Programare DINAMICĂ. TEORIE și APLICAȚII

Radu Vișinescu, Violeta Vișinescu

În acest articol propunem o prezentare a tehnicii programării dinamice, urmată de o serie de aplicații cu grad mic de dificultate, care pot fi folosite pentru inițierea elevilor în rezolvarea acestor tipuri de probleme.

Folosirea tehnicii programării dinamice presupune intuiție și abilități matematice. De multe ori rezolvările date prin această tehnică au ordin de complexitate polinomial.

La un nivel superior, toate tehnicile de programare pot fi privite ca particularizări ale programării dinamice.

Considerații teoretice

Tehnica programării dinamice se aplică problemelor de optim referitor la un anumit criteriu dat în enunț. În cele mai multe cazuri soluția care satisface acest optim nu este unică. În literatura de specialitate există două variante de prezentare generală a acestei tehnici. Prima dintre ele este mai degrabă de natură deductivă pe când a doua folosește gândirea inductivă.

În prima variantă apare conceptul de ***subproblemă***. Sunt considerate următoarele două aspecte care caracterizează o rezolvare prin programare dinamică:

- Problema se poate descompune recursiv în mai multe subprobleme care sunt caracterizate de optime parțiale, iar optimul global se obține prin combinarea acestor optime parțiale.
- Subproblemele respective se suprapun. La un anumit nivel, două sau mai multe subprobleme necesită, pentru a fi rezolvate, rezolvarea unei aceeași subprobleme. Pentru a evita risipa de timp rezultată în urma unei implementări recursive, optimele parțiale se vor reține treptat, în maniera bottom-up, în anumite structuri de date (tabele).

Fie următoarea problemă: *Considerăm o rețea de orașe legate prin șosele. Se cere determinarea pentru fiecare pereche de orașe A, B a drumului de lungime minimă care le unește.*

Problema se poate aborda folosind metoda programării dinamice. Dacă drumul minim care leagă orașele A și B trece prin orașul C, atunci și subdrumurile: $A \ldots C$ și $C \ldots B$ sunt de lungime minimă, afirmație care poate fi demonstrată folosind metoda reducerii la absurd.

Am descompus astfel problema în subprobleme. Mai departe, dacă avem drumuri minime de forma $A \ldots C \ldots B_1$ și $A \ldots C \ldots B_2$, atunci subproblemele (A, B_1) și (A, B_2) necesită rezolvarea aceleași subprobleme (A, C). Ca urmare, problema satisface și cel de-al doilea aspect general.

A doua variantă de prezentare face apel la conceptele intuitive de sistem, stare și decizie. O problemă este abordabilă folosind tehnica programării dinamice dacă satisface principiul de optimalitate sub una din formele prezentate în continuare.

Fie secvența de stări $S_0, S_1, \ldots, S_n$ ale sistemului.

- Dacă $d_1, d_2, \ldots, d_n$ este un șir optim de decizii care duc la trecerea sistemului din starea S_0 în starea S_n, atunci pentru orice i $(1 \le i \le n)$ $d_{i+1}, d_{i+2} \ldots, d_n$ este un șir optim de decizii care duc la trecerea sistemului din starea S_i în starea S_n. Astfel, decizia d_i depinde de deciziile anterioare $d_{i+1}, \ldots d_n$. Spunem că se aplică metoda ***înainte***.
- Dacă $d_1, d_2, \ldots, d_n$ este un șir optim de decizii care duc la trecerea sistemului din starea S_0 în starea S_n, atunci pentru orice i $(1 \le i \le n)$ $d_1, d_2 \ldots, d_i$ este un șir optim de decizii care duc la trecerea sistemului din starea S_0 în starea S_i. Astfel, decizia d_{i+1} depinde de deciziile anterioare

d_1,... d_i. Spunem că se aplică metoda ***înapoi***.

