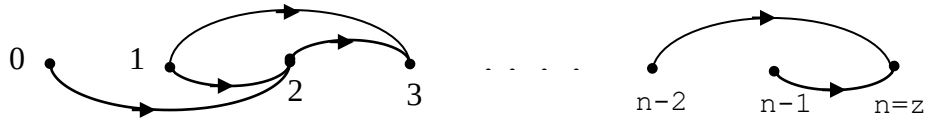
Dorim să calculăm F_n pentru un n oarecare.

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

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

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

Un prim graf de dependenţe este următorul:

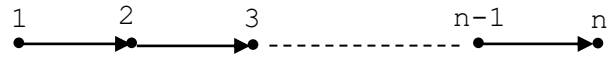


Să observăm că o mai bună alegere a mulţimii B simplifică structura PD-arborelui.

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

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

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



şi obţinem algoritmul binecunoscut:

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

Exemplul 2. *Calculul sumei* $a_1+\ ...+a_n$

Este evident că trebuie calculate anumite sume parţiale.

O primă posibilitate este să considerăm un graf în care fiecare vârf este o submulţime $\{i_1,...,i_k\}$ a lui $\{1,2,...,n\}$, cu valoarea asociată $a_{i_1}+a_{i_2}+...+a_{i_k}$. Această abordare este nerealizabilă: numărul de vârfuri ar fi exponenţial.

O a doua posibilitate este ca vârfurile să corespundă mulţimilor $\{i,i+1,...,j\}$ cu $i\le j$ şi cu valoarea ataşată $s_{i,j}=a_i+...+a_j$. Vom nota un astfel de vârf prin $(i:j)$. Dorim să calculăm valoarea $a_1+...+a_n$ vârfului $z=(1:n)$. Putem considera mai mulţi PD-arbori:

- Arborele liniar constituit din vârfurile cu $i=1$. Obţinem relaţiile de recurenţă:

$s_{1,1}=a_1$;

$s_{1,j}=s_{1,j-1}+a_j$, $\forall j=2,3,...,n$

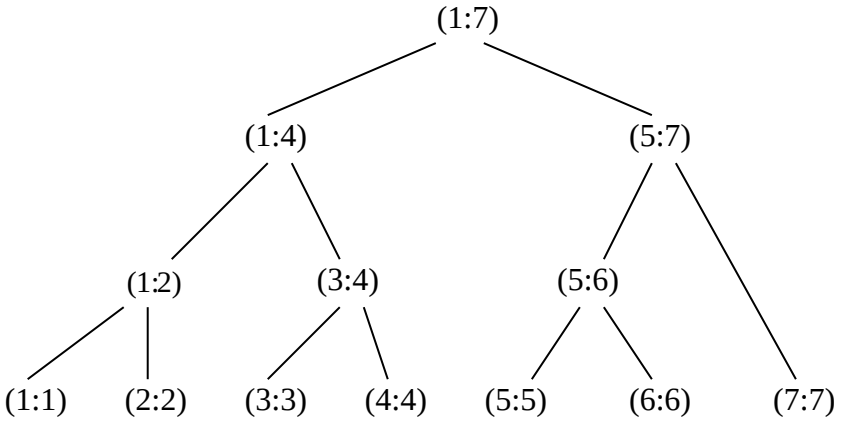
care corespund asociativităţii la stânga: $(...((a_1+a_2)+a_3)+...)$.

- Arborele liniar constituit din vârfurile cu $j=n$. Acest arbore corespunde asociativităţii la dreapta a sumei:

$s_{n,n}=a_n$

$s_{i,n}=a_i+s_{i-1,n}$, $\forall i=n-1,n-2,...1$.

- Arborele binar strict în care fiecare vârf (afară de frunze) are descendenţii $(i:k)$ şi $(k+1:j)$ cu $k=\lfloor (i+j)/2 \rfloor$. Prezentăm acest arbore pentru $n=7$:



Relaţiile de recurenţă sunt:

$s_{ii}=a_i$

$s_{i,j}=s_{ik}+s_{k+1,j}$ pentru $i<j$.

iar algoritmul constă în parcurgerea pe niveluri, de jos în sus, a arborelui; nu este folosit vreun tablou suplimentar:

```
k←1
while k<n
   k2 ← k+k; i←1
   while i+k≤n
      a_i ← a_i+a_{i+k}; i ← i+k2
   k ← k2
```

Rezultatul este obţinut în a_1. Evoluţia calculelor apare în următorul tabel:

n	k2	k	i	
7	2	1	1	$a_1 \leftarrow a_1+a_2$
			3	$a_3 \leftarrow a_3+a_4$
			5	$a_5 \leftarrow a_5+a_6$
			7	
	4	2	1	$a_1 \leftarrow a_1+a_3$
			5	$a_5 \leftarrow a_5+a_7$
			9	
	8	4	1	$a_1 \leftarrow a_1+a_5$
			9	

Algoritmul de mai sus nu este atât de stupid şi inutil pe cât apare la prima vedere pentru o problemă atât de simplă.

