BURSELE AGORA 2004/ 2005. RUNDELE #13-#30

Vă prezentăm în continuare soluțiile problemelor propuse spre rezolvare la 18 dintre rundele ediției a șasea a concursului de programare organizat de revista noastră.

P050307: Găuri

Vom determina toate regiunile formate din zerouri și vom eticheta elementele acestor regiuni folosind numere naturale, începând cu 1.

Pentru fiecare regiune determinată vom păstra dimensiunile acestora (numărul de elemente) într-un vector ai cărui indici sunt numerele folosite pentru etichetarea dimensiunilor regiunilor. Pentru a determina aceste regiuni vom folosi un simplu algoritm de umplere.

O gaură poate fi privită ca fiind o regiune formată din zerouri care nu conține nici un element aflat pe marginea matricei. Vom parcurge cele patru margini ale matricei și vom seta la 0 dimensiunile tuturor regiunilor care conțin elemente de pe aceste margini. Astfel, practic am eliminat toate regiunile care nu sunt găuri.

În final vom parcurge vectorul dimensiunilor și vom determina elementul care are valoarea maximă. Valoarea acestui element va fi scrisă în fișierul de ieșire.

Analiza complexității

Datele de intrare constau în dimensiunile matricei și valorile tuturor elementelor acesteia. Așadar ordinul de complexitate al operației de citire este $O(M \cdot N)$.

Operația de determinare a regiunilor formate din zerouri folosind un algoritm de umplere are ordinul de complexitate $O(M \cdot N)$.

Pentru a elimina regiunile care nu sunt găuri vom parcurge cele patru margini ale matricei, așadar această operație are ordinul de complexitate $O(M) + O(N) + O(M) + O(N) = O(M + N)$.

În total pot exista cel mult $[M \cdot N / 2] + 1$ regiuni (dacă matricea are aspectul unei table de șah). Așadar, operația de determinare a dimensiunii celei mai mari regiuni are ordinul de complexitate $O(M \cdot N)$.

Datele de ieșire constau într-un singur număr, operația de scriere a acestuia având ordinul de complexitate $O(1)$.

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

P050308: Lagăr

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 *Pădurea Aurie* (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 conțin 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 paznicul 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 (*Pădurea Aurie*). 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 (*Pădurea Aurie* și oricare dintre celelalte). În continuare vom aplica algoritmul pentru a determina lungimile traseului pe care se află trei turnuri, patru 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 valoare 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 $O(n) + O(n^3) + O(n^4) + O(n^2) + O(n) = O(n^4)$.

P050310: Dispozitive

În principiu, problema ar putea fi enunțată astfel: *dându-se un număr întreg să se determine cel mai mic multiplu al său care este format doar din anumite cifre date*.

Soluția acestei probleme este destul de simplă și se bazează pe câteva observații matematice.

Pentru început vom demonstra că, dacă există un multiplu al numărului Nr care este format doar din cifrele $x_1, x_2, ..., x_m$, atunci cel mai mic astfel de multiplu are cel mult Nr cifre.

Vom presupune, prin absurd, că cel mai mic multiplu M are $Nr + 1$ cifre. Eliminând ultima cifră, vom obține un număr care, împărțit la Nr, duce la obținerea unui rest R_1. Repetând procesul de eliminare a cifrelor, se obțin resturile $R_2, ..., R_{Nr}$.

Dacă cele Nr resturi sunt distincte, atunci unul dintre ele este 0, deci există un multiplu care conține mai puțin de $Nr + 1$ cifre (cel mult Nr cifre).

Dacă resturile nu sunt distincte, atunci cel puțin două dintre ele (fie acestea R_i și R_j, $i < j$) sunt egale. Multiplul M are forma $m_1...m_i...m_j...m_{Nr+1}$; datorită faptului că resturile R_i și R_j sunt egale, înseamnă că prin eliminarea cifrelor $m_{i+1}, ..., m_j$ se obține un număr care este, la rândul său, multiplu al numărului Nr. Acest număr are forma $m_1...m_i m_{j+1}...m_{Nr+1}$, deci are cel mult Nr cifre.

Putem trage concluzia că ipoteza inițială este falsă; așadar, dacă există un multiplu al numărului Nr care să fie format din anumite cifre date, atunci cel mai mic astfel de multiplu conține cel mult Nr cifre.

Ipoteza potrivit căreia multiplul ar fi format din $Nr + 1$ cifre nu reduce generalitatea deoarece, dacă presupunem că numărul ar fi format din $Nr + k$ cifre, folosind același procedeu putem obține, pe rând, multiplii formați din cel mult $Nr + k - 1$ cifre, $Nr + k - 2$ cifre și așa mai departe până se ajunge la un multiplu format din cel mult $Nr + 1$ cifre.

În continuare vom descrie modul în care poate fi determinat un multiplu format din cel mult Nr cifre.

Vom crea o coadă care va conține numere prin împărțirea cărora la Nr se obțin resturi distincte. Corespunzător fiecărui rest care poate fi obținut, în coadă se va afla cel mai mic număr care duce la obținerea restului respectiv.

Inițial vom insera în coadă numere formate dintr-o singură cifră (cifrele pe care le avem la dispoziție), ordonate crescător. La fiecare pas, vom alege primul element din coadă și vom încerca să adăugăm la sfârșitul său, pe rând, una dintre cifrele disponibile. Pentru a obține cele mai mici numere, cifrele vor fi considerate în ordine crescătoare.

În cazul în care un număr astfel determinat duce, prin împărțirea la Nr, la obținerea unui rest care nu a mai fost obținut anterior, atunci numărul determinat este adăugat în coadă.

După considerarea tuturor cifrelor, elementul curent este eliminat din coadă. Datorită acestor eliminări trebuie păstrat un vector de valori logice care să indice dacă un rest a fost sau nu obținut anterior.

În momentul în care se obține restul 0, cel mai mic multiplu este determinat și execuția algoritmului se oprește.

Noile resturi pot fi determinate foarte repede folosind o altă observație matematică: dacă restul împărțirii unui număr a la Nr este r, atunci restul împărțirii la Nr a numărului obținut prin adăugarea cifrei x la sfârșitul numărului a este rest$[(r \cdot 10 + x) / Nr]$.

În cazul în care nu am reușit să obținem restul 0 considerând numere formate din cel mult Nr cifre, atunci suntem siguri că nu există nici un multiplu format doar din cifrele date deoarece, dacă astfel de multiplii ar fi existat, atunci cel puțin unul dintre ei ar fi avut cel mult Nr cifre.

Analiza complexității

Datele de intrare constau din numărul Nr de pe mecanism (care are în-

totdeauna patru cifre, chiar dacă este posibil ca primele să fie 0) și cele N cifre pe care le avem la dispoziție. Ca urmare, ordinul de complexitate al operației de citire a datelor este $O(N)$.

În continuare, pentru fiecare rest vom încerca să adăugăm cele N cifre. Așadar, avem ordinul de complexitate $O(N)$ pentru fiecare rest.

Datorită faptului că numărul resturilor considerate este de cel mult Nr, ordinul de complexitate al operației de determinare a multiplului este $O(Nr \cdot N)$.

Datele de ieșire constau în cele cel mult Nr cifre ale multiplului sau doar în cifra 0. Ca urmare, ordinul de complexitate al operației de scriere a datelor de ieșire este $O(Nr)$.

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

P050311: Sfere

Să presupunem că am ales k sfere; vom încerca să determinăm cu cât crește numărul de zone în care este împărțit spațiul după alegerea celei de-a $(k + 1)$-a sfere.

