ONI 2002, Brăila

Olimpiada NAȚIONALĂ de Informatică

Vă prezentăm în continuare soluțiile problemelor propuse la ONI 2002 la clasele a XI-a și a XII-a. Aceste soluții au fost realizate de redacția GInfo pe baza soluțiilor oficiale prezentate de autorii problemelor și au fost verificate folosind seturile de date cu care au fost testate soluțiile concurenților.

P050213: Arbore

Vom spune că un nod este ***rezolvat*** dacă și numai dacă el *face parte din exact un ciclu*. Similar, vom spune că un subarbore este ***rezolvat*** dacă și numai dacă *toate nodurile acestuia sunt rezolvate*.

Vom numi ***fir*** un subarbore care este *lanț* (fiecare nod, cu excepția frunzei, are un singur fiu). Un fir este ilustrat în figura 1.

Figura 1: Fir

Se observă că un fir poate fi rezolvat foarte simplu prin *unirea rădăcinii sale cu frunza sa*. Evident, firul va putea fi rezolvat numai dacă el *conține cel puțin două noduri*. Procedeul este ilustrat în figura 2.

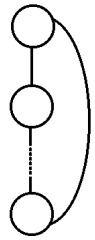


Figura 2: Fir rezolvat

Pentru a rezolva această problemă vom alege, pentru început, un nod care va fi rădăcina arborelui. Vom spune că un subarbore este ***potențial rezolvabil*** dacă *toți fiii rădăcinii subarborelui* (în număr de cel puțin doi) *sunt rădăcini ale unor fire*.

În continuare vom arăta modul în care poate fi rezolvat un nod potențial rezolvabil. Vom alege cele mai scurte două fire (acestea pot conține unul, două sau mai multe noduri) și *vom uni printr-o muchie frunzele acestor fire*. Astfel vom obține un ciclu care va conține nodurile din componența celor două fire, precum și nodul potențial rezolvabil. Eventualele fire rămase (dacă sunt formate din cel puțin trei noduri) vor fi rezolvate așa cum se arată în figura 2. Se observă că, în cazul în care există mai mult de două fire care conțin unul sau două noduri, subarborele nu poate fi rezolvat și problema *nu are soluție*. Rezolvarea unui subarbore potențial rezolvabil este ilustrată în figura 3.

După rezolvarea unui subarbore, acesta este ***eliminat*** din arbore și se va alege un alt nod potențial rezolvabil. Procedeul continuă până la *rezolvarea tuturor nodurilor* sau până la *detectarea unui subarbore care nu poate fi rezolvat*.

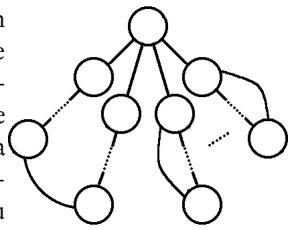


Figura 3: Rezolvarea unui subarbore potențial rezolvabil

În cele ce urmează vom prezenta modul în care poate fi implementat algoritmul descris. Pentru început vom alege o modalitate de reprezentare a arborelui. Vom construi o *listă de fii* și vom păstra, pentru fiecare nod, *indicele primului și al ultimului fiu* în această listă. Prin convenție vom considera că pentru un nod care nu are nici un fiu (frunză) indicele ultimului fiu va fi mai mic decât indicele primului fiu; de asemenea, pentru fiecare nod vom păstra *nodul părinte*. Datorită faptului că arborele este dat prin șirul muchiilor, va trebui să îl construim. Pentru aceasta vom realiza o parcurgere în lățime a arborelui, ceea ce permite construirea listei fiilor pe măsura traversării nodurilor arborelui.

În continuare vom alege rădăcina arborelui; aceasta poate fi orice nod care are *gradul mai mare decât 1*. Vom continua cu identificarea fiilor; evident, toate frunzele sunt fire de lungime 1. Pentru fiecare frunză vom "urca în arbore" atâta timp cât nodul curent are un singur fiu (părintele frunzei va fi rădăcina unui fir de lungime 2, părintele părintelui va fi rădăcina unui fir de lungime 3 și așa mai departe). Folosind acest procedeu vom determina și lungimile firelor. Prin convenție, vom considera că dacă un nod nu face parte dintr-un fir, atunci lungimea corespunzătoare va fi -1.

