# TAP
# Curs 5: PROGRAMARE DINAMICĂ

Tehnici avansate de programare

Lect.dr. Iulia Banu
Departamentul de Informatică,
Universitatea din București

semestrul 1, 2017

# Rezumat curs Programare dinamica

# Programare dinamică

- Aplicabilă pentru probleme de optim sau de numărare.
- Presupune, asemănător metodei divide et impera, împărțirea problemei inițiale în subprobleme asemănătoare, de dimensiune mai mică.
- Subproblemele și dependențele dintre acestea pot fi reprezentate printr-un graf orientat. Acestuia i se asociază un arbore astfel încât rezolvarea problemei inițiale presupune parcurgerea acestuia în postordine.

# Programare dinamică

- Aplicabilă pentru pentru probleme care presupun rezolvarea de relații de recurență. O subproblemă poate fi rezolvată după ce toate subproblemele de care depinde aceasta au fost rezolvate.
- De obicei relațiile de recurență se obțin din respectarea unui principiu de optimalitate (rezolvarea optimă a subproblemelor).
- O subproblemă poate apărea în rezolvarea mai multor probleme. Pentru optimizarea timpului de execuție este de evitat rezolvarea aceleași probleme de mai multe ori.

# Rezolvarea recurențelor top-down, bottom-up

Pentru a evita rezolvarea aceleași probleme de mai multe ori, recurențele se pot aborda astfel:

- **top-down** Dacă o subproblemă a mai fost rezolvată se returnează rezultatul obținut anterior, dacă subproblema nu a mai fost rezolvată se calculează și se memorează rezultatul.
- **bottom-up (programare dinamică)** Se stabilește ordinea de rezolvare a subproblemelor astfel încât atunci când este rezolvată o subproblemă, toate subproblemele de care depinde aceasta au fost deja rezolvate. Primele probleme rezolvate sunt cele pentru care rezultatul este calculat direct.

# Rezolvarea recurențelor top-down, bottom-up

Asupra unui număr întreg pozitiv $n$ se pot face următoarele operații:

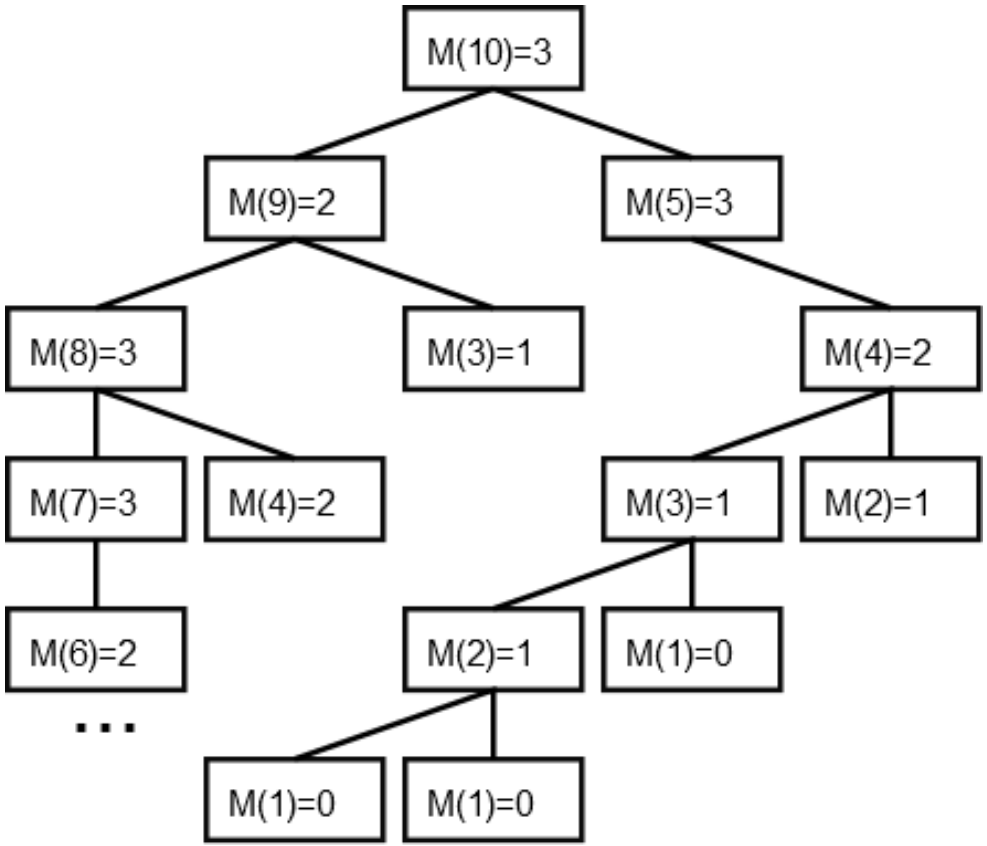
- $n := n - 1$
- dacă $n\%2 = 0$ atunci $n := n/2$
- dacă $n\%3 = 0$ atunci $n := n/3$

Să se obțină numărul minim de operații M(n) după care n va reține valoarea 1. Spre exemplu M(10) = 3.