Numărul maxim de zone se obține dacă oricare două sfere se intersectează, adică nu există nici o pereche de sfere tangente, nu există nici o sferă care să se afle complet în interiorul unei alte sfere, nu există trei sfere care să aibă un cerc comun și oricare patru sfere nu au nici un punct comun.

Sfera considerată se va intersecta cu celelalte k sfere după k cercuri. Vom încerca să determinăm acum numărul maxim de zone în care aceste k cercuri pot împărți suprafața sferei.

Deoarece dimensiunile sferelor nu au nici o importanță, putem considera că noua sferă are o rază mult mai mare decât cea a celorlalte k sfere. La limită, vom considera că raza acestei sfere este infinită. Datorită faptului că, în acest caz, sfera degenerează într-un plan (un plan poate fi considerat ca fiind o sferă de rază infinită) problema în spațiu se reduce la o problemă în plan.

Enunțul acestei noi probleme este următorul: *determinați numărul maxim de zone în care poate fi împărțit planul prin intermediul a* n *cercuri.*

Cercurile împart planul într-un număr maxim de zone dacă oricare două cercuri se intersectează, adică nu există nici o pereche de cercuri tangente, nu există nici un cerc care să se afle complet în interiorul unui alt cerc și nu există nici un grup de trei cercuri care să aibă un punct comun.

Considerăm că am ales k cercuri și încercăm să determinăm numărul de zone suplimentare care apar dacă alegem cel de-al $(k + 1)$-lea cerc. Deoarece cercul se intersectează cu fiecare dintre celelalte k cercuri în câte două puncte, rezultă că circumferința sa va fi împărțită în $2 \cdot k$ regiuni. Ca urmare, numărul de zone suplimentare care apar după considerarea unui nou cerc este egal cu dublul numărului cercurilor considerate anterior.

Așadar, relația de recurență, cu ajutorul căreia să se poată determina numărul de zone în care poate fi împărțit un plan, este $c_{k+1} = c_k + 2 \cdot k$. Deoarece un cerc împarte planul în două zone (interiorul cercului și exteriorul său), avem $c_1 = 2$.

Astfel obținem:

$$\begin{aligned} c_n &= 2+2+4+6+8+\ldots+2\cdot(n-1) \\ &= 2+2\cdot(1+2+3+\ldots+n-1) \\ &= 2+2\cdot\frac{n\cdot(n-1)}{2} \\ &= n^2-n+2. \end{aligned}$$

Așadar, numărul maxim de zone în care k cercuri pot împărți suprafața unei sfere este de $k^2 - k + 2$. Ca urmare, prin considerarea celei de-a $(k + 1)$-a sfere numărul de zone este mărit cu $k^2 - k + 2$.

Așadar, relația de recurență, cu ajutorul căreia se poate determina numărul de zone în care poate fi împărțit spațiul, este $s_{k+1} = s_k + k^2 - k + 2$.

Deoarece o sferă împarte spațiul în două zone (interiorul și exteriorul ei), $s_1 = 2$. Efectuând calculele obținem:

$$\begin{aligned} s_n &= 2+(1^2-1+2)+(2^2-2+2)+(3^2-3+3)+\ldots+[(n-1)^2-(n-1)+2] \\ &= 2+[1^2+2^2+3^2+\ldots+(n-1)^2]-[1+2+3+\ldots+(n-1)]+2\cdot(n-1) \\ &= 2+\frac{n\cdot(n-1)\cdot(2\cdot n-1)}{6}-\frac{n\cdot(n-1)}{2}+2\cdot(n-1) \\ &= \frac{12+2\cdot n^3-3\cdot n^2+n-3\cdot n^2+3\cdot n+12\cdot n-12}{6} \\ &= \frac{2\cdot n^3-6\cdot n^2+16\cdot n}{6} \\ &= \frac{n\cdot(n^2-3\cdot n+8)}{3}. \end{aligned}$$

Așadar, numărul maxim de zone în care poate fi împărțit spațiul cu ajutorul a n sfere este:

$$\frac{n\cdot(n^2-3\cdot n+8)}{3}.$$

Analiza complexității

Citirea numărului n se realizează în timp constant, deci are ordinul de complexitate $O(1)$, iar operația de determinare a numărului maxim de zone în care poate fi împărțit spațiul are ordinul de complexitate tot $O(1)$, deoarece este utilizată o simplă formulă matematică. Operația de scriere a rezultatului are, de asemenea, ordinul de complexitate $O(1)$.

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

P050312: Templu

Pentru simplitate, vom prezenta un enunț echivalent al problemei: *se consideră un șir de numere binare; folosind acest șir se construiește o matrice în care prima linie este dată de șirul considerat, iar fiecare dintre următoarele linii sunt obținute prin permutarea la stânga cu o poziție a elementelor din linia precedentă; în continuare, liniile matricei obținute sunt sortate lexicografic (valoarea* 0 *precede valoarea* 1*); dându-se elementele de pe ultima coloană a matricei obținute după sortare, să se determine elementele de pe prima linie a acestei matrice.*

Algoritmul constă în trei pași:

- se numără elementele cu valoarea 0 și elementele cu valoarea 1 de pe ultima coloană și, pe baza rezultatului numărării, se construiește prima coloană a matricei; în continuare vom presupune că avem p elemente cu valoarea 0 și q elemente cu valoarea 1;
- se creează un șir a astfel:
  - pentru $i \le p$, elementul a_i va avea ca valoare linia pe care se află cel de-al i-lea element cu valoarea 0 de pe ultima coloană;
  - pentru $j \le q$, elementul a_{p+j} va avea ca valoare linia pe care se află cel de-al j-lea element cu valoarea 1 de pe ultima coloană;

- începând cu prima linie, se parcurg liniile folosind vectorul construit a (de la linia k se trece la linia a_k) și sunt reconstituite elementele de pe prima linie pe baza valorilor de pe ultima coloană.

În cele ce urmează vom demonstra corectitudinea algoritmului prezentat. Numim succesor al unui șir șirul obținut printr-o permutare cu o poziție la stânga a elementelor șirului.

Pentru a demonstra corectitudinea algoritmului, va trebui să arătăm că șirul a reprezintă exact această relație de succesiune între șiruri. După demonstrarea acestui fapt, este evident că al treilea pas al algoritmului reconstituie corect elementele de pe prima linie.

Considerăm liniile (din matricea obținută după sortare) care încep cu un element care are valoarea 0. Deoarece, aceste șiruri sunt ordonate în matricea sortată, după efectuarea unei permutări la stânga cu o poziție, șirurile obținute (care se termină cu un element care are valoarea 0) sunt și ele sortate.

Aceeași proprietate este valabilă și pentru liniile care încep cu un element care are valoarea 1.

Deoarece aceste proprietăți sunt echivalente cu relația de succesiune, corectitudinea algoritmului este demonstrată.

Analiza complexității

Vom demonstra în continuare că timpul de execuție al algoritmului este unul liniar. Pentru aceasta vom arăta că fiecare dintre cei trei pași se efectuează în timp liniar.

Numărarea elementelor de pe ultima coloană se realizează printr-o singură parcurgere, deci liniaritatea este evidentă.

Inițializarea tabloului a necesită doar o singură parcurgere a acestuia și a ultimei coloane, deci timpul este liniar și în acest caz.

Cel de-al treilea pas se execută și el în timp liniar deoarece la fiecare tranziție se determină câte un element de pe prima linie a matricei sortate.

Evident, operațiile de citire și de scriere se efectuează, ambele, în timp liniar.

