Münster, Germania

Olimpiada de informatică a EUROPEI CENTRALE

În continuare vă prezentăm soluțiile problemelor propuse spre rezolvare la acest concurs. Aceste soluții au fost realizate de redacția GInfo pe baza soluțiilor oficiale din limba engleză prezentate de către autorii problemelor.

P060307: Hanoi

Această problemă se reduce la rezolvarea altor două subprobleme. Prima subproblemă constă în a determina tija destinație pentru discuri, iar a doua constă în a determina numărul total de mutări care vor fi efectuate.

Prima subproblemă este relativ simplă, deoarece întotdeauna este mai avantajos să lăsăm discul N (cel mai mare) pe tija pe care acesta se află inițial. Altfel, ar trebui să se mute discul N pe o altă tijă care în momentul respectiv ar trebui să fie liber. Această mutare nu va schimba starea jocului (în afară de renumerotarea tijelor) și astfel nu poate face parte din soluția optimă. În continuare vom nota această tijă destinație prin *dest*.

În continuare vom prezenta modul de rezolvare a celei de-a doua subprobleme care constă în determinarea numărului de mutări.

O abordare algoritmică simplă va fi dedusă din următoarele observații:

- în timpul mutării discului i, pozițiile discurilor N, N - 1, ..., i + 1 (având diametre mai mari) nu au importanță. De exemplu, dacă discurile N, N - 1, ..., i se situează deja pe tija *dest*, nu are sens să se mute nici unul dintre ele încă o dată.
- în scopul de a muta discul i în locul său final pe tija *dest*, toate discurile N, N - 1, ..., i + 1 trebuie să fie deja prezente. Rezultă că discurile trebuie mutate în locul lor final în ordinea descrescătoare a dimensiunilor lor.
- dacă discurile N, N - 1, ..., i + 1 se află deja pe tija *dest*, toate discurile i - 1, ..., 1 trebuie să nu se afle pe aceeași tijă pe care se află discul i, și de asemenea nici pe tija *dest*, datorită regulilor de mutare. Rezultă că aceștia, mai întâi trebuie să fie mutate pe o altă tijă.

Toate aceste restricții conduc la următorul algoritm recursiv:

```
1  Subalgoritm MutăDisc(disc, TijăDest,
                                    TijăDisc):
2    TijăSursă ← TijăDisc[disc]
3    TijăAux ← 1 + 2 + 3 - TijăSursă - TijăDest
4    pentru AltDisc = disc - 1, 1 execută:
5      MutăDisc(AltDisc,TijăAux)
6    sfârșit pentru
7    NumMutări ← rest[(NumMutări + 1)/1000000]
8    TijăDisc[disc] ← TijăDest {mutare directă}
9  sfârșit subalgoritm
10
11 Algoritm Soluție(N, TijăDisc)
12   NumMutări ← 0
13   pentru disc = N - 1, 1, -1 execută:
14     MutăDisc(disc, TijăDisc[N])
15   sfârșit pentru
16 {NumMutări conține acum rezultatul}
17 sfârșit algoritm
```

Pe baza observațiilor de mai sus este evident că acest algoritm generează ordinea corectă a mutărilor. De asemenea, este vizibil că acest lucru încă nu este suficient. Numărul `NumMutări` al mutărilor nu crește niciodată cu mai mult decât 1, deci subalgoritmul `MutăDisc` este apelat o dată pentru fiecare mutare. Deoarece este știut că numărul mutărilor în cazul problemei *turnurilor din Hanoi* crește exponențial cu $O(2^N)$, timpul de execuție va fi cel puțin $O(2^N)$, ceea ce este mult prea mult pentru $N \leq 100.000$.

Așa cum a fost precizat și în enunț, sunt necesare exact 2^i - 1 mutări pentru a muta discurile i, i - 1, ..., 1 de pe o tijă pe alta (aceleași tije pentru toate discurile). Deci, în loc să realizăm aceste 2^i - 1 mutări separat, vom muta direct aceste i discuri pe tija destinație și vom mări numărul mutărilor cu 2^i - 1.