- Dacă d_1, d_2,..., d_n este un șir optim de decizii care duc la trecerea sistemului din starea S_0 în starea S_n, atunci pentru orice i ($1 \le i \le n$) d_{i+1}, d_{i+2} ..., d_n este un șir optim de decizii care duc la trecerea sistemului din starea S_i în starea S_n și d_1, d_2 ..., d_i este un șir optim de decizii care duc la trecerea sistemului din starea S_0 în starea S_i. Spunem că se aplică metoda ***mixtă***.

Considerăm o a doua problemă: *Se consideră un triunghi format din N linii (*$1 \le N \le 100$*), fiecare linie conținând numere întregi din domeniul* {1, 2, ..., 99}.

Exemplu

```
          7
        3   8
      8   1   0
    2   7   4   4
4   5   2   6   5
```

Determinați cea mai mare sumă de numere aflate pe un drum între numărul de pe prima linie și unul dintre numerele de pe ultima linie. Succesorii posibili ai unui număr pe acest drum sunt situați sub acesta, o poziție la stânga sau o poziție la dreapta.

Sistemul este reprezentat de structura de date corespunzatoare problemei, iar decizia este reprezentată de alegerea de a merge la un anumit pas la stânga sau la dreapta.

Problema îndeplinește criteriul de optimalitate în prima sa formă: dacă suntem pe un nivel $k + 1$ și am luat toate deciziile de a ajunge în mod optim la toate elementele de pe acest nivel, atunci decizia d_k corespunzătoare nivelului k se construiește de la stânga la dreapta în modul următor: pentru fiecare element x de pe nivelul k se consideră cei doi succesori aflați pe nivelul $k + 1$ deja procesat și este ales acela pentru care suma până în acel moment este maximă.

Indiferent de varianta de prezentare, rezolvarea prin programare dinamică presupune găsirea și rezolvarea unui sistem de recurențe.

În prima variantă avem recurențe între subprobleme, în a doua avem variantă recurențe în șirul de decizii. Deoarece rezolvarea prin recursivitate duce de cele mai multe ori la ordin de complexitate exponențial, se aleg anumite tabele auxiliare pentru reținerea optimelor parțiale și spațiul de soluții se parcurge în ordinea crescătoare a dimensiunilor subproblemelor.

Folosirea acestor tabele dă și numele tehnicii respective.

Pentru prima problemă folosim o matrice auxiliară *COST* în care vom calcula costurile minime între oricare perechi de orașe. Sistemul de recurențe este următorul:

- $COST(A, B) = D(A, B)$ pentru C = 0;
- $COST(A, B) =$ **min**({$COST(A, B)$, $COST(A, C) + COST(C, B)$ | pentru C = 1, 2, ... , N}).

Pentru a doua problemă, triunghiul va fi memorat în matricea A sub diagonală, stânga poziției (P, Q) va fi $(P + 1, Q)$, iar dreapta va fi $(P + 1, Q + 1)$. Folosim o matrice auxiliară S iar sistemul de recurențe va fi:

- $S(P, N) = A(P, N)$ pentru $1 \le P \le N$
- $S(P, Q) =$ **max**{$S(P + 1, Q)$, $S(P + 1, Q + 1)$} $+ A(P, Q)$ pentru $P < N$, $1 \le Q \le P$.

În afara cazurilor în care recurențele sunt evidente, este necesară și o justificare sau demonstrație a faptului că aceste recurențe sunt cele corecte.

Pentru cea de-a doua problemă rezolvarea este în mod clar corectă. Pentru prima vom demonstra valididatea sistemul de recurențe:

Demonstrăm prin inducție că matricea costurilor generate la pasul K conține costurile minime între oricare două orașe, pentru acele drumuri care trec numai prin orașe intermediare cu etichete mai mici sau egale cu K. Propoziția care va fi demonstrată este notată prin P_K,

Propoziția P_0 este evident adevărată.

Presupunem că propoziția P_{K-1} este și ea adevărată. Fie A și B două orașe oarecare; distingem două cazuri:

- drumul minim prin primele K orașe nu trece prin orașul K; atunci: $COST(A, B) \le COST(A, K) + COST(K, B)$ și, conform ipotezei de inducție, relația se verifică.
- drumul minim prin primele K orașe trece prin orașul K; atunci trece numai o singură dată prin orașul K și avem: $COST(A, B) \ge COST(A, K) + COST(K, B)$; din definiția recurențelor rezultă că și P_K este adevărată.