Acum, la fiecare pas vom căuta un nod care conține cel puțin doi fii nerezolvați care sunt rădăcini ale unor fire. Vom determina numărul firelor de lungime 1 sau 2 și, în cazul în care există cel mult două astfel de fire, vom rezolva subarborele așa cum se arată în figura 3. Dacă există mai mult de două astfel de fire vom scrie în fișierul de ieșire valoarea -1 și vom opri execuția programului. După rezolvarea subarborelui, acesta este eliminat din arbore prin mar-

carea tuturor nodurilor sale ca fiind rezolvate. Acum părintele rădăcinii subarborelui ar putea să facă parte dintr-un fir. În cazul în care părintele a rămas cu un singur fiu care este rădăcină a unui fir, atunci părintele devine rădăcină a unui fir de lungime cu 1 mai mare. Vom "urca" din nou în arbore până la întâlnirea unui nod care are mai mulți fii nerezolvați și vom păstra, pentru fiecare nod, lungimea firului cu rădăcina în nodul respectiv. O a doua posibilitate este ca părintele rădăcinii subarborelui eliminat să devină frunză. Se va "urca" în arbore în mod asemănător, dar noua frunză va fi rădăcina unui fir de lungime 1.

Există posibilitatea ca, la un moment dat, să rămânem cu un arbore format dintr-un singur fir. Dacă acesta are lungimea mai mare decât 2, atunci el va fi rezolvat așa cum se arată în figura 2. În caz contrar, problema nu are soluție.

De fiecare dată când rezolvăm un subarbore, vom păstra o listă cu muchiile adăugate. În final, vom scrie în fișierul de ieșire numărul muchiilor adăugate, precum și extremitățile muchiilor din listă.

Analiza complexității

Citirea datelor de intrare se realizează în timp **liniar**, deoarece implică citirea extremităților celor N - 1 muchii ale arborelui.

Construirea arborelui prin parcurgerea în lățime implică, pentru fiecare nod parcurs, studierea tuturor muchiilor. Deoarece avem N noduri și N - 1 muchii, ordinul de complexitate al operației este $O(N^2)$. Pe parcursul construirii arborelui se creează șirul părinților, cel al fiilor, cel al gradelor, precum și cele care păstrează indicii primului și ultimului fiu pentru fiecare nod.

Pentru păstrarea informațiilor inițiale despre fire, vom parcurge, în cel mai defavorabil caz, toate nodurile arborelui, deci operația se realizează în timp **liniar**. Trebuie observat faptul că acest ordin de complexitate include și timpul consumat pentru actualizarea necesară după eliminarea unui nod din arbore.

Alegerea unui nod potențial rezolvabil implică traversarea listei fiilor pentru a verifica dacă sunt rădăcini ale unor fire, deci ordinul său de complexitate este $O(N)$. La fiecare pas vom elimina cel puțin trei noduri, deci vom realiza cel mult $N / 3$ astfel de căutări. Ca urmare, ordinul de complexitate al tuturor alegerilor este $O(N^2)$.

Pentru rezolvarea arborelui este necesară parcurgerea tuturor nodurilor acestuia. Un nod rezolvat nu va fi parcurs încă o dată. Așadar, ordinul de complexitate al acestei operații este $O(N)$.

Ordinul de complexitate al operațiilor de eliminare a nodurilor este tot $O(N)$ pentru că fiecare nod este eliminat o singură dată.

Scrierea datelor în fișierul de ieșire se realizează tot în timp **liniar** pentru că implică scrierea extremităților a cel mult $N / 3$ muchii.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(N) + O(N^2) + O(N) + O(N^2) + O(N) + O(N) + O(N) = O(N^2)$.

Dificultate	57
Algoritmi	★★★☆☆
Structuri de date	★★☆☆☆
Originalitate	★★☆☆☆
Cunoștințe	★★★☆☆
Implementare	★★★☆☆

www.ginfo.ro/revista/12_7/surse/arbore.cpp

P050214: Decodificare

Inițial vom determina numărul de poziții în care permutarea identică și permutarea care trebuie determinată coincid.

În continuare vom încerca să determinăm pozițiile în permutare ale tuturor numerelor cuprinse între 1 și N. Pentru fiecare număr i, dacă poziția sa nu a fost determinată la un pas anterior, va trebui să determinăm poziția acestuia în permutarea căutată. Pentru aceasta, în permutarea identică, vom interschimba, pe rând, numărul i cu fiecare dintre celelalte elemente. Astfel, vom obține permutări de forma $(1, 2, 3, ..., i - 1, j, i + 1, ..., j - 1, i, j + 1, ..., N)$. Numărul pozițiilor, în care această permutare și permutarea care trebuie considerată coincid, poate crește dacă și numai dacă numărul i ajunge pe poziția sa sau numărul care trebuie să se afle pe poziția i ajunge în această poziție. Așadar, pentru doar două dintre aceste permutări numărul pozițiilor considerate poate să crească. Există trei cazuri:

- Numărul pozițiilor care coincid nu crește în nici un caz. În această situație putem trage concluzia că numărul i se află deja pe poziția sa, deci va fi al i-lea element în permutarea care trebuie determinată.
- Numărul pozițiilor care coincid crește o singură dată. Fie j numărul cu care a fost interschimbat i în permutarea care a dus la creșterea numărului de poziții care coincid. Este ușor de observat că, în permutarea care trebuie determinată, numărul i se va afla pe poziția j, iar numărul j se va afla pe poziția i.
- Numărul pozițiilor care coincid crește de două ori. Fie j și k pozițiile pe care a ajuns numărul i în cele două permutări. Va trebui să determinăm pe care dintre aceste două poziții se află i în permutarea care trebuie determinată. Știm cu siguranță că unul dintre numerele j și k se află pe poziția i în permutarea care trebuie determinată. Va trebui să studiem două cazuri:
  - ♦ i trece pe poziția j, j pe poziția k și k pe poziția i;
  - ♦ i trece pe poziția k, j pe poziția i și k pe poziția j.

  În unul dintre aceste două cazuri numărul pozițiilor care coincid va fi cu cel puțin 2 mai mare decât numărul pozițiilor care coincid în permutarea identică. Așadar, vom verifica una dintre permutări; dacă numărul pozițiilor care coincid crește cu cel puțin 2, atunci vom ști care este poziția numărului i în permutarea care trebuie determinată și care este numărul care se află pe poziția i în această permutare.

După determinarea pozițiilor tuturor numerelor, vom afișa permutarea determinată.

Analiza complexității

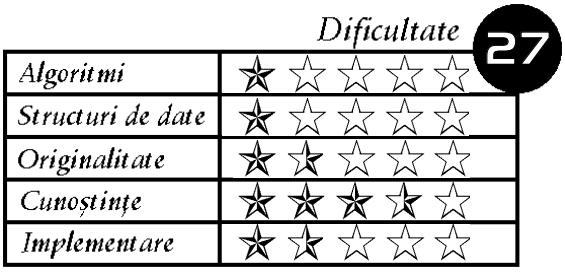
Pentru început, trebuie precizat faptul că apelul funcției `Check` implică traversarea unui vector cu N elemente care conține permutarea care trebuie determinată. Așadar ordinul de complexitate al unui astfel de apel este $O(N)$.

Preluarea dimensiunii șirului se realizează în timp constant, deci are ordinul de complexitate $O(1)$.

Identificarea poziției unui număr implică realizarea a cel mult N - 1 interschimbări și, eventual, a unei permutări care va diferi de permutarea identică în trei poziții. Pentru fiecare dintre aceste permutări se va realiza un apel al funcției `Check`, deci operația de determinare a poziției unui număr în permutarea căutată are ordinul de complexitate $O(N) \cdot O(N) = O(N^2)$. În total, vom efectua cel mult N determinări, deci ordinul de complexitate al operației de determinare a permutării căutate este $O(N) \cdot O(N^2) = O(N^3)$.

Scrierea datelor în fișierul de ieșire implică traversarea permutării determinate, deci ordinul de complexitate al operației este $O(N)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(N^3) + O(N) = O(N^3)$.



www.ginfo.ro/revista/12_7/surse/decod.cpp

P050215: SETI

Pentru a determina numărul de apariții ale cuvintelor din dicționar în mesajul dat, vom ordona alfabetic toate șirurile (cuvintele) de 16 caractere consecutive din mesaj. Apoi, pentru fiecare cuvânt din dicționar vom determina, prin metoda căutării binare, prima și ultima apariție a cuvântului în șirul sortat. Evident, dacă un cuvânt din dicționar conține k litere, atunci, în momentul în care determinăm aceste două poziții, vom lua în considerare doar primele k litere ale șirurilor.

O primă observație ar fi că, datorită faptului că nu avem suficientă memorie la dispoziție, șirul sortat trebuie să conțină indici și nu cuvinte propriu-zise. Astfel, al i-lea element al șirului sortat va conține poziția la care începe al i-lea cuvânt în șirul sortat.

Datorită limitei de timp foarte stricte, algoritmii clasici de sortare (*quicksort*, *heapsort*) nu sunt suficient de performanți. Pentru această problemă este recomandabilă folosirea algoritmului *radixsort* (vezi *CLR*[1], p. 152).