Când mutăm discul i pe tija X, acest lucru îl facem fie deoarece trebuie să eliberăm tija pentru un disc de diame-

tru mai mare, fie deoarece tija *X* este tija destinație *dest*. În ambele cazuri vrem să mutăm toate discurile mai mici *i* - 1, *i* - 2, ..., 1 pe *X* unul după altul. În subalgoritmul `MutăDisc`, când am ajuns în linia 7, toate aceste discuri mai mici se află pe aceeași tijă. Deoarece acestea vor fi mutate pe `TijăDest` oricum, schimbăm subalgoritmul `MutăDisc` în `MutăDiscuri` cu care le mutăm, și concomitent, folosind ordinea în care se află, calculăm în mod direct numărul mutărilor necesare. Aceasta conduce la următoarea soluție îmbunătățită:

```
1  {mută toate discurile 1, 2, ..., maxDisc pe TijăDest}
2  Subalgoritm MutăDiscuri(maxDisc, TijăDest,
                                        TijăDisc):
3    TijăSursă ← TijăDisc[maxDisc]
4    TijăAux ← 1 + 2 + 3 - TijăSursă - TijăDest
5    dacă maxDisc < 1 atunci
6      *ieșire
7    sfârșit dacă
8    dacă (TijăSursă = TijăDest) atunci
9      MutăDiscuri(maxDisc - 1, TijăDest)
10   altfel
11     MutăDiscuri(maxDisc - 1, TijăAux)
12   sfârșit dacă
13 {acum maxDisc poate fi mutat direct de pe TijăSursă}
14 {pe TijăDest (o mutare) și discurile mai mici pot fi}
15 {mutate de pe TijăAux pe TijăDest folosind 2^maxDisc
16 {- 1 mutări, rezultă un total de 2^maxDisc mutări}
17  pentru i = 1, maxDisc execută:
18    TijăDisc[i] ← TijăDest
19    NumMutări ← rest[(NumMutări +
              PutereaLuiDoi(disc)) / 1000000]
20  sfârșit pentru
21 sfârșit subalgoritm
22
23 Algoritm Soluție(N, TijăDisc)
24   NumMutări ← 0
25   MutăDiscuri(N - 1, TijăDisc[N])
26 sfârșit algoritm
```

În această versiune îmbunătățită, subalgoritmul `MutăDiscuri` nu se autoapelează decât o singură dată pentru fiecare valoare `maxDisc` egală cu *N*, *N* - 1, ..., 1, adică în total de *N* ori. Pentru fiecare nivel al recursiei timpul de execuție este în medie $O(N)$ datorită ciclului **pentru** din linia 17.

Ideea este de a calcula puterile lui 2 și de a le memora într-un șir, acesta realizându-se în timp constant. Bineînțeles, aceștia se pot calcula modulo 106 pentru a evita întregi mai mari decât cei reprezentabili pe 32 de biți. Astfel, timpul total de execuție al algoritmului este $O(N^2)$, suficient de rapid doar pentru primele 12 teste.

Prin verificarea explicită a faptului că discurile 1, ..., *i* care trebuie mutate se află sau nu pe aceeași tijă cu un ciclu de tip **for**, se poate realiza un algoritm $O(N^3)$ care rezolvă primele 6-8 teste. (Această soluție nu o vom prezenta).

În continuare vom îmbunătăți algoritmul $O(N^2)$ pentru a realiza unul *liniar*. Pentru a realiza acest lucru vom observa că ciclul din linia 17 este inutil și poate fi omis, deoarece valorile `TijăDisc[]` care se scriu în acest ciclu, *nu se citesc niciodată*. Valorile `TijăDisc[]` sunt citite doar *înainte* de apelurile recursive, deci nu are importanță dacă acestea sunt modificate în timpul sau după aceste apeluri. Deoarece pe noi ne interesează doar numărul mutărilor necesare și știm că în final toate discurile se vor afla pe tija destinație, actualizarea pozițiilor discurilor nu ne interesează și poate fi omisă. Deoarece acest ciclu este singurul care în subalgoritmul `MutăDiscuri` consumă mai mult decât un timp *constant*, și acesta este apelat doar de *N* ori, avem o soluție $O(N)$. Evident, nici o soluție nu poate fi mai bună (excepție făcând un factor constant), deoarece cu citirea datelor oricum se consumă un timp $O(N)$, iar scrierea rezultatului se face, de asemenea, într-un timp $O(N)$ deoarece numărul care reprezintă soluția este aproximativ egal cu $O(2^N)$, deci are $O(N)$ cifre.