**?** Care ar fi o strategie Greedy pentru rezolvarea acestei probleme? Este strategia Greedy corectă?

```
int calculM(int n){
      int n1, n2, n3;
      if (n == 1)
         return 0;
      else{
         n2 = n3 = n;
         n1 = calculM(n-1);
         if (n%2 == 0)
            n2 = calculM(n/2);
         if (n%3 == 0)
            n3 = calculM(n/3);
         return 1+min(min(n1,n2),n3);
      }
   }
```

# Memorare, top-down



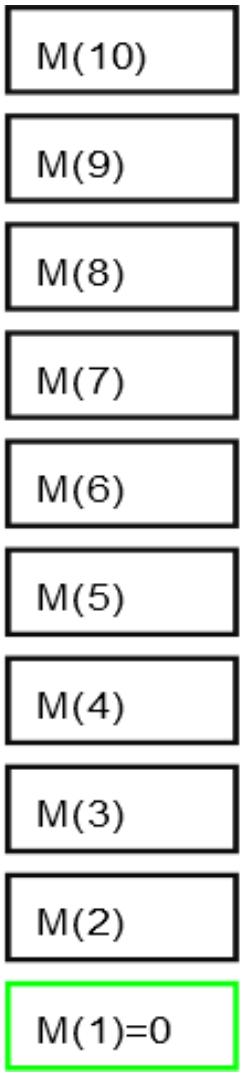

Cum evităm rezolvarea aceleași subprobleme de mai multe ori? Spre exemplu, procedura M(4) va fi apelată de două ori.

Dacă o subproblemă a mai fost rezolvată se returnează rezultatul obținut anterior, dacă subproblema nu a mai fost rezolvată se calculează și se memorează rezultatul.

```
int calculM(int n){
int n1, n2, n3;
   if (n == 1) return 0;
   else{  n2 = n3 = n;
      if (nrm[n-1] == 0)
         nrm[n-1] = n1 = calculM(n-1);
      n1 = nrm[n-1]; //se utilizeaza rezultatul calculat anterior
      if (n%2 == 0){
         if (nrm[n/2] == 0)
            nrm[n/2] = calculM(n/2);
         n2 = nrm[n/2];
      }
      if (n%3 == 0){
         if (nrm[n/3] == 0)
            nrm[n/3] = calculM(n/3);
         n3 = nrm[n/3];
      }
      return 1 + min(min(n1,n2),n3);
   }
}
```

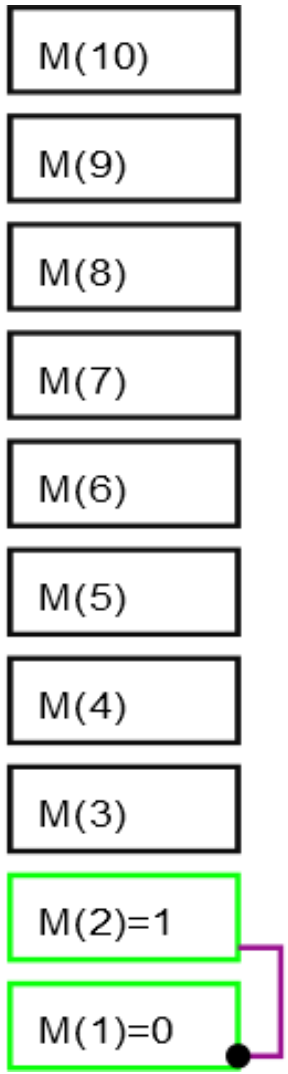
# Bottom-up, programare dinamică

Se stabilește ordinea de rezolvare a subproblemelor astfel încât atunci când este rezolvată o subproblemă, toate subproblemele de care depinde aceasta au fost deja rezolvate. Primele probleme rezolvate sunt cele pentru care rezultatul este calculat direct.



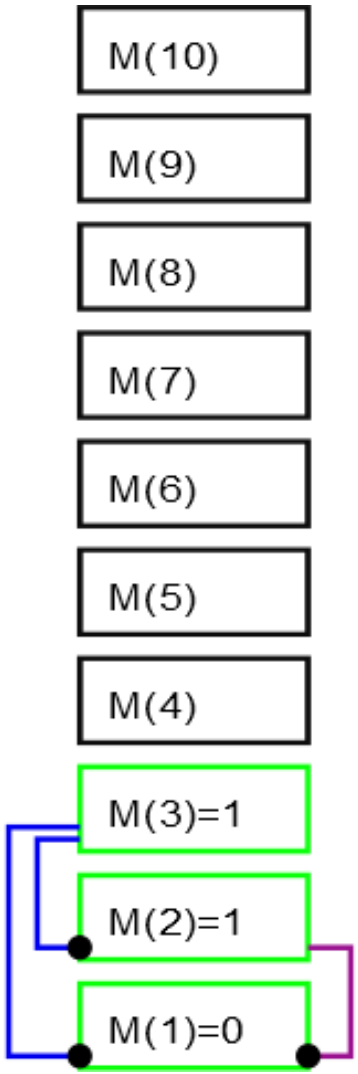
# Bottom-up, programare dinamică

Se stabilește ordinea de rezolvare a subproblemelor astfel încât atunci când este rezolvată o subproblemă, toate subproblemele de care depinde aceasta au fost deja rezolvate. Primele probleme rezolvate sunt cele pentru care rezultatul este calculat direct.



