Rzeszów, iulie 2004

OLIMPIADA de Informatică a EUROPEI CENTRALE 2004

Vă prezentăm în cadrul acestui articol soluțiile oficiale ale celor șase probleme propuse spre rezolvare la ultima ediție a Olimpiadei de Informatică a Europei Centrale.

P040607: Nori

Această problemă este una dintre acele probleme de geometrie ale cărei soluții corecte se bazează pe aceeași idee.

Este ușor de observat faptul că pentru a determina de câte ori este întreruptă comunicația, trebuie să luăm în considerare punctele în care raza atinge marginile norilor. Acestea sunt singurele puncte în care "starea" comunicației se schimbă.

În enunț se precizează că avem o rază care pornește din punctul de coordonate (0, 0) și o mulțime de nori care se deplasează cu o viteză descrisă de un vector ale cărui componente sunt v_x și v_y.

Sunt necesare cunoștințe elementare de fizică pentru a ne da seama că mișcarea este relativă, deci putem considera că norii își păstrează poziția, iar raza se deplasează cu o viteză descrisă de un vector ale cărui componente sunt $-v_x$ și $-v_y$.

Această nouă formulare a problemei este mult mai convenabilă și va fi folosită în cele ce urmează. În figura 1 este prezentată "traiectoria" razei.

Această traiectorie este o semilinie care pornește din originea sistemului de coordonate. Intersecția dintre această semilinie și poligoanele care reprezintă norii este dată de o mulțime de segmente (care fac parte din semilinie).

Poate apărea și situația în care intersecția dintre traiectorie și un nor este dată de un singur punct. În acest caz vom considera că intersecția este un segment de lungime 0.

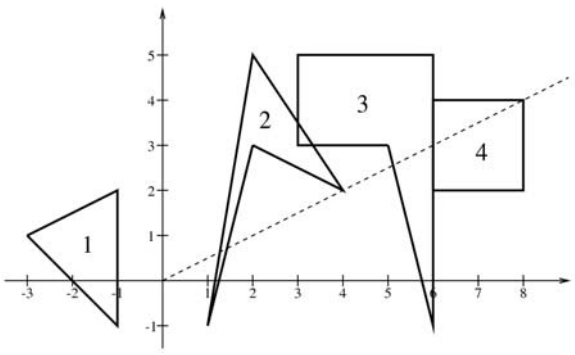

Figura 1: Exemplu pentru traiectoria razei LASER

Pentru cazul din figura 1, intersecția dintre linie și norul identificat prin 2 este dată de două segmente, dintre care unul are lungimea 0.

Fiecare segment care face parte din intersecția dintre traiectorie și nori reprezintă un interval de timp în care comunicația este întreruptă.

Ca prim pas al soluției, vom determina separat, pentru fiecare nor, segmentele care fac parte din intersecție. Ulterior, vom lua în considerare reuniunea segmentelor obținute pentru fiecare nor în parte.

Evident, este posibil ca, în momentul efectuării reuniunii, anumite segmente să se lipească.

Este cazul segmentelor corespunzătoare intersecției cu norii 3 și 4 din figura 1.

Soluția problemei va fi dată de numărul total al segmentelor obținute după efectuarea reuniunii.

Pentru exemplul din figura 1 vom avea:

- două segmente pentru norul 2, dintre care unul are lungimea 0;
- un segment pentru norul 3;
- un segment pentru norul 4.

Răspunsul corect va fi 3, deoarece avem trei segmente după efectuarea reuniunii.

Determinarea segmentelor pentru fiecare nor

Pentru determinarea segmentelor de intersecție cu un nod, este necesară

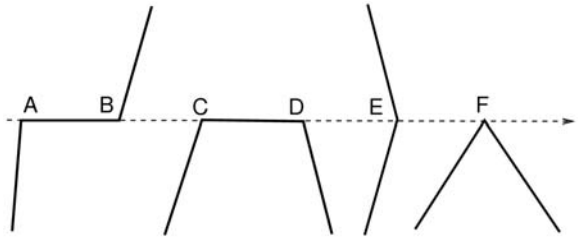

Figura 2: Cazurile speciale care trebuie luate în considerare în momentul determinării segmentelor care reprezintă intersecția dintre traiectorie și un nor

determinarea punctelor de intersecție dintre semilinia care reprezintă traiectoria și laturile poligonului care reprezintă norul.

Va trebui să luăm în considerare fiecare latură a poligonului și să verificăm dacă aceasta se intersectează cu semilinia.

Pentru un nor care este reprezentat printr-un poligon cu k vârfuri, ordinul de complexitate al acestei operații este $O(k)$.

Totuși, chiar și după determinarea intersecțiilor, nu putem imediat să ne dăm seama câte segmente există deoarace forma norilor este arbitrară (nu știm decât faptul că poligonul care reprezintă norul nu poate fi încrucișat - laturile nu se pot intersecta decât în vârfurile poliginului, iar două laturi care se intersectează trebuie să fie consecutive în sens trigonometric sau antitrigonometric).

În principiu, ar trebui să sortăm punctele de intersecție în funcție de distanța lor față de originea sistemului de coordonate, să le grupăm două câte două și să numărăm segmentele astfel formate.

Această abordare este corectă în cazul în care semilinia nu intersectează poligonul într-un vârf. Dacă printre punctele de intersecție se află și vârfuri ale poligonului, atunci trebuie tratate câteva cazuri speciale, care sunt prezentate în figura 2.

Ultimele două cazuri pot fi rezolvate foarte ușor. Punctul E este adăugat o singură dată, iar unctul F este adăugat de două ori. Se observă că această abordare este corectă indiferent în care parte se află interiorul poligonului care reprezintă norul.

Pentru rezolvarea primelor două cazuri va trebui să definim ***tipul*** fiecărui punct de intersecție.

Punctele de intersecție care nu sunt vârfuri ale poligonului sunt puncte ***standard***.

De asemenea, punctele adăugate pentru ultimele două cazuri sunt și ele puncte standard.

Pentru primul caz punctul A este adăugat de două ori.

Prima dată acesta este adăugat ca punct standard, iar apoi ca punct ***segment***.

Punctul B este adăugat o singură dată, ca punct segment.

Pentru cel de-al doilea caz, punctele C și D sunt adăugate o singură dată, ca puncte segment.

În momentul în care generăm punctele de intersecție pentru un anumit nor, vom sorta punctele obținute prin procedeele descrise anterior, în funcție de distanța față de originea sistemului de coordonate, păstrând ordinea pentru punctele care apar de două ori.

Cu alte cuvinte, după sortare, cele două puncte corespunzătoare punctului A din primul caz vor apărea, după sortare, ca un punct standard urmat de un punct segment.

În acest moment putem determina segmentele care reprezintă intersecțiile.

Pentru aceasta vom examina punctele din secvența ordonată, în ordinea obținută după ordonare. Punctele standard din secvență reprezintă extremități drepte sau stângi (alternativ) ale unor segmente.

Se observă faptul că punctele segment apar întotdeauna în perechi în secvența ordonată.

În momentul în care detectăm o astfel de pereche de puncte (le vom nota prin X și Y) vom studia cel mai recent punct standard considerat anterior.

În cazul în care acest punct reprezintă extremitatea stângă a unui segment (nu are încă pereche), punctele X și Y sunt ignorate.

În caz contrar (punctul reprezintă extremitatea dreaptă a unui segment sau nu există nici un punct standard considerat anterior), segmentul determinat de punctele X și Y este adăugat în lista segmentelor care reprezintă intersecția.

Este evident faptul că factorul dominant în determinarea timpului de execuție necesar prelucrării unui nor este dat de operația de sortare a punctelor de intersecție.

Așadar, pentru un nor reprezentat de un poligon cu k vârfuri care se intersectează cu traiectoria în q puncte, ordinul de complexitate al operației de determinare a segmentelor de intersecție este $O(k + q \cdot \log q)$.

Determinarea numărului de întreruperi ale comunicației

Așa cum am menționat anterior, este suficient să luăm în considerare segmentele determinate pentru fiecare nor și să determinăm numărul segmentelor din reuniune (lipind anumite segmente dacă este necesar).