Soluția, care are ordinul de complexitate $O(N)$, este:

```
1  {mută toate discurile 1, 2, ..., maxDisc pe TijăDest}
2  Subalgoritm MutăDiscuri(maxDisc, TijăDest,
                    TijăDisc, PutereaLuiDoi):
3    TijăSursă ← TijăDisc[maxDisc]
4    TijăAux ← 1 + 2 + 3 - TijăSursă - TijăDest
5    dacă maxDisc < 1 atunci
6      *ieșire
7    sfârșit dacă
8    dacă (TijăSursă = TijăDest) atunci
9      MutăDiscuri(maxDisc - 1, TijăDest)
10   altfel
11     MutăDiscuri(maxDisc - 1, TijăAux)
12   sfârșit dacă
13 {acum maxDisc poate fi mutat direct de pe TijăSursă}
14 {pe TijăDest (o mutare) și discurile mai mici pot fi}
15 {mutate de pe TijăAux pe TijăDest folosind 2^maxDisc
16 {- 1 mutări, rezultă un total de 2^maxDisc mutări}
17  pentru i = 1, maxDisc execută:
18    TijăDisc[i] ← TijăDest
19    NumMutări ← rest[(NumMutări +
              PutereaLuiDoi(disc)) / 1000000]
20  sfârșit pentru
21 sfârșit subalgoritm
22
23 Algoritm Soluție(N, TijăDisc)
24   NumMutări ← 0
25   PutereaLuiDoi[0] ← 1
26   pentru i=1, N execută:
27     PutereaLuiDoi[i] ← rest[(2 *
              PutereaLuiDoi[i - 1]) / 1000000]
28   sfârșit pentru
25   MutăDiscuri(N - 1, TijăDisc[N],
                              PutereaLuiDoi)
26 sfârșit algoritm
```

P060308: Cursa

Această problemă constă în două subprobleme independente.

Prima subproblemă constă în determinarea numărului total de depășiri. Deoarece toate vitezele navelor sunt constante, după un anumit moment de timp nu vor mai avea loc depășiri, iar navele vor fi sortate în ordinea crescătoare a vitezelor. Două nave având aceeași viteză vor fi ordonate în funcție de pozițiile lor de pornire.

Pentru a efectua sortarea în funcție de viteza navelor, se va folosi algoritmul de sortare *bubblesort*, deoarece este un algoritm stabil și navele având aceeași viteză își vor păstra ordinea după sortare. Datorită restricțiilor impuse asupra fișierului de intrare, adică navele sunt sortate în funcție de poziția de pornire, navele având aceeași viteză, după sortare, vor ocupa același loc.

De fapt, prima subproblemă este echivalentă cu a determina numărul total de interschimbări efectuate de algoritmul de sortare *bublesort*.

Cel mai simplu algoritm constă în a parcurge șirul de nave în ordinea pozițiilor lor în cursă. La fiecare pas se va actualiza numărul de nave care preced poziția curentă și câte dintre acestea vor depăși nava curentă. Această soluție are ordinul de complexitate $O(N \cdot V_{max})$, unde V_{max} reprezintă viteza maximă pentru o navă impusă în enunțul problemei. În continuare prezentăm algoritmul de rezolvare, în pseudocod, pentru această subproblemă:

```
nDepășiri ← 0
pentru i = VMAX, 1, -1 execută:
   nNaveCuV[i] ← 0
sfârșit pentru
pentru i = 0, N - 1 execută:
   nNaveCuV[vitezănava[i]] ← nNaveCuV[vitezănava[i]] + 1
sfârșit pentru
pentru i = N - 1, 0, -1 execută:
   nNaveCuV[vitezănava[i]] ← nNaveCuV[vitezănava[i]] - 1
//count passes of ships that are behind position i
   pentru v = viteză[nava[i]] + 1, VMAX execută:
      nDepășiri ← nDepășiri + nNaveCuV[v]
   sfârșit pentru
   nDepășiri ← rest[nDepășiri / 1000000];
sfârșit pentru
```

A doua subproblemă constă în a determina primele 10000 de depășiri în ordine cronologică.

