Plovdiv, iulie 2004

Olimpiada BALCANICĂ de Informatică 2004

Vă prezentăm în continuare soluțiile oficiale ale celor șase probleme propuse spre rezolvare la ultima ediție a Olimpiadei Balcanice de Informatică.

P040613: Monede

Problema poate fi rezolvată în mai multe moduri, dar fiecare modalitate de rezolvare va duce la obținerea răspunsului corect în timpul permis doar pentru o parte dintre teste.

O caracteristică oarecum neobișnuită a problemei este faptul că, indiferent care ar fi fost metoda aleasă, nu s-ar fi obținut punctajele corespunzătoare tuturor testelor.

Pentru a se obține punctajul maxim, era necesar să fie combinate cel puțin două dintre metodele care vor fi prezentate.

Așadar, putea fi implementat un algoritm bazat pe programarea dinamică și o căutare bine aleasă, sau algoritmul bazat pe programarea dinamică și metoda algebrică pe care o vom prezenta.

Căutare exhaustivă

O primă metodă constă în determinarea soluției optime și compararea ei cu soluția obținută folosind algoritmul *greedy* prezentat în enunț. Evident, această metodă directă poate fi aplicată numai pentru testele "mici".

Căutarea bine-aleasă

Această metodă constă în testarea anumitor cazuri speciale, cum ar fi căutarea unei soluții formate din monede de unul sau două tipuri sau căutarea unei soluții care nu este optimă, dar care este mai bună decât cea furnizată de algoritmul *greedy*.

Programare dinamică

Metoda pe care o vom prezenta va avea ordinul de complexitate $O(M \cdot S)$, iar spațiul de memorie necesar va avea ordinul $O(S)$.

Vom nota prin t_i numărul minim de monede care este suficient pentru a plăti suma i, folosind cele M tipuri de monede date.

Dacă valorile t_1, t_2, ..., t_{i-1} sunt deja calculate, valoarea t_i poate fi calculată pe baza următoarei observații: există un tip de monede a_j care este folosit pentru a obține suma i folosind un număr de monede; dacă folosim acea monedă și monedele folosite pentru a obține suma $i - a_j$, atunci vom obține suma i folosind un număr minim de monede.

Secvența în pseudocod care urmează va prezenta metoda descrisă.

Se folosește un vector t care conține S elemente și ale cărui valori au semnificația de mai sus.

Mai este necesar un vector p, care conține tot S elemente, iar p_i va conține tipul a_j care a fost ales pentru a se obține valoarea minimă pentru plata sumei i.

Numărul operațiilor efectuate este proporțional cu produsul $M \cdot S$.

```
algoritm determinăMinim
  t0 ← 0
  t1 ← 1
  p0 ← 0
  pentru i ← 2, S
                executută
    w ← ∞
    pentru j ← 1, M executută
      dacă aj ≤ i atunci
        k ← i - aj
        dacă w > tk + 1 atunci
          w ← tk + 1
          pi ← j
        sfârșit dacă
      sfârșit dacă
    sfârșit pentru
    ti ← w
  sfârșit pentru
sfârșit algoritm
```

Alți algoritmi

Există mai mulți algoritmi bazați pe metoda programării dinamice care pot duce la rezolvarea problemei, dar spațiul de memorie utilizat este mai mare sau numărul operațiilor efectuate este mai mare. De exemplu spațiul de memorie ajunge la ordinul $O(M \cdot S)$, sau ordinul de complexitate al algoritmului ajunge la $O(M \cdot S^2)$.

Metoda algebrică

Vom determina modul de plată a sumei a_k folosind doar monede de tipul a_1, a_2, ..., a_{k-1}.

Această sumă poate fi obținută astfel:

$$a_k = \sum_{i=1}^{k-1} b_{k,i} a_i$$

Pentru fiecare pereche de valori (k, j), vom nota:

$$P_{k,j} = a_j + \sum_{i=j}^{k-1} b_{k,i} a_i$$

Se poate demonstra faptul că dacă există o sumă care poate fi obținută folosindu-se un număr mai mic de monede decât numărul furnizat de algoritmul *greedy*, atunci există cel puțin o valoare $P_{k,j}$ pentru care această afirmație este de asemenea adevărată.

Cu alte cuvinte, numărul de monede obținut folosind algoritmul *greedy* este mai mare decât numărul de monede indicat în formula anterioară.

Așadar, este suficient să verificăm valorile $P_{k,j}$ obținute. Obținem un algoritm al cărui ordin de complexitate este $O(M^2)$, dar, în unele situații, s-ar putea ca algoritmul să nu ducă la rezolvarea problemei deoarece este posibil ca toate valorile găsite să nu fie cuprinse în intervalul $[x, y]$.

Date de test

Tabelul prezintă datele de test folosite în concurs. Se observă că pentru ultimele teste algoritmii al căror ordin de complexitate depinde de S nu sunt acceptabili.

#	M	a_M	x	y
1	7	21	26	50
2	12	234	1	999
3	13	313	513	548
4	13	313	5130	5480
5	30	205	201	999
6	42	899	999500	999999
7	44	800000	99500	999999
8	44	800000	1	999999
9	58	3000000	5010000	6000000
10	91	4000000	5000000	7000000

P040614: Echipa

Există o mulțime de posibilități de a rezolva această problemă. Soluțiile pot fi clasificate în trei categorii în funcție de ordinul de complexitate al algoritmilor folosiți pentru rezolvare. Vom prezenta în continuare câte un algoritm reprezentativ pentru fiecare dintre categorii.

Algoritmul banal

Cel mai intuitiv algoritm urmărește pur și simplu definițiile conceptelor de ***mai bun*** și ***excelent*** din enunțul problemei.

Vom parcurge toți concurenții și pentru fiecare concurent X vom determina dacă acesta este sau nu excelent.

Verificarea constă într-o nouă parcurgere a tuturor concurenților și, pentru fiecare concurent Y, se verifică dacă Y este mai bun decât X.

Datorită faptului că putem păstra cu ușurință în memorie pozițiile concurenților de la fiecare concurs, putem verifica dacă X este mai bun decât Y comparând pozițiile celor doi la cele trei concursuri.