De cele mai multe ori, dacă pe lângă calcularea valorii optime se cere și o secvență de decizii care duce la obținerea optimului, se impune folosirea de memorie suplimentară.

Pentru a doua problemă, după construirea matricei S, drumul care duce la obținerea sumei maxime poate fi calculat pornind din vârf și coborând la fiecare pas în direcția la care valoarea respectivă este maximă. Așadar determinarea unui drum optim nu necesită memorie suplimentară.

În cazul primei probleme, este utilă folosirea unei matrice auxiliare de predecesori care se construiește în modul următor:

Inițial avem $PRED(I, J) = I$, dacă există drum direct de la I la J sau $PRED(I, J) = 0$ în caz contrar.

La fiecare pas K vom avea $PRED(I, J) = PRED(I, J)$, dacă $COST(I, J) \le COST(I, K) + COST(K, J)$ sau $PRED(I, J) = PRED(K, J)$, dacă $COST(I, J) > COST(I, K) + COST(K, J)$.

Pentru orașele A și B, un drum de cost minim care le unește va fi dat de valorile succesive: B, $PRED(A, B)$, $PRED(A, PRED(A, B))$, $PRED(A, PRED(A, PRED(A, B)))$,... , A.

Acest algoritm este cunoscut în literatura de specialitate drept algoritmul ***Roy-Floyd***.

Aplicații

În cele ce urmează vom prezenta enunțurile mai multor probleme și vom arăta, pentru fiecare dintre acestea, modul în care pot fi aplicate cunoștințele teoretice prezentate.

Problema #1

Fie un șir $X = (x_1, x_2, ..., x_n)$ de lungime N ale cărui elemente sunt numere întregi.

Numim subșir al șirului X o succesiune de elemente din X, în ordinea în care acestea apar în X: $X_{i_1}, X_{i_2}, ..., X_{i_k}$ unde $1 \le i_1 < i_2 < ... < i_k \le N$.

Se cere determinarea lungimii maxime a unui astfel de subșir, cu proprietatea suplimentară de a fi crescător. De asemenea, trebuie precizat unul dintre aceste șiruri crescătoare de lungime maximă.

Rezolvare

Fie $X_{i_1}, X_{i_2}, ..., X_{i_k}$ o soluție a problemei. În acest caz $X_{i_1}, X_{i_2}, ..., X_{i_k}$ este o subsoluție optimă pentru subsecvența care începe de la poziția i_1, $X_{i_2}, X_{i_3}, ..., X_{i_k}$ este o subsoluție optimă pentru subsecvența ce începe de la poziția i_2 ș.a.m.d. Afirmația se poate justifica folosind metoda reducerii la absurd.

Se va aplica metoda înainte, determinând o soluție a fiecărei subprobleme corespunzătoare subsecvenței formate din elementele vectorului cuprinse între pozițiile K și N, pentru $1 \le K \le N$.

Folosim vectorul L, unde L_K reprezintă lungimea maximă a subsoluției pentru subsecvența care începe la poziția K și opțional vectorul $SUCC$ pentru memorarea succesorilor în soluția generală a problemei.

Sistemul de recurențe va fi:

- $L_N = 1$, $SUCC_N = 0$;
- pentru $1 \le K < N$:

  $L_K = \max\{L_P + 1 \mid X_K \le X_P\}$ pentru $P > K$

  sau

  $L_K = 1$, dacă pentru oricare $P > K$ avem $X_K > X_P$,

  $SUCC_K$ = un indice P care maximizează formula anterioară

  sau

  $SUCC_K = 0$, dacă $L_K = 1$.