P050313: Raze magice

Pentru simplitate, vom prezenta mai întâi un enunț echivalent al acestei probleme: *o placă dreptunghiulară de dimensiuni $L_1 \times L_2$ este împărțită într-o rețea ortogonală de pătrate de latură* 1, *în care sunt marcate N puncte albe și N puncte negre, toate având coordonate întregi; oricare trei dintre cele $2 \cdot N$ puncte nu sunt coliniare. În scopul obținerii unui circuit imprimat prin corodarea plăcii, fiecare punct alb urmează a fi unit printr-un segment de dreaptă cu un punct negru oarecare, dar astfel încât oricare două dintre cele N segmente să nu se intersecteze.*

Datorită faptul că punctele au coordonate numere reale cu trei zecimale exacte, fiecare valoare se va înmulți cu 1000 și coordonatele vor deveni numere întregi cuprinse între 1 și 1000000. Așadar, în această situație avem $L_1 = L_2 = 1000000$.

Există mai multe soluții pentru rezolvarea problemei. Prima se bazează pe metoda *backtracking*: se încearcă o conectare a punctului alb (elfului) curent cu un punct negru (orc) neconectat încă, astfel încât segmentul (raza) format(ă) să nu se intersecteze cu nici unul dintre segmentele (razele) deja formate. Este evident că o astfel de soluție nu se va încadra în limita de timp admisă, așadar nu vom mai insista asupra ei.

Cea de-a doua soluție folosește un algoritm care evită metoda *backtracking*. Pentru început vom conecta punctele într-un anumit mod. De exemplu, am putea conecta punctele (elfii și orcii) în ordinea dată la intrare: primul punct alb (elf) cu primul punct negru (orc) etc.

Vom identifica prin V_i punctele albe (elfii) și cu F_i punctele negre (orcii). Vom verifica apoi dacă există două segmente V_1F_1 și V_2F_2 care se intersectează. În acest caz, conexiunile se vor modifica și noile segmente vor fi V_1F_2 și V_2F_1. Este evident că aceste segmente nu se vor mai intersecta.

Vom demonstra în continuare că, prin schimbarea conexiunilor, suma lungimilor celor două segmente se reduce. Vom nota cu O intersecția segmentelor V_1F_1 și V_2F_2. Știm că într-un triunghi lungimea unei laturi este întotdeauna mai mică decât suma lungimilor celorlalte două. Așadar, în triunghiul V_1OF_2 avem $V_1F_2 < V_1O + OF_2$, iar în triunghiul V_2OF_1 avem $V_2F_1 < V_2O + OF_1$. Adunând cele două inegalități obținem $V_1F_2 + V_2F_1 < V_1O + OF_1 + V_2O + OF_2$, adică $V_1F_2 + V_2F_1 < V_1F_1 + V_2F_2$.

Așadar, după fiecare modificare a conexiunilor suma totală a lungimilor segmentelor se reduce. Deoarece această sumă nu va fi niciodată negativă, înseamnă că după un număr finit de pași nu vom mai avea intersecții.

Se conturează astfel următorul algoritm de rezolvare a problemei: atâta timp cât există două segmente care se intersectează, se modifică conexiunile pentru punctele corespunzătoare.

Ordinul de complexitate al algoritmului prezentat anterior este liniar în suma lungimilor segmentelor, dar nu este polinomial în numărul de puncte, deoarece după modificarea conexiunilor pentru două segmente, segmentele nou formate se pot intersecta cu alte segmente, deci numărul de intersecții nu se reduce neapărat, ci numai suma totală a lungimilor.

Există, totuși, o modalitate de rezolvare a problemei care să ducă la obținerea unui timp de execuție polinomial în numărul de puncte. Vom încerca să demonstrăm că există o linie care trece printr-un elf și printr-un orc, astfel încât numărul elfilor aflați de o parte a liniei este egal cu numărul orcilor aflați de aceeași parte.

Este relativ simplu să determinăm soluția dacă reușim să găsim o astfel de dreaptă. Pentru rezolvare vom folosi metoda *divide et impera*. Dreapta va împărți elfii și orcii în două mulțimi fiecare dintre ele conținând un număr egal de elfi și orci. Așadar, același algoritm poate fi aplicat pentru cele două mulțimi fără a exista peri-

colul ca vreun segment trasat în una dintre mulțimi să se intersecteze cu un segment trasat în cealaltă mulțime sau cu segmentul care unește cele două puncte care determină dreapta care separă mulțimile.

Problema mai dificilă este determinarea dreptei care are proprietatea cerută. Vom considera punctul P având coordonata y cea mai mică, indiferent dacă reprezintă poziția unui elf sau a unui orc. Vom sorta toate celelalte puncte în funcție de unghiul format de dreapta care unește P de acest punct și dreapta orizontală care trece prin P (unghiul polar).

Vom avea doi indicatori V și F, primul va indica numărul elfilor luați în considerare, iar al doilea numărul orcilor. Dacă punctul P reprezintă un orc, atunci inițializăm V cu 0 și F cu 1, iar dacă reprezintă un elf inițializăm V cu 1 și F cu 0.

Vom considera, pe rând, punctele sortate; dacă punctul curent reprezintă un elf incrementăm indicatorul V, iar dacă reprezintă un orc incrementăm indicatorul F. În momentul în care vom avea $V = F$ vom ști că sub dreapta care unește punctul P cu punctul curent se află un număr egal de elfi și orci.

Vom demonstra în continuare că, la un moment dat, vom ajunge în situația în care $V = F$. Presupunem, fără a restrânge generalitatea, că în punctul P se află un elf.

Dacă în primul punct se află un orc atunci, ajungem în situația $V = F = 1$, deci am determinat deja o dreaptă cu proprietatea cerută. Dacă în ultimul punct se află un orc atunci, fie am găsit o dreaptă anterior, fie acest ultim punct și punctul P determină dreapta pe care o căutăm. Așadar, cazul în care primul sau ultimul punct reprezintă un orc este rezolvat.

Să presupunem acum că atât primul, cât și ultimul punct reprezintă un elf. În acest caz inițial vom avea $V = 1$ și $F = 0$ ($V - F = 1$), iar înaintea considerării ultimului punct $V = n - 1$ și $F = n$ ($V - F = -1$).

Deoarece valoarea $V - F$ nu se poate modifica decât cu 1 la fiecare pas, pentru a ajunge de la 1 la -1 ea trebuie neapărat să treacă și prin 0. Așadar, la un moment dat ne vom afla în situația pentru care $V = F$, deci vom avea dreapta cerută.

Vom descrie acum o metodă pe baza căreia punctele se pot sorta în funcție de unghiul polar pe care îl formează cu punctul P. Să presupunem că, în sistemul de coordonate cu originea în P, avem două puncte A și B de coordonate (x_1, y_1) și $x_2, y_2)$.

Pentru a sorta punctele în funcție de unghi avem trei cazuri.

Dacă punctele A și B se află în primul cadran al sistemului de coordonate cu originea în P, atunci punctul A se află înaintea punctului B în șirul sortat dacă și numai dacă $y_1 / x_1 < y_2 / x_2$, adică $y_1 \cdot x_2 < y_2 \cdot x_1$, deoarece coordonatele orizontale sunt pozitive. (Am înlocuit împărțirile cu înmulțiri pentru a mări viteza de execuție și a evita împărțirea cu zero.)

Dacă punctele A și B se află în cadrane diferite, atunci punctul din primul cadran îl va precede pe cel din al doilea cadran.