Așadar, putem determina dacă X este sau nu excelent într-un timp de ordinul $O(N)$. Ca urmare, vom determina toți concurenții excelenți într-un timp de ordinul $O(N^2)$.

Este acum foarte simplu să numărăm concurenții excelenți. Obținem un algoritm care rulează în timp pătratic și care, în concurs, ar fi obținut 20-30 de puncte în funcție de detaliile de implementare și prezența sau absența optimizărilor.

Un alt algoritm simplu

Putem îmbunătăți ordinul de complexitate al algoritmului anterior dacă luăm în considerare următoarea observație: dacă Y nu este excelent (datorită faptului că există un concurent Z care este mai bun decât Y) atunci, în momentul în care verificăm dacă un concurent X este sau nu excelent, putem "sări" peste Y datorită faptului că, dacă Y este mai bun decât X, atunci și Z va fi mai bun decât X.

Algoritmul este o variantă ușor modificată a celui anterior. Este suficient să păstrăm o listă a concurenților despre care știm că sunt excelenți.

În momentul în care dorim să verificăm dacă un concurent este sau nu excelent, nu va mai trebui să verificăm pentru fiecare concurent Y dacă Y este mai bun decât X. Este suficient să verificăm toți concurenții Y aflați în lista concurenților excelenți.

Astfel, ordinul de complexitate a operației de verificare a faptului dacă un concurent este sau nu excelent se reduce de la $O(N)$ la $O(K)$, unde K este numărul total al concurenților excelenți.

Dacă se efectuează această modificare, ordinul de complexitate al algoritmului devine $O(N \cdot K)$ și, folosind acest algoritm, un concurent ar fi obținut 40-50 de puncte.

Un algoritm bazat pe arbori de indici

Pentru a obține un algoritm mai rapid va trebui să folosim o structură de date cunoscută și sub numele de *arbore de indici*.

Arborele de indici pe care îl vom folosi pentru a rezolva această problemă trebuie să permită efectuarea următoarelor operații:

- adăugarea unei perechi de numere (cheie și valoare) în arborele de indici;
- furnizarea celei mai mici valori corespunzătoare unei chei care este mai mică decât o valoare dată X.

Arborele de indici pe care îl vom utiliza va permite efectuarea acestor două operații într-un timp de ordinul $O(\log N)$. Vom prezenta în continuare modul în care vom construi un astfel de arbore.

Putem privi arborele de indici ca fiind un arbore binar complet care conține 20 de niveluri, deci are $2^{19} = 524.288$ de frunze.

Fiecare nod al acestui arbore (cu excepția rădăcinii) va avea exact un

părinte și, de asemenea, fiecare nod (cu excepția frunzelor) va avea exact doi fii.

Vom atribui fiecare indice posibil (de la 1 la 524.288) frunzelor arborelui, în ordine. Fiecare frunză va fi "răspunzătoare" pentru informațiile corespunzătoare indicelui care i-a fost atribuit.

Apoi, fiecare nod al arborelui va fi "răspunzător" pentru indicii corespunzători fiilor săi. De exemplu, părintele primelor două frunze va fi "răspunzător" pentru indicii `1` și `2`, părintele său va fi răspunzător pentru indicii `1`, `2`, `3` și `4`, părintele acestuia va fi răspunzător pentru indicii `1`, `2`, `3`, `4`, `5`, `6`, `7` și `8` etc.

Similar, frunzele "responsabile" pentru indicii `35` și `36` vor avea un părinte care va fi "răspunzător" pentru ambii indici (`35` și `36`). Acest părinte va avea un frate care va fi "răspunzător" pentru indicii `33` și `34`, iar părintele acestor două noduri va fi "răspunzător" pentru indicii `33`, `34`, `35` și `36`.

Continuând în acest mod ajungem la rădăcină care va fi "răspunzătoare" pentru întregul interval [`1`, `524.288`], iar cei doi fii ai săi vor fi "răspunzători" pentru intervalele [`1`, `262.144`], respectiv [`262.145`, `524.288`].

Acum, cunoscând intervalul de indici pentru care este "răspunzător" fiecare nod, definim *responsabilitatea* astfel: fiecare nod păstrează cea mai mică valoare corespunzătoare unui indice din intervalul pentru care este "răspunzător" nodul.

Inițial, toate frunzele vor conține valoarea ∞. Implicit, toate nodurile vor conține aceeași valoare.

În momentul în care se dorește adăugarea unei perechi în arborele de indici, este suficient să actualizăm frunza corespunzătoare indicelui dat împreună cu toți strămoșii săi (toate nodurile răspunzătoare pentru indicele dat sunt strămoși ai frunzei respective).

Actualizarea constă în compararea noii valori și a valorii curente. Dacă valoarea curentă este mai mare decât noua valoare, atunci ea este modificată.

Datorită faptului că există $\log_2 N$ strămoși, obținem ordinul de complexitate dorit $O(\log N)$ pentru operația de actualizare.

În momentul în care dorim să efectuăm o interogare de tipul celei descrise (a doua operație menționată), este suficient să alegem cea mai mică valoare dintre valorile păstrate de un frate stâng al unuia dintre strămoșii frunzei corespunzătoare valorii X date.

Deși pare complicat, de fapt este foarte simplu. Vom explica în detaliu în ce constau operațiile descrise.

Fiecare nod (cu excepția rădăcinii) are un frate. Vom numi *frate stâng* nodul corespunzător unui interval ale cărui extremități sunt mai mici; celălalt frate va fi numit *frate drept*. De exemplu, nodurile corespunzătoare intervalelor [`1`, `4`] și [`5`, `8`] sunt frați. Primul nod va fi fratele stâng deoarece valorile `1` și `4` sunt mai mici decât valorile `5` și `8`.

Vom considera acum un indice arbitrar Y care este mai mic decât indicele dat X. Cele două frunze corespunzătoare acestor doi indici vor avea un așa-numit ***cel mai mic strămoș comun*** Z (deoarece oricare două noduri au cel puțin un strămoș comun: rădăcina). Acest cel mai mic strămoș comun este cel mai apropiat strămoș de cele două frunze (aflat pe un nivel cu număr de ordine cât mai mare - numărătoarea nivelurilor începe de la rădăcină).