Pentru aceasta vom sorta segmentele în funcție de distanțele dintre extremitățile stângi și originea sistemului de coordonate.

Considerăm acum segmentele în ordinea obținută după sortare și numărăm segmentele ale căror extremități stângi sunt mai îndepărtate de orgine decât cea mai îndepărtată extremitate dreaptă a unui segment considerat anterior.

Ordinul de complexitate al algoritmului prezentat este $O(m + p \cdot \log p)$., unde m reprezintă numărul total al vârfurilor poligoanelor care reprezintă norii, iar p reprezintă numărul total al punctelor de intersecție determinate pentru toți norii.

O capcană...

Soluția prezentată conține o capcană bine ascunsă. Așa cum am arătat, algoritmul efectuează sortări ale punctelor, ceea ce implică efectuarea de comparări între coordonatele acestora.

Uneori, intersecțiile sunt atât de apropiate încât nici măcar numerele reale reprezentate pe 80 de biți nu oferă o acuratețe suficientă.

Pentru a se evita această problemă era necesară reprezentarea coordonatelor punctelor ca rapoarte între două numere întregi a câte 64 de biți. Folosind această reprezentare, pentru a efectua comparările, era necesară implementarea operației de înmulțire pentru două numere întregi reprezentate pe câte 64 de biți, deci rezultatul putea conține până la 128 de biți.

P040608: Bomboane

Pentru început vom reformula problema în termeni specifici combinatoricii.

Trebuie să determinăm numărul posibilităților de a distribui cel mult r obiecte identice în n cutii diferite, astfel încât cea de-a i-a cutie să conțină cel mult m_i obiecte, unde i variază între 1 și n.

Dacă notăm această valoare prin $C(r)$, atunci determinarea soluției va implica calcularea valorilor de tipul $C(b) - C(a - 1)$, sau doar $C(b)$ dacă avem $a = 0$.

În continuare vom prezenta câteva elemente teoretice referitoare la distribuirea obiectelor identice în cutii distincte fără a se impune nici o restricție.

Propoziția #1

Numărul posibilităților de distribuire a r obiecte identice în n cutii distincte este C_{r+n-1}^{n-1}.

Demonstrație

Numărul posibilităților de distribuire a r obiecte în n cutii corespunde numărului posibilităților de a aranja r obiecte și n - 1 bare, număr care este egal cu C_{r+n-1}^{n-1}.

Aceste bare împart obiectele în n grupuri, iar fiecare grup corespunde unei cutii.

Propoziția #2

Numărul posibilităților de distribuire a cel mult r obiecte identice în n cutii distincte este C_{r+n}^{n}.

Demonstrație

Veridicitatea propoziției reiese imediat pe baza primei propoziții. Distribuirea a cel mult r obiecte în n cutii corespnde distribuirii a exact n obiecte în $n + 1$ cutii.

Indiferent câte obiecte avem în a $(n + 1)$-a cutie, în primele n vom avea cel mult r.

Cu alte cuvinte, cea de-a $(n + 1)$-a cutie va conține toate obiectele care nu au fost distribuite în nici una dintre primele n cutii.

Notăm prin U numărul posibilităților de a distribui, fără nici o restricție, r obiecte identice în n cutii.

De asemenea, notăm prin A_i o mulțime care conține toate posibilitățile de a distribui cel mult r obiecte identice în n cutii astfel încât cea de-a i-a cutie să conțină mai mult de m_i obiecte.

Complementul mulțimii A_i (care va fi notat prin $\overline{A_i}$) va conține toate posibilitățile de a distribui cel mult r obiecte în n cutii astfel încât cea de-a i-a cutie să conțină cel mult m_i obiecte.

Ca urmare, soluția problemei va fi dată de cardinalul mulțimii obținute prin intersecția tuturor mulțimilor de forma $\overline{A_i}$, unde i variază între 1 și n:

$$\left|\overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}\right|$$

Această valoare poate fi determinată pe baza principiului includerii și excluderii. Obținem următorul rezultat:

$$\left|\overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}\right| = |U| - |A_1| - |A_2| - \cdots - |A_n| + \\ + |A_1 \cap A_2| + \cdots + |A_{n-1} \cap A_n| - \\ - \cdots + (-1)^n |A_1 \cap A_2 \cap \cdots \cap A_n|$$

Pentru a rezolva problema va trebui să determinăm toate valorile de forma $\left|A_{i_1} \cap A_{i_2} \cap \cdots A_{i_k}\right|$ pentru toate mulțimile $\{i_1, i_2, \ldots i_k\} \subseteq \{1, 2, \ldots, n\}$.

Mulțimea $A_{i_1} \cap A_{i_2} \cap \cdots A_{i_k}$ conține toate posibilitățile de a distribui r obiecte în n cutii astfel încât pentru orice valoare $j \in \{i_1, i_2, \ldots i_k\}$, cea de-a j-a cutie să conțină mai mult de m_j obiecte.

Vom prezenta acum modul în care poate fi obținută o astfel de distribuire. Pentru început vom amplasa câte $m_j + 1$ obiecte în fiecare cutie $j \in \{i_1, i_2, \ldots i_k\}$.

Apoi, distribuim obiectele rămase în toate cele n cutii. Din propoziția #2 rezultă că numărul posibilităților este:

$$\left|A_{i_1} \cap A_{i_2} \cap \cdots A_{i_k}\right| = C_{n+r-(m_{i_1}+1)-(m_{i_2}+1)-\cdots-(m_{i_k}+1)}^{n}$$

Folosind această formulă putem deduce că rezultatul final va fi dat de:

$$S(r) = \sum_{I \subseteq \{1,\ldots,n\}} (-1)^{|I|} \cdot C_{n+r-\sum_{i \in I}(m_i+1)}^{n}$$

Coeficienți binomiali

Principala dificultate care apare în momentul implementării constă în calcularea modulo 2004 a valorilor coeficienților bonomiali.

Există câteva posibilități de rezolvare a acestei probleme, fiecare având un anumit ordin de complexitate și fiind mai mult sau mai puțin simplă.

Triunghiul lui Pascal

Putem utiliza cunoscuta relație de recurență care duce la obținerea triunghiului lui *Pascal*:

$$C_p^q = \begin{cases} 1 & \text{pentru } p = 0 \text{ sau } q = 0 \\ C_{p-1}^{q} + C_{p-1}^{q-1} & \text{pentru } p > 0 \text{ și } q > 0 \end{cases}$$

Folosind această relație, putem determina, folosind metoda programării dinamice, toate valorile C_p^q (modulo 2004) pentru toate valorile $p \le b - \sum_{i \in I}(m_i + 1) + n$.

Valorile păstrate vor fi ulterior utilizate pentru determinarea rezultatului $S(r)$.

Această preprocesare necesită păstrarea unui vector cu cel mult $b + n$ elemente, valoare care se încadrează în limita asupra memoriei disponibile impusă în enunț.

Totuși, ordinul de complexitate al operației de preprocesare este $O(n \cdot b)$, ordin care este mai mare decât cel al operației prin care sunt determinate valorile $S(b)$ și $S(a - 1)$.

Astfel, întreaga soluție va avea ordinul de complexitate $O(n \cdot b + 2^n)$.

O recurență mai "inteligentă"

Vom prezenta în continuare o abordare similară exponențierii binare[1].

Pentru a calcula C_p^q vom determina valorile C_p^i pentru i cuprins între 0 și q. Pentru $p = 1$ vom avea

$C_p^0 = C_p^1 = 1$ și $C_p^i = 0$ pentru $i > 1$.

Pentru $p > 1$ vom putea utiliza următoarea formulă:

$$C_p^i = \begin{cases} C_{p-1}^i + C_{p-1}^{i-1} & \text{pentru } n \text{ par} \\ \sum_{j=0}^{i} C_{p/2}^j \cdot C_{p/2}^{i-j} & \text{pentru } n \text{ impar} \end{cases}$$

Ca urmare, ordinul de complexitate al operației de determinare a valorii coeficientului binomial C_p^q este $O(q^2 \cdot \log p)$.