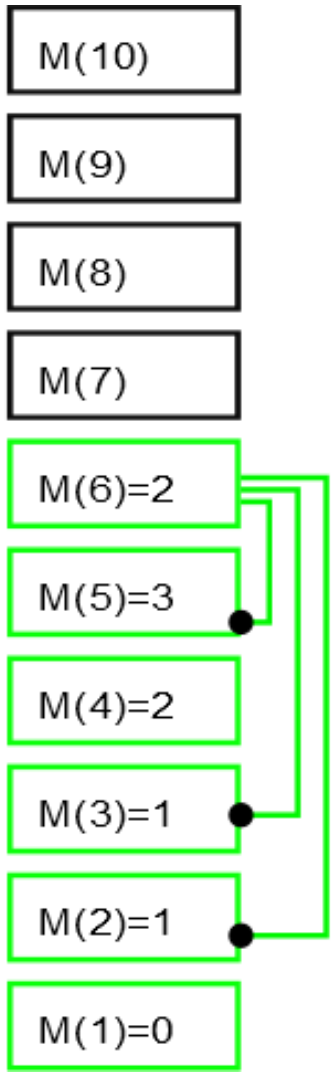
# Bottom-up, programare dinamică

Se stabilește ordinea de rezolvare a subproblemelor astfel încât atunci când este rezolvată o subproblemă, toate subproblemele de care depinde aceasta au fost deja rezolvate. Primele probleme rezolvate sunt cele pentru care rezultatul este calculat direct.



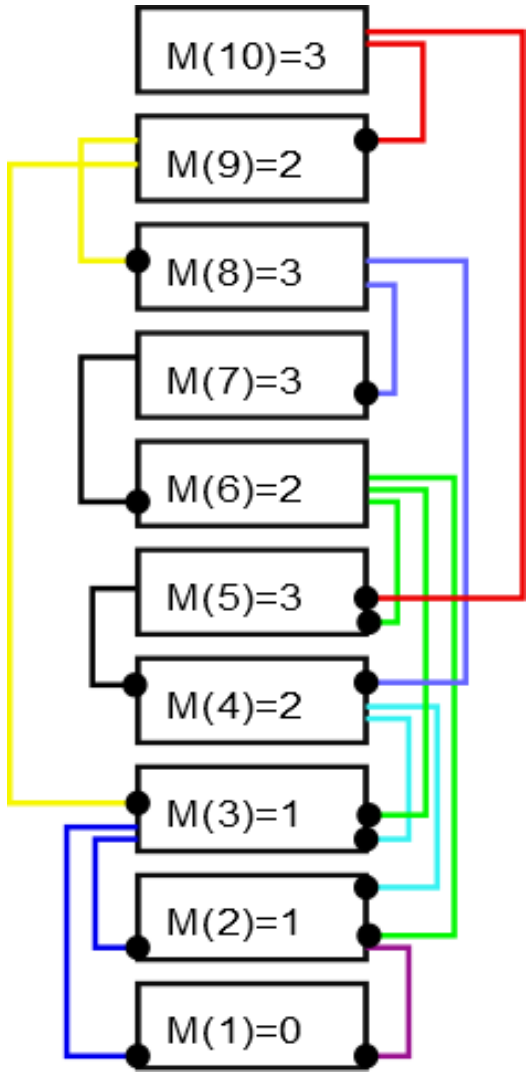
# Bottom-up, programare dinamică

Se stabilește ordinea de rezolvare a subproblemelor astfel încât atunci când este rezolvată o subproblemă, toate subproblemele de care depinde aceasta au fost deja rezolvate. Primele probleme rezolvate sunt cele pentru care rezultatul este calculat direct.



# Bottom-up, programare dinamică

Se stabilește ordinea de rezolvare a subproblemelor astfel încât atunci când este rezolvată o subproblemă, toate subproblemele de care depinde aceasta au fost deja rezolvate. Primele probleme rezolvate sunt cele pentru care rezultatul este calculat direct.



```
int calculV(int n){
      int n1, n2, n3;
      for (int i = 2; i<=n; i++){
         n2 = n3 = n;
         n1 = nrm[i-1];
         if (n%2 == 0)
            n2 = nrm[i/2];
         if (n%3 == 0)
            n3 = nrm[i/3];
         nrm[i] = 1 + min(min(n1,n2),n3);
      }
      return nrm[n];
   }
```

# Prezentare generală - Graf de dependențe, pd-arbore

Fie $A$ și $B$ două mulțimi oarecare.Fiecărui element $x \in A$ urmează să i se asocieze o valoare $v(x) \in B$.