Datorită faptului că Y este mai mic decât X și datorită faptului că nu există nici un strămoș comun mai mic decât Z, este întotdeauna adevărată afirmația potrivit căreia fiul drept al lui Z este un strămoș al lui Y, iar fiul stâng al lui Z este un strămoș al lui X.

Așadar, fiul stâng al acestui strămoș al lui X este "răspunzător" pentru Y (reamintim faptul că oricare strămoș al lui Y este "răspunzător" pentru Y).

Am demonstrat astfel că, examinând frații stângi ai strămoșilor lui X, putem acoperi ***toți*** indicii mai mici decât X datorită faptului că toți indicii Y au fost acoperiți în acest fel (în argumentația anterioară indicele Y a fost ales arbitrar).

În concluzie, este suficient să verificăm valorile păstrate de către frații stângi ai strămoșilor și să o alegem pe cea mai mică dintre ele.

Numărul strămoșilor este $\log_2 N$, deci obținem ordinul de complexitate dorit $O(\log N)$ pentru operația de interogare.

După implementarea structurii de date descrise, putem folosi algoritmul care va fi descris în cele ce urmează.

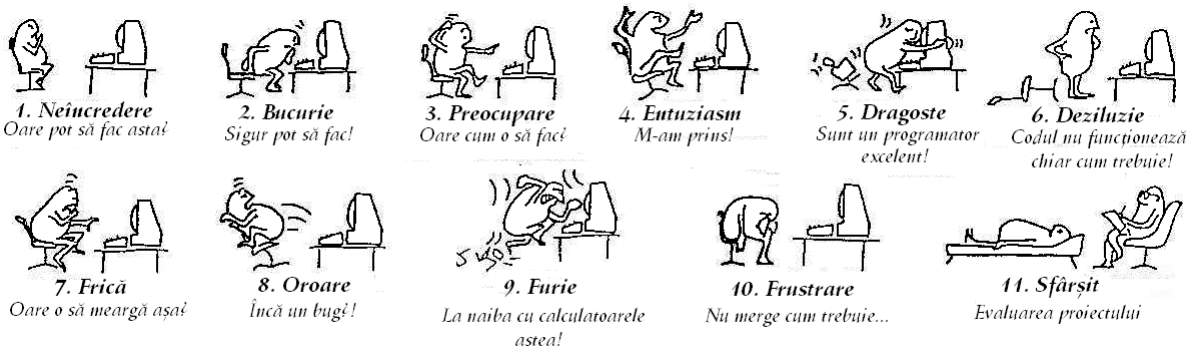
Păstrăm o listă a concurenților în ordinea în care s-au clasat aceștia în primul concurs. De asemenea, păstrăm câte un vector care vor conține pozițiile concurenților în celelalte două concursuri.

În continuare parcurgem concurenții în ordinea în care apar aceștia în listă. Pentru fiecare concurent X vom adăuga în arborele de indici o pereche formată din pozițiile corespunzătoare lui X și vom efectua o interogare pentru indicele X (care reprezintă poziția lui X în al doilea concurs).

În urma interogării vom obține cea mai mică valoare corespunzătoare unui concurent al cărui indice este mai mic decât X.

Datorită faptului că indicii reprezintă pozițiile din cel de-al doilea concurs și valorile reprezintă pozițiile din cel de-al treilea concurs, vom obține cea mai bună clasare în cel de-al treilea concurs a unui concurent care s-a clasat înaintea lui X în cel de-al doilea concurs.

De asemenea, datorită faptului că parcurgem concurenții în ordinea clasării lor în primul concurs, suntem siguri că toți concurenții adăugați în arbore înaintea lui X au terminat pri-



mul concurs pe o poziție mai bună decât X.

Comparăm acum valoarea obținută în urma interogării cu valoarea corespunzătoare lui X (care reprezintă poziția lui X în al treilea concurs).

Dacă valoarea obținută este mai mică decât cea corespunzătoare lui X, atunci știm cu siguranță că există un concurent care a fost mai bun decât X în toate cele trei concursuri. În caz contrar, nu există un astfel de concurent, deci X este un concurent excelent.

În final vom număra concurenții excelenți și vom afișa rezultatul la ieșirea standard.

Fiecare operație efectuată asupra arborelui de indici are ordinul de complexitate $O(\log N)$. Pentru fiecare concurent efectuăm două astfel de operații, deci ordinul de complexitate al întregului algoritm este $O(N \cdot \log N)$. Un astfel de algoritm ar fi dus la obținerea punctajului maxim în concurs.

În continuare vom prezenta versiunea în pseudocod a algoritmului prezentat. Cele 20 de niveluri ale arborelui sunt reprezentate sub forma unor vectori. Arborele va fi format dintr-un vector de vectori care conține 20 de elemente (numerotate de la 0 la 19 începând cu nivelul pe care se află frunzele).

```
algoritm Echipa
  // citirea numărului de concurenți
  citește n
  // citirea concurenților în ordinea
  // clasării în primul concurs
  pentru i ← 1, n execută
    citește unu_i
  sfârșit pentru
  // citirea concurenților în ordinea
  // clasării în al doilea concurs și
  // construirea vectorului care
  // conține pozițiile lor în al
  // doilea concurs
  pentru i ← 1, n execută
    citește X
    doi_i ← X
  sfârșit pentru
  // citirea concurenților în ordinea
  // clasării în al treilea concurs și
  // construirea vectorului care
  // conține pozițiile lor în al
  // treilea concurs
  pentru i ← 1, n execută
    citește X
    trei_i ← X
  sfârșit pentru
  // inițializarea numărului de
  // concurenți excelenți
  excelenți ← 0
  // inițializarea nodurilor
  // arborelui de indici
  pentru i ← 0, 19 execută
    pentru j ← 0, 2^i execută
      arbore_{i,j} ← ∞
    sfârșit pentru
  sfârșit pentru
  // parcurgerea concurenților în
  // ordinea clasării în primul
  // concurs
  pentru i ← 1, n execută
    X ← unu_i
    excelent ← adevărat
    poziție ← trei_X
    pentru nivel ← 0, 19
                        execută
      // adăugarea în arborele
      // de indici a perechii de
      // poziții în al doilea și al
      // treilea concurs
      dacă doi_X <
        arbore_{nivel,poziție} atunci
        arbore_{nivel,poziție} ← doi_X
      sfârșit dacă
      // interogarea pentru
      // poziția în al treilea concurs
      dacă poziție este impar
          și arbore_{nivel,poziție-1}
                < doi_X atunci
        // concurentul nu poate
        // fi excelent
        excelent ← fals
        întrerupe execuția ciclului
      sfârșit dacă
      // trecerea la părinte
      poziție ← [poziție / 2]
    sfârșit pentru
    // se verifică dacă am găsit un
    // concurent excelent
    dacă excelent atunci
      // crește numărul
      // concurenților excelenți
      excelenți ← excelenți + 1
    sfârșit dacă
  sfârșit pentru
  // afișarea rezultatului
  scrie excelenți
sfârșit algoritm
```

Date de test

Tabelul prezintă datele de test folosite în concurs. Se observă că viteza de execuție a diferiților algoritmi depinde de numărul total al concurenților (n) și numărul concurenților excelenți (k).