Astfel, obținem ordinul de complexitate $O(2^n \cdot n^2 \cdot \log b)$ pentru întregul algoritm de rezolvare a problemei. Spațiul de memorie necesar are ordinul $O(n)$.

O altă abordare

O a treia posibilitate este utilizarea formulei de calcul pentru combinări:

$$C_p^q = \frac{p!}{q!(p-q)!}.$$

Din nefericire, valoarea numărătorului poate deveni foarte mare.

O posibilitate de a rezolva acest inconvenient este implementarea operațiilor cu numere mari (adunare și scădere pentru două numere mari, înmulțire și împărțire dintre un număr mare și un număr mic). Această metodă ignoră faptul că ne interesează rezultatul modulo 2004.

Dacă luăm în considerare acest aspect, putem utiliza următoarea abordare:

- se determină descompunerea în factori primi a valorii $q!$ (datorită faptului că valorile q care apar în rezolvare pot fi cel mult egale cu 10, toate cele zece descompuneri posibile pot fi determinate manual și păstrate dub forma unor șiruri constante);
- se construiește un vector A care conține q elemente și ale cărui elemente au inițial valorile $A_i = p + 1 - i$;
- se iau în considerare succesiv factorii primi; pentru fiecare factor x de ordin k efectuăm următorii pași de k ori: determinăm prima valoare A_i divizibilă cu x, împărțim la p valoarea A_i și păstrăm rezultatul în același element A_i al vectorului;
- se înmulțesc (modulo 2004) valorile finale ale elementelor vectorului creat; în urma acestei înmulțiri se obține valoarea modulo 2004 a coeficientului binomial C_p^q.

Folosind această metodă, ordinul de complexitate al operației de determinare a valorii C_p^q este $O(q^2 \cdot \log q)$.

Astfel, obținem ordinul de complexitate $O(2^n \cdot n^2 \cdot \log n)$ pentru întregul algoritm de rezolvare a problemei.

O soluție alternativă: recursivitate și programare dinamică

Dacă notăm prin $c_k(r)$ numărul posibilităților de distribuire a r obiecte în primele k cutii, atunci următoarea relație de recurență este adevărată:

$$c_0(r) = [r = 0];$$

$$c_k(r) = \sum_{i=0}^{m_k} c_{k-1}(r-i).$$

Folosind această formulă și metoda programării dinamice obținem un algoritm cu ordinul de complexitate $O(b \cdot M)$, unde $M = \sum_{i=1}^{n} m_i$. Spațiul de memorie necesar are ordinul $O(b)$.

Putem îmbunătăți performanțele dacă, în vederea calculării valorilor c_k, păstrăm sume parțiale de termeni c_{k-1}.

Dacă notăm $C_{k-1}(r) = \sum_{i=0}^{r} c_{k-1}(i)$, atunci avem:

$c_k(r) = C_{k-1}(r) - C_{k-1}(r - m_k - 1)$.

Evident, va trebui să ținem cont de faptul că avem $C_{k-1}(r - m_k - 1) = 0$ pentru situațiile în care avem $r - m_k - 1 < 0$.

Ordinul de complexitate al acestei soluții este $O(b \cdot n)$.

P040609: Excursii

Vom prezenta două posibile variante de rezolvare a problemei. Prima dintre ele este relativ directă, în timp ce a doua profită de particularitățile problemei.

Cuplaj maxim în graf bipartit

Problema poate fi foarte ușor reformulată în termeni ai teoriei grafurilor. Ea se reduce la determinarea unui cuplaj maxim într-un graf bipartit $G = (V, E)$.

Un cuplaj este definit ca o submulțime M a mulțimii E a muchiilor, astfel încât oricare două muchii diferite din M nu au nici un punct comun. Problema determinării cuplajului maxim necesită determinarea unui cuplaj M care să conțină cât mai multe muchii posibil.

Pentru problema de față vom avea un graf bipartit $G = (V \cup W, E)$, unde V este mulțimea grupurilor, W este mulțimea excursiilor, iar E este mulțimea muchiilor.

Două noduri ale grafului $v \in V$ și $w \in W$ vor fi unite printr-o muchie în cazul în care grupul v poate pleca în excursia w.

Așadar, avem:

$$(i,j) \in E \Leftrightarrow i \in V \wedge j \in W \wedge l_j \le s_i \le u_j$$

Un cuplaj M în acest graf corespunde realizării unei corspondențe între grupuri și excursii. Dacă avem $(v, w) \in M$, înseamnă că grupul v pleacă în excursia w.

Fiecare cuplaj din graful bipartit corespunde unei corespondențe valide între grupuri și excursii.

Definiția cuplajului ne asigură că fiecare grup va pleca în cel mult o excursie și că în fiecare excursie va pleca cel mult un grup.

Scopul nostru este de a determina un cuplaj maximal P, astfel încât $P \subseteq E$ și

$$\forall_{(i,j)\in P} \forall_{(a,b)\in P} ((i,j) = (a,b) \vee (i \neq a \wedge j \neq b))$$

[1] *exponențierea binară este modalitatea de calcul a valorii a^n pe baza următoarei relații:*

$$a^n = \begin{cases} 1 & \text{pentru } n = 0 \\ a^{n/2} \cdot a^{n/2} & \text{pentru } n \text{ par} \\ a \cdot a^{[n/2]} \cdot a^{[n/2]} & \text{pentru } n \text{ impar} \end{cases}$$

Evident, cardinalul mulțimii P trebuie să fie cât mai mare posibil.

Există o mulțime de algoritmi foarte cunoscuți care pot fi utilizați pentru determinarea cuplajului maxim. Un exemplu în acest sens este algoritmul *Hopcroft-Karp*.

Totuși, acești algoritmi se vor dovedi prea lenți pentru a rezolva problema de față. Ei nu folosesc o particularitate a problemei și anume faptul că mulțimea E este *convexă*, deci vom avea întotdeauna:

$\forall_{(i,j),(k,j)\in E}(i+1,j),(i+2,j),\ldots,(k-1,j)\in E.$

Folosind această observație putem rezolva problema utilizând metoda *greedy*.

Metoda *greedy*

Algoritmul greedy care poate fi utilizat pentru rezolvarea acestei probleme este schițat în continuare:

- se sortează grupurile în ordine crescătoare, în funcție de dimensiunile lor;
- pentru fiecare grup i:
  - se determină o excursie j care nu a mai fost utilizată, astfel încât $u_j \geq s_i \geq l_j$ și valoarea u_j este cât mai mică posibil (s_i reprezintă numărul membrilor grupului i, l_j reprezintă dimensiunea minimă a grupului j, iar u_j reprezintă dimensiunea maximă a grupului j);
  - dacă o astfel de excursie există, atunci grupul i va pleca în excursia j (de acum excursia j este considerată a fi utilizată), iar numărul total al excursiilor va fi incrementat;
- se scriu rezultatele (perechile determinate).

Demonstrarea corectitudinii

După realizarea corespondenței dintre un grup și o excursie, excursia este eliminată din lista excursiilor.

Să presupunem că am realizat în mod optim corespondențele pentru primele k - 1 grupuri și dorim să realizăm acum corespondența pentru cel de-al k-lea grup.

De asemenea, presupunem că grupul k poate pleca în una dintre excursiile a_1, a_2, ..., a_p (aceste excursii nu au fost utilizate anterior) și că avem $u_{a_1} \leq u_{a_2} \leq \cdots \leq u_{a_p}$.

Datorită faptului că am reușit să realizăm în mod optim corespondențele pentru primele k - 1 grupuri, știm cu siguranță că există cel puțin o soluție optimă S. Vom arăta că există o soluție optimă în care se realizează o corespondență înre grupul k și excursia a_1 și corespondențele pentru toate grupurile anterioare sunt cele deja stabilite.

Există trei posibilități:

- în cadrul soluției S, al k-lea grup pleacă în excursia a_1;
- în cadrul soluției S, al k-lea grup pleacă într-o excursie a_x, și avem $x \neq 1$.
- în cadrul soluției S, pentru al k-lea nu se realizează nici o corespondență cu nici o excursie.