În cel mai rău caz au loc $O(N^2)$ depășiri. Un algoritm care compară toate perechile de nave determinând depășirile posibile și sortând aceste depășiri are ordinul de complexitate $O(N^2 \cdot \log N)$ și nu s-ar fi încadrat în timp.

Algoritmul pe care-l vom prezenta în continuare are ordinul de complexitate $O(N + c \cdot \log N)$, unde c reprezintă numărul total de depășiri care trebuie determinate.

Se observă că o navă i poate depăși nava j dacă nava i este prima în spatele navei j și are o viteză mai mare. Ideea de rezolvare constă în a menține o listă de nave și un *heap* care conține, pentru fiecare navă i, nava j care se află imediat în fața ei și momentul de timp când nava i va depăși nava j. Dacă nava i nu poate depăși nava j, atunci acest eveniment nu va fi pus în *heap*. Elementul care se află în vârful *heap*-ului reprezintă următoarea depășire care va avea loc.

La începutul algoritmului, lista de nave se ordonează în funcție de poziția lor la start și se va construi *heap*-ul, elementul din vârful aceastuia reprezentând prima depășire care are loc.

În continuare va trebui să simulăm primele 10000 de depășiri, astfel, la fiecare pas, efectuăm secvențele următoare de instrucțiuni:

- eliminăm din *heap* primul eveniment de forma nava i depășește nava j și îl vom tipări;
- dacă nava i depășește altă navă în viitor, vom insera în *heap* acest eveniment;
- dacă nava care se află imediat în spatele navei j va depăși nava i (care va depăși nava j înainte de a depăși nava i) va trebui să actualizăm acest eveniment, lucru care se realizează prin urcarea acestuia în *heap*.

Algoritmul de rezolvare a acestei probleme în pseudocod este următorul:

```
sortează(nave)
pentru i = 1, N - 1 execută:
   dacă nave[i] depășește nave[i+1] atunci
      pune eveniment în heap
   sfârșit dacă
sfârșit pentru
pentru i = 1, 10000 execută:
   dacă heap-ul este gol atunci
      *întrerupe ciclu
   sfârșit dacă
   scoate primul eveniment din heap
   tipărește eveniment
   interschimbă navele eveniment.i și eveniment.j
   dacă nave[eveniment.i] depășește nave[eveminent.i+1] atunci
      pune acest eveniment în heap;
   sfârșit dacă
   dacă nave[eveniment.j-1] depășește nave[eveminent.j] atunci
      actualizează evenimentul navei j
   sfârșit dacă
sfârșit pentru
```

P060309: Pătrat

Algoritmul de rezolvare pentru această problemă, pe care-l vom prezenta în continuare, nu efectuează mai mult de $3 \cdot N$ interogări, și astfel se încadrează între limite.

Mai întâi, trebuie să observăm că fiecare drum dintre nodul (1, 1) și un nod v dat are aceeași lungime.

Notăm cu $D(v)$ distanța dintre (1, 1) și v. Dacă $D(v) > L$, știm că nu pot fi soluții nici nodurile din dreapta lui v, nici cele situate sub v, deoarece distanțele până la nodul (1, 1) sunt mai mari.

Vom începe explorarea grafului pornind de la nodul (1, 1) și ne vom muta în jos până la atingerea unui nod v pentru care $D(v) \geq L$. Dacă $D(v) = L$, am găsit o soluție. Dacă $D(v) > L$, știm că nici nodul curent și nici unul dintre nodurile aflate în dreapta nodului curent sau sub el, nu pot fi soluții. Mai mult, știm că distanța până la nodul aflat deasupra este prea mică. Astfel, vom trece la nodul din dreapta-sus.