Dacă punctele A și B se află în al doilea cadran al sistemului de coordonate cu originea în P, atunci punctul A se află înaintea punctului B în șirul sortat dacă și numai dacă $y_1 / x_1 < y_2 / x_2$, adică $y_1 \cdot x_2 > y_2 \cdot x_1$, deoarece coordonatele orizontale sunt negative.

Datorită faptului că am ales punctul P având coordonata y minimă, punctele A și B nu se pot afla în al treilea sau al patrulea cadran.

Analiza complexității

Datele de intrare constau în coordonatele a $2 \cdot n$ puncte; operația de citire a acestora are ordinul de complexitate $O(n)$.

Pentru determinarea unei drepte va trebui să realizăm o sortare a punctelor și o parcurgere a punctelor sortate. Ordinul de complexitate al operației de sortare este $O(n \cdot \log n)$, iar cel al operației de parcurgere este $O(n)$. Ca urmare ordinul de complexitate al operației de determinare a unei drepte este $O(n \cdot \log n)$. În total vom determina n astfel de drepte, operația de determinare a tuturor având, ca urmare, ordinul de complexitate $O(n^2 \cdot \log n)$.

Datele de ieșire constau în afișarea a n perechi de numere, operație al cărei ordin de complexitate este $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(n) + O(n^2 \cdot \log n) + O(n) = O(n^2 \cdot \log n)$.

P050314: Joacă

Această problemă poate fi rezolvată folosind o structură de date numită arbore indexat binar. Deși această structură a mai fost prezentată în cadrul revistei noastre, vom descrie acum din nou detaliat modul în care ea poate fi folosită.

Subsecvențe

Prin subsecvență înțelegem un subșir ale cărui elemente se află pe poziții consecutive în șirul inițial. De exemplu, (2, 3, 4) este o subsecvență a șirului (1, 2, 3, 4, 5) formată din al doilea, al treilea și al patrulea element al șirului, în timp ce (1, 2, 4) nu este o subsecvență deoarece nu este formată din elemente aflate pe poziții consecutive.

Vom identifica o subsecvență prin extremitățile sale (poziția cea mai din stânga și poziția cea mai din dreapta din șir). O subsecvență care începe în poziția a și se termină în poziția b va fi notată prin <a, b>. Pentru șirul (1, 2, 3, 4, 5) subsecvența (2, 3, 4) va fi notată prin <2, 4> dacă numerotarea elementelor începe cu 1 sau prin <1, 3> dacă numerotarea începe de la 0.

Deseori, avem nevoie de informații referitoare la subsecvențele unui șir cum ar fi suma elementelor sau produsul. La prima vedere, rezolvarea acestei probleme pare destul de simplă (de exemplu, pentru sumă se parcurg elementele subsecvenței și se adună). Totuși, acest algoritm este ineficient datorită faptului că necesită parcurgerea întregii subsecvențe. Vom arăta că există algoritmi pentru care o astfel de parcurgere nu este necesară.

Dacă am efectua o singură dată sau de un număr limitat de ori o ast-

fel de parcurgere, algoritmul ar putea părea performant, viteza de execuție fiind relativ mare. Problema se complică dacă elementele șirului se modifică în timp real.

Modificări în timp real

Să presupunem că există două tipuri de operații care pot fi efectuate asupra unui șir. Primul tip constă în modificarea valorii unui element, în timp ce al doilea reprezintă interogări (cereri de informații) referitoare la anumite subsecvențe ale șirului. De obicei, cele două tipuri de operații sunt "amestecate" în sensul că nu vor fi doar modificări urmate din interogări, ci putem avea o modificare, urmată de două interogări, urmate de cinci modificări, urmate de alte două interogări etc.

Așadar, putem spune că elementele șirului se modifică în timp real și interogările se referă la starea curentă a șirului (cea din momentul cererii de informații).

Un enunț echivalent

În cele ce urmează vom prezenta un enunț al problemei, particularizat pentru cazul în care interogările se referă la suma elementelor subsecvențelor.

Se consideră un șir de numere întregi care are toate elementele nule. O modificare a unui element constă în adunarea unei valori la valoarea curentă (pot fi realizate și scăderi care au forma adunării unor valori negative). Pe parcursul modificărilor pot apărea interogări referitoare la suma elementelor unei subsecvențe a șirului.

Problema poate fi enunțată în multe alte forme. Una dintre ele ar putea fi următoarea:

Un furnizor lucrează cu N *distribuitori; în fiecare moment distribuitorii pot efectua plăți către furnizor sau pot cumpăra produse de la acesta. De asemenea, în fiecare moment furnizorul poate cere informații referitoare la suma totală a datoriilor pe care le au magazinele cu numere de ordine cuprinse între două valori date.*

Un exemplu

Vom exemplifica acum evoluția în timp real a unui șir format din cinci elemente, prezentând valorile șirului după fiecare modificare și rezultatul fiecărei interogări.

Operația *Inițializare* constă în setarea la 0 a valorilor tuturor elementelor. Operația *Adună*(*i*, *x*) constă în adunarea valorii *x* la valoarea curentă a celui de-al *i*-lea element. Operația *Sumă*(*a*, *b*) furnizează suma elementelor subsecvenței <*a*, *b*>.

Operație	*Șir/Rezultat*
Inițializare	0 0 0 0 0
Adună(1, 3)	3 0 0 0 0
Adună(4, 5)	3 0 0 5 0
Sumă(3, 3)	0
Sumă(4, 4)	5
Adună(4, 2)	3 0 0 7 0
Adună(2, 3)	3 3 0 7 0
Sumă(2, 5)	10
Sumă(2, 4)	10
Adună(5, 2)	3 3 0 7 2
Sumă(2, 4)	10
Sumă(2, 5)	12
Adună(3, 1)	3 3 1 7 2
Adună(4, -2)	3 3 1 5 2
Sumă(2, 5)	11
Adună(1, -3)	0 3 1 5 2
Adună(5, 5)	0 3 1 5 7
Sumă(1, 2)	3
Sumă(3, 4)	6
Adună(2, -1)	0 2 1 5 7
Adună(3, 3)	0 2 4 5 7
Sumă(5, 5)	7
Sumă(1, 2)	2
Sumă(1, 5)	18

Cazul unidimensional

Din modul în care a fost enunțată problema, rezultă că operațiile sunt efectuate asupra unui tablou unidimensional; așadar, acesta este cazul unidimensional al problemei. Vom prezenta în continuare trei algoritmi care pot fi folosiți pentru rezolvarea problemei și apoi le vom studia performanțele.

Algoritmul ineficient

Cea mai intuitivă metodă de rezolvare constă în păstrarea unui vector cu valorile șirului, modificarea lor atunci când este necesar și calcularea sumelor în momentul în care apar interogări.

Pentru a prezenta algoritmul vom considera că o modificare este codificată prin valoarea 1, iar o interogare prin valoarea 2. Ar fi necesară o a treia operație (codificată prin 3) care ar indica faptul că nu mai există modificări sau interogări, deci execuția poate fi încheiată.