Prezentăm versiunea în pseudocod a algoritmului prin care se construiesc vectorii L și $SUCC$:

```
L_N ← 1
SUCC_N ← 0
pentru K ← N - 1, 1 pas -1
                          execută
  MAX ← 1
  pentru P ← K + 1, N execută
    dacă X_K ≤ X_P atunci
      MAX ← L_P + 1
      POZ ← P
    sfârșit dacă
    L_K ← MAX
    dacă MAX = 1 atunci
      SUCC_K ← 0
    altfel
      SUCC_K ← POZ
    sfârșit dacă
  sfârșit pentru
sfârșit pentru
```

Lungimea optimă o reprezintă valoarea maximă din vectorul L. Dacă această valoare se află pe poziția POZ, atunci soluția optimă este obținută cu ajutorul vectorului $SUCC$. Varianta în pseudocod este:

```
MAX ← -1
pentru K ← 1, N execută
  dacă L_K > MAX atunci
    MAX ← L_K
    POZ ← K
  sfârșit dacă
sfârșit pentru
scrie MAX
cât timp POZ ≠ 0 execută
  scrie X_POZ
  POZ ← SUCC_POZ
sfârșit cât timp
```

Problema se poate rezolva și folosind metoda înapoi. În ambele situații ordinul de complexitate al algoritmului de rezolvare este $O(N^2)$.

Problema #2

Se consideră o clădire cu N niveluri. În interiorul acesteia se află M lifturi. Fiecare lift L_i face legătura între nivelul de plecare P_i și nivelul de sosire U_i. Avem întotdeauna relația $P_i < U_i$.

Inițial toate lifturile se află la nivelurile lor de plecare și întotdeauna lifturile circulă numai în sus. O persoană își propune să plece de la nivelul 1 și să ajungă la nivelul N, folosind cât mai puține lifturi.

Determinați (dacă există) acest număr minim de lifturi precum și un șir posibil de lifturi care respectă această cerință. Șirul va fi dat prin numerele de ordine ale lifturilor care îl compun.

Date de intrare

Fișierul de intrare **lift.in** are următoarea structură:

- pe prima linie se află numerele N și M, separate printr-un spațiu;
- pe fiecare dintre următoarele M linii se află perechi de valori P_i, U_i care reprezintă nivelurile între care circulă liftul corespunzător; numărul de ordine al liftului este determinat de poziția în fișier a perechii de niveluri între care circulă acesta.

Date de ieșire

Fișierul de ieșire **lift.out** trebuie să aibă următoarea structură:

- pe prima linie se va afla numărul minim NR al lifturilor sau valoarea -1 dacă problema nu are soluție;
- pe a doua linie, în cazul în care există soluții, se vor afla numerele de ordine ale unei secvențe posibile de lifturi care respectă cerința problemei, în ordinea în care persoana folosește aceste lifturi.

Restricții și precizări:

- $1 \le N \le 40$;
- $1 \le M \le 500$.

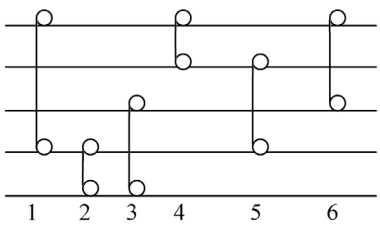
Exemplu

lift.in
```
5 6
2 5
1 2
1 3
4 5
2 4
3 5
```

lift.out
```
2
2 1
```

sau

lift.out
```
2
3 6
```



Problema a fost propusă spre rezolvare elevilor din clasele a XI-a și a XII-a la ediția 2005 a *Olimpiadei Locale de Informatică* din județul *Prahova*.

Rezolvare

Dacă $L_1, L_2, \ldots L_{K-2}, L_{K-1}, L_K$ este un șir optim de lifturi pentru care avem $U_{L_K} = N$, atunci $L_1, L_2, \ldots L_{K-2}, L_{K-1}$, este un șir optim până la nivelul $U_{L_{K-1}} = P_{L_K}$, $L_1, L_2, \ldots L_{K-2}$ este un șir optim de lifturi până la nivelul $U_{L_{K-2}}$ ș.a.m.d.