O altă dificultate care apare este datorată tot insuficienței memoriei. Șirul sortat de indici ar trebui să conțină numere reprezentate pe patru octeți (teoretic ar fi suficienți trei) deoarece textul poate conține peste 100000 de cuvinte. Pentru a evita această problemă, vom împărți textul în patru blocuri și vom aplica algoritmul descris pentru fiecare dintre ele. Fiecare bloc va conține cel mult $2^{15} = 32768$ litere. Datorită faptului că numărul de litere este cel mult $2047 \cdot 64 = 131008 < 2^{17} = 131072$, sunt suficiente patru blocuri. Vom avea și patru șiruri sortate, dar fiecare element al acestor șiruri va putea fi reprezentat pe doi octeți.

Datorită împărțirii textelor în blocuri apar probleme pentru cuvintele care fac parte din două blocuri. Pentru a le evita, vom face în așa fel încât ultimele 15 litere dintr-un bloc să fie adăugate ca primele 15 litere ale blocului următor. Dimensiunea totală a textului va crește cu până la 45 de caractere, dar va ajunge doar la cel mult $131053 < 2^{17}$, deci nu apar blocuri suplimentare. Se observă că la începutul celui de-al doilea bloc sunt copiate 15 litere din primul, deci ultimele 15 litere ale acestui al doilea grup trebuie să fie mutate în al treilea. Alte 15 litere ajung să fie ultimele în al doilea bloc, deci trebuie copiate în al treilea. Așadar, la începutul celui de-al treilea bloc vor fi ultimele 30 de litere ale celui de-al doilea bloc. Folosind același raționament, ajungem la concluzia că la începutul celui de-al patrulea bloc vor fi ultimele 45 de litere ale celui de-al treilea bloc.

Așadar, pentru rezolvarea problemei, mai întâi vom împărți textul inițial în grupuri de 32768 de litere. Apoi vom realiza copierea grupurilor de 15, 30 și 45 de litere dintr-un bloc în altul și vom construi cele patru șiruri sortate, folosind algoritmul *radixsort*. În continuare, pentru fiecare cuvânt din dicționar, vom determina prima și ultima apariție a sa în fiecare dintre șirurile sortate (dacă nu există nici o apariție vom spune, prin convenție, că poziția ultimei apariții este mai mică decât poziția primei apariții). Numărul de cuvinte dintre cele două poziții va fi adăugat la numărul de apariții în text ale cuvântului din dicționar. Pe măsură ce sunt determinate aparițiile cuvintelor din dicționar, se vor scrie în fișierul de ieșire numerele corespunzătoare.

Analiza complexității

Citirea textului care trebuie analizat se realizează linie cu linie, deci ordinul de complexitate al operației este $O(N)$. Deoarece dicționarul conține M cuvinte, ordinul de complexitate al operației de citire a acestor cuvinte este $O(M)$.

Construirea blocurilor se realizează pe parcursul citirii, ordinul de complexitate al operației fiind $O(N)$. Aceeași complexitate ($O(N)$) o are și operația de transformare a blocurilor astfel încât ultimele 15 litere dintr-un grup să fie identice cu primele 15 din următorul.

Algoritmul *radixsort* este ***liniar***, deci ordinul de complexitate al operației de sortare este $O(N)$.

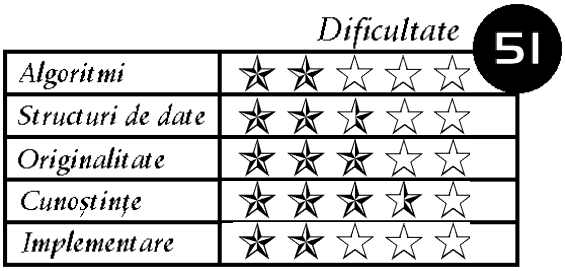
Pentru fiecare cuvânt din dicționar vom efectua cel mult opt căutări binare, ordinul de complexitate al unei căutări fiind $O(\log N)$. Pentru cele 8 căutări vom avea ordinul de complexitate $8 \cdot O(\log N) = O(\log N)$. Deoare-

1. *Cormen T. H., Leiserson C. E., Rivest R. L., Introducere în Algortmi, Computer Libris Agora, Cluj-Napoca, 2000*

ce pentru fiecare dintre cele M cuvinte din dicționar se vor realiza căutări binare, ordinul de complexitate al operației de determinare a numărului de apariții pentru toate cuvintele din dicționar este $O(M) \cdot O(\log N) = O(M \cdot \log N)$.