Într-adevăr, calculele pentru fiecare reluare a ciclului `while` interior sunt executate asupra unor seturi de date disjuncte. De aceea, în ipoteza că pe calculatorul nostru dispunem de mai multe procesoare, calculele pe fiecare nivel al arborelui (mergând de jos în sus) pot fi executate în paralel. Ca urmare, timpul de calcul va fi de ordinul $O(\log n)$, deci mai bun decât cel secvenţial, al cărui ordin este $O(n)$.

Exemplul 3. *Determinarea subşirului crescător de lungime maximă.*

Se consideră vectorul $a=(a_1,\ldots,a_n)$. Se cer lungimea celui mai lung subşir crescător, precum şi toate subşirurile crescătoare de lungime maximă.

Introducem notaţiile:

nr = lungimea maximă căutată;

$lung(i)$ = lungimea maximă a subşirului crescător ce începe cu a_i.

$A=\{1,2,\ldots,n\}$; $X=\{n\}$;

$A_i=\{i+1,\ldots,n\}$ şi $f_i=lung(i)$, $\forall i<n$;

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

Determinarea lui nr se face astfel:

```
nr ← 1; lung(n) ← 1
for i=n-1,1,-1
   lung(i) ← 1+max{lung(j)|j>i & a_i<a_j}
   nr ← max{nr,lung(i)}
```

Determinarea tuturor subşirurilor crescătoare de lungime maximă se face printr-un backtracking recursiv optimal. Subşirurile se obţin în vectorul `s`, iar `poz` reprezintă ultima poziţie completată din `s`.

```
for i=1,n
   if lung(i)=nr
   then poz←1; s(1)←ai; scrie(i)
```

unde procedura `scrie`, în care `i` este poziţia curentă din `a`, are forma:

```
procedure scrie(i)
   if poz=nr
   then write(s)
   else for j=i+1,n
          if ai<aj & lung(i)= 1+lung(j)
          then poz++; s(poz)←a(j); scrie(j); poz--
end;
```

Exemplul 4. *Înmulţirea optimă a unui şir de matrici.*

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

Presupunem că înmulţirea a două matrici se face în modul uzual, adică produsul matricilor $A(m,n)$ 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 matrici $A_1 \times A_2 \times A_3 \times A_4$ 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.

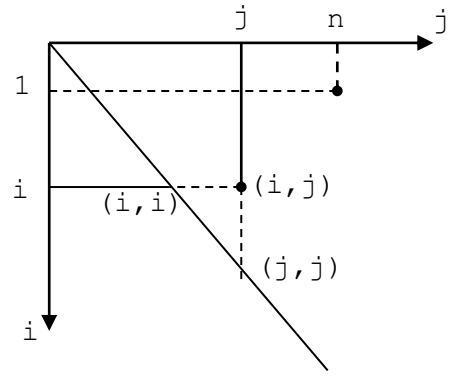
Fie $cost(i,j)$ numărul minim de înmulţiri elementare pentru calculul produsului $A_i \times \ldots \times A_j$. Punând în evidenţă ultima înmulţire de matrici, obţinem relaţiile:

$cost(i,i) = 0, \ \forall i=1,2,\ldots,n$

$cost(i,j) = \min \{cost(i,k)+cost(k+1,j)+d_i \times d_{k+1} \times d_{j+1} \mid i \le k < j\}.$

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 parcurgem în ordine coloanele 2,...,`n`, iar pe fiecare coloană `j` să mergem în sus de la diagonală până la `(i,j)`.

```
for j=2,n
   for i=j-1,1,-1
      cost(i,j) calculat ca mai sus;
      fie k valoarea pentru care se realizează minimul
      cost(j,i)←k
write cost(1,n)
```

(se observă că am folosit partea inferior triunghiulară a matricii pentru a memora indicii pentru care se realizează minimul).