#	n	k
1	50	26
2	1000	150
3	20000	19042
4	50000	23049
5	100000	1033
6	200000	132974
7	300000	522
8	400000	84652
9	500000	4875
10	500000	286073

P040615: Drum minim

Rezolvarea acestei probleme se bazează pe algoritmul lui *Dijkstra* de determinare a drumului minim într-un graf. Vom prezenta mai multe soluții bazate pe acest algoritm.

Algoritmul banal

Cea mai simplă soluție constă în regăsirea informațiilor necesare pentru a construi întregul graf folosind biblioteca pusă la dispoziție și aplicarea algoritmului lui *Dijkstra* pentru acest graf. O astfel de soluție ar fi dus la obținerea a 20 de puncte deoarece, în acest caz, avem $K = T$.

Un algoritm simplu

Algoritmul anterior poate fi optimizat dacă ne asigurăm că nu cerem informații pe care nu le vom utiliza. Așadar, vom efectua apeluri `getXYM` și `getAdj` numai dacă avem nevoie de rezultatele returnate de funcțiile respective.

De asemenea, execuția se va încheia în momentul în care este găsit un drum până la destinație și nu va cere informații despre vecini sau despre intersecții aflate la o distanță față de S (punctul de plecare) mai mare decât cea la care se află destinația F.

O astfel de implementare ar fi dus la obținerea a 40 de puncte.

Un algoritm euristic

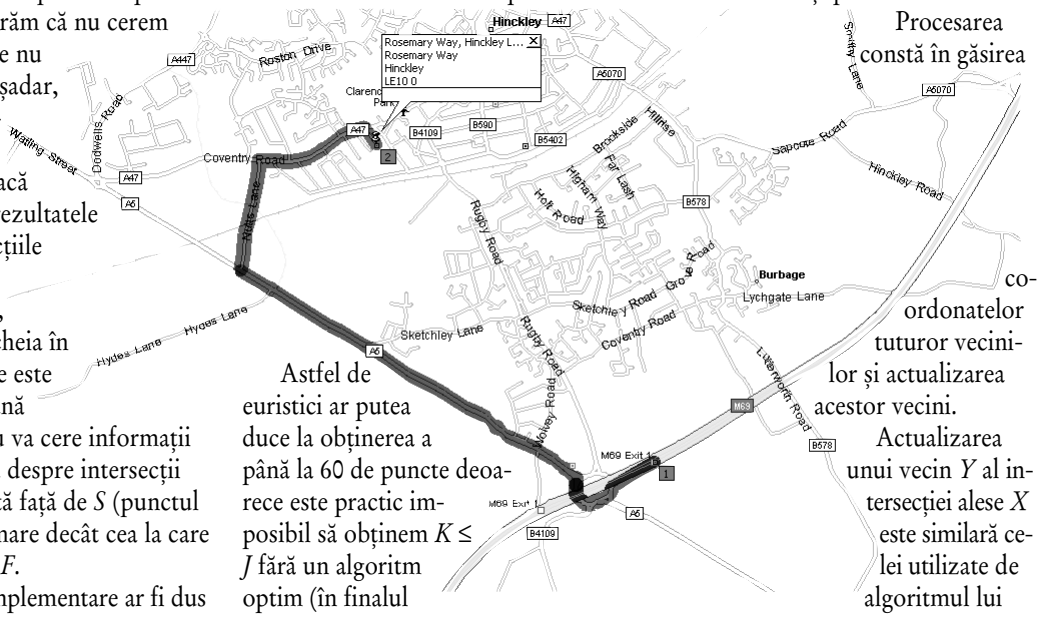
Se știe că algoritmul lui *Dijkstra* este optim, în general. Totuși, această afirmație nu este aplicabilă pentru problema de față deoarece avem informații adiționale: pe lângă muchii și vârfuri, cunoaștem coordonatele intersecțiilor.

Folosind aceste informații obținem o limită inferioară pentru lungimea oricărui drum care unește două intersecții: drumul nu poate fi mai scurt decât distanța euclidiană dintre cele două intersecții.

Astfel putem obține un algoritm mai rapid decât cel al lui *Dijkstra*.

Un exemplu ar fi să formăm un dreptunghi sau o elipsă în jurul punctelor S și F și să ignorăm toate intersecțiile aflate în afara zonei alese. Totuși, trebuie să fim atenți să nu lăsăm în afara zonei posibile drumuri minime.

Dacă este furnizat un răspuns fără ca programul să se asigure că acesta reprezintă într-adevăr drumul minim, acest răspuns nu este luat în considerare. Aceasta înseamnă că dacă există o intersecție X care ar duce la obținerea unui drum minim în cazul în care X este legată direct de destinație, atunci va trebui să determinăm cel puțin coordonatele vecinilor lui X pentru a ne asigura că nu există un alt drum mai scurt care trece prin X.



Astfel de euristici ar putea duce la obținerea a până la 60 de puncte deoarece este practic imposibil să obținem $K \leq J$ fără un algoritm optim (în finalul prezentării soluției va fi descris modul în care este aleasă valoarea J).

Algoritmul optim

Am afirmat că trebuie neapărat să verificăm anumite tipuri de intersecții (trebuie să obținem informații referitoare la vecinii lor).

Ideea care stă la baza acestui algoritm este aceea de a verifica numai intersecțiile care ar trebui luate în considerare. Aceasta va duce la obținerea unui algoritm optim care nu efectuează nici un apel inutil.