Folosim doi vectori auxiliari *NR* și *PRED* care memorează numărul minim de lifturi, respectiv predecesorul unui lift în șirul de lifturi. Prezentăm rezolvarea în pseudocod:

```
NR_1 ← 0
PRED_1 ← 0
pentru I ← 2, N execută
  MIN ← ∝
  pentru J ← 1, I - 1 execută
    pentru K ← 1, M execută
      dacă P_K = J și U_K = I
                          atunci
        dacă MIN > NR_J și
                NR_J ≥ 0 atunci
          MIN ← NR_J
          POZ ← K
        sfârșit dacă
      sfârșit dacă
    sfârșit pentru
  sfârșit pentru
  dacă MIN ≠ ∝ atunci
    NR_I ← MIN + 1
    PRED_I ← POZ
  altfel
    NR_I ← -1
    PRED_I ← 0
  sfârșit dacă
sfârșit pentru
dacă NR_N = -1 atunci
  scrie -1
altfel
  scrie NR_N
  drum(N)
sfârșit dacă

subalgoritm drum(NIV)
  dacă PRED_NIV > 0 atunci
    drum(PRED_NIV)
    scrie PRED_NIV
sfârșit dacă
sfârșit subalgoritm
```

Algoritmul are ordinul de complexitate $O(N^2 \cdot M)$ datorită faptului că am memorat proprietățile lifturilor sub forma unei liste de perechi.

Propunem refacerea algoritmului, memorând lifturile care ajung la etajul *I* într-o listă.

Ordinul de complexitate ar trebui să devină $O(N + M)$.

Problema #3

Se consideră un teren denivelat, codificat cu ajutorul unei matrice *A*, care conține *N* linii și *M* coloane, ale cărei elemente sunt numere naturale.

Se numește drum *Est-Sud* o succesiune de celule ale matricei care îndeplinesc condiția ca, la fiecare pas, după o celulă de coordonate (*I*, *J*) să urmeze fie celula dinspre *Est* (*I* + 1, *J*), fie celula dinspre *Sud* (*I*, *J* + 1).

Determinați un drum *Est-Sud* de lungime maximă având în plus proprietatea că valorile situate în celulele sale sunt în ordine crescătoare.

Date de intrare

Fișierul de intrare **drum.in** are următoarea structură:

- pe prima linie se află valorile *N* și *M*, separate printr-un spațiu;
- pe următoarele *N* linii, se află valorile corespunzătoare ale matricei, câte *M* numere pe o linie, separate între ele prin câte un spațiu.

Date de ieșire

Fișierul de ieșire **drum.out** trebuie să aibă următoarea structură:

- pe prima linie valoarea *L*, lungimea maximă a drumului.
- pe următoarele *L* linii, perechi *I*, *J* de coordonate care identifică celulele care alcătuiesc drumul.

Restricții și precizari:

- $1 \le N$;
- $M \le 100$;
- $0 \le A_{IJ} \le 1000$

Exemplu

drum.in
```
4 4
1 20 3 10
4 5 6 6
1 7 3 1
```

drum.out
```
5
1 1
2 1
2 2
2 3
3 4
```

Rezolvare:

Dacă $(I_1, J_1), (I_2, J_2), (I_3, J_3), \ldots, (I_K, J_K)$ este un drum de lungime maximă care pornește de la celula (I_1, J_1), atunci $(I_2, J_2), (I_3, J_3), \ldots, (I_K, J_K)$ este un drum de lungime maximă care pornește de la celula $((I_2, J_2)$, drumul $(I_3, J_3), \ldots, (I_K, J_K)$ este un drum de lungime maximă care pornește de la celula (I_3, J_3) ș.a.m.d.

Folosim două matrice auxiliare *L* și *LEG*, care vor păstra informațiile necesare determinării drumului.

Valoarea L_{IJ} reprezintă lungimea maximă a unui drum care pornește din celula de coordonate *I* și *J*, iar LEG_{IJ} poate avea una dintre valorile 0 , 1 sau 2.

Valoarea 0 indică faptul că următoarea celulă care face parte din drum este înspre *Sud*, valoarea 1 indică faptul că următoarea celulă care face parte din drum este înspre *Est*, iar valoarea 2 indică faptul că celula corespunzătoare este ultima celulă a drumului.