Dacă prima condiție este îndeplinită, atunci totul este în regulă și avem o soluție optimă care conține corespondențele pe care le dorim.

Dacă pentru grupul k nu se stabilește nici o corespondență cu o excursie, atunci este necesar ca excursia a_1 să fi fost pusă în corespondență cu un alt grup. În caz contrar, prin stabilirea corespondenței între grupul k și excursia a_1 am obține o soluție mai bună.

Să presupunem că, în cadrul soluției S, excursia a_1 este pusă în corespondență cu un grup z.

Evident, grupul z nu a fost încă procesat deoarece astfel excursia a_1 ar fi fost deja eliminată din listă (excursiile sunt eliminate după ce sunt utilizate).

Așadar, dacă realizăm o corespondență între al k-lea grup și excursia a_1 și nu vom realiza ulterior nici o corespondență pentru grupul z, vom avea în continuare o soluție optimă (numărul corespondențelor este același).

Ultima posibilitate este ca al k-lea grup să fie pus în corespondență cu o excursie a_x, unde $x \neq 1$. Distingem două cazuri:

- pentru excursia a_1 nu este stabilită nici o corespondență;
- excursia a_1 este pusă în corespondență cu un grup z, care nu a fost încă procesat.

Dacă pentru excursia a_1 nu este stabilită nici o corespondență, atunci putem să modificăm pur și simplu corespondența pentru al k-lea grup. Acestuia nu îi va mai corespunde excursia a_x, ci excursia a_1, iar numărul corespondențelor nu se va modifica.

Dacă excursia a_1 este pusă în corespondență cu un grup z și acest grup nu a fost încă procesat, atunci putem intersechimba cele două coresponențe. Celui de-al k-lea grup îi va corespunde excursia a_1, iar grupului z îi va corespunde excursia a_x.

Putem realiza această interschimbare deoarece, în cazul în care al k-lea grup este pus în corespondență cu excursia a_1 și grupul z este pus în corespondență cu excursia a_x, atunci avem:

$$l_{a_1} \leq s_k \leq u_{a_1};$$
$$l_{a_x} \leq s_z \leq u_{a_x}.$$

Din definiția valorilor a_1 și a_x obținem $l_{a_x}, l_{a_1} \leq s_k$ și $u_{a_1} \leq u_{a_x}$.

Pe baza acestor relații ajungem la următoarea inegalitate:

$$l_{a_1}, l_{a_x} \leq s_k \leq s_z \leq u_{a_1} \leq u_{a_x}$$

Așadar, putem realiza corespondența dintre al k-lea grup și excursia a_1, precum și corespondența dintre grupul z și excursia a_x fără a reduce numărul total al corespondențelor realizate.

Am reușit astfel să demonstrăm că dacă algoritmul realizează în mod optim corespondențele pentru primele k - 1 grupuri, atunci el va realiza în mod optim și corespondența pentru al k-lea grup.

Folosind metoda inducției matematice și presupunând cu pentru zero grupuri corespondențele sunt realizate optim, tragem concluzia că algoritmul anterior determină întotdeauna răspunsul corect.

Detalii de implementare

Soluția constă în utilizarea unui *heap* H pentru a păstra excursiile posibile

pentru grupul curent. Dacă grupul se modifică, atunci anumite excursii sunt adăugate și alte excursii sunt eliminate.

Rădăcina *heap*-ului va conține întotdeauna o excursie pentru care valoarea u_i este cea mai mică dintre valorile corespunzătoare tuturor elementelor din *heap*.

Algoritmul constă în următorii pași:

- se ordonează crescător grupurile în funcție de valorile s_i;
- se ordonează crescător excursiile în funcție de valorile l_j;
- pentru fiecare grup i (în ordinea dată de valorile s_i):
  - se adaugă în *heap*-ul H toate excursiile care nu au fost adăugate anterior și pentru care numărul minim de participanți l_j este mai mic sau egal cu numărul membrilor grupului i;
  - se elimină din *heap*-ul H toate excursiile pentru care numărul maxim de participanți u_j este mai mic decât numărul membrilor grupului i;
  - se elimină din *heap*-ul H elementul cu valoarea minimă (numai dacă *heap*-ul nu este vid); excursia eliminată este pusă în corespondență cu grupul i.

Ordinul de complexitate al algoritmului prezentat este $O(n \cdot \log n + m \cdot \log m)$, iar spațiul de memorie necesar are ordinul $O(n + m)$.

P040610: Fotbal

Vom arăta modalitatea de determinare a soluției cu ajutorul unor leme pe care le vom și demonstra.

Limita inferioară

Vom arăta acum care este numărul minim al situațiilor în care o echipă joacă două meciuri consecutive pe teren propriu sau două meciuri consecutive în deplasare.

Lema #1

Limita inferioară a numărului situațiilor în care o chipă joacă două meciuri consecutive pe teren propriu sau două meciuri consecutive în deplasare este n - 2, unde n este numărul total al echipelor.

Demonstrație

Presupunem, fără a restrânge generalitatea, că în prima etapă echipele 1, ..., n / 2 joacă pe teren propriu, iar echipele n / 2 + 1, ..., n joacă în deplasare.

Când oricare două dintre primele n / 2 echipe trebuie să dispute un meci, una dintre ele va trebui să joace două meciuri consecutive acasă sau două meciuri consecutive în deplasare.

Definim *faza* ca fiind proprietatea echipelor de a juca meciurile, alternativ, pe teren propriu și în deplasare.

Inițial, faza primelor n / 2 echipe arată că acestea vor juca meciurile din etapele impare pe teren propriu și meciurile din etapele pare în deplasare, iar faza ultimelor n / 2 echipe arată că acestea vor juca meciurile din etapele pare pe teren propriu și meciurile din etapele impare în deplasare.

Pentru ca două dintre primele n / 2 echipe să poată disputa un meci, una dintre aceste echipe trebuie să își modifice faza.

Evident, prin modificarea fazei, o echipă joacă două meciuri consecutive pe teren propriu sau două meciuri consecutive în deplasare.

Să presupunem că echipa care își modifică faza este echipa n / 2. Trebuie observat faptul că, până în momentul în care această echipă își modifică faza, ea poate juca cu oricare dintre primele n / 2 - 1, cu condiția ca acestea să nu își fi modificat faza anterior.

Acest raționament poate fi apoi repetat pentru fiecare dintre primele n / 2 - 1 echipe, deoarace și ele trebuie să dispute partide unele cu altele.

Mai exact, fiecare dintre aceste echipe va trebui să își modifice o dată faza pentru a putea juca cu echipele cu număr de ordine mai mic.

Așadar, vor exista cel puțin n / 2 - 1 situații în care o echipă joacă două meciuri consecutive pe teren propriu sau două meciuri consecutive în deplasare.

Întregul raționament poate fi repetat și pentru ultimele n / 2 echipe (situația este simetrică celei analizate anterior). Astfel obținem alte cel puțin n / 2 - 1 situații, ceea ce duce la un număr total de cel puțin n - 2 situații.

Aplicând această lemă obținem o limită inferioară a numărului situațiilor în care o echipă joacă două meciuri consecutive pe teren propriu sau două meciuri consecutive în deplasare. De fapt, vom arăta că este întotdeauna posibilă obținerea unei programări a meciurilor astfel încât numărul total al acestor situații să fie exact n - 2.

Vom prezenta mai întâi modul în care putem stabili programarea meciurilor, vom arăta că sunt respectate toate condițiile impuse și vom demonstra că apare exact n - 2 situații în care o echipă joacă două meciuri consecutive pe teren propriu sau două meciuri consecutive în deplasare.

Stabilirea programului

Pentru a simplifica formulele matematice, vom numerota echipele și etape începând cu 0 (ultima echipă este n - 1, iar ultima etapă este n - 2).

Definim programul ca o funcție p_k: {0, 1, ..., n - 2} → {0, 1, ..., n - 1} pentru k = 0, 1, ..., n - 2.