Vom numi valoarea P a unei intersecții X, lungimea celui mai scurt drum de la S la F care trece prin X. Această valoare va fi dată de cel mai scurt drum cunoscut de la S la X, la care se adaugă distanța de la X la F.

Inițial, pentru fiecare intersecție (cu excepția intersecției S), valoarea P va fi ∞ deoarece nu există nici un drum cunoscut la nici una dintre aceste intersecții. Pentru intersecția S, valoarea P va fi egală cu distanța dintre S și F.

Pe baza definiției valorii P vom construi un algoritm similar celui al lui *Dijkstra*. Vom împărți intersecțiile în două mulțimi: intersecții procesate și intersecții neprocesate. Apoi vom repeta operația care va fi descrisă în continuare.

Operația constă în alegerea intersecției neprocesate care are cea mai mică valoare P și procesarea ei.

Procesarea constă în găsirea coordonatelor tuturor vecinilor și actualizarea acestor vecini.

Actualizarea unui vecin Y al intersecției alese X este similară celei utilizate de algoritmul lui *Dijkstra*: verificăm dacă se îmbunătățește valoarea P a lui Y dacă se ajunge în Y direct din X (pe un drum direct care leagă X și Y).

Datorită faptului că distanța dintre Y și F este constantă, actualizarea valorii P a intersecției Y este echivalentă cu actualizarea celui mai scurt drum până la Y.

Acesta este motivul pentru care criteriul pentru actualizarea valorii P a intersecției Y este identic cu cel folosit în cadrul algoritmului lui *Dijkstra*.

După actualizarea vecinilor intersecției X, această intersecție este mutată în mulțimea intersecțiilor procesate.

Procedeul se încheie în momentul în care *F* devine intersecția neprocesată cu cea mai mică valoare *P* (exact ca în cazul algoritmului lui *Dijkstra*).

Vom demonstra în cele ce urmează corectitudinea algoritmului propus. Demonstrația este foarte asemănătoare celei care demonstrează corectitudinea algoritmului lui *Dijkstra*.

În momentul în care actualizăm valoarea *P* a unei intersecții *Y* care este o intersecție neprocesată vecină intersecției *X*, valoarea *P* a intersecției *Y* va fi întotdeauna mai mare decât valoarea *P* a intersecției *X*.

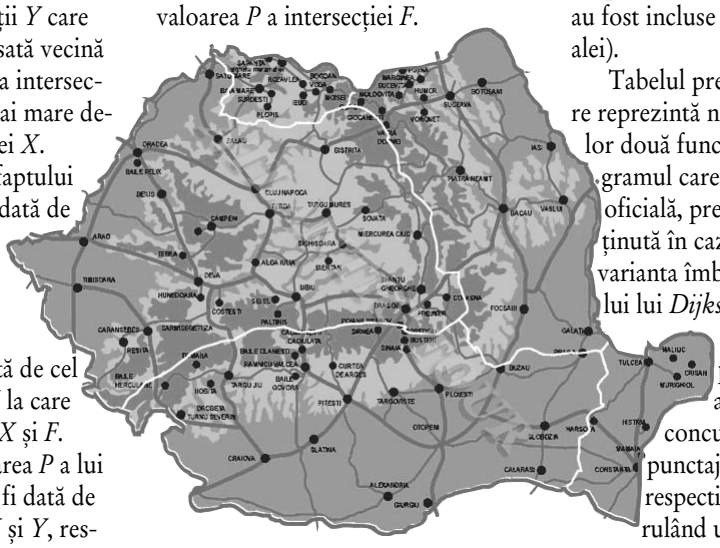
Aceasta se datorează faptului că valoarea *P* a lui *Y* este dată de cel mai scurt drum până la *X* la care se adaugă distanța dintre *X* și *Y* și distanța dintre *Y* și *F*, iar valoarea *P* a lui *X* este dată de cel mai scurt drum până la *X* la care se adaugă distanța dintre *X* și *F*.

Diferența dintre valoarea *P* a lui *Y* și valoarea *P* a lui *X* va fi dată de suma distanțelor dintre *X* și *Y*, respectiv *Y* și *F*, din care se scade distanța dintre *X* și *F*. Folosind inegalitatea triunghiului (suma lungimilor a două laturi este întotdeauna mai mare decât lungimea celei de-a treia laturi) deducem imediat că această diferență este întotdeauna un număr pozitiv.

Astfel am demonstrat că valoarea *P* a unei intersecții neprocesate nu poate deveni niciodată mai mică decât valoarea *P* a intersecției care este aleasă pentru a fi procesată. Putem concluziona că în momentul în care alegem o intersecție *X* pentru procesare, avem deja adevărata valoare *P* a acesteia deoarece această valoare nu poate fi îmbunătățită.

În continuare putem afirma că, în momentul în care este aleasă pentru procesare intersecția *F*, avem deja valoarea *P* a acesteia, deci putem furniza răspunsul fiind siguri că acesta este cel corect.

Toate intersecțiile pe care eram nevoiți să le procesăm au fost deja luate în considerare deoarece valorile *P* ale acestora sunt mai mici decât valoarea *P* a intersecției *F*.



Demonstrarea optimalității algoritmului este mai simplă. Am procesat doar intersecții ale căror valori *P* sunt mai mici decât cea a intersecției *F*.

Fiecare astfel de intersecție trebuia procesată deoarece ar fi putut duce la obținerea unui drum minim până la *F*.

Nu am procesat nici o intersecție a cărei valoare *P* este mai mare decât valoarea *P* a intersecției *F*, deci nu am efectuat nici o procesare inutilă.

Din aceste motive, algoritmul descris este optim în sensul că folosește numărul optim de apeluri pentru funcțiile `getXYM` și `getAdj`.

Date de test

Datele de test se bazează pe harta unui oraș real. În tabelul de pe această pagină este prezentat numărul intersecțiilor (*N*), numărul străzilor (*M*), precum și clasa de străzi incluse (o valoare *R* cuprinsă între 1 și 10; valoarea 1 indică faptul că au fost incluse doar străzile principale, în timp ce valoarea 10 indică faptul că au fost incluse chiar și cele mai mici alei).