Algoritmul descris în pseudocod este:

```
pentru I ← 1, N execută
  L_IM ← 1
  LEG_IM ← 2
sfârșit pentru
pentru J ← 1, M execută
  L_NJ ← 1
  LEG_NJ ← 2
sfârșit pentru
pentru I ← N - 1, 1 pas -1
                        execută
  pentru J ← M - 1, 1 pas -1
                        execută
    dacă A_IJ ≤ A_I+1,J atunci
      L1 ← L_I+1,J+1
    altfel
      L1 ← 0
    sfârșit dacă
    dacă A_IJ ≤ A_I,J+1 atunci
      L2 ← L_I,J+1+1
```

```
    altfel
      L2 ← 0
    sfârșit dacă
    dacă L1 > L2 atunci
      DIR ← 0
      L0 ← L1
    altfel
      DIR ← 1
      L0 ← L2
    sfârșit dacă
    dacă L0 = 0 atunci
      L_IJ ← 1
      LEG_IJ ← 2
    altfel
      L_IJ ← L0
      LEG_IJ ← DIR
    sfârșit dacă
  sfârșit pentru
sfârșit pentru
X ← 0
Y ← 0
MAX ← 0
pentru I ← 1, N execută
  pentru J ← 1, M execută
    dacă L_IJ > MAX atunci
      MAX ← L_IJ
      X ← I
      Y ← J
    sfârșit dacă
  sfârșit pentru
sfârșit pentru
scrie MAX
I ← X
J ← Y
cât timp (LEG_IJ ≠ 2) execută
  scrie I, J
  dacă LEG_IJ = 0 atunci
    I ← I + 1
  altfel
    J ← J + 1
  sfârșit dacă
sfârșit cât timp
scrie I, J
```

Algoritmul are ordinul de complexitate $O(N \cdot M)$.

Problema #4

Considerăm o tablă de joc de formă dreptunghiulară având N linii și M coloane. Piesele jocului se mișcă începând de la o poziție inițială aflată pe prima linie până la o poziție finală aflată pe ultima linie a tablei.

Este dată o listă de mutări posibile sub forma unor cvadruple: X_1, Y_1, X_2, Y_2 cu semnificația că piesa situată în punctul de coordonate (X_1, Y_1) se poate muta în punctul de coordonate (X_2, Y_2); în plus de fiecare dată avem relația: $X_1 < X_2$ (piesele avansează spre ultima linie a tablei).

Să se determine numărul secvențelor posibile de mutări pe care le poate efectua o piesă plecând de la un punct situat pe prima linie și ajungând la un punct situat pe ultima linie a tablei.

Date de intrare

Fișierul de intrare **mutari.in** are următoarea structură:

- pe prima linie se află numerele N, M, K care indică numărul de linii ale tablei, numărul de coloane ale tablei, respectiv numărul de elemente ale listei de mutări; cele trei numere sunt separate prin câte un spațiu;
- pe următoarele K linii se află cvadruple de forma X_1, Y_1, X_2, Y_2 care indică o mutare posibilă; cele patru numere sunt separate prin spații.

Date de ieșire

Fișierul de ieșire **mutari.out** trebuie să aibă următoarea structură:

- o singură linie pe care se va afla numărul NR, indicând numărul secvențelor posibile de mutări.

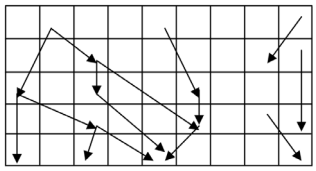
Exemplu

mutari.in

```
5 9 15
1 2 3 1
1 2 2 3
2 3 3 3
3 1 5 1
3 1 4 3
4 3 5 3
2 3 4 6
3 3 5 5
4 3 5 5
1 5 3 6
3 6 4 6
4 6 5 5
1 9 2 8
2 9 4 9
4 8 5 9
```

mutari.out

```
6
```



Restricții si precizări:

- $1 \le N$;
- $M \le 200$;
- $1 \le K \le 1000$.