Versiunea în pseudocod este prezentată în continuare:

```
algoritm ModificareIneficientă
  inițializări
  scrie 'Introduceți numărul
              de elemente: '
  citește N
  pentru i ← 1, N execută
    a_i ← 0
  sfârșit pentru
  scrie 'Introduceți codul
                operației: '
  citește cod
  cât timp cod ≠ 3 execută
    dacă cod = 1 atunci
      scrie 'Introduceți
        indicele elementului
                modificat: '
      citește ind
      scrie 'Introduceți
          valoarea care va fi
             adunată (valoare
              negativă pentru
                  scăderi): '
      citește val
      a_ind ← a_ind + val
    altfel
      scrie 'Introduceți ex-
          tremitățile subsec-
                  venței: '
      citește st, dr
      suma ← 0
      pentru i ← st, dr
                        execută
        suma ← suma + a_i
      sfârșit pentru
      scrie 'Suma elemente-
              lor secvenței
               este ', suma
    sfârșit dacă
    scrie 'Introduceți codul
                operației:'
    citește cod
  sfârșit cât timp
sfârșit algoritm
```

Se observă că modificările se efectuează în timp constant deoarece implică doar accesarea unui element al vectorului și modificarea valorii sale.

Datorită faptului că pentru calcularea sumei elementelor unei subsecvențe este necesară parcurgerea tuturor elementelor subsecvenței, această operație are ordinul de complexitate $O(l)$, unde l este lungimea subsecvenței.

Vector de sume

Prima încercare de optimizare constă în găsirea unui algoritm mai rapid pentru calcularea sumei elementelor unei subsecvențe. O posibilitate relativ simplă este păstrarea unui vector b ale cărui elemente b_i reprezintă suma primelor elemente ale șirului a. Pentru a găsi suma elementelor unei subsecvențe $<st, dr>$ vom efectua o simplă diferență: $b_{dr} - b_{st} - 1$.

Pentru a calcula sumele pentru subsecvențe de forma $<1, dr>$ vom avea nevoie de elementul b_0 care va avea întotdeauna valoarea 0. Astfel, pentru o subsecvență de această formă suma elementelor va fi $b_{dr} - b_0 = b_{dr}$.

Așadar, folosind acest artificiu, ordinul de complexitate al unei interogări va fi $O(1)$ pentru că este necesară doar o simplă scădere pentru furnizarea rezultatului. Din nefericire, în momentul efectuării unei modificări pentru elementul i, toate elementele b_j pentru care $j \geq i$ trebuie să își modifice valoarea.

Ca urmare, operația de modificare a celui de-al i-lea element nu se mai realizează în timp constant, deoarece trebuie modificate $N - i + 1$ valori. Așadar, ordinul de complexitate devine liniar.

Astfel, am reușit să înlocuim algoritmul liniar de determinare a sumei elementelor unei subsecvențe cu un algoritm având ordinul de complexitate $O(1)$ cu prețul creșterii timpului de execuție a algoritmului de modificare de la unul constant la unul liniar.

Se observă că nu mai avem nevoie de șirul a folosit pentru algoritmul anterior, fiind suficientă păstrarea valorilor elementelor șirului b.

Prezentăm versiunea în pseudocod a acestui algoritm, folosind aceleași coduri pentru operațiile efectuate:

```
algoritm ModificareVector
  inițializări
  scrie 'Introduceți numărul
                de elemente: '
  citește N
  pentru i ← 1, N execută
    b_i ← 0
  sfârșit pentru
  scrie 'Introduceți codul
                  operației: '
  citește cod
  cât timp cod ≠ 3 execută
    dacă cod = 1 atunci
      scrie 'Introduceți
         indicele elementului
                  modificat: '
      citește ind
      scrie 'Introduceți
          valoarea care va fi
             adunată (valoare
              negativă pentru
                  scăderi): '
      citește val
      pentru i ← ind, N
                          execută
        b_i ← b_i + val
      sfârșit pentru
    altfel
      scrie 'Introduceți ex-
          tremitățile subsec-
                    venței: '
      citește st, dr
      scrie 'Suma elemente-
        lor secvenței este ',
                    b_dr - b_st-1
    sfârșit dacă
    scrie 'Introduceți codul
                  operației:'
    citește cod
  sfârșit cât timp
sfârșit algoritm
```

Arbori indexați binar

Practic, pentru algoritmii anteriori una dintre cele două operații se efectuează în timp constant, în timp ce cealaltă se efectuează în timp liniar. Așadar, dacă numărul de modificări este aproximativ egal cu cel al interogărilor, timpul de execuție va fi aproximativ același.

Dacă numărul modificărilor este mult mai mare decât cel al interogărilor, atunci este preferabilă folosirea algoritmului naiv.

Dacă, dimpotrivă, numărul interogărilor este mult mai mare decât cel al modificărilor, atunci algoritmul bazat pe vectorul de sume este mai performant. Totuși, în cazul mediu, ambii algoritmi sunt liniari.

În cele ce urmează vom prezenta o structură de date care permite efectuarea în timp logaritmic atât a modificărilor, cât și a interogărilor. Structura de date pe care o propunem este numită ***arbore indexat binar***.

Așadar, vom avea o structură arborescentă care va permite efectuarea interogărilor în timp logaritmic în condițiile în care și modificările se efectuează în timp logaritmic.

Pentru a folosi această structură de date, va trebui să considerăm că elementele șirului sunt numerotate începând cu 1.

Arborele va fi păstrat sub forma unui vector c în care fiecare element i va conține suma elementelor subsecvenței $<i - 2^k + 1, i>$, unde k este numărul zerourilor terminale din reprezentarea binară a lui i.

Așadar, elementele de pe pozițiile impare ale arborelui vor păstra suma elementelor unor subsecvențe formate dintr-un singur element (aflat pe o poziție impară în șirul a).

În elementele de pe pozițiile de forma $4 \cdot k + 2$ (un zero terminal în reprezentarea binară a poziției) vom păstra suma elementelor unor subsecvențe formate din două elemente.

În elementele de pe pozițiile de forma $8 \cdot k + 4$ (două zerouri terminale în reprezentarea binară a poziției) vom păstra suma elementelor unor subsecvențe formate din patru elemente.

În general, dacă reprezentarea binară a pozițiilor au p zerouri terminale, atunci elementele din arbore vor păstra sume ale elementelor unor subsecvențe cu 2^p elemente.

Vom considera că, la un moment dat, șirul (format din 15 elemente) este (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15).

Valorile vectorului *c* (cel care reprezintă arborele indexat binar) sunt:

Element	*Poziție*	*Semnificație*	*Val*
c_1	0001	*Sumă*(1, 1)	1
c_2	0010	*Sumă*(1, 2)	3
c_3	0011	*Sumă*(3, 3)	3
c_4	0100	*Sumă*(1, 4)	10
c_5	0101	*Sumă*(5, 5)	5
c_6	0110	*Sumă*(5, 6)	11
c_7	0111	*Sumă*(7, 7)	7
c_8	1000	*Sumă*(1, 8)	36
c_9	1001	*Sumă*(9, 9)	9
c_{10}	1010	*Sumă*(9, 10)	19
c_{11}	1011	*Sumă*(11, 11)	11
c_{12}	1100	*Sumă*(9, 12)	42
c_{13}	1101	*Sumă*(13, 13)	13
c_{14}	1110	*Sumă*(13, 14)	27
c_{15}	1111	*Sumă*(15, 15)	15

Vom prezenta în continuare modul în care se efectuează modificările, exemplificând pe arborele prezentat anterior. În acest scop, vom modifica valoarea celui de-al doilea element din 2 în 8 (se adună valoarea 6).

Se observă că trebuie modificate toate valorile vectorului *c* care reprezintă sume ale unei subsecvențe care conține al doilea element al șirului; acestea sunt: c_2, c_4 și c_8.

Reprezentările binare ale acestor trei poziții sunt 0010, 0100 și 1000. Se observă în continuare că, fiecare astfel de reprezentare (evident, cu excepția primeia) poate fi obținută din cea anterioară prin adunarea valorii 2^r, unde *r* reprezintă numărul de zerouri terminale ale reprezentării binare a poziției anterioare.