Pentru $i \in \{0, 1, ..., n - 2\}$, alegem $j \in \{0, 1, ..., n - 2\}$ astfel încât $i + j \equiv k \pmod{n - 1}$. Dacă $i \neq j$, vom stabili valoarea $p_k(i) = j$, iar dacă $i = j$, vom stabili $p_k(i) = n - 1$.

În acest moment putem prezenta programul meciurilor pentru o etapă k. Vom nota $l = [(k + 1) / 2]$.

Dacă k este un număr par, atunci meciurile din cea de-a k-a etapă vor fi de forma: $(l, p_k(l))$, $(l + 1, p_k(l + 1))$, ..., $(l + n / 2 - 1, p_k(l + n / 2 - 1))$.

Dacă k este impar, atunci meciurile din cea de-a k-a etapă vor fi de forma: $(p_k(l), l)$, $(p_k(l + 1), l + 1)$, ..., $(p_k(l + n / 2 - 1), l + n / 2 - 1)$.

Demonstrația corectitudinii acestei programări va fi prezentată în continuare.

Corectitudinea și optimalitatea

Vom prezenta acum două leme care vor demonstra că în fiecare etapă o echipă are întotdeauna programat un singur meci și că între oricare două echipe se desfășoară exact un meci.

Lema #2

Prin programarea descrisă anterior, în fiecare etapă, fiecare echipă cuprinse între 0 și n - 1 dispută exact o partidă (pentru fiecare etapă programarea este corectă).

Demonstrație

Pentru a nu complica formulele matematice, vom lua în considerare separat cazurile în care valoarea k este pară și cazurile în care această valoare este impară.

Dacă k este un număr par, atunci există o valoare m astfel încât $k = 2 \cdot m$, deci avem $l = [(2 \cdot m + 1) / 2] = m$.

În aceste condiții perechile pe care le construim sunt: $(m, p_k(m))$, $(m + 1, p_k(m + 1))$, ..., $(m + n / 2 - 1, p_k(m + n / 2 - 1))$.

Dacă utilizăm definiția funcției p_k, atunci perechile sunt: $(m, n - 1)$, $(m + 1, (m - 1) \bmod (n - 1))$, ..., $(m + n / 2 - 1, (m - n / 2 + 1) \bmod (n - 1))$, unde prin **mod** am notat restul împărțirii întregi. Datorită faptului că împărțitorul este întotdeauna n - 1, resturile vor fi întotdeauna cuprinse între 0 și n - 2.

Numerele $m - n / 2 - 1$, ..., $m - 1$, m, $m + 1$, ..., $m + n / 2 - 1$ sunt n - 1 numere întregi consecutive, deci resturile operațiilor de împărțire a acestora la n - 1 sunt distincte. Aceste resturi vor reprezenta toate echipele cuprinse într 0 și n - 2, iar echipa n - 1 apare exact o dată în listă. Ca urmare, fiecare echipă apare exact o dată în lista de perechi.

Această afirmație este adevărată deoarece $k \le n - 2$, $m + n / 2 - 1 = k / 2 + n / 2 - 1 \le (n - 2) / 2 + n / 2 - 1 = n - 2$, deci nici una dintre echipele aflate pe prima poziție în perechi nu va fi n - 1 și, evident, nici una dintre echipele care apar pe a doua poziție în perechi (cu excepția primei perechi) nu poate fi n - 1.

Dacă k este un număr impar, atunci există o valoare m astfel încât $k = 2 \cdot m + 1$, deci avem $l = [(2 \cdot m + 2) / 2] = m + 1$.

În aceste condiții perechile pe care le construim sunt: $(p_k(m + 1), m + 1)$, $(p_k(m + 2), m + 2)$, ..., $(p_k(m + n / 2), m + n / 2)$.

Din nou, dacă utilizăm definiția funcției p_k, obținem perechile: $(m, m + 1)$, $((m - 1) \bmod (n - 1), m + 2)$, ..., $((m - n / 2) \bmod (n - 1), m + n / 2 - 1)$, $(n - 1, m + n / 2)$.

Valoarea n - 1 apare în ultima pereche deoarece avem $m - n / 2 + 1 = m + n / 2 - (n - 1)$, deci $m - n / 2 + 1 \equiv m + n / 2$ (modulo n - 1).

La fel ca și în cazul anterior, echipele cu numere de ordine cuprinse între 0 și n - 2 apar exact o dată în aceste perechi și, datorită faptului că avem $k = 2 \cdot m + 1 \le (n - 2)$, $m \le (n - 2) / 2$ (n este întotdeauna par), echipa cu numărul de ordine n - 1 apare și ea exact o dată.

Astfel, am încheiat demonstrarea acestei leme.

Lema #3

Dacă luăm în considerare întreaga programare, atunci fiecare pereche de echipe joacă exact un meci (întreaga programare este corectă).

Demonstrație

Fie o valoare $i \in \{0, 1, ..., n - 2\}$; echipele împotriva cărora va disputa meciurile echipa i vor avea numerele de ordine j_k astfel încât $i + j_k \equiv k$ (modulo n - 1), iar dacă valoarea j_k este egală cu i, atunci echipa oponentă va fi n - 1.

Afirmația anterioară este dedusă pe baza faptului că echipa i joacă exact o dată în fiecare etapă (potrivit lemei #2).

Ca urmare, aceste numere j_k vor fi: $(-i) \bmod (n - 1)$, $(-i + 1) \bmod (n - 1)$, ..., $(-i + n - 2) \bmod (n - 1)$.

Numerele $-i$, $-i + 1$, ..., $-i + n - 2$ sunt n - 1 numere întregi consecutive, deci resturile împărțirilor lor întregi la n - 1 sunt distincte (fiecare rest, de la 0 la n - 2, apare exact o dată).

Printre aceste resturi se va afla și valoarea i; pentru etapa corespunzătoare oponentul echipei i va fi echipa n - 1.

În concluzie, cu excepția valorii i, fiecare număr cuprins între 0 și n - 1 va apărea exact o dată între numerele $p_0(i)$, $p_1(i)$, $p_{n-2}(i)$.

Din această parte a demonstrației putem deduce că echipa n - 1 va juca de asemenea exact o dată cu fiecare dintre primele n - 2 echipe.

Această afirmație este adevărată deoarece pentru fiecare $i \in \{0, 1, ..., n - 2\}$ există o etapă în care echipa i joacă cu echipa n - 1.

Aplicând lema #2 ajungem la concluzia că echipa n - 1 joacă exact un meci în fiecare etapă.

Așadar, fiecare echipă joacă cu echipe diferite în fiecare etapă, deci lema #3 este adevărată.

Pe baza lemelor #2 și #3 deducem că programarea este corectă. Teorema următoare va demonstra că această programare este și optimă.

Teoremă

Programarea construită anterior este optimă, în sensul că există exact n - 2 situații în care o echipă joacă două meciuri consecutive pe teren propriu sau două meciuri consecutive în deplasare.

Demonstrație

Similar cazului demonstrației lemei #2 vom considera două cazuri distincte. Pentru primul caz vom "trece" de la o etapă cu număr de ordine par la o etapă cu număr de ordine impar, iar pentru al doilea vom "trece" de la o etapă cu număr de ordine impar la o etapă cu număr de ordine par.

Dacă luăm în considerare etapele k și $k + 1$, iar k este par, atunci, folosind perechile descrise în demonstrația lemei #2 (cu $m = k / 2$), obținem următoarele perechi pentru etapa k: $(m, n - 1)$, $(m + 1, (m - 1) \bmod (n - 1))$, $(m + 2, (m - 2) \bmod (n - 1))$,..., $(m + n / 2 - 1, (m - n / 2 + 1) \bmod (n - 1))$.

De asemenea, obținem următoarele perechi pentru etapa $k + 1$: $(m, m + 1)$, $((m - 1) \bmod (n - 1), m + 2)$, ..., $((m - n / 2) \bmod (n - 1), m + n / 2 - 1)$, $(n - 1, m + n / 2)$.