Generarea datelor de ieșire implică scrierea a M numere, deci ordinul de complexitate al operației este $O(M)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N) + O(M) + O(N) + O(N) + O(N) + O(M \cdot \log N) + O(M) = O(M \cdot \log N + N)$.



www.ginfo.ro/revista/12_7/surse/seti.cpp

P050216: Suma divizorilor

Numărul A poate fi scris ca produs de factori primi sub forma:

$$A = d_1^{p_1} \cdot d_2^{p_2} \cdot \ldots \cdot d_n^{p_n},$$

unde $d_1, ..., d_n$ sunt divizorii primi ai lui A, iar $p_1, ..., p_n$ sunt puterile la care apar aceștia în descompunerea lui A în factori primi.

Ca urmare, numărul A^B poate fi scris sub forma:

$$A^B = d_1^{p_1 \cdot B} \cdot d_2^{p_2 \cdot B} \cdot \ldots \cdot d_n^{p_n \cdot B}$$

Așadar, divizorii numărului A^B au forma $\prod_{i=1}^{n} d_i^{q_i}$, unde q_i variază între 0 și $p_i \cdot B$.

Suma acestor divizori poate fi scrisă sub forma:

$$\sum_{\substack{q_i=0,\ldots,p_i \cdot B \\ i=1,\ldots,n}} \prod_{i=1}^{n} d_i^{q_i}.$$

Această sumă de produse poate fi scrisă sub forma unui produs de sume astfel:

$$\prod_{i=1}^{n} \sum_{j=0}^{p_i \cdot B} d_i^j.$$

Așadar, suma divizorilor unui număr de forma A^B este dată de un produs de sume. Aceste sume au forma $1 + q + q^2 + q^3 + ... + q^m$, deci sunt sume ale unei progresii geometrice de rație q. Este cunoscut faptul că valoarea unei astfel de sume, pentru $q > 1$ este $\frac{q^{m+1}-1}{q-1}$, iar pentru $q = 1$ este $m + 1$.

Datorită faptului că se cere determinarea restului împărțirii la 9901 a sumei divizorilor, toate operațiile efectuate vor fi *modulo* 9901.

Vor trebui folosite relațiile:

`(a + b) mod n = ((a mod n) + (b mod n)) mod n` și

`(a · b) mod n = ((a mod n) · (b mod n)) mod n`.

De asemenea, datorită faptului că numărul 9901 este prim, pentru orice număr nenul a va exista un număr a^{-1} (numit ***invers***) astfel încât ($a \cdot a^{-1} \bmod 9901 = 1$). Așadar, pentru a efectua o împărțire cu un anumit număr vom realiza, de fapt, o înmulțire cu inversul acestuia.

Pentru a rezolva problema vom începe cu descompunerea numărului A în factori primi și identificarea puterilor la care apar aceștia în descompunere. Vom înmulți aceste puteri cu B și apoi vom calcula sumele progresiilor corespunzătoare.

Pentru calculul sumei unei progresii avem nevoie de efectuarea unei ridicări la putere (evident, *modulo* 9901). O modalitate rapidă de calcul se bazează pe următoarea formulă recursivă:

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

O modalitate de determinare a inversului unui număr a (*modulo* 9901) este considerarea numerelor de forma $9901 \cdot k + 1$ și alegerea primului număr b de această formă al cărui rest la împărțirea cu a este 1. Inversul va fi dat de valoarea b / a. Datorită faptului că 9901 este prim va exista întotdeauna un astfel de număr și se poate demonstra că valoarea k corespunzătoare este mai mică decât 9901.

O altă observație necesară este faptul că scăderea valorii 1 (necesară pentru calcularea sumei progresiilor) este echivalentă cu adunarea valorii 9900 (*modulo* 9901).

Pe parcursul determinării sumei progresiilor, acestea vor fi înmulțite, *modulo* 9901. În final, rezultatul va fi scris în fișierul de ieșire.

Analiza complexității

Datele de intrare constau doar în două numere, deci operația de citire a acestora se realizează în timp ***constant***.

Pentru determinarea divizorilor primi și a puterilor la care apar aceștia în descompunerea numărului A vom lua în considerare numărul 2 și numerele impare cuprinse între 3 și cel mult $\sqrt{A}$, așadar ordinul de complexitate al acestei operații este $O(\sqrt{A})$.