Așadar, ar trebui să efectuăm astfel de adunări până în momentul în care poziția obținută este mai mare decât dimensiunea șirului și să creștem cu 6 valorile tuturor elementelor de pe pozițiile de la fiecare pas.

Vom exemplifica procedeul și pentru al treilea element. Prima poziție este 0011; numărul zerourilor terminale este 0, deci vom aduna valoarea $2^0 = 1$. Noua poziție va fi 3 + 1 = 4, deci am ajuns la poziția 4 a cărei reprezentare binară este 0100. Numărul zerourilor terminale este 2, deci vom aduna valoarea $2^2 = 4$, poziția devenind 4 + 4 = 8. Reprezentarea binară este 1000, deci avem trei zerouri terminale. Valoarea adunată va fi $2^3 = 8$, deci ajungem în poziția 8 + 8 = 16, așadar am depășit dimensiunea șirului.

Se observă că elementele c_3, c_4 și c_8 sunt cele a căror valoare depinde de valoarea elementului de pe a treia poziție.

Algoritmul care realizează operația de modificare este următorul:

- Se identifică poziția elementului care trebuie modificat.
- Cât timp poziția curentă este cel mult egală cu dimensiunea șirului:
  - Se modifică valoarea elementului de pe poziția curentă.
  - Se determină numărul *k* al zerourilor terminale din reprezentarea binară a poziției curente.
  - Noua poziție se determină adunând valoarea 2^k la poziția curentă.

Se observă că numărul elementelor modificate este cel mult egal cu numărul cifrelor reprezentării binare a numărului de elemente din șir, deci operația are ordinul de complexitate $O(\log N)$.

Mai trebuie prezentat modul în care este efectuată operația de interogare. Vom încerca să aplicăm o metodă similară celei propuse în cazul algoritmului care folosește un vector de sume. Pentru aceasta, dacă dorim să determinăm suma elementelor subsecvenței <*st*, *dr*>, vom determina sumele elementelor subsecvențelor <1, *st* - 1> și <1, *dr*>, efectuând apoi o simplă diferență.

Operațiile efectuate sunt, oarecum, inversele celor de la operația de modificare. Vom porni din poziția dată și, la fiecare pas, vom determina următoarea poziție scăzând valoarea 2^k, unde *k* reprezintă numărul zerourilor terminale din reprezentarea binară a poziției curente. Ne vom opri în momentul în care poziția curentă devine 0.

De exemplu, pentru a determina suma elementelor subsecvenței <1, 11> vom porni din poziția 11 a cărei reprezentare binară este 1011. Numărul zerourilor terminale este 0, deci vom scădea valoarea $2^0 = 1$, ajungând în poziția 11 - 1 = 10. Reprezentarea binară a acesteia este 1010, deci numărul zerourilor terminale este 1. Valoarea scăzută este $2^1 = 2$, deci se ajunge în poziția 10 - 2 = 8, a cărei reprezentare binară este 1000. De data aceasta avem trei zerouri terminale, deci vom scădea valoarea $2^3 = 8$ ajungând în poziția 8 - 8 = 0.

Așadar, am "trecut" prin pozițiile 11, 10 și 8. Adunând valorile elementelor corespunzătoare (c_{11}, c_{10} și c_8) obținem 11 + 19 + 36 = 66, adică exact valoarea căutată.

Este ușor de observat că c_{11} reprezintă suma elementelor subsecvenței <11, 11>, c_{10} reprezintă suma elementelor subsecvenței <9, 10>, iar c_8 reprezintă suma elementelor subsecvenței <1, 8>, așadar, în momentul calculării sumei, fiecare element de pe o poziție mai mică sau egală cu 11 este adunat o singură dată.

Pentru a determina suma elementelor unei subsecvențe de forma <1, *dr*> vom folosi următorul algoritm:

- Se identifică elementul de pe poziția *dr*.
- Cât timp poziția curentă este diferită de 0:
  - Se adună la suma totală valoarea elementului de pe poziția curentă.
  - Se determină numărul *k* al zerourilor terminale din reprezentarea binară a poziției curente.
  - Noua poziție se determină scăzând valoarea 2^k din poziția curentă.

Evident, pentru a determina suma elementelor unei subsecvențe de forma <*st*, *dr*> vom aplica de două ori algoritmul prezentat: o dată pentru subsecvența <1, *dr*> și a doua oară pentru subsecvența <1, *st* - 1>, efectuând la final diferența valorilor obținute.

De exemplu, pentru determinarea sumei elementelor subsecvenței <4, 13> vom calcula mai întâi sumele elementelor subsecvențelor <1, 13> și <1, 3>.

Acestea sunt: *Sumă*(1, 13) = c_{13} + c_{12} + c_8 = 13 + 42 + 36 = 91 și *Sumă*(1, 3) = c_3 + c_2 = 3 + 3 = 6, diferența lor fiind 91 - 6 = 85 care este egală cu valoarea *Sumă*(4, 13).

Se observă că numărul elementelor adunate este cel mult egal cu numărul cifrelor reprezentării binare a poziției elementului dat, deci operația are ordinul de complexitate *O*(log *N*).

Din nou, se observă că nu avem nevoie de păstrarea elementelor șirului *a*, fiind suficientă păstrarea valorilor vectorului *c* (cel care reprezintă arborele indexat binar).

Varianta în pseudocod a algoritmului de rezolvare a problemei folosind un arbore indexat binar este următoarea:

```
Algoritm ModificareEficientă
  inițializări
  scrie 'Introduceți numărul
              de elemente: '
  citește N
  pentru i ← 1, N execută
    c_i ← 0
  sfârșit pentru
  scrie 'Introduceți codul
               operației: '
  citește cod
  cât timp cod ≠ 3 execută
    dacă cod = 1 atunci
      scrie 'Introduceți
         indicele elementului
                modificat: '
      citește ind
      scrie 'Introduceți
          valoarea care va fi
             adunată (valoare
              negativă pentru
                 scăderi): '
      citește val
      poz ← 0
      cât timp ind ≤ N
                      execută
        c_ind ← c_ind + val
        cât timp ind & 2^poz =
                    0 execută
          poz ← poz + 1
        sfârșit cât timp
        ind ← ind + 2^poz
        poz ← poz + 1
      sfârșit cât timp
    altfel
      scrie 'Introduceți ex-
          tremitățile subsec-
                  venței: '
      citește st, dr
      s_1 ← 0
      poz ← 0
      cât timp dr > 0 execută
        s_1 ← s_1 + c_dr
        cât timp dr & 2^poz =
                    0 execută
          poz ← poz + 1
        sfârșit cât timp
        dr ← dr - 2^poz
        poz ← poz + 1
      sfârșit cât timp
      st ← st - 1
      s_2 ← 0
      poz ← 0
      cât timp st > 0 execută
        s_2 ← s_2 + c_st
        cât timp st & 2^poz =
                    0 execută
          poz ← poz + 1
        sfârșit cât timp
        st ← st - 2^poz
        poz ← poz + 1
      sfârșit cât timp
      scrie 'Suma elemente-
        lor secvenței este ',
                      s_1 - s_2
    sfârșit dacă
    scrie 'Introduceți codul
                  operației:'
    citește cod
  sfârșit cât timp
sfârșit algoritm
```

Cazul bidimensional

Problema pentru o singură dimensiune poate fi modificată astfel încât să lucrăm în spațiul bidimensional. Enunțul noii probleme ar putea fi următorul:

Se consideră o matrice de numere întregi care are toate elementele nule. O modificare a unui element constă în adunarea unei valori la valoarea curentă (pot fi realizate și scăderi care au forma adunării unor valori negative). Pe parcursul modificărilor pot apărea interogări referitoare la suma elementelor unei submatrice.