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

```
procedure sol(p,u)
   if p=u
   then write(p)
   else k←cost(u,p)
        write('('); sol(p,k); write(',');
        sol(k+1,u); write(')')
end;
```

Pentru evaluarea timpului de lucru, vom calcula numărul de comparări efectuate. Aceste este:

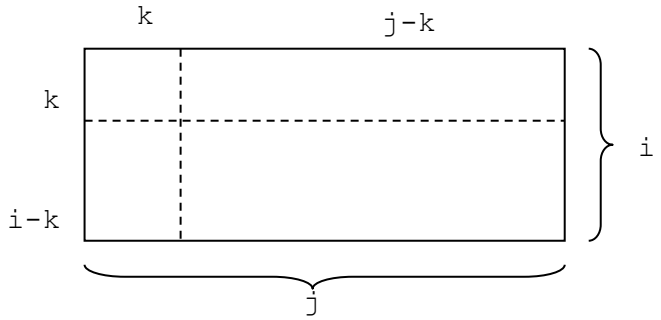
$$\sum_{j=2}^{n}\sum_{i=1}^{j-1}(j-i+1)=\sum_{j=2}^{n}\left[j(j-1)-\frac{(j-1)(j-2)}{2}\right]=O(n^3)$$

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.

Fie a_{ij} = numărul minim de pătrate în care poate fi descompus un dreptunghi de laturi i şi j. Evident $a_{ij}=a_{ji}$. Rezultatul căutat este a_{mn}.

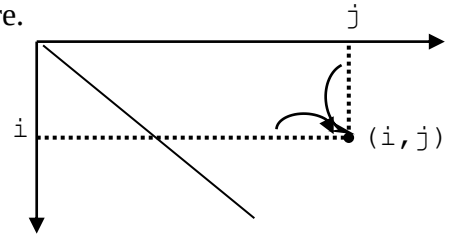
Vârfurile grafului de dependenţe sunt (i,j), iar valorile asociate sunt a_{ij}.



Pentru calculul lui a_{ij} avem de ales între a face:

- o tăietură pe verticală; costurile sunt: $a_{ik}+a_{i,j-k}$, $k\le \lfloor j/2 \rfloor$;
- o tăietură pe orizontală; costurile sunt: $a_{k,j}+a_{i-k,j}$, $k\le \lfloor i/2 \rfloor$.

Rezultă că valoarea a_{ij} 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:

$a_{i,1}=i,\ \forall i=1,\ldots,m$

$a_{1,j}=j,\ \forall j=1,\ldots,n$

$a_{ii}=1,\ \forall=1,\ldots,m$

$a_{ij} = \min\{\alpha,\beta\}$, unde

$\alpha = \min\{a_{ik}+a_{i,j-k} \mid k\le \lfloor j/2 \rfloor\}$, iar $\beta = \min\{a_{k,j}+a_{i-k,j} \mid k\le \lfloor i/2 \rfloor\}$.

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,m
   for j=i+1,n
```

calculul lui a_{ij} conform celei de a patra dependenţe de mai sus

```
      if j≤m then aji←aij
```

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

Exemplul 6. *Verificarea apartenenţei unui cuvânt la limbajul generat de o gramatică independentă de context.*

Fie $G=(N,T,S,P)$ o gramatică independentă de context şi $w \in T^*$. Se cere să se determine dacă $w \in L(G)$.

Putem presupune că gramatica este în forma normală a lui Chomsky, adică producţiile au numai formele $A \to BC$ şi $A \to a$, cu $A, B \in N$ şi $a \in T^*$.

Fie $w=a_1a_2...a_n$. Pentru orice i,j cu $1 \le i \le j \le n$ notăm prin $M(i,j)$ mulţimea:

$M(i,j) = \{ A \in N \mid A \Rightarrow a_i...a_j \}$, unde $\Rightarrow$ semnifică derivare în oricâţi paşi.

Evident, $w \in L(G)$ dacă şi numai dacă $S \in M(1,n)$, deci ceea ce urmărim este determinarea mulţimii $M(1,n)$.

Observăm că:

$M(i,i) = \{ A \in N \mid A \to a_i \in P \}$ pentru toţi $i=1,2,...,n$.

Pentru $1 \le i < j \le n$, mulţimea $M(i,j)$ se construieşte astfel:

```
M(i,j) ← ∅
for k=i,j-1
   for toţi B∈M(i,k)
      for toţi C∈M(k+1,j)
         if A→BC ∈ P
         then M(i,j) ← M(i,j) ∪ {A}
```

Se observă că dependenţele sunt cele de la înmulţirea optimă a matricilor, deci se poate urma aceeaşi ordine a calculelor.