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

**problema inițială $v(z)$.**

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

**mulțimea subproblemelor pentru care rezultatul, v(x) este calculat direct: $X$.**

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

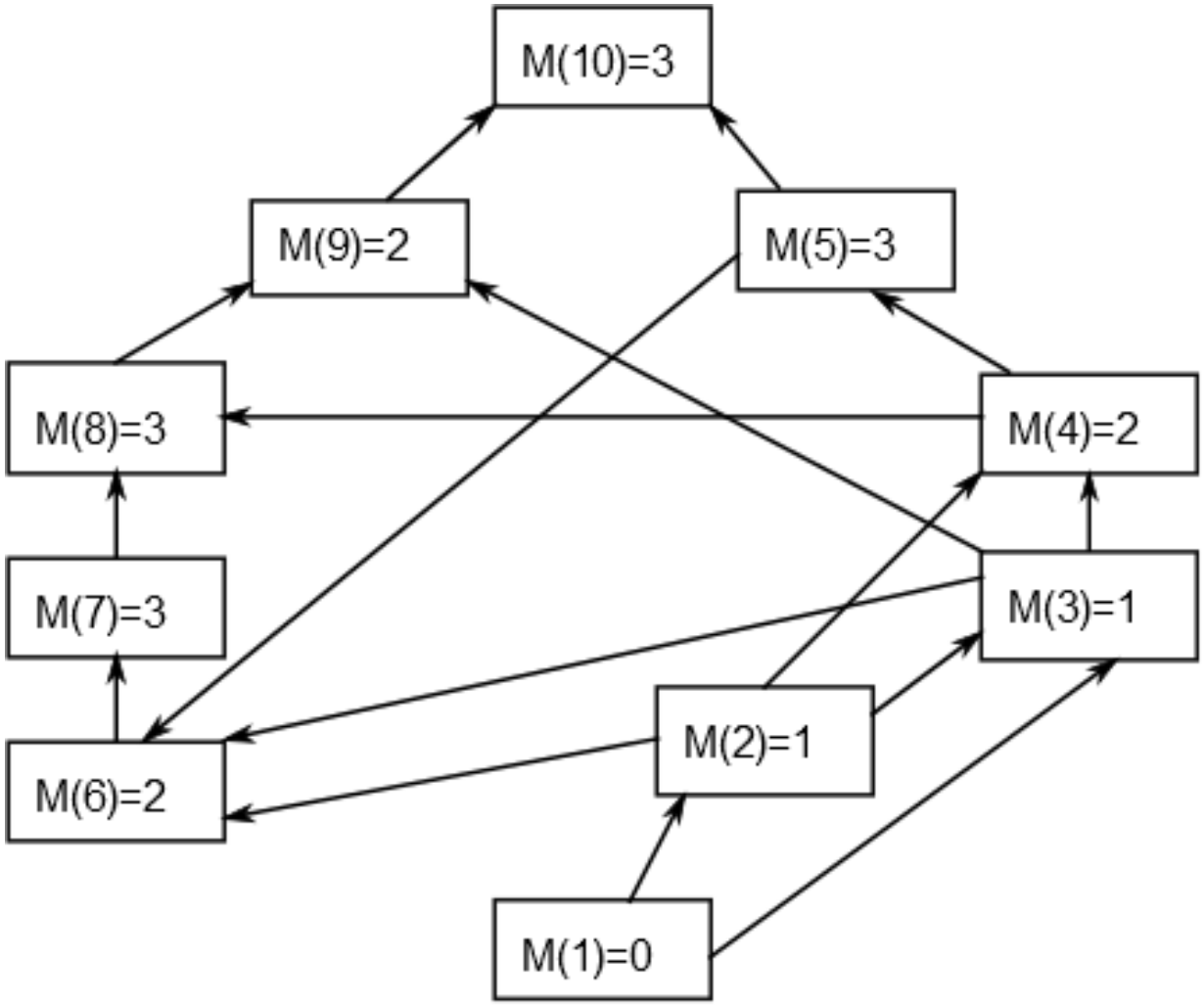
**recurența: $v(x) = f_x(v(a_1), \ldots, v(a_k))$**

unde $A_x = \{a_1, \ldots, a_k\} \subset A$ este mulțimea elementelor din $A$ de a căror valoare depinde $v(x)$, iar $f(x)$ este o funcție care specifică această dependență.

# Prezentare generală - Graf de dependențe, pd-arbore

- Putem reprezenta **subproblemele/soluțiile subproblemelor** ca **noduri într-un graf**. $x/v(x)$
- Există o submulțime de noduri **X** care reprezintă subprobleme pentru care rezultatul poate fi calculat direct.
- O subproblemă $x$ are ca **descendenți**, subproblemele de a căror rezolvare depinde. $A_x = \{a_1, \ldots, a_k\} \subset A$
- Ne interesează doar valorile vârfurilor $y$ de a căror valoare depinde $v(z)$ (direct sau indirect), mulțimea vârfurilor de la care există drum până la $z$: **vârfuri observabile din z ,** $G_z$.

# Prezentare generală - Graf de dependențe, pd-arbore



# Prezentare generală - Graf de dependențe, pd-arbore

Problema are soluție dacă și numai dacă:

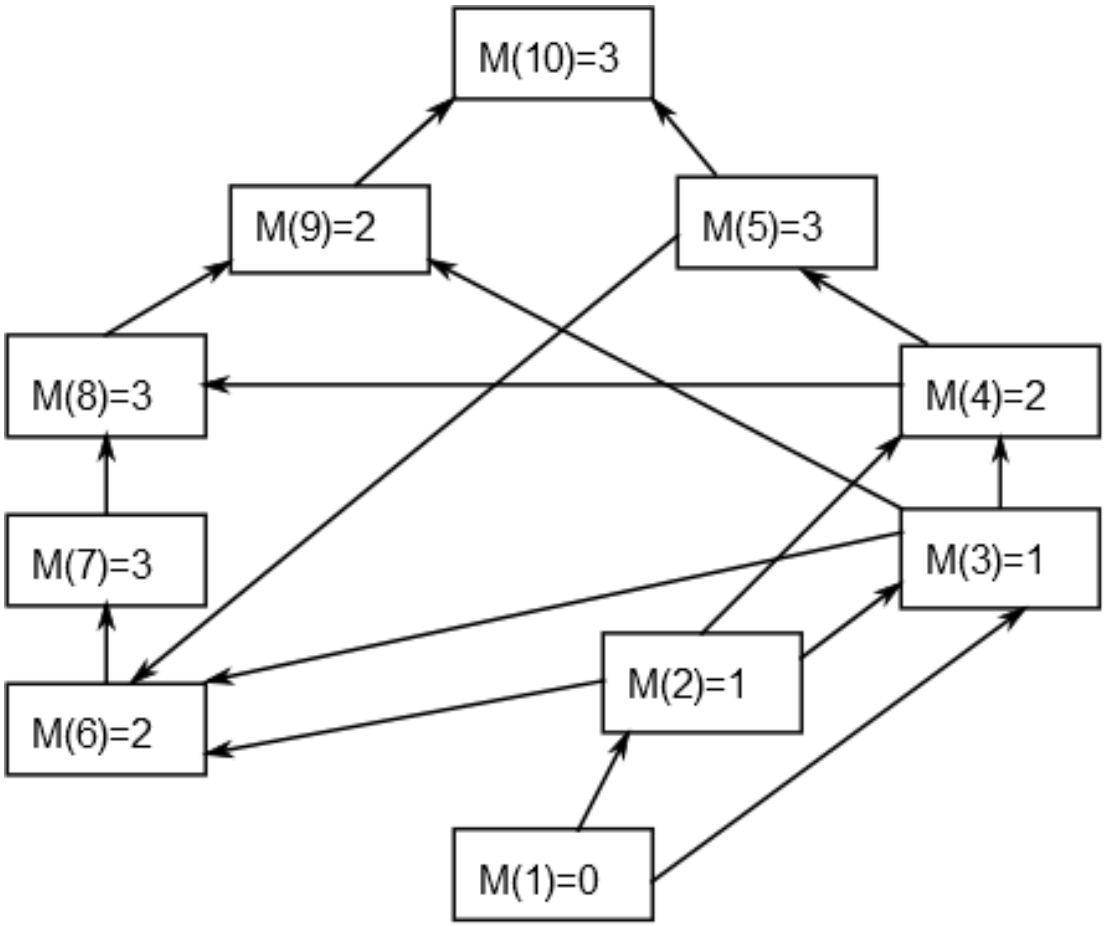
- $G_z$ nu are circuite;
- vârfurile din $G_z$ în care nu sosesc arce fac parte din $X$.

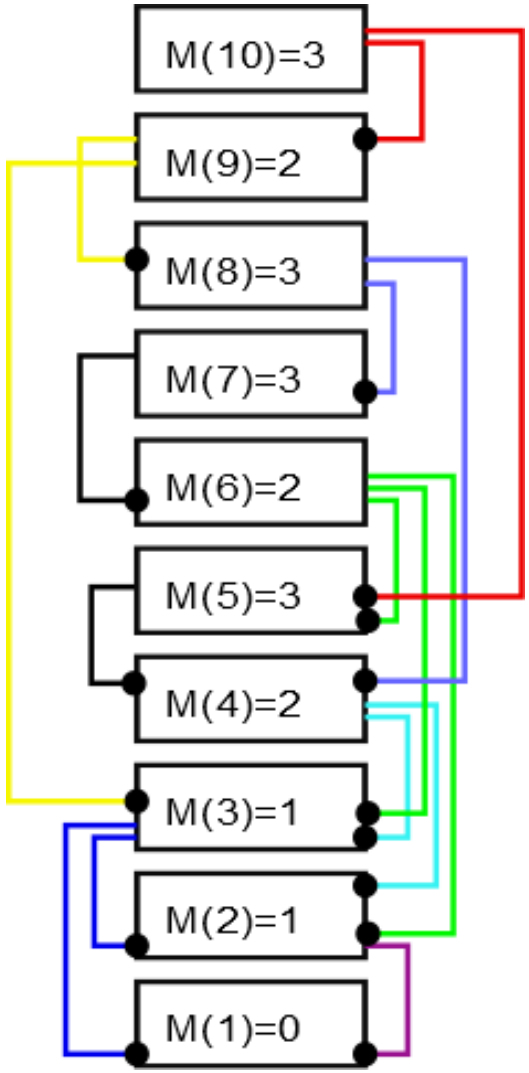
$G_z$ poate fi așezat pe niveluri: **PD-arbore**.
Nodurile $x$ din $G_z$ pentru care drumul de lungime maximă de la $x$ la $z$ are lungimea $l$ vor fi așexate pe nivelul $l$. Rădăcina este $z$.

Rezolvarea problemei v(z) constă în **parcurgerea în postordine** a PD-arborelui.

În exemplul anterior, parcurgerea în postordine a PD-arborelui se reduce la parcurgerea unui vector.

# Prezentare generală - Graf de dependențe, pd-arbore




Lungime maximă de la $M(1)$ la $M(10)$ este 9. Lungime maximă de la $M(3)$ la $M(10)$ este 7.

# Algoritmul general

```
for toate varfurile x ∈ A
        if x ∈ X
                vizitat[x] := true
        else
                vizitat[x] := false
postord(z)

procedure postord(x)
        for j ∈ A(x)
                if vizitat[j]=false
                        postord(j)
        calculeaz v(x) ;
        vizitat[x] := true
end
```

# Recurențe pe vector, exemple și aplicații

- Se dă un vector $v$ de numere naturale. Se pornește de la poziția $i = 1$ a lui $v$. Se cere numărul minim de pași în care se poate ajunge pe poziția $n$ dacă v[i] = k are semnificația: de la poziția $i$ se poate ajunge (vizita) la oricare dintre pozițiile $i + 1, i + 2, i + k$, într-un singur pas.