Așadar, în acest caz, interogările se referă la suma valorilor dintr-o "zonă dreptunghiulară" care face parte din matrice.

Evident, pentru reprezentarea datelor vom folosi tablouri bidimensionale în locul celor unidimensionale (matrice în locul vectorilor). Din nou, enunțul poate fi reformulat în diferite moduri. O variantă este chiar enunțul original a acestei probleme.

Vom adapta cei trei algoritmi descriși pentru cazul unidimensional pentru a rezolva noua problemă și vom studia apoi performanțele acestora.

De data aceasta, pentru operația de modificare vom avea nevoie de doi parametri care vor reprezenta linia și coloana pe care se află elementul a cărui valoare va fi modificată.

Pentru interogări vom avea nevoie de patru parametri care vor reprezenta coordonatele colțurilor din stânga-sus și dreapta-jos ale unui dreptunghi.

Vom exemplifica acum cazul bidimensional al problemei folosind o matrice care conține trei linii și trei coloane.

Operație	*Matrice/Rezultat*
Inițializare	0 0 0
	0 0 0
	0 0 0
Adună(2, 2, 3)	0 0 0
	0 3 0
	0 0 0
Adună(1, 3, 5)	0 0 5
	0 3 0
	0 0 0
Sumă(2, 1, 3, 3)	3
Adună(2, 1, 1)	0 0 5
	1 3 0
	0 0 0
Sumă(1, 2, 2, 3)	8
Adună(2, 2, -1)	0 0 5
	1 2 0
	0 0 0
Sumă(1, 2, 2, 3)	7
Adună(2, 2, 5)	0 0 5
	1 7 0
	0 0 0
Adună(2, 2, -5)	0 0 5
	1 2 0
	0 0 0
Adună(3, 2, 4)	0 0 5
	1 2 0
	0 4 0
Sumă(1, 1, 3, 3)	12

Algoritmul ineficient

Primul algoritm descris pentru cazul unidimensional poate fi adaptat foarte ușor pentru a rezolva cazul bidimensional al problemei. Vom păstra valorile matricei, le vom modifica dacă este necesar și vom calcula sumele cerute de interogări.

Versiunea în pseudocod este prezentată în continuare:

```
algoritm ModificareIneficientă
  inițializări
  scrie 'Introduceți numărul
                de elemente: '
  citește M, N
  pentru i ← 1, M execută
    pentru 1 ← 1, M execută
      a_ij ← 0
    sfârșit pentru
  sfârșit pentru
  scrie 'Introduceți codul
                operației: '
  citește cod
  cât timp cod ≠ 3 execută
    dacă cod = 1 atunci
      scrie 'Introduceți
          indicii elementului
                modificat: '
      citește indx, indy
      scrie 'Introduceți
          valoarea care va fi
             adunată (valoare
              negativă pentru
                  scăderi): '
      citește val
      a_indx,indy ← a_indx,indy+val
    altfel
      scrie 'Introduceți co-
       ordonatele colțurilor:'
      citește st, sus,
                      dr, jos
      suma ← 0
      pentru i ← sus, jos
                      execută
        pentru j ← st, dr
                      execută
          suma ← suma + a_ij
        sfârșit pentru
      sfârșit pentru
      scrie 'Suma elemente-
           lor dreptunghiului
                este ', suma
    sfârșit dacă
    scrie 'Introduceți codul
                operației:'
    citește cod
  sfârșit cât timp
sfârșit algoritm
```

Modificările se efectuează tot în timp constant, deoarece implică doar accesarea unui element al matricei și modificarea valorii sale.

Datorită faptului că pentru calcularea sumei elementelor dintr-un dreptunghi este necesară parcurgerea tuturor elementelor dreptunghiului, această operație are ordinul de complexitate $O(l \cdot h)$, unde l și h sunt lungimile laturilor dreptunghiului.

Matrice de sume

Adaptarea algoritmului care folosește un vector de sume este destul de simplă. Vom păstra o matrice b ale cărei elemente b_{ij} vor conține sumele corespunzătoare dreptunghiurilor cu colțul din stânga-sus în poziția (1, 1) și cel din dreapta-jos în poziția (i, j).

Este ușor de observat că pentru a calcula valoarea *Sumă*(st, sus, dr, jos) poate fi folosită relația:

Sumă(st, sus, dr, jos) =
Sumă(1, 1, dr, jos) -
Sumă(1, 1, st - 1, jos) -
Sumă(1, 1, dr, sus - 1) +
Sumă(1, 1, st - 1, sus - 1).

Se observă că prin scăderea valorii *Sumă*(1, 1, st-1, jos) se scad valorile tuturor elementelor aflate la dreapta dreptunghiului considerat, iar prin scăderea valorii *Sumă*(1, 1, dr, sus - 1) se scad valorile elementelor aflate deasupra dreptunghiului.

Aceste operații implică scăderea de două ori a tuturor elementelor aflate la stânga și deasupra dreptunghiului considerat, motiv pentru care va trebui să adunăm valoarea *Sumă*(1, 1, st - 1, sus - 1).

Folosind acest artificiu avem nevoie de accesarea a patru elemente ale matricei b, deci operația se realizează în timp constant.

Din nou, în momentul modificării valorii unui element de coordonate (x, y), va trebui să modificăm valorile tuturor elementelor matricei b de coordonate (i, j) pentru care $i \geq x$ și $j \geq y$. Așadar, vom modifica (M - x + 1) · (N - y + 1) elemente, ordinul de complexitate devenind $O(M \cdot N)$.

Pentru a putea determina valori pentru dreptunghiuri în care colțul din stânga-sus are una dintre coordonate egală cu 1 va trebui să considerăm că valorile elementelor matricei sunt nule dacă cel puțin unul dintre cei doi indici este 0.

Prezentăm versiunea în pseudocod a algoritmului descris:

```
algoritm ModificareMatrice
  inițializări
  scrie 'Introduceți numărul
                de elemente: '
  citește M, N
  pentru i ← 1, M execută
    pentru 1 ← 1, M execută
      b_ij ← 0
    sfârșit pentru
  sfârșit pentru
  scrie 'Introduceți codul
                operației: '
  citește cod
  cât timp cod ≠ 3 execută
    dacă cod = 1 atunci
      scrie 'Introduceți
          indicii elementului
                modificat: '
      citește indx, indy
      scrie 'Introduceți
          valoarea care va fi
             adunată (valoare
              negativă pentru
                  scăderi): '
      citește val
      pentru i ← indx, M
                      execută
        pentru j ← indy, N
                      execută
          b_ij ← b_ij + val
        sfârșit pentru
      sfârșit pentru
    altfel
      scrie 'Introduceți co-
       ordonatele colțurilor:'
      citește st, sus,
                      dr, jos
      scrie 'Suma elemente-
       lor dreptunghiului es-
        te ', b_dr,jos - b_st-1,jos -
             b_dr,sus-1 + b_st-1,sus-1
    sfârșit dacă
    scrie 'Introduceți codul
                operației:'
```

```
    citește cod
  sfârșit cât timp
sfârșit algoritm
```

Arbore de arbori indexați binar

Din nou, avem doi algoritmi în care una dintre operații este eficientă, în timp ce cealaltă necesită un timp de execuție mai mare. Folosind arborii indexați binar, vom putea găsi algoritmi care realizează ambele operații într-un timp de ordinul $O(\log M \cdot \log N)$, unde M și N sunt dimensiunile matricei.