Numărul sumelor calculate este dat de numărul divizorilor primi ai numărului A. Folosind aproximarea lui *Stirling* se poate deduce că există aproximativ $\ln A$ numere prime cuprinse între 2 și A. Așadar, vom calcula $O(\log A)$ sume. Calculul unei sume implică o ridicare la putere. Folosind algoritmul recursiv prezentat anterior, o ridicare la puterea n are ordinul de complexitate $O(\log n)$. Puterea la care poate apărea un factor prim în descompunerea numărului A este cel mult $\log_2 A$; în descompunerea lui A^B puterea poate fi cel mult $B \cdot \log_2 A$. Ca urmare, o ridicare la putere va avea ordinul de complexitate $O(\log (B \cdot \log A))$. Celelalte operații necesare pentru calculul sumelor se realizează în timp constant (considerăm că determinarea inversului are ordinul de complexitate $O(1)$ deoarece operația necesită cel mult 9901 pași; există și posibilitatea construirii și utilizării unui tablou de constante a, unde a_i să conțină inversul numărului i modulo 9901), deci ordinul de

complexitate al determinării sumei tuturor divizorilor este $O(\log A) \cdot O(\log (B \cdot \log A)) = O(\log A \cdot \log (B \cdot \log A))$.

În fișierul de ieșire trebuie scris un singur număr, deci ordinul de complexitate al operației de scriere a datelor de ieșire este $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(1) + O(\sqrt{A}) + O(\log A \cdot \log (B \cdot \log A)) + O(1) = O(\sqrt{A} + \log A \cdot \log(B \cdot \log A))$.

Dificultate	37
Algoritmi	★☆☆☆☆
Structuri de date	☆☆☆☆☆
Originalitate	★★★☆☆
Cunoștințe	★★★★★
Implementare	★★☆☆☆

www.ginfo.ro/revista/12_7/surse/sumdiv.cpp

P050217: Sistem

Problema poate fi enunțată în termeni ai teoriei grafurilor astfel: *Să se determine numărul grafurilor neorientate distincte în care fiecare nod are gradul 2.*

Este ușor de determinat numărul grafurilor conexe cu proprietatea cerută; practic, orice permutare $(p_1, p_2, ..., p_N)$ a mulțimii poate duce la obținerea unei soluții. Dacă unim printr-o muchie vârfurile p_i și p_{i+1} (i = 1, …, n - 1) și vârfurile p_1 și p_N obținem un graf (*hamiltonian*) în care toate vârfurile au gradul 2. Luând în considerare toate cele $N!$ permutări obținem $N!$ grafuri care respectă proprietatea cerută. Totuși, nu toate aceste grafuri sunt distincte; dacă alegem o anumită permutare atunci toate celelalte permutări circulare ale acesteia duc la obținerea aceluiași graf. De exemplu, permutările (1, 2, ..., N) și (N, 1, ..., N - 1) corespund aceluiași graf. Există N permutări circulare ale unei permutări date, deci numărul total al grafurilor distincte se reduce de N ori, devenind (N - 1)!. În plus, dacă parcurgem în ordine inversă o anumită permutare obținem același graf deoarece grafurile sunt neorientate. De exemplu, permutările (1, 2, ..., N) și (1, N, ..., 2) duc la obținerea aceluiași graf. Fiecare permutare poate fi parcursă în sens invers într-un singur mod, deci numărul total al grafurilor distincte se reduce de 2 ori. În concluzie, numărul grafurilor conexe cu proprietatea cerută este (N - 1)! / 2. Această valoare este valabilă pentru grafuri care conțin cel puțin trei noduri; nu există grafuri cu proprietatea cerută care să conțină unul sau două noduri.

Urmează să determinăm numărul total al grafurilor (conexe și neconexe) care au proprietatea cerută. Vom nota cu Nr_k numărul grafurilor care respectă proprietatea cerută și conțin k noduri. Vom avea $Nr_1 = Nr_2 = 0$ deoarece nu există nici un astfel de graf care să conțină unul sau două noduri. Prin convenție vom considera $Nr_0 = 1$ (se poate considera că graful vid respectă proprietatea cerută deoarece am putea spune că "*toate cele 0 noduri au gradul 2*", deși această afirmație este, practic, lipsită de sens; totuși această convenție se va dovedi utilă pentru simplificarea calculelor).

Vom considera, pe rând, că nodul 1 face parte din componente conexe cu 3, 4, ..., N noduri. Vom determina numărul de grafuri în care acest nod face parte din componente conexe cu k elemente (k este cuprins între 3 și N) și apoi vom aduna aceste valori.

Dacă nodul 1 face parte dintr-o componentă conexă cu k elemente, atunci există C_{N-1}^{k-1} posibilități de a alege celelalte k - 1 noduri care fac parte din aceeași componentă conexă. Folosind rezultatul demonstrat anterior, deducem că numărul componentelor conexe cu k elemente care respectă proprietatea cerută este (k - 1)! / 2. Indiferent care sunt cele k noduri care formează componenta conexă, celelalte N - k noduri pot forma Nr_{N-k} grafuri cu proprietatea cerută. În concluzie, dacă nodul 1 face parte dintr-o componentă conexă cu k elemente, avem $Nr_{N-k} \cdot C_{N-1}^{k-1} \cdot \frac{(k-1)!}{2}$ grafuri distincte cu proprietatea cerută.