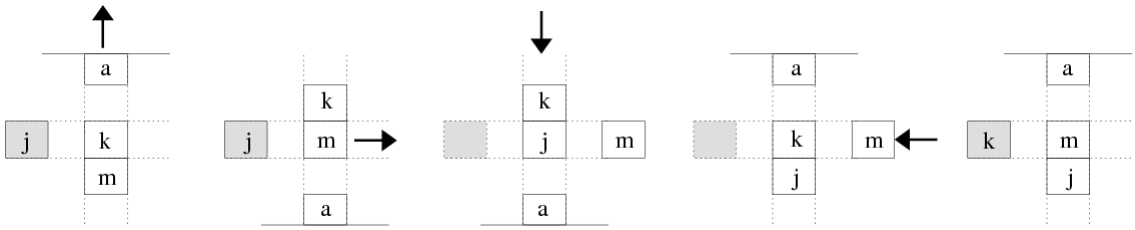

Figura 3: Cele patru deplasări necesare pentru a muta piesa k în poziția sa corectă (cea haşurată), în condițiile în care piesa se află deja pe linia corectă.

Putem observa imediat că numai echipa cu numărul de ordine m joacă, în această situație, două meciuri consecutive pe teren propriu și numai echipa cu numărul de ordine $m + n / 2$ joacă două meciuri consecutive în deplasare. Pentru a deduce acest lucru trebuia ținut cont de faptul că $m - n / 2 + 1 \equiv m + n / 2$ (modulo $n - 1$) (pe care l-am mai amintit).

Așadar, pentru oricare două etape consecutive dintre care prima are numărul de ordine par vom avea două echipe care își vor modifica faza (vor juca două meciuri consecutive pe teren propriu sau în deplasare). Datorită faptului că avem $(n / 2) - 1$ astfel de perechi de etape, numărul total al situațiilor în care o echipă joacă două meciuri consecutive pe teren propriu sau două meciuri consecutive în deplasare este $2 \cdot (n / 2 - 1) = n - 2$.

Dacă luăm în considerare etapele k și $k + 1$, iar k este impar, atunci, folosind perechile descrise în demonstrația lemei #2 (cu $m = [k / 2]$), obținem următoarele perechi pentru etapa k: $(m, m + 1)$, $((m - 1) \bmod (n - 1), m + 2)$, ..., $((m - n / 2) \bmod (n - 1), m + n / 2 - 1)$, $(n - 1, m + n / 2)$.

Pentru etapa $k + 1$ vom obține, tot pe baza lemei #2, următoarele perechi: $(m + 1, n - 1)$, $(m + 2, m \bmod (n - 1))$, $(m + 3, (m - 1) \bmod (n - 1))$, ..., $(m + n / 2, (m - n / 2 + 2) \bmod (n - 1))$.

Dacă studiem primele echipe din a doua mulțime de perechi și perechile aflate pe a doua poziție în prima mulțime de perechi, observăm că aceste mulțimi de echipe sunt identice.

Aceasta implică faptul că, dacă se utilizează o astfel de programare, după o etapă cu număr de ordine impar toate echipele își păstrează faza, deci nu apare nici o situație în care o echipă joacă două meciuri consecutive pe teren propriu sau două meciuri consecutive în deplasare.

După ce am luat în considerare ambele cazuri, am dedus că apar, în total, exact $n - 2$ situații în care o echipă joacă două meciuri consecutive pe teren propriu sau două meciuri consecutive în deplasare, ceea ce demonstrează teorema.

Soluția corectă și optimă are ordinul de complexitate $O(n^2)$ și necesită un spațiu de memorie de dimensiune constantă. Mai trebuie observat faptul că operația de scriere a datelor de ieșire are ordinul de complexitate $O(n^2)$.

P040611: Puzzle

Puzzle-ul prezentat în această problemă pare a fi foarte asemănător cu celebrul cub *Rubik*. Într-adevăr, abordarea pe care o vom prezenta va putea fi utilizată și pentru a rezolva cubul *Rubik*.

Algoritmul pe care îl vom descrie constă în două faze:

- aranjarea pieselor corespunzătoare pe toate liniile cu excepția ultimei linii;
- rearanjarea pieselor de pe ultima linie.

Prima fază

În această primă fază vom plasa piesele succesive 1, 2, ..., $n^2 - n$ în pozițiile corespunzătoare.

Să presupunem că am reușit să mutăm deja piesele 1, 2, ..., k în pozițiile corecte. Dorim acum să amplasăm piesa k în poziția sa corectă, fără ca vreuna dintre piesele anterioare să fie mutată din poziția sa.

Astfel, putem distinge patru situații:

- piesa se află deja în poziția corectă;
- piesa se află pe linie corectă;
- piesa se află pe coloana corectă;
- piesa nu se află nici pe linia, nici pe coloana corectă.

În prima situația nu trebuie efectuată nici o operație.

În cazul în care piesa se află pe linia corectă, procedăm așa cum se arată în figura 3.

Se observă că, după efectuarea celor patru deplasări, doar trei piese își modifică poziția: j, k și m.

Dacă piesa se află pe coloana corectă, procedăm într-o manieră simi-

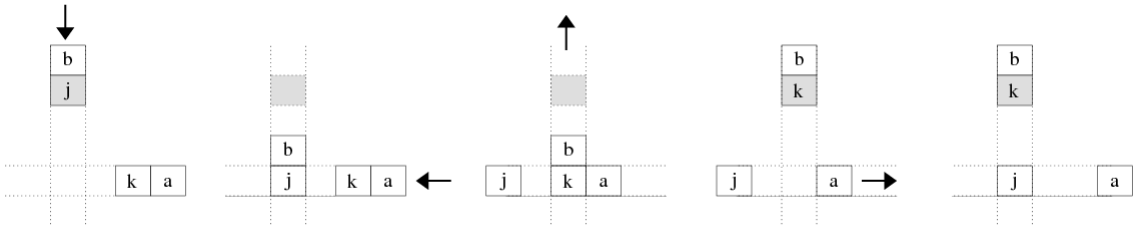

Figura 4: Cele patru deplasări necesare pentru a muta piesa k în poziția sa corectă (cea haşurată), în condițiile în care piesa nu se află nici pe linia, nici pe coloana corectă.

lară. Vom avea nevoie tot de patru deplasări și în acest caz.

Dacă piesa nu se află nici pe linia, nici pe coloana corectă, atunci vom efectua tot patru deplasări, după cum se poate vedea în figura 4.

Trebuie observat faptul că după cele patru deplasări prezentate în figură, doar două piese își modifică poziția: *j* și *k*.

Cu aceasta am încheiat prezentarea operațiilor efectuate în prima fază.

A doua fază

Înainte de a arăt modul în care sunt rearanjate piesele de pe ultima linie, trebuie să ne gândim puțin la problema formulată de rege - există ***întotdeauna*** o modalitate de a rezolva *puzzle*-ul?

Permutări

Vom prezenta acuma câteva elemente fundamentale referitoare la permutări.

Fie o permutare f: {1, ..., n} → {1, ..., n}. Dacă pornim de la un element arbitrar $x \in \{1, ..., n\}$ și iterăm funcția f obținând $f(x)$, $f(f(x))$, $f(f(f(x)))$, ..., după câteva iterații (cel mult n), vom obține din nou valoarea x. Rezultă imediat că fiecare permutare este o colecție de cicluri.

De exemplu, permutarea (2, 4, 5, 1, 3) este formată din două cicluri și putem scrie (2, 4, 5, 1, 3) = [2, 4, 1][5, 3]. De asemenea, mai spunem că permutarea (2, 4, 5, 1, 3) este produsul ciclurilor [2, 4, 1] și [5, 3].

Ciclurile de lungime 2 au o semnificație specială. Ele sunt numite și ***transpoziții***.

Fiecare ciclu poate fi scris ca un produs de transpoziții:
[1, 2, ..., k] = [1, k][1, k - 1]...[1, 3][1, 2]

Practic, în formula anterioară aplicăm succesiv transpozițiile de la dreapta spre stânga.

Se observă că orice permutare poate fi scrisă sub forma unui produs de transpoziții.

Descompunerea în transpoziții nu este unică. De exemplu, avem (2, 4, 5, 1, 3) = [4, 1][2, 1][5, 3] = [2, 4][4, 1][5, 3].