Acum vom prelucra iterativ vecinul din dreapta, de fiecare dată când distanța dintre nodul curent și acesta este prea mică sau prelucrăm vecinul de deasupra dacă distanța este prea mare. De fiecare dată se poate extrage un grup de noduri din mulțimea soluțiilor posibile, așa cum se poate vedea în figura 1.

Dacă trebuie să avansăm într-o direcție în care nu mai există noduri, putem trage concluzia că pentru acest test nu există soluție.

Am uitat un caz: Ce se întâmplă atunci când nu putem ajunge, mutându-ne în jos, la un nod față de care distanța este mai mare sau egală cu L? Răspunsul este simplu: vom continua mutările în dreapta și vom proceda ca mai sus până când găsim un nod v pentru care avem $D(v) > L$.

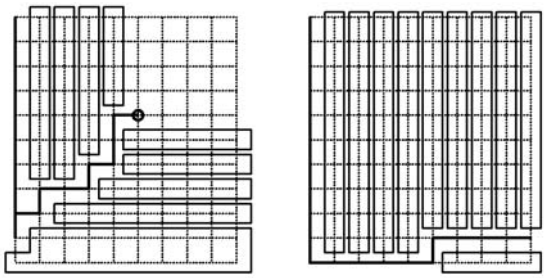

Figura 1

Afirmația de la începutul analizei este adevărată, adică nu vom executa decât cel mult $3 \cdot N$ prelucrări, deoarece ne vom deplasa în jos de cel mult N - 1 ori, ne vom deplasa în dreapta de cel mult N - 1 ori și în sus de cel mult N - 1 ori.

P060310: Colierul de perle

Problema constă în găsirea mutărilor care conduc la câștig. O mutare conduce la câștig, dacă jucătorul mută optim în continuare.

Dacă presupunem că ambii jucători joacă optim, mutările fie conduc la câștig, fie la pierderea jocului din moment ce în acest joc nu există remiză.

În plus, o mutare conduce la câștig, dacă un pitic poate da șiragul de mărgele unui alt pitic din tribul lui care are la dispoziție o mutare de câștig, sau dacă el poate da șiragul de mărgele unui alt pitic din celălalt trib care nu are la dispoziție o mutare de câștig.

O stare a jocului este determinată de starea șiragului de mărgele și a piticului curent:

```
Subalgoritm AreMutaredeCâștig(c1c2...ck,p)
  returnează (∃p' ∈ list_p,c1 astfel încât (culoare_p =
  culoare_p' și AreMutaredeCâștig(c2...ck,p'))
  sau (culoare_p ≠ culoare_p' și nu
  AreMutaredeCâștig(c2...ck,p')))
sfârșit subalgoritm.
```

Datorită restricțiilor impuse în enunțul problemei, `AreMutaredeCâștig(ck,p')` returnează întotdeauna valoarea **adevărat** deoarece, la sfârșitul recursiilor, în listă rămâne numai diamantul, iar `p'` este sigur un pitic de culoare verde.

În subalgoritmul anterior, $list_{p,c}$ reprezintă lista albă pentru piticul `p`, dacă `c` are valoarea `1` și lista neagră pentru piticul `p`, dacă `c` are valoarea `0`, iar $culoare_p$ reprezintă culoarea piticului `p`.

Pe baza acestor observații se pot determina ușor mutările care conduc la câștig folosind metoda programării dinamice într-o abordare *bottom-up*.

Pentru a determina mutările care duc la câștig vom folosi un tablou bidimensional auxiliar de dimensiune $L \times N$ iar fiecare element $c_{i,p}$ al acestuia va avea valoarea `0` dacă, pornind din starea curentă (s-au efectuat i-1 mutări), piticul p nu are mutare de câștig și valoarea 1 dacă, pornind din starea curentă, piticul p are mutare de câștig.

Acest tablou se construiește traversând indicii din șirag în ordine inversă, adică de la L la 1.

În continuare, având construit tabloul c, la fiecare mutare a unui pitic verde se va alege un pitic de pe lista acestuia, indicată de mărgeaua la care s-a ajuns, astfel încât să fie verificată condiția impusă în cadrul subalgoritmului prezentat anterior, adică fie un pitic verde care să aibă mutare de câștig, fie un pitic roșu care să nu aibă mutare de câștig.