Tabelul prezintă și valoarea *K* care reprezintă numărul apelurilor celor două funcții efectuate de programul care implementează soluția oficială, precum și valoarea *D* obținută în cazul în care se folosea varianta îmbunătățită a algoritmului lui *Dijkstra*.

Ultima coloană prezintă valoarea *J* care a fost acceptabilă în concurs pentru a se obține punctajul maxim pentru testul respectiv. Ea a fost calculată rulând un algoritm ușor modificat care determină toate drumurile minime dintre *S* și *F*. Scopul acestei rulări a fost eliminarea factorului aleator (de exemplu, ar fi putut apărea diferențe în funcție de modul în care se tratau situațiile în care două intersecții neprocesate aveau aceeași valoare *P* minimă).

#	N	M	R	T=N+M	D	K	J
1	8	12	10	20	15	13	14
2	23	40	10	63	36	28	29
3	2039	4552	10	6591	6487	2208	2209
4	5277	11810	6	17087	16139	12418	12419
5	6560	15229	10	21789	18979	7352	7353
6	138	122	1	260	29	19	19
7	639	870	3	1509	371	291	293
8	1829	3176	4	5005	1046	383	386
9	6140	13667	6	19807	15385	5269	5272
10	6856	15889	10	22745	21551	7652	7625

Date de test pentru problema Drum minim

P040616: Cursă

Avem un graf planar care descrie harta unui oraș și trebuie să determinăm un drum care respectă restricțiile impuse în enunțul problemei.

Algoritmul banal

Este posibilă generarea tuturor drumurilor care pornesc din vârful identificat prin `1` și ajung la același nod și alegerea celor care nu conțin schimbări de direcție spre dreapta și în care lungimea celei mai scurte străzi este minimă.

O astfel de soluție, chiar cu anumite optimizări, are un timp de execuție exponențial și nu ar duce la obținerea a mai mult de 20 de puncte.

Algoritmul lui Dijkstra modificat

Cea mai bună modalitate de rezolvare a acestei probleme este să o împărțim în subprobleme. Vom încerca să satisfacem constrângerile una câte una.

Prima proprietate a drumului este aceea că cea mai scurtă stradă din cadrul acestuia trebuie să fie cât mai lungă posibil.

În general, dacă trebuie să căutăm un drum de la un vârf la altul, folosim modificări aplicate algoritmului lui *Dijkstra*. Pentru această problemă vom efectua modificarea care va fi descrisă în continuare.

La fiecare pas vom realiza cunoscuta extensie a mulțimii vârfurilor procesate, alegând un vârf pentru care drumul optim curent de la vârful `1` este optim (lungimea celei mai scurte străzi este maximă).

În continuare recalculăm lungimile drumurilor până la vârfurile adiacente vârfului ales. O mică dificultate apare datorită faptului că trebuie să determinăm un drum de la vârful `1` la el însuși. Ea poate fi ușor depășită dacă creăm un duplicat al acestui vârf.

Drumul trebuie să satisfacă o condiție suplimentară: la fiecare intersecție drumul trebuie să continue înainte (în sens pur geometric) sau să continue spre dreapta cu un unghi care are mai puțin de 180°.

Să presupunem că drumul trece prin intersecțiile A, B și C ale căror coordonate sunt (x_A, y_A), (x_B, y_B), respectiv (x_C, y_C). În aceste condiții direcția se schimbă spre stânga dacă intersecția C se află pe aceeași linie ca și intersecțiile A și B sau se află în semiplanul stâng determinat de aceeași linie.

Vom nota prin S valoarea $(x_B - x_A) \cdot (y_C - y_A) - (x_C - x_A) \cdot (y_B - y_A)$. Dacă avem $S > 0$, atunci C se află în semiplanul stâng determinat de dreapta care trece prin intersecțiile A și B.

Dacă avem $S = 0$ și $x_A < x_B < x_C$ sau $x_C < x_B < x_A$ sau $y_A < y_B < y_C$ sau $y_C < y_B < y_A$, atunci intersecția B se află în interiorul segmentului determinat de intersecțiile A și C.

Pentru a rezolva această subproblemă va trebui ca, pentru fiecare vârf, să păstrăm cele mai "bune" drumuri care ajung în vârful respectiv direct de la toate vârfurile incidente.

Dintre toate drumurile determinate la un moment dat o vom alege pe cea mai "bună" și vom modifica valorile vârfurilor incidente (dacă este cazul) luând în considerare și vârful anterior.

De asemenea va trebui să verificăm dacă se respectă condiția potrivit căreia schimbarea direcției nu poate fi decât înspre stânga.

Aceasta reprezintă o modificare semnificativă a algoritmului lui *Dijkstra* datorită faptului că în varianta clasică nu există cicluri în drumurile minime.

În cazul nostru s-ar putea ca în cadrul drumului anumite vârfuri să apară de mai multe ori. Totuși, dacă un vârf se repetă, atunci de fiecare dată când acesta apare "intrăm" în acel vârf prin inter-mediul unei alte muchii.

În algoritmul modificat vom avea cel mult M iterații. Așadar o implementare simplă va duce la obținerea ordinului de complexitate $O(M^2)$.

Totuși, o implementare folosind un *heap* binar va duce la obținerea ordinului de complexitate $O(M \cdot \log N)$.

Date de test

Tabelul următor prezintă principalele caracteristici ale datelor de test folosite pentru evaluarea soluțiilor concurenților.

#	M	N
1	6	7
2	10	12
3	40	500
4	70	1400
5	300	3000
6	800	10000
7	1000	13500
8	1500	20000
9	1800	18000
10	2000	25000

P040617: Joc

Acesta este un exemplu tipic de aplicare a metodei programării dinamice. Vom prezenta în cele ce urmează patru algoritmi bazați pe această metodă, fiecare dintre ei rezolvând problema.

Totuși, cei patru algoritmi au ordine de complexitate diferite și, din acest motiv, ar duce la obținerea unui anumit număr de puncte pentru datele de test folosite.

Algoritmul banal

Definim o subproblemă (x, y) ca fiind o configurație în care avem x numere rămase în prima secvență și y numere rămase în cea de-a doua.

Am dori să determinăm costul minim pentru fiecare sub problemă (x, y), unde x variază între 1 și L_1, iar y variază între 1 și L_2. Acest cost minim va fi notat prin $f(x, y)$.