Problema a fost propusă spre rezolvare elevilor din clasa a X-a la ediția 2005 a *Olimpiadei Locale de Informatică* din județul *Prahova*.

Rezolvare

Problema poate fi considerată o generalizare a problemei determinării numărului de modalitați de a urca o scară cu N trepte, la fiecare pas urcându-se una sau două trepte.

Folosim vectorii *XP*, *YP*, *XF*, *YF* pentru memorarea extremităților arcelor (aceste arce reprezintă mutările posibile) și o matrice auxiliară A cu N linii și M coloane.

Prezentăm în continuare rezolvarea în pseudocod:

```
sortează crescător vectorii XP, YP, XF,
  YF în funcție de valorile vectorului
                                    XP
inițializează cu 0 toate elementele
                                matricei A
pentru J ← 1, K execută
  dacă XP_J = 1 atunci
    A_1,YP_J ← 1
  sfârșit dacă
sfârșit pentru
pentru J ← 1, K execută
  A_XF_J,YF_J ← A_XF_J,YF_J + A_XP_J,YP_J
sfârșit pentru
S ← 0
pentru J ← 1, M execută
  S ← S + A_NJ
sfârșit pentru
scrie S
```

Algoritmul are ordinul de complexitate $O(K \cdot \log K + N \cdot M)$. Datorită existenței relațiilor de recurență și a rezolvării lor în modul *bottom-up* putem spune că problema se încadrează în tehnica programării dinamice.

Problema #5

Fie $X = (X_1, X_2, ..., X_n)$ și $Y = (Y_1, Y_2, ..., Y_m)$ două șiruri care conțin n, respectiv m numere întregi. Determinați un subșir comun al acestor șiruri, care are lungimea maximă.

Exemplu

```
X = (2,7,7,6,2,8,4,1,3,5,6)
Y = (10,2,5,6,7,7,4,3,5,8)
```

O soluție este:

```
Z = (2,7,7,4,3,5)
```

Rezolvare

Fie $Z = (Z_1, Z_2, ..., Z_{k-1}, Z_k)$ un astfel de subșir. Există o pereche (i, j) pentru care avem $Z_k = X_i$ și $Z_k = Y_j$. În acest caz Z este optim pentru subsecvențele $X_1,..., X_i$ și $Y_1, ..., Y_j$. De asemenea, există o pereche (i', j') pentru care avem $Z_{k-1} = X_{i'}$ și $Z_{k-1} = Y_{j'}$. Mai mult, vom avea întotdeauna $i' < i$ și $i' < i$. În acest caz $Z_1, ..., Z_{k-1}$ este optim pentru subsecvențele $X_1, ..., X_{i'}$ și $Y_1, ..., Y_{j'}$.

Se aplică metoda înapoi; trebuie să ajungem pas cu pas de la perechea (i', j') la perechea (i, j).

Considerăm toate subsecvențele de forma $X_1,..., X_i$ și $Y_1, ..., Y_j$. Reținem pentru o pereche de indici (i, j) valoarea A_{ij} care indică lungimea maximă a unui subșir comun al șirurilor $X_1,..., X_i$ și $Y_1, ..., Y_j$.

Pentru $i = 0$ sau $j = 0$ vom avea întotdeauna $A_{0k} = A_{h0}=0$, pentru toate valorile k și h pentru care $1 \le k \le m$ și $1 \le h \le n$.

Pe cazul general, dacă $X_i = Y_j$, atunci $Z_k = X_i = Y_j$ și $A_{ij} = A_{i-1,j-1} + 1$, deoarece $i' < i - 1$ și $j' < j - 1$.

Dacă $X_i \neq Y_j$ trebuie să eliminăm unul dintre capete. S-ar putea să avem $X_i = Y_{j-1}$ sau $Y_j = X_{i-1}$ (este posibil ca ambele egalități să fie adevărate simultan). În general, putem considera că $A_{ij} = \max(A_{i,j-1}, A_{i,j-1})$.