  Care este numărul minim de pași pentru
  v = {5, 6, 4, 1, 4, 1, 2, 2, 1, 1} ?

  Pentru v = {5, 6, 4, 1, 4, 1, 2, 2, 1, 1} se pot face minim 3 pași. Se avansează din poziția 1 în poziția 2. Din poziția 2 se avansează în poziția 8. Din poziția 8 se poate ajunge în poziția 10.

# Recurențe pe vector, exemple și aplicații

Vom calcula pentru orice i valoarea N(i)=numărul minim de pași în care se poate ajunge de pe pozitia 1 pe poziția i. Rezolvarea problemei este N(n).

**?** Care sunt valorile de care depinde N(i)

**?** Optimalitate Pentru a ajunge pe poziția $i$ în număr minim de pași, care este cea mai bună alegere pentru penultima pozitie $j < i$ vizitată? Se poate alege orice $j$ a.î $i$ să fie accesibil direct din $j$ ( $j + v[j] >= i$ )
Se alege $j$ a.î numărul de pași în care se ajunge la poziția $j$ să fie minim.

$$N[i] = 1 + \min\{N[j] \mid j < i \text{ și } j + v[j] >= i \}$$

# Recurențe pe vector, LIS

- Se dă un vector $v$ de numere întregi. Să se determine un subșir crescător al lui $v$ de lungime maximă.

  Pentru v={ 6, 3, 5, 10, 12, 2, 9, 15, 14, 7, 4, 8,13 }
  lungimea maximă a unui șir crescător este
  LMAX = 5: {3, 5, 7, 8, 13} sau {3, 5, 10, 12, 14}.

  Vom calcula pentru orice i valoare L(i)=lungimea celui mai lung șir crescător care se termină pe poziția $i$.

  Alternativa se poate L(i)=lungimea celui mai lung șir crescător care începe pe poziția $i$.

  $L[i] = 1 + \min\{L[j] \mid j < i \text{ și } v[j] < v[j]\}$

  **?** Putem calcula eficient numărul de subșiruri care au lungimea LMAX?

# Recurențe pe vector, exemple și aplicații

- Se dă un șir de caractere $s$. Să se determine cel mai lung subșir al lui s fară repetiții.

  Pentru șirul a = "xaxyxayy" LMAX=3. ("yxa") Pentru șirul a = "xaxyxazy" LMAX=4. ("yxaz")

  Vom calcula pentru orice i valoare L(i)=lungimea celui mai lung subșir fără repetiții care se termină pe poziția i. LMAX este max{L(i)}

# Recurențe pe vector, exemple și aplicații

Vom calcula pentru orice i valoare L(i)=lungimea celui mai lung subșir fără repetiții care se termină pe poziția i. LMAX este max{L(i)}

**?** Care sunt valorile de care depinde L(i)

**?** Optimalitate Daca s[i] nu apare pe o pozitie $j < i$ atunci
L(i) = L(i-1) + 1.

Dacă cea mai mare poziție pe care mai apare s[i] mai mică decât $i$, e $j = poz(i) < i$ atunci

L(i) = L(i-1) + 1 dacă $i - L[i-1] > j$
și
L(i) = i - j dacă $i - L[i-1] <= j$

# Exemple și aplicații

- Se dă un șir de n numere naturale. Se pot alege subșiruri *s* ale șirului *v* cu proprietatea că dintre oricare 3 elemente consecutive ale lui v, cel puțin unul este ales în s. Să se găsească cea mai mică sumă *smin* pe care o pot avea elementele unei submulțimi *s* astfel construite.

Pentru v = {1, 2, 3, 4, 1, 3, 2, 4} se poate alege
s = {v[2], v[5], v[7]} = {2, 1, 2} cu suma 5.

Pentru v = {5, 3, 7, 2, 4, 1, 2} se va alege s = {v[2], v[4], v[6]} cu suma 6.

Vom calcula pentru orice i, $1 \leq i \leq n$
S[i]=suma minimă care se poate forma dacă a fost ales în s al i-lea element din *v*.
S[1]= v[1]; S[2]=min{v[1],v[2]}; S[3] = min{v[1],v[2],v[3]}.
S[i]=v[i] + min{s[i-1],s[i-2],s[i-3]}.
Soluția problemei este min{s[n],s[n-1],s[n-2]}.

# Recurențe pe vector, exemple și aplicații

- Orice număr natural se poate reprezenta ca sumă de pătrate (n = 1*1 + 1*1 + ...). Să se găsească numărul minim de pătrate a căror sumă este n.

Vom calcula pentru orice $i < n$ valoarea:

N(i)=numărul minim de pătrate în care poate fi împărțit $i$.

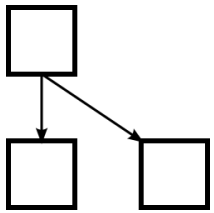
Se consideră $N(0) = 0; N(1) = 1$

$$N(i) = 1 + \min_{k^2 \leq i} \{N(i - k^2)\}$$

# Trasee de sumă maximă

Se consideră un triunghi de numere naturale nenule t 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;



- Din celula (i,j) putem merge doar în (i+1,j) sau (i+1,j+1).
- Traseul se oprește pe oricare dintre celulele ultimei linii.