Nici numărul transpozițiilor care apar în două descompuneri diferite ale unei permutări nu trebuie să fie același. De exemplu, putem avea [1, 3][1, 7][1, 3] = [3, 4][4, 5][5, 6][6, 7][5, 6][4, 5][3, 4].

Totuși, există anumite proprietăți importante. Următoarea propoziție este destul de cunoscută.

Propoziție

Dacă o permutare f poate fi descompusă sub forma unui produs de transpoziții și numărul acestor transpoziții este par, atunci permutarea f nu poate fi descompusă sub forma unui produs de transpoziții în care numărul transpozițiilor este impar.

Ca urmare, putem împărți permutările în două grupuri: permutări ***pare*** care se descompun în produse din care fac parte un număr par de transpoziții și permutări ***impare*** care se descompun în produse din care fac parte un număr par de transpoziții.

Să considerăm o permutare care este formată dintr-un singur ciclu. Se observă că lungimea ciclului este pară dacă și numai dacă permutarea este impară.

Permutările și puzzle-ul

Este evident că putem trata configurația inițială ca fiind o permutare a celor n^2 piese. Vom nota această configurație prin π_0. Fiecare deplasare circulară este și ea o permutare și este formată dintr-un singur ciclu. Configurația finală este dată de permutarea identică *id*.

Dorim să efectuăm o serie de deplasări asupra configurației inițiale pentru a obține permutarea *id*.

Cu alte cuvinte, dorim să determinăm o secvență de permutări circulare s_2, s_2, ..., s_k, corespunzătoare unor deplasări ale liniilor sau ale coloanelor, q astfel încât să avem $s_k s_{k-1} ... s_2 s_1 \pi_0 = id$.

Să presupunem că valoarea n este impară. În acest caz fiecare s_i este o permutare pară. Datorită faptului că *id* este o permutare pară, reiese că *puzzle*-ul poate fi rezolvat numai dacă π_0 este o permutare pară.

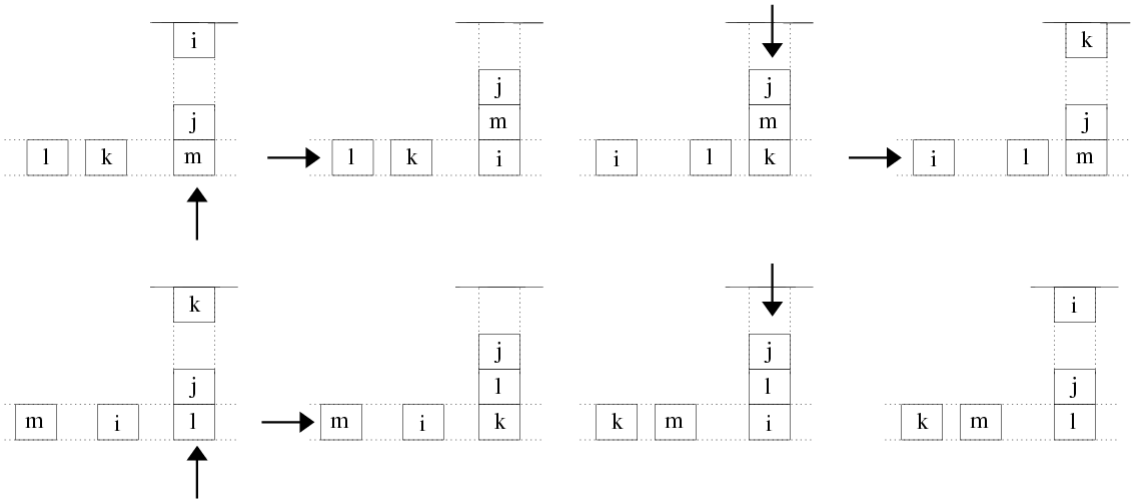

Figura 5: Rearanjarea ultimei linii: piesa k ajunge în poziția piesei l, piesa l ajunge în poziția piesei m și piesa m ajunge în poziția piesei k.

Operația de verificare a faptului că, în condițiile în care n este un număr impar, configurația inițială este reprezentată de un număr impar, are ordinul de complexitate $O(n^2)$.

Totuși, ce putem face dacă valoarea n este pară sau permutarea π_0 este pară? După prima fază obținem cel mult o linie care nu este aranjată corect (ultima linie). Se observă că, dacă n este un număr par și configurația obținută după prima fază este impară, atunci această configurație poate deveni pară dacă deplasăm cu o poziție ultima linie.

Așadar, în cele ce urmează, putem presupune că după prima fază am obținut o configurație a pieselor care este reprezentată de o permutare pară.

În acest codiții putem muta piesele succesive $n^2 - n + 1$, $n^2 - n + 2$, ..., n^2 în pozițiile corecte după cum se arată în continuare.

Considerăm piesa cu numărul de ordine k. Presupunem că o piesă cu numărul de ordine l se află pe poziția k. Trebuie să existe o piesă cu numărul de ordine m, astfel încât $m > k$ și $m \neq l$, deoarece altfel permutarea ar fi formată dintr-o singură transpoziție și ar fi impară.

În figura 5 se arată că sunt suficiente șapte mutări pentru a aplica permutarea $[k, l, m]$. Ca urmare, piesa k ajunge pe poziția sa corectă și fiecare piesă $j < k$ rămâne pe poziția sa corectă.

Se observă că numărul deplasărilor efectuate dacă se utilizează acest algoritm este întotdeuna cel mult egal cu $4 \cdot n \cdot (n - 1) + 7 \cdot n < 400.000$.

Detalii de implementare

Există o mulțime de posibilități de a implementa o soluție corectă, dar există câteva "trucuri" care îmbunătățesc semnificativ viteza de execuție.

Dacă păstrăm o tablă cu n^2 piese și simulăm fiecare mutare prin mutarea pieselor, atunci ordinul de complexitate devine $O(n^3)$.

Acest ordin de complexitate poate crește la $O(n^4)$ dacă nu păstrăm pozițiile pieselor (avem nevoie de un timp de ordinul $O(n^2)$ pentru a găsi fiecare dintre cele $O(n^2)$ piese).

Există o soluție cu ordinul de complexitate $O(n^2)$, care se bazează pe o observație simplă. Am prezentat trei modalități pentru a muta o piesă în poziția sa corectă.

Fiecare modalitate constă în grupuri de patru sau șapte deplasări, dar rezultatul acestor deplasări este modificarea pozițiilor a două sau trei piese.

Așadar, pozițiile pieselor după efectuarea unui astfel de grup de deplasări pot fi actualizate în timp constant.

P040612: Cherestea

Pentru început trebuie să observăm faptul că nu este eficient să construim fabrici de cherestea între copaci. Într-un astfel de caz costul va scădea cu siguranță dacă fabrica ar fi mutată în dreptul copacului aflat mai sus.

Înainte de a prezenta algoritmul de rezolvare, vom prezenta câteva definiții utile.

Vom nota prin $Cost(a, b)$ costul total al transportului în situația în care cele două fabrici de cherestea sunt construite în dreptul copacilor a și b, unde $a < b$.

În plus, presupunând că fabrica de cherestea aflată mai jos este situată în punctul i, poziția optimă a celeilalte fabrici va fi notată prin $opt(i)$. Dacă există mai multe poziții optime, atunci va fi aleasă cea mai mică dintre ele.

Va trebui să determinăm o pereche (a, b) astfel încât valoarea $Cost(a, b)$ să fie minimă.

Primul nostru obiectiv va fi determinarea unei modalități rapide de a calcula valorile $Cost(a, b)$.

Presupunem că stocăm următoarele valori:

- $D(i)$ - distanța dintre copacul 1 și copacul n; $D(n + 1)$ va conține distanța dintre copacul 1 și fabrica de cherestea aflată la poalele dealului;
- $W(i)$ - suma greutăților copacilor 1, 2, ..., i;
- $CostTotal$ - costul transportării tuturor copacilor la fabrica de la poalele dealului (când nu există nici o altă fabrică).