Se observă că matricea A se poate construi în urma unei parcurgeri pe linii și coloane.

Prezentăm în continuare rezolvarea în pseudocod:

```
pentru i ← 1, n execută
  A_i0 ← 0
sfârșit pentru
pentru j ← 1, m execută
  A_0j ← 0
sfârșit pentru
pentru i ← 1, n execută
  pentru j ← 1, m execută
    dacă X_i = Y_j atunci
      A_ij ← A_i-1,j-1 + 1
    altfel
      dacă A_i-1,j > A_i,j-1 atunci
        A_ij ← A_i-1,j
      altfel
        A_ij ← A_i,j-1
      sfârșit dacă
    sfârșit dacă
  sfârșit pentru
sfârșit pentru
```

Subșirul propriu-zis se construiește tot pe baza matricei A. Se parcurge un drum de la punctul de coordonate (n, m) către marginile din stânga și de sus ale matricei. La fiecare pas, dacă $X_i = Y_j$ se reține această valoare, altfel se merge către maximul coordonatelor precedente $(I - 1, J)$ și $(I, J - 1)$. Prezentăm rezolvarea în pseudocod:

```
i ← n
j ← m
cât timp i > 0 și j > 0 execută
  dacă X_i = Y_j atunci
    scrie X_i
    I ← I - 1
    J ← J - 1
  altfel
    dacă A_i-1,j = A_ij atunci
      i ← i - 1
    altfel
      j ← j - 1
    sfârșit dacă
  sfârșit dacă
sfârșit cât timp
```

Problema #6

Un graf orientat pe niveluri are nodurile împărțite în $k \ge 2$ mulțimi V_1, V_2, ..., V_k.

În prima mulțime se află nodul sursă s, în ultima mulțime se află nodul terminal t. Arcele unesc noduri din mulțimi succesive și au costuri pozitive.

Se dorește determinarea unui drum de cost minim de la sursă la destinație.

Exemplu

$V_1 = \{1\}$, $V_2 = \{2, 3, 4, 5\}$, $V_3 = \{6, 7, 8\}$, $V_4 = \{9, 10, 11\}$, $V_5 = \{12\}$.
Arce: (1, 2) cost 9; (1, 3) cost 7; (1, 4) cost 3; (1, 5) cost 2; (2, 6) cost 4; (2, 7) cost 2; (2, 8) cost 1; (3, 6) cost 2; (3, 7) cost 7; (4, 8) cost 11; (5, 7) cost 11; (5, 8) cost 8; (6, 9) cost 6; (6, 10) cost 5; (7, 9) cost 4; (7, 10) cost 3; (8, 10) cost 5; (8, 11) cost 6; (9, 12) cost 4; (10,12) cost 2; (11,12) cost 5.

Un drum de cost minim este: 1, 2, 7, 10, 12 și are costul 16.

Rezolvare

Se aplică metoda înainte. Relația de recurență este: $cost_{nivel, J} = \min\{c_{JL} + cost_{nivel+1, L}\}$ pentru L în $V_{nivel+1}$ și (J, L) arc. Algoritmul de determinare a costului minim, în pseudocod, este:

```
cost_N ← 0
pentru J ← N - 1, 1 pas -1
                          execută
  parcurgem lista de adiacentă a
                          nodului J
  fie K nodul pentru care valoarea
            c_JK + cost_K este minimă
  cost[J] ← c_JK + cost_K
  d[J] ← K
sfârșit pentru
```

Pentru determinarea drumului de cost minim putem utiliza algoritmul:

```
drum_1 ← 1
drum_H ← N
pentru J ← 2, H - 1 execută
  drum_J ← d_drum_J-1
sfârșit pentru
```

Ordinul de complexitate al algoritmului este $O(N + M)$ dacă memorăm graful prin liste de adiacență. Algoritmul se poate generaliza pentru arce între mai multe niveluri.

Bibliografie

1. Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, *Computer Algorithms,* Computer Science Press 1998.
2. Emanuela Cerchez, *Informatică*, Editura Polirom, 2002
3. ***, probleme propuse la concursurile de programare.