1
3 2
7 6 9
0 5 8 4

# Trasee de sumă maximă

Se va calcula M[i][j] = suma maximă a unui traseu care începe din T[1][1] și se termină cu celula (i,j).

M[1][1] = T[1][1]
M[i][j] = max{M[i-1][j], M[i-1][j-1]} pentru $i <= j$.

T

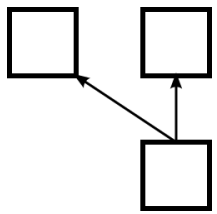
| | | | |
|---|---|---|---|
| 1 | | | |
| 3 | 2 | | |
| 7 | 6 | 9 | |
| 0 | 5 | 8 | 4 |

M

| | | | |
|---|---|---|---|
| 1 | | | |
| 4 | 3 | | |
| 11 | 10 | 12 | |
| 11 | 16 | 20 | 16 |

M

| | | | |
|---|---|---|---|
| 1 | | | |
| 4 | 3 | | |
| 11 | 10 | 12 | |
| 11 | 16 | 20 | 16 |



**?** Care este ordinea în care calculăm valorile lui M?

# Distanța de editare (Levenstein)

Se consideră două șiruri de caractere $A$ și $B$. Putem face următoarele tipuri de modifică asupra primului cuvânt:

a. inserare: se adaugă în cuvânt un caracter pe o poziție (oarecare)

b. ștergere: se șterge o literă din cuvânt (de pe o poziție, nu toate aparițiile)

c. înlocuire: se înlocuiește o literă de pe o poziție din cuvânt cu altă literă

| a. inserare | b. ștergere | c. înlocuire |
|---|---|---|
| ac | aaba | abc |
| abc | aba | aqc |

Distanța de editare d(A,B) = numărul minim de modificări ce se pot face succesiv asupra șirului A astel încât să se obțină B.

# Distanța de editare (Levenstein)

Distanța de editare d(A,B) = numărul minim de modificări ce se pot face succesiv asupra șirului A astel încât să se obțină B.

d("abac","apaq") = 2
abac → abaq → apaq

d("aba","aacd") = 3
aba → aa → aac → aacd

Se va calcula M[i][j] = numărul minim de modificări necesare pentru a transforma subșirul format cu primele i litere ale lui A, A[1 ... i] în subșirul format cu primele j litere ale lui B, B[1 ... j].
M[0][j] = j
(costul transformării șirului vid în șirul B[1 ... j ], se adaugă j caractere)

M[i][0] = i
(costul transformării șirului A[1 ... i] în șirul vid, se șterg j caractere)

# Distanța de editare (Levenstein)

Se va calcula M[i][j] = numărul minim de modificări necesare pentru a transforma subșirul format cu primele i litere ale lui A, A[1 ... i] în subșirul format cu primele j litere ale lui B, B[1 ... j].

M[i][j] = M[i-1][j-1] dacă A[i] = B[j]

M[i][j] = 1 + min {M[i][j-1], M[i][j-1], M[i-1][j-1] } dacă A[i] $\neq$ B[j]

# Exerciții

- Să se calculeze cel mai lung subșir comun a două șiruri de caractere a,b.
  Exemplu: a = "abxcayb" b= "aybcx" - un șir comun de lungime maximă 3 este "abc".

- Se dau două șiruri *s* și *t* de lungimi *n*, respectiv *m*. Să se numere de câte ori apare t ca subșir al lui *s*.

  Exemplu: s = "abxcaybc" t= "abc" - sunt 4 subșiruri ale lui s egale cu t: {s[1],s[2],s[4]}, {s[1],s[2],s[8]}, {s[1],s[7],s[8]}, {s[5],s[7],s[8]}.

  Se va calcula nr[i][j] = numărul de apariții ale șirului t[1..i] ca subșir al lui s[i..j].

  nr[i][j] = nr[i-1][j-1] + nr[i][j-1] dacă t[i] = s[j]

  nr[i][j] = nr[i][j-1] dacă t[i] $\neq$ s[j]

# Exerciții

- Se dau două șiruri $s$ și $p$. Șirul $s$ conține litere mici ale alfabetului, șirul $p$ conține literele mici ale alfabetului și simbolurile $*$ și _.
  Se se testeze dacă $p$ este un pattern care recunoaște $s$.
  Simbolul $*$ înlocuiește orice șir (inclusiv șirul vid), simbolul _ înlocuiește o singură literă.

  Exemplu: Pentru șirurile s="abcxy" și p="*x_" se afișează DA, pentru șirurile s="abcxy" și p="*x_y" se afișează NU.

# Exerciții

- Se dau trei șiruri $a, b, c$. Se se testeze dacă $c$ este rezultatul interclasării șirurilor $a, b$.

  Exemplu: Pentru șirurile a="xyz", b="uv", c="xyuzv" se va afișa DA. Pentru șirurile a="xyz", b="uv", c="yuxzv" se va afișa NU.

- Se dau două numere naturale $n$ și $k$. Să se afișeze numărul de permutări de $n$ elemente care au exact $k$ inversiuni.