Cea mai simplă modalitate constă în încercarea tuturor mutărilor posibile ca primă mutare pentru o subproblemă.

Putem determina costul minim asociat unei mutări pe baza următoarei formule:

$$f(x, y) = \min\{(S_1 - K_1) \cdot (S_2 - K_2) + f(x - K_1, y - K_2)\}$$

în cadrul căreia folosim toate valorile K_1 și K_2 care satisfac condițiile.

Procedeul constă în simpla însumare a costului mutării și a costului subproblemei rămase.

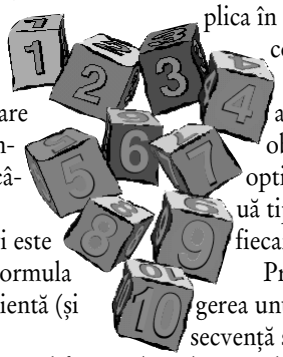
Costul minim asociat tuturor mutărilor posibile pentru subproble-

ma (x, y) va fi dat întotdeauna de valoarea $f(x, y)$.

Algoritmul poate fi implementat foarte simplu și are ordinul de complexitate $O(N^4)$. Un astfel de algoritm ar fi obținute 20-30 de puncte, în funcție de detaliile de implementare și diferitele optimizări utilizate.

Observații utile

Există câteva observații care duc la simplificarea problemei și obținerea unei soluții mai rapide.



Prima observație constă în faptul că putem transforma fiecare secvență în alta care conține numerele din secvența inițială din care se scade câte o unitate.

Beneficiul acestei scăderi este faptul că în loc să utilizăm formula $(S_1 - K_1) \cdot (S_2 - K_2)$ va fi suficientă (și necesară) formula $S_1 \cdot S_2$.

După efectuarea acestei simplificări, putem observa imediat o alta. Practic, nu are rost să luăm două sau mai multe numere din fiecare secvență în același timp (la fiecare mutare vom avea $K_1 = 1$ și/sau $K_2 = 1$).

Vom demonstra acum această afirmație. Să considerăm o posibilă mutare pentru care avem $K_1 > 1$ și $K_2 > 1$.

Alegând două sau mai multe numere din fiecare secvență înseamnă, de fapt, că putem "împărți" această mutare în două mutări.

Să considerăm două astfel de mutări. Prima dintre ele constă în alegerea a K_1 numere din prima secvență (suma acestora este S_1) și a K_2 numere din a doua secvență (suma acestora este S_2). A doua mutare constă în alegerea a altor K_3 numere din prima secvență (suma acestora este S_3) și a altor K_4 numere din a doua secvență (suma acestora este S_4).

Costul total al acestor două mutări va fi, având în vedere prima observație, $S_1 \cdot S_2 + S_3 \cdot S_4$.

Dacă am fi efectuat două mutări diferite am fi avut un cost total de $(S_1 + S_3) \cdot (S_2 + S_4) = S_1 \cdot S_2 + S_3 \cdot S_4 + S_1 \cdot S_4 + S_3 \cdot S_2 > S_1 \cdot S_2 + S_3 \cdot S_4$.

Pe baza acestei observații putem deduce că numărul total al mutărilor posibil este de ordinul $O(N)$.

Ordinul de complexitate al întregului algoritm devine $O(N^3)$. Un astfel de algoritm ar fi obținut în concurs 30-50 de puncte.

Reducerea buclei interioare

Pentru a îmbunătăți performanțele algoritmului vom utiliza o tehnică foarte cunoscută pentru optimizarea algoritmilor bazați pe metoda programării dinamice și anume metoda reducerii buclei interioare. Vom explica în cele ce urmează în ce constă această tehnică.

Algoritmul descris anterior (care folosește observațiile care permit optimizarea) încearcă două tipuri de mutări pentru fiecare subproblemă (x, y).

Primul tip constă în alegerea unui număr A din prima secvență și a K_2 numere din cea de-a doua. Valoarea optimă pentru K_2 este determinată alegându-se acea valoare care minimizează costul $A \cdot S_2 + f(x - 1, y - K_2)$ unde A este ultimul număr din prima secvență.

Notăm poziția din cea de-a două secvență care duce la obținerea optimului prin $g(x, y)$, valoare care este egală cu $y - K_2$.

Aplicăm aceeași metodă pentru cel de-al doilea tip de mutări în care se minimizează un cost de forma $B \cdot S_1 + f(x - K_1, y - 1)$ și notăm poziția optimului prin $h(x, y)$. Am notat prin B ultimul număr din cea de-a doua secvență.

Algoritmul compară cele două mutări optime (câte una din fiecare tip) și o alege pe cea mai bună dintre ele.

Vom reduce buclele interioare ale algoritmului anterior pe baza următoarei observații: $g(x, y - 1) \le g(x, y)$.

Vom demonstra această afirmație folosind metoda reducerii la absurd. Presupunem că avem $g(x, y - 1) > g(x, y)$. În acest caz avem $g(x, y) < g(x, y - 1) < y - 1 < y$. Așadar vom lua în considerare atât valoarea $g(x, y)$, cât și valoarea $g(x, y - 1)$ în momentul în care vom calcula valorile $f(x, y)$ și $f(x, y - 1)$.

Mai mult, când rezolvăm subproblema (x, y) preferăm valoarea $g(x, y)$. De asemenea, am preferat valoarea $g(x, y - 1)$ când am rezolvat subproblema $(x, y - 1)$.

Dacă notăm prin $s(u, v)$ suma numerelor aflate între pozițiile u și v în cea de-a doua secvență, pe baza observațiilor de mai sus obținem:

$$A \cdot s(g(x, y - 1) + 1, y) + f(x - 1, g(x, y - 1)) < A \cdot s(g(x, y) + 1, y) + f(x - 1, g(x, y))$$

$$A \cdot s(g(x, y - 1) + 1, y - 1) + f(x - 1, g(x, y - 1)) > A \cdot s(g(x, y) + 1, y - 1) + f(x - 1, g(x, y))$$

Se observă imediat că dacă scădem valoarea $A \cdot s(y, y)$ din ambii termeni ai primei inegalități, obținem două inegalități contradictorii.