P060311: Registru de deplasare

Ideea de rezolvare a acestei probleme constă în a reprezenta poarta *XOR* ca o funcție liniară cu valori în mulțimea $Z_2 = \{0, 1\}$. Plecând de la această idee vom avea de rezolvat un sistem liniar cu N ecuații și N necunoscute folosind metoda lui *Gauss*.

Observăm că, în mulțimea Z_2, înmulțirea a două numere este echivalentă cu aplicarea operatorului binar **ȘI** între numere, iar adunarea este echivalentă cu aplicarea operatorului binar **SAU-Exclusiv**.

Această observație ne permite să descriem efectul combinat al comutatoarelor s_i și al porții *XOR* folosind următoarea expresie liniară:

$$s_1 \cdot a_1 + s_2 \cdot a_2 + ... + s_N \cdot a_N.$$

Fie o_i ($i \geq 1$) cel de-al i-lea bit transmis de poarta *XOR* (care reprezintă de fapt cea de-a i-a valoare de pe cea de-a doua linie a fișierului de ieșire). Primii N biți transimiși de poarta *XOR*, o_1, o_2, ..., o_N, reprezintă valorile inițiale ale biților registrului de deplasare a_1, a_2, ..., a_N. Primul bit transmis care necesită calcule este o_{N+1}. Când o_{N+1} este calculat, biții a_i ai registrului de deplasare sunt ocupați (în

ordine inversă) de valorile o_1, o_2, ..., o_N. Pe baza expresiei liniare construite anterior, obținem prima ecuație a sistemului:

$$o_{N+1} = s_N \cdot o_1 + s_{N-1} \cdot o_2 + ... + s_1 \cdot o_N.$$

Pentru a calcula valorile o_{N+2}, o_{N+3}, ..., $o_{2 \cdot N}$ se vor construi ecuații similare. Vom inversa ordinea comutatoarelor definind $s'_i = s_{n-i+1}$ pentru a putea manipula mai ușor aceste ecuații. Astfel, avem de rezolvat sistemul de ecuații:

$$s'_1 \cdot o_1 + s'_2 \cdot o_2 + ... + s'_N \cdot o_N = o_{N+1}$$
$$s'_1 \cdot o_2 + s'_2 \cdot o_3 + ... + s'_N \cdot o_{N+1} = o_{N+2}$$
$$...$$
$$s'_1 \cdot o_N + s'_2 \cdot o_{N+1} + ... + s'_N \cdot o_{2 \cdot N-1} = o_{2 \cdot N}$$

În continuare vom construi matricea M de dimensiune $N \times N + 1$ care va conține coeficienții ecuațiilor și termenii liberi.

Rezolvarea sistemului de ecuații constă în a obține valoarea 0 sub diagonala principală prin transformări liniare ale liniilor și interschimbări ale acestora și reconstituirea valorilor s'_i în ordine inversă conform metodei lui *Gauss* al cărei ordin de complexitate este $O(N^3)$.

Dacă sistemul de ecuații nu are soluție, atunci se va scrie în fișierul de ieșire valoarea -1 și în caz contrar se vor scrie valorile s'_i, $1 \le i \le N$, în ordine inversă.

P060312: Călătoria

Problema cere să se afișeze toate cele mai lungi subsecvențe comune ale două șiruri de caractere. Prima idee care ar putea fi fructificată constă în utilizarea *metodei programării dinamice*, în forma ei standard, pentru a determina lungimea maximă a subsecvențelor comune, urmând ca apoi să se construiască, pe rând, fiecare astfel de subsecvență.

Fie $A = (a_1, a_2, ..., a_n)$ primul șir și $D = (b_1, b_2, ..., b_m)$ cel de-al doilea șir. Fie $c_{i,j}$ lungimea celei mai lungi subsecvențe comune ale șirurilor $(a_1, a_2, ..., a_i)$ și $(b_1, b_2, ..., b_j)$. Atunci:

$$c_{i,j} = \begin{cases} 0, & \text{dacă } i = 0 \text{ sau } j = 0 \\ c_{i-1,j-1} + 1, & \text{dacă } i, j > 0 \text{ și } a[i] = b[j] \\ \max(c_{i-1,j}, c_{i,j-1}), & \text{dacă } i, j > 0 \text{ și } a[i] \neq b[j] \end{cases}.$$

Lungimea celei mai lungi subsecvențe comune a șirurilor A și B este dată de $c_{n,m}$.