Se observă utilitatea convenției $Nr_0 = 1$ deoarece, pentru componenta conexă formată din toate cele N noduri ale grafului, obținem:

$Nr_{N-N} \cdot C_{N-1}^{N-1} \cdot \frac{(N-1)!}{2} = 1 \cdot 1 \cdot \frac{(N-1)!}{2} = \frac{(N-1)!}{2}$, adică exact numărul grafurilor conexe cu proprietatea cerută. Cazul $k = N$ ar fi putut fi tratat ca fiind unul particular, dar folosirea convenției duce la simplificarea formulelor de calcul.

Numărul total al grafurilor este obținut prin însumarea acestor valori pentru k = 3, …, N, deci vom avea $Nr_N = \sum_{k=3}^{N} Nr_{N-k} \cdot C_{N-1}^{k-1} \cdot \frac{(k-1)!}{2}$ grafuri cu proprietatea cerută.

Chiar și pentru valori mici ale lui N, acest număr nu va putea fi reprezentat folosind tipuri standard de date, deci vor trebui simulate operații aritmetice cu numere mari. Folosind un artificiu de calcul, nu va trebui să implementăm decât adunarea a două numere mari și înmulțirea dintre un număr mare și un număr mai mic decât N.

Vom avea nevoie de adunarea numerelor mari pentru calcularea sumei care duce la obținerea valorilor Nr_k. Valorile însumate sunt obținute prin înmulțirea unui număr mare (Nr_{N-k} cu mai multe numere mai mici decât 100). Aceste numere sunt obținute folosind următorul artificiu:

$$C_{N-1}^{k-1} \cdot \frac{(k-1)!}{2} = \frac{(N-1)!}{(k-1)!(N-k)!} \cdot \frac{(k-1)!}{2} = \frac{(N-1)!}{2 \cdot (N-k)!} = \frac{(N-k+1) \cdot (N-k+2) \cdot \ldots \cdot (N-1)}{2}$$

Așadar, șirul numerelor cu care se vor realiza înmulțiri este $N - k + 1$, $N - k + 2$, N - 1. Deoarece k este cel puțin 3, acest șir are cel puțin două elemente; elementele șirului sunt numere consecutive, deci unul dintre primele două este par. Acest număr par va fi împărțit cu 2 și se va obține un șir de elemente care vor fi înmulțite, pe rând, cu un număr mare.

Pentru determinarea valorii Nr_N este necesară calcularea tuturor valorilor Nr_k ($k < N$): acestea vor fi păstrate

într-un șir de numere mari, iar în final valoarea Nr_N va fi scrisă în fișierul de ieșire.

Analiza complexității

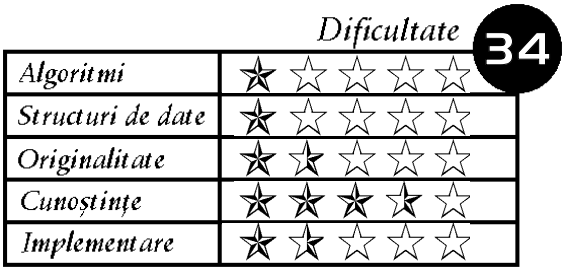
Fiecare operație aritmetică se efectuează în timp constant deoarece numărul cifrelor unui număr (dacă se folosește baza 10000) este mai mic decât 100.

Citirea datelor de intrare este realizată în timp ***constant*** pentru că se citește un singur număr.

Pentru calcularea unei valori Nr_K avem nevoie de însumarea a k - 3 termeni. Valorile Nr_i necesare sunt cunoscute de la pașii anteriori; pentru determinarea termenilor care trebuie însumați avem nevoie de i - 1 înmulțiri. Așadar, în total vom avea $(k - 4) + (k - 5) + ... + 1 = (k - 4)\cdot(k - 3)/2$ înmulțiri. În concluzie, ordinul de complexitate al operației de determinare a numărului Nr_k este $O(k^2)$. Vom determina N astfel de numere, ordinul de complexitate al întregii operații fiind $O(N^3)$ deoarece avem $k = O(N)$.