# Problema rucsacului (discretă)

Se consideră G greutatea unui rucsac, $n$ număr natural, și c=$\{c_1, \ldots c_n\}$, g=$\{g_1, \ldots g_n\}$ și și greutățile a n obiecte. Un obiect se încarcă în întregime în rucsac. Să se încarce rucsacul astfel încât costul total al obietelorîncărcate să fie maxim.

Exemplu: Pentru G=8, g =$\{3, 4, 4, 6\}$ și c =$\{3, 9, 10, 18\}$ profitul maxim este 19, valoare care se obține încărcând obiectele 2 și 3.

Vom calcula S[i][g] = costul maxim care se poate obține dacă într-un rucsac de greutate G încărcăm o submulțime a obiectelor 1 ... i.

**?** Ce valoare a matricei S conține soluția problemei.

# Înmulțire optimă de matrici

Avem de calculat produsul de matrice $A_1 \cdot A_2 \cdot \dots \cdot A_n$, unde dimensiunile lor sunt respectiv $(d_1, d_2), (d_2, d_3), ...., (d_n, d_{n+1})$. Produsul matricelor A(m,n) și B(n,p) necesită $m \cdot n \cdot p$ înmulțiri elementare.

Știind că înmulțirea matricelor este asociativă,care este ordinea în care trebuie înmulțite matricele astfel încât numărul de înmulțiri elementare să fie minim?

Exemplu: $(A_1 \cdot A_2) \cdot (A_3 \cdot A_4)$ unde d1 = 10, d2=10, d3=20, d4=50, d5=5 necesită $10 \cdot 10 \cdot 20 + 20 \cdot 50 \cdot 5 + 10 \cdot 20 \cdot 5 = 8000$ înmulțiri elementare.

$((A_1 \cdot A_2) \cdot A_3) \cdot A_4$ necesită $10 \cdot 10 \cdot 20 + 10 \cdot 20 \cdot 50 + 10 \cdot 50 \cdot 5 = 14500$ înmulțiri elementare.

# Înmulțire optimă de matrici

Memorăm pentru $i < j$
M[i][j] = numărul minim de înmulțiri elementare necesar pentru a înmulții matricele $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$,

M[i][i] = 0 M[i][i+1] = $d_i \cdot d_{i+1} \cdot d_{i+2}$

Optimalitate Dacă $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ se calculează pentru $i < j-1, i \leq k < j$ astfel: $(A_i \cdot \dots A_k) \cdot (A_{k+1} \dots \cdot \dots \cdot A_j)$ atunci numărul minim de operații efectuate este:

M[i][k] + M[k+1][j] + $d_i \cdot d_{k+1} \cdot d_j$

Astfel M[i][j] = $\min_{i \leq k < j} \{M[i][k] + M[k+1][j] + d_i \cdot d_{k+1} \cdot d_{j+1}\}$

# Înmulțire optimă de matrici

| | | | |
|---|---|---|---|
| 0 | - | - | - |
| - | 0 | - | - |
| - | - | 0 | - |
| - | - | - | 0 |

| | | | |
|---|---|---|---|
| 0 | 2000 | - | - |
| 1 | 0 | 10000 | - |
| - | 2 | 0 | 5000 |
| - | - | 3 | 0 |

| | | | |
|---|---|---|---|
| 0 | 2000 | 12000 | - |
| 1 | 0 | 10000 | 6000 |
| 2 | 2 | 0 | 5000 |
| - | 2 | 3 | 0 |

| | | | |
|---|---|---|---|
| 0 | 2000 | 12000 | 6500 |
| 1 | 0 | 10000 | 6000 |
| 2 | 2 | 0 | 5000 |
| 1 | 2 | 3 | 0 |

# Exerciții

- Se consideră un șir s format cu literele a, b, c. Asupra lui s se pot face succesiv următoarele transformări: subșirul "ab" poate fi înlocuit cu "c", subșirul "bc" poate fi înlocuit cu "a" iar subșirul "ca" poate fi înlocuit cu "b". Să se afle lungimea minimă la care poate ajunge s după aplicări succesive ale înlocuirilor de mai sus.

  Exemplu: s="abcab" se va transforma într-un șir de lungime 1: "a**bc**ab" → "aa**ab**" → "a**ac**" → "**ab**" → "c".

- Se consideră un șir de caractere *s*. Să se afle numărul minim de caratere ce trebuie adăgat în *s* pentru ca acesta să devină palindrom.

  Exemplu: În s="axyba" se vor adăuga două caractere "by" începând de la poziția 2.

# Exerciții

- Se dau două șiruri de caractere *a* și *b*. Să se determine în câte moduri se poate adăuga o literă în șirul *a*, obținându-se șirul a', astfel încât LCS(a,b) = LCS(a',b) - 1. Unde LCS(a,b) este lungimea maximă a unui șir comun al șirurilor a, b.

  Exemplu: Pentru șirurile a="xz", b="xtxz" se poate insera litera x pe pozițiile 0 sau 1 sau litera t pe poziția 1.

- Considerăm următorul joc pentru două persoane. Tabla de joc este o secvență de *n* numere întregi pozitive, iar cei doi jucători mută alternativ. Când un jucător mută, el selectează un număr ori de la stânga ori de la dreapta secvenței. Numărul selectat este șters de pe tablă. Jocul se termină când toate numerele au fost selectate. Să se obțină suma maximă pe care o poate selecta primul jucător.