Pe baza acestor recurențe algoritmul de programare dinamică este următorul:

```
...
pentru i = 1, n execută:
  c[i,0] ← 0
sfârșit pentru
pentru j=0, m execută:
  c[0,j] ← 0
sfârșit pentru
pentru i=1, n execută:
  pentru j=1, m execută:
    dacă a[i] = b[j] atunci
      c[i,j] ← c[i-1,j-1]+1
    altfel
      c[i,j] ← Max(c[i-1,j],c[i,j-1])
    sfârșit dacă
  sfârșit pentru
sfârșit pentru
...
```

În scopul construirii celei mai lungi subsecvențe, trebuie să reconstruim rezultatul pe baza tabloului *c*. Această reconstituire se realizează cu algoritmul:

```
Subalgoritm Reconstituire(i,j,secvență):
  dacă i = 0 sau j = 0 atunci
    adaugă secvență mulțimii celor mai lungi
                                    subsecvențe
    *ieșire
  sfârșit dacă
  dacă a[i] = b[j] atunci
    alipește a[i] secvenței
    Reconstituire(i-1,j-1,secvență)
    *ieșire
  sfârșit dacă
  dacă c[i-1,j] = c[i,j] atunci
    Reconstituire(i-1,j,secvență)
  sfârșit dacă
  dacă c[i,j-1] = c[i,j] atunci
    Reconstituire(i,j-1,secvență)
  sfârșit dacă
sfârșit subalgoritm
```

Studiind mai atent pseudocodul de mai sus, observăm că în cazul în care $c_{i-1,j} = c_{i,j-1}$, subalgoritmul de reconstituire se apelează de două ori. Chiar dacă avem cel mult 1000 de cele mai lungi subsecvențe comune, acest lucru nu spune nimic despre numărul de posibilități în care acestea se pot construi. Modalitatea de reconstituire de mai sus va încerca toate posibilitățile de a construi toate cele mai lungi subsecvențe comune. De exemplu, dacă avem două șiruri de caractere "aaaaaaabccccccccd" și "abbbbbbbcddddddd" există 1778966 posibilități de a construi singura cea mai lungă subsecvență comună "abcd".

Cum am putea evita construirea aceleiași secvențe de mai multe ori? Am putea să le construim în paralel cu determinarea lungimilor. Avem nevoie de un tablou de mulțimi în care să păstrăm toate cele mai lungi subsecvențe comune ale șirurilor $(a_1, a_2, ..., a_i)$ și $(b_1, b_2, ..., b_j)$.

Algoritmul bazat pe programare dinamică acum arată în felul următor:

```
...
pentru i = 1, n execută:
  c_{i,0} ← 0
  mulțime_{i,0} ← ∅
sfârșit pentru
pentru j = 0, m execută:
  c[0,j] ← 0
```

```
    mulțime[0,j] ← ∅
  sfârșit pentru
  pentru i = 1, n execută:
    pentru j=1, m execută:
      dacă a_i = b_j atunci
        c_{i,j} ← c_{i-1,j-1} + 1
        mulțime[i,j] ← mulțime[i-1,j-1]
        alipește a_i la sfârșitul tuturor secvențelor din
                                          mulțime_{i,j}
      altfel
        c_{i,j} ← Max(c_{i-1,j}, c_{i,j-1})
        dacă c_{i-1,j} > c_{i,j-1} atunci
          mulțime_{i,j} ← mulțime_{i-1,j}
        altfel
          dacă c_{i,j-1} > c_{i-1,j} atunci
            mulțime_{i,j} ← mulțime_{i,j-1}
          altfel
            mulțime_{i,j} ← mulțime_{i-1,j} ∪ mulțime_{i,j-1}
          sfârșit dacă
        sfârșit dacă
      sfârșit dacă
    sfârșit pentru
  sfârșit pentru
  ...
```