Datele de ieșire constau într-un singur număr mare, deci operația de scriere a acestora se realizează în timp *constant*.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(1) + O(N^3) + O(1) = O(N^3)$.



www.ginfo.ro/revista/12_7/surse/sistem.cpp

P050218: Comitat

O posibilitate de rezolvare a acestei probleme este folosirea metodei programării dinamice. Vom încerca să determinăm un tablou tridimensional A, unde A_{ijk} va indica lungimea minimă a unui traseu care pornește din *Mordor* (turnul 0), ultimele două turnuri de pe acest traseu sunt j și k și traseul conține în total i turnuri.

Se observă că lungimile traseelor care conțin i turnuri pot fi determinate cu ajutorul lungimilor traseelor care conține i - 1 turnuri. Astfel, vom lua în considerare toate traseele pe care se află i - 1 turnuri, și ultimul turn al traseului este j și îl vom alege pe cel mai scurt dintre acestea. La lungimile acestor trasee vom adăuga distanța dintre turnurile j și k. Datorită faptului că un călăreț se poate deplasa doar paralel cu axele de coordonate, distanța dintre două turnuri i și j va fi dată de $|x_i - x_j| + |y_i - y_j|$. Fie p, penultimul turn de pe traseul care se termină în turnul j; evident, unghiul determinat de turnurile p, i și j nu trebuie să anuleze convexitatea traseului. Pentru a putea reconstitui traseul vom păstra, pentru fiecare triplet (i, j, k) numărul de ordine p al turnului care se află pe traseu înaintea turnurilor j și k.

În final, vom obține toate elementele tabloului tridimensional A. Evident, traseul trebuie să fie un poligon închis, deci ultimul turn trebuie să fie 0 (*Mordor*). Așadar, vom studia valorile A_{ij0}; vom determina cea mai mare valoare i pentru care există o valoare j astfel încât valoarea A_{ij0} este cel mult egală cu lungimea maximă a traseului.

După identificarea valorilor i și j vom putea determina turnurile de pe traseu cu ajutorul predecesorilor păstrați pentru fiecare triplet (i, j, k).

Pentru a mări viteza de execuție a programului vom determina, pentru fiecare triplet (i, j, k) dacă cele trei turnuri corespunzătoare se pot afla pe poziții consecutive în traseu (se respectă condiția de convexitate) și vom folosi aceste informații pe parcurs în loc să verificăm convexitatea de fiecare dată.

Inițial putem determina toate traseele pe care se află două turnuri (*Mordor* și oricare dintre celelalte). În continuare vom aplica algoritmul pentru a determina lungimile traseului pe care se află 3 turnuri, 4 turnuri etc.

Analiza complexității

Datele de intrare constau în numărul turnurilor, coordonatele acestora și lungimea maximă a traseului, deci ordinul de complexitate al operației de citire este $O(n)$.

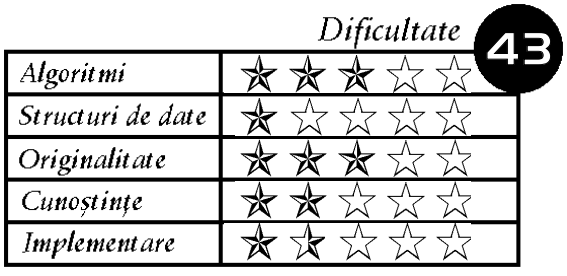
Va trebui să verificăm pentru fiecare triplet (i, j, k) dacă se respectă condiția de convexitate, operație care are ordinul de complexitate $O(n^3)$, deoarece există $(n + 1)^3$ astfel de triplete (i, j și k variază între 0 și n).

Pentru a determina o valoarea A_{ijk} vor trebui luați în considerare toți predecesorii posibili, deci ordinul de complexitate al acestei operații este $O(n)$. Deoarece tabloul este tridimensional sunt realizate $O(n^3)$ astfel de operații, deci ordinul de complexitate al operației de determinare a tabloului A (și a tabloului care conține predecesorii) este $O(n)\cdot O(n^3) = O(n^4)$.

Identificarea valorii A_{ij0} care indică lungimea traseului care conține cele mai multe turnuri se realizează în timp ***pătratic*** deoarece, în cel mai defavorabil caz, sunt luate în considerare toate perechile (i, j).

Refacerea traseului (și scrierea turnurilor de pe traseu în fișierul de ieșire) pe baza tabloului predecesorilor se realizează în timp ***liniar***, deoarece vor exista cel mult n turnuri pe acest traseu.

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate $O(n) + O(n^3) + O(n^4) + O(n^2) + O(n) = O(n^4)$.



www.ginfo.ro/revista/12_7/surse/comitat.cpp