Operațiile de determinarea a tuturor acestor valori au ordinul de complexitate total $O(n)$. Pentru a le calcula este suficient să plecăm de la primul copac și să ne deplasăm înspre poalele dealului, adunând greutăți și distanțe.

Se observă faptul că după această preprocesare se pot calcula valorile $Cost(a, b)$ în timp constant, pentru orice pereche (a, b).

Următoarea formulă ușor de verificat este adevărată:

$Cost(a, b) = CostTotal - W(a) \cdot (D(b) - D(a)) - W(b) \cdot (D(n + 1) - D(b))$.

Algoritmul pătratic

În acest moment putem prezenta un algoritm care are ordinul de complexitate $O(n^2)$. Se execută în timp liniar preprocesarea prezentată și apoi se verifică toate posibilitățile de a alege pozițiile celor două fabrici de cherestea. Pentru fiecare alegere avem nevoie de un timp constant pentru a calcula costul, iar numărul acestor perechi este $O(n^2)$.

Un algoritm mai rapid

Din nefericire abordarea prezentată anterior nu este suficient de rapidă. Vom prezenta o observație cheie care ne va ajuta să îmbunătățim semnificativ performanțele.

Observație

Pentru fiecare $i < j$, avem $opt(i) < opt(j)$.

Cu alte cuvinte, dacă mutăm fabrica aflată mai jos înspre poalele dealului, fabrica aflată mai sus nu se poate muta înspre vârf.

Nu este foarte dificil să demonstrăm această afirmație. Să considerăm următoarea lemă:

Lemă

Pentru fiecare $k < k'$ și $i < j$, dacă avem $Cost(k', i) \leq Cost(k, i)$, atunci $Cost(k', j) \leq Cost(k, j)$.

Demonstrație

Inegalitatea $Cost(k', i) \leq Cost(k, i)$ arată că este mai eficient să transportăm copacii 1, 2, ..., k de la poziția k la poziția k' decât să transportăm copacii $k + 1$, $k + 2$, ..., k' de la poziția k' la poziția i.

Este evident că pentru transportarea copacilor $k + 1$, $k + 2$, ..., k' de la poziția k' la poziția j costul va fi cel puțin la fel de mare.

Putem deduce imediat că avem, într-adevăr, $Cost(k', j) \le Cost(k, j)$.

Să introducem acum în lema anterioară $k' = opt(i)$. Inegalitatea $Cost(k', i) \le Cost(k, i)$ este adevărată (este ipoteza). Lema ne arată că oricare $k < k'$ nu poate fi $opt(j)$ deoarece ar duce la obținerea uni cost mai mare decât k'. Așadar, am ajuns la concluzia că observația prezentată este întotdeauna adevărată.

În cele ce urmează vom prezenta modul în care putem obține un algoritm mai rapid pe baza acestei observații.

Prezentarea algoritmului

Să presupunem că am reușit să calculăm valoarea $a^* = opt([n / 2])$. Știm că pentru orice $i < [n / 2]$ vom avea $opt(i) \le a^*$ și că pentru orice $j > [n / 2]$ vom avea $opt(j) \ge a^*$.

Așadar, putem crea o rutină recursivă $Solve(a_{min}, a_{max}, b_{min}, b_{max})$ care determină $a \in \{a_{min}, ..., a_{max}\}$ și $b \in \{b_{min}, ..., b_{max}\}$ astfel încât valoarea $Cost(a, b)$ este minimă.

Rutina determină valoarea $a^* = opt([(b_{max} - b_{min}) / 2])$, într-un timp de ordinul $O(a_{max} - a_{min})$.

În continuare se apelează recursiv:

$Solve(a_{min}, a^*, b_{min}, [(b_{max} - b_{min}) / 2] - 1)$

și

$Solve(a^*, a_{max}, [(b_{max} - b_{min}) / 2] + 1, b_{max})$.

Evident, apelul inițial va fi:

$Solve(1, n, 1, n)$.

Analiza complexității

Se observă imediat că numărul perechilor verificate se reduce. Vom încerca acum să estimăm acest număr al perechilor.

Să considerăm arborele binar al apelurilor recursive. Adâncimea acestuia este mărginită de valoarea $[\log n] + 1$ deoarece valoarea $b_{max} - b_{min}$ este înjumătățită la fiecare pas.

La fiecare nivel al aborelui cele n valori posibile pentru a sunt împărțite în mai multe "segmente" astfel încât două segmente consecutive au exact o valoare comună.

Rezultă că numărul total al valorilor a luate în considerare la fiecare nivel are ordinul $O(n)$.

Ca urmare, ordinul de complexitate al operației de determinare a valorilor $opt(\cdot)$ de la fiecare nivel este $O(n)$.

În concluzie, ordinul de complexitate al întregului algolritm este $O(n \cdot \log n)$ deoarece avem $O(\log n)$ niveluri în arbore.

Alte soluții

Există mai multe alte abordări pentru această problemă, dar nu toate sunt corecte. De asemenea, există generalizări ale problemei.

Soluții naive

Există o mulțime de algoritmi cu ordinul de complexitate $O(n^3)$ sau chiar $O(n^4)$.

Aceștia iau în considerare toate variantele de amplasare pentru cele două fabrici de cherestea și determină costurile pentru aceste poziții cu ajutorul unor algoritmi lenți care pot fi liniari sau chiar pătratici dacă se utilizează metoda forței brute.

Mai multe fabrici

În continuare vom prezenta un algoritm care rezolvă problema generalizată: trebuie amplasate k fabrici de cherestea.

Pentru fiecare i cuprins între 1 și n și fiecare l cuprins între 1 și k vom nota prin T_{il} costul optim pentru amplasarea a k fabrici de cherestea dacă se iau în considerare doar copacii cu numere de ordine cuprinse între 1 și i. De asemenea, presupunem că există o fabrică suplimentară aflată în dreptul celui de-al $(i + 1)$-lea copac.

Pentru a rezolva problema vom aplica metoda programării dinamice. Valoarea T_{il} poate fi determinată într-un timp de ordinul $O(i)$. Practic, vom verifica toate posibilitățile de amplasare a celei de-a i-a fabrici și vom utiliza valorile deja păstrate $T_{1,l-1}$, ..., $T_{i-1,l-1}$.

Ordinul de complexitate al întregului algoritm este $O(k \cdot n^2)$. pentru $k = 2$ obținem un alt algoritm pătratic de rezolvare a problemei inițiale.

Soluții greedy eronate

Putem încerca să determinăm poziția optimă a unei fabrici de cherestea în timp liniar. Una dintre cele două fabrici va fi amplasată în acea poziție.

Acum, putem verifica toate celelalte poziții pentru cea de-a doua fabrică și să o alegem pe cea pentru care obținem un cost total minim.

Ordinul de complexitate al acestui algoritm este $O(n)$ dar, din nefericire, algoritmul nu funcționează corect decât pentru o mică parte dintre situații.

Despre un algoritm liniar...

Problema generalizată este cunoscută sub denumirea de ***problema k-medianei***.

Această problemă cere alegerea a k vârfuri ale unui graf ale cărui vârfuri și muchii au atașate costuri. În cele k vârfuri vor fi amplasate facilități, iar vârfurile trebuie să fie alese astfel încât costul total să fie minim.

Costul total este dat de suma produselor dintre costurile vârfurile și distanțele (costul drumului minim) până la cea mai apropiată facilitate.

Pentru cazul general această problemă este *NP*-completă, indiferent dacă graful este orientat sau nu.

Pentru arbori problema poate fi rezolvată în timp polinomial.

Pentru un drum orientat există un algoritm cu ordinul de complexitate $O(n \cdot k)$. Acest algoritm a fost prezentat în lucrarea *Placing Resources on a Growing Line*, publicat de către ***V. Auletta***, ***D. Parente*** și ***G. Persiano*** în *Journal of Algorithms*, vol. 26, no. 1, 1998.

Pentru cazul $k = 2$ obținem un algoritm liniar de rezolvare a problemei de la concurs. Totuși, din nefericire algoritmul este mult prea complex pentru a putea fi implementat în cadrul unui concurs de programare.