Ca urmare, ipoteza potrivit căreia $g(x, y - 1) > g(x, y)$ este falsă, deci avem întotdeauna $g(x, y - 1) \le g(x, y)$.

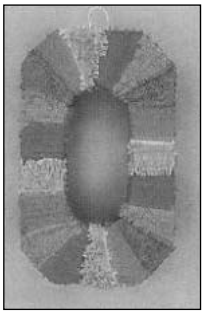
Așadar, în momentul în care vom calcula valoarea $g(x, y)$ nu mai trebuie să luăm în considerare toate valorile cuprinse între 1 și $y - 1$, ci doar pe cele cuprinse între $g(x, y - 1)$ și $y - 1$.

Putem aplica același raționament și pentru cea de-a doua buclă interioară. Vom ajunge la concluzia că în momentul în care vom calcula valoarea $h(x, y)$ nu mai trebuie să luăm în considerare toate valorile cuprinse între 1 și $x - 1$, ci doar pe cele cuprinse între $h(x - 1, y)$ și $x - 1$.

În cel mai defavorabil caz, acest algoritm are ordinul de complexitate $O(N^3)$. Totuși, în practică, ordinul de complexitate se apropie de $O(N^2)$. Din acest motiv, o rezolvare bazată pe acest algoritm ar fi dus la obținerea a 60-80 de puncte în concurs.

Eliminare buclei interioare

Pentru a obține un algoritm al cărui ordin de complexitate să fie întotdeauna $O(N^2)$ vom încerca să îmbunătățim algorimul anterior eliminând complet buclele interioare.



În demonstrația anterioară am arătat că dacă facem o comparație între două valori "candidate" pentru $g(x, y)$ și pentru $g(x, y - 1)$ vom obține întotdeauna același rezultat deoarece în caz contrar am ajunge în situația în care cele două inegalități cu termeni identici, ar avea semne diferite.

Tragem concluzia că vom avea $g(x, y) = g(x, y - 1)$ sau valoarea $g(x, y)$ se află în afara valorilor valide pentru $g(x, y - 1)$.

Valorile valide pentru $g(x, y - 1)$ sunt cuprinse între 1 și $y - 2$, iar valorile valide pentru $g(x, y)$ sunt cuprinse între 1 și $y - 1$. Ca urmare vom avea întotdeauna fie $g(x, y) = g(x, y - 1)$, fie $g(x, y) = y - 1$.

Așadar, este suficient acum să verificăm aceste două posibile valori, ceea ce elimină complet bucla interioară.

De asemenea, aplicăm același raționament pentru $h(x, y)$. În acest caz cele două valori posibile sunt $h(x - 1, y)$ sau $x - 1$.

O ultimă observație constă în faptul că nu putem aplica această procedură pentru valorile $g(x, 2)$ și $h(2, y)$ deoarece $g(x, 1)$ și $h(1, y)$ sunt cazuri speciale care nu permit aplicarea logicii recurente descrise.

Așadar, exact pentru aceste valori va trebui să aplicăm procedura de determinare descrisă pentru algoritmul anterior.

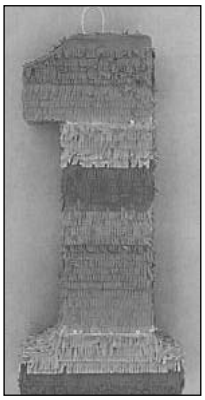
Ordinul de complexitate al acestui algoritm este cu siguranță $O(N^2)$ deoarece fiecare subproblemă este rezolvată într-un timp constant. O implementare corectă a acestui algoritm ar fi dus la obținerea punctajului maxim.

Date de test

Cei mai importanți parametri ai datelor de test sunt lungimile celor două șiruri. Următorul tabel descrie datele folosite pentru testarea programelor participanților.

#	M	N
1	20	20
2	110	80
3	200	130
4	400	80
5	1000	333
6	510	910
7	1200	1400
8	700	1800
9	1998	1370
10	2000	1999

P040618: Coduri



Pentru început vom menționa, ca fapt divers, faptul că, în general, codurile circulare de anvergură 1 sunt numite coduri ***șarpe în cutie***.

Cartea *Combinatorial Algorithms* scrisă de ***E. Reingold, J. Nievergelt*** și ***N. Deo*** și publicată la editura *Prentice Hall* în 1977 conține definițiile și lemele prezentate în cele ce urmează.

Definiție

Două coduri sunt echivalente dacă fiecare dintre ele poate fi obținut din celălalt pe baza următoarelor transformări:

- rearanjarea coloanelor;
- inversarea elementelor unei coloane.

Lemă

Fiecare cod este echivalent cu un cod al cărui prim șir este `00...0`.

Lemă

Fiecare cod de lungime n și anvergură k pentru care avem $M > 2 \cdot k + 3$ este echivalent cu un cod ale cărui prime $k + 3$ șiruri sunt:

```
000...00...0
100...00...0
110...00...0
   ...
111...10...0
```

Lemă

Fiecare cod este echivalent cu un cod ale cărui coloane sunt sortate în ordine lexicografică.

Pe baza acestor leme au fost obținute rezultatele din tabelul de pe această pagină.

Valorile marcate cu * indică faptul că rezultatul este cu siguranță cel optim.

Valorile îngroșate indică datele de test folosite în concurs, iar în paranteză sunt prezentate valorile maxime obținute de concurenți în cadrul concursului.

Pentru obținerea codurilor circulare pe baza primelor două leme prezentate putea fi folosită eficient metoda *backtracking*.

Notă

Acestea sunt soluțiile oficiale ale problemelor propuse spre rezolvare la ediția 2004 a Olimpiadei Balcanice de Informatică. Ele au fost preluate de pe site-ul oficial al competiției și au fost traduse de către membrii redacției GInfo.

Lungimea \ Anvergura	1	2	3	4	5	6
4	8*	8*	8*	-	-	-
5	**14* (14)**	10*	10*	10*	-	-
6	**26* (26)**	**16* (16)**	12*	12*	12*	-
7	**48* (46)**	**24* (24)**	14*	14*	14*	14*
8	**96 (82)**	**36* (34)**	22*	16*	16*	16*
9	170	**54 (50)**	30*	24*	18*	18*
10	340	**80 (72)**	**46 (40)**	28	20	20
11	620	154	58	40	30	22

Lungimile codurilor circulare