A mai rămas să analizăm dimensiunea spațiului de memorare, necesară tabloului de mulțimi. Există cel mult 1000 de cele mai lungi subsecvențe comune având lungimea maximă egală cu 80, și astfel dimensiunea tabloului este 80 × 80. Acesta ar necesita 512 *MB* dacă s-ar folosi memoria statică, sau un pic mai puțină dacă în loc de un tablou alocat static s-ar folosi o listă înlănțuită, alocată dinamic. De altfel, cu tabloul plin s-ar fi rezolvat nouă din cele 10 teste, folosind memoria pusă la dispoziție, având dimensiunea 16 *MB*.

În continuare observăm că în ciclul "**pentru** i = 1, n", la un moment dat, numai linia curentă și cea precedentă sunt necesare în timpul prelucrării. Astfel este posibil să se folosească un tablou de dimensiune 2 × 80 și mulțimile vor fi indexate în tablou cu i *modulo* 2. Astfel forma finală a algoritmului este:

```
  ...
  pentru i = 1, n execută:
    c[i,0] ← 0
  sfârșit pentru
  pentru j =0, m execută:
    c_{0,j} ← 0
    mulțime_{0,j} ← ∅
    mulțime_{1,j} ← ∅
  sfârșit pentru
  pentru i = 1, n execută:
    pentru j=1, m execută:
      dacă a_i = b_j atunci
        c_{i,j} ← c_{i-1,j-1} + 1
        mulțime_{i mod 2,j} ← mulțime_{1-(i mod 2),j-1}
        alipește a_i la sfârșitul tuturor secvențelor din
                                          mulțime_{i mod 2,j}
      altfel
        c_{i,j} ← Max(c_{i-1,j}, c_{i,j-1})
        dacă c_{i-1,j} > c_{i,j-1} atunci
          mulțime_{i mod 2,j} ← mulțime_{1-(i mod 2),j}
        altfel
          dacă c_{i,j-1} > c_{i-1,j} atunci
            mulțime_{i mod 2,j} ← mulțime_{i mod 2,j-1}
          altfel
            mulțime_{i mod 2,j] ← mulțime_{1-(i mod 2),j} ∪
                                  mulțime_{i mod 2,j-1}
          sfârșit dacă
        sfârșit dacă
      sfârșit dacă
    sfârșit pentru
  sfârșit pentru
  Afișează toate secvențele din mulțime_{n mod 2,m}
  ...
```

Astfel am realizat o rezolvare și mai rapidă. Ideea este de a folosi o versiune modificată a algoritmului de reconstituire prezentat.

Vom încerca să construim doar acele cele mai lungi subsecvențe comune x în care x_i reprezintă prima apariție după litera x_{i-1} în ambele șiruri de caractere. Celelalte construcții nu ar conduce la altă cea mai lungă subsecvență comună. Pentru a obține rezultatul cerut, aplicăm căutare pe baza traversării în lățime.

Subalgoritmul de reconstituire este următorul:

```
Subalgoritm Reconstituire2(i,j,secvență):
  dacă i = 0 sau j = 0 atunci
    adaugă secvență mulțimii celor mai lungi
                                    subsecvențe
    *ieșire
  sfârșit dacă
  inserează (i,j) în coadă
  cât timp coada nu este vidă execută:
    (k,l) ← primul element din coadă
    dacă a_k = b_l atunci
      dacă Reconstituire2 nu a fost apelat pentru
      litera a_k alipită în fața lui secvență atunci
        alipește a_k la secvență
        Reconstituire2(i-1,j-1,secvență)
      altfel
        dacă c_{k-1,l} = c_{k,l} și (k-1,l) nu este deja în
        coadă atunci
          adaugă (k-1,l) în coadă
        sfârșit dacă
        dacă c_{k,l-1} = c_{k,l} și (k,l-1) nu este deja în
        coadă atunci
          adaugă (k,l-1) în coadă
        sfârșit dacă
      sfârșit dacă
    sfârșit dacă
  sfârșit cât timp
sfârșit subalgoritm
```