Pentru a rezolva problema vom crea un arbore de arbori indexați binar. Așadar, vom avea o matrice c ale cărei elemente c_{ij} vor reprezenta sumele elementelor din dreptunghiul care are colțul din stânga-sus în poziția $(i - 2^k + 1, j - 2^l + 1)$, iar cel din dreapta-jos în poziția (i, j). Valoarea k reprezintă numărul zerourilor terminale din reprezentarea binară a lui i, iar l reprezintă numărul zerourilor terminale din reprezentarea binară a lui j.

De exemplu, pentru matricea:

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

elementele arborelui de arbori indexați binar vor avea următoarele semnificații:

Sumă(1, 1, 1, 1) *Sumă*(1, 1, 1, 2) *Sumă*(1, 3, 1, 3) *Sumă*(1, 1, 1, 4)
Sumă(1, 1, 2, 1) *Sumă*(1, 1, 2, 2) *Sumă*(1, 3, 2, 3) *Sumă*(1, 1, 2, 4)
Sumă(3, 1, 3, 1) *Sumă*(3, 1, 3, 2) *Sumă*(3, 3, 3, 3) *Sumă*(3, 3, 3, 4)
Sumă(1, 1, 4, 1) *Sumă*(1, 1, 4, 2) *Sumă*(1, 3, 4, 3) *Sumă*(1, 1, 4, 4).

Așadar, valorile matricei care reprezintă arborele binar vor fi:

1 3 3 6
6 14 10 36
9 19 11 23
28 60 36 136

Să presupunem că trebuie să modificăm valoarea elementului de coordonate (2, 1) care face parte dintr-o matrice cu 8 linii și 4 coloane.

Vom aplica de două ori principiul descris pentru cazul unidimensional al problemei.

Vom identifica linia i pe care se află elementul și vom efectua modificări în toate coloanele de pe acea linie în care acestea trebuie efectuate. Pentru aceasta vom identifica coloana j, vom determina numărul k al zerourilor terminale și apoi vom aduna la j valoarea 2^k. Ajungem într-o nouă poziție și continuăm procedeul până în momentul în care valoarea coloanei curente depășește numărul coloanelor matricei.

În acest moment vom determina numărul zerourilor terminale l din reprezentarea binară a valorii i și ne vom "muta" pe linia $i + 2^l$. Vom reveni la coloana j și vom parcurge aceeași pași ca și cei parcurși pentru linia anterioară.

Vom continua până în momentul în care valoarea liniei curente va fi mai mare decât numărul de linii din matrice.

Pe fiecare linie vom modifica cel mult $log_2 N$ elemente; în plus, vom modifica elemente de pe cel mult $log_2 M$ linii; așadar, ordinul de complexitate al operației de modificare a unui element este $O(\log M \cdot \log N)$.

Ca urmare, dacă se modifică elementul de coordonate (3, 1), atunci vor trebui modificate următoarele elemente ale matricei care reprezintă arborele de arbori indexați binar: c_{31}, c_{32}, c_{34}, c_{41}, c_{42}, c_{44}, c_{81}, c_{82} și c_{84}.

În cazul în care dorim să efectuăm o interogare vom folosi aceeași metodă. Singura diferență este, din nou, faptul că valorile de forma 2^k sunt scăzute în loc să fie adunate.

Datorită faptului că, de data aceasta, avem nevoie de patru sume pentru a furniza rezultatul, algoritmul va fi aplicat de patru ori. Ordinul de complexitate al algoritmului va fi tot $O(\log M \cdot \log N)$.

Vom prezenta în continuare versiunea în pseudocod a algoritmului de rezolvare a problemei în cazul bidimensional, folosind un arbore de arbori indexați binar.

```
algoritm ModificareEficientă
  inițializări
  scrie 'Introduceți numărul
              de elemente: '
  citește M, N
  pentru i ← 1, M execută
    pentru 1 ← 1, M execută
      c_ij ← 0
    sfârșit pentru
  sfârșit pentru
  scrie 'Introduceți codul
                operației: '
  citește cod
  cât timp cod ≠ 3 execută
    dacă cod = 1 atunci
      scrie 'Introduceți
          indicii elementului
                modificat: '
      citește indx, indy
      scrie 'Introduceți
          valoarea care va fi
             adunată (valoare
              negativă pentru
                  scăderi): '
      citește val
      pozi ← 0
      cât timp indx ≤ M
                      execută
        pozj ← 0
        j ← indy
        cât timp j ≤ N
                      execută
          c_indx,j ← c_indx,j +
                           val
          cât timp indx &
             2^pozj = 0 execută
            pozj ← pozj + 1
          sfârșit cât timp
          j ← j + 2^pozj
          pozj ← pozj + 1
        sfârșit cât timp
        cât timp indx & 2^pozi
                 = 0 execută
          pozi ← pozi + 1
        sfârșit cât timp
        indx ← indx + 2^pozi
        pozi ← pozi + 1
      sfârșit cât timp
    altfel
      scrie 'Introduceți co-
       ordonatele colțurilor:'
      citește st, sus,
                    dr, jos
```

```
s ← 0
x ← jos
y ← dr
pozi ← 0
cât timp x > 0 execută
  pozj ← 0
  j ← y
  cât timp y > 0
                 execută
    s ← s + c_xj
    cât timp x & 2^pozj
           = 0 execută
      pozj ← pozj + 1
    sfârșit cât timp
    j ← j - 2^pozj
    pozj ← pozj + 1
  sfârșit cât timp
  cât timp x & 2^pozi =
             0 execută
    pozi ← pozi + 1
  sfârșit cât timp
  x ← x - 2^pozi
  pozi ← pozi + 1
sfârșit cât timp
s_1 ← s
s ← 0
x ← jos
y ← st - 1
pozi ← 0
cât timp x > 0 execută
  pozj ← 0
  j ← y
  cât timp y > 0
                 execută
    s ← s + c_xj
    cât timp x & 2^pozj
           = 0 execută
      pozj ← pozj + 1
    sfârșit cât timp
    j ← j - 2^pozj
    pozj ← pozj + 1
  sfârșit cât timp
  cât timp x & 2^pozi =
             0 execută
    pozi ← pozi + 1
  sfârșit cât timp
  x ← x - 2^pozi
  pozi ← pozi + 1
sfârșit cât timp
s_2 ← s
s ← 0
x ← sus - 1
y ← dr
pozi ← 0
cât timp x > 0 execută
  pozj ← 0
  j ← y
  cât timp y > 0
                 execută
    s ← s + c_xj
    cât timp x & 2^pozj
           = 0 execută
      pozj ← pozj + 1
    sfârșit cât timp
    j ← j - 2^pozj
    pozj ← pozj + 1
  sfârșit cât timp
  cât timp x & 2^pozi =
             0 execută
    pozi ← pozi + 1
  sfârșit cât timp
  x ← x - 2^pozi
  pozi ← pozi + 1
sfârșit cât timp
s_3 ← s
s ← 0
x ← sus - 1
y ← dr - 1
pozi ← 0
cât timp x > 0 execută
  pozj ← 0
  j ← y
  cât timp y > 0
                 execută
    s ← s + c_xj
    cât timp x & 2^pozj
           = 0 execută
      pozj ← pozj + 1
    sfârșit cât timp
    j ← j - 2^pozj
    pozj ← pozj + 1
  sfârșit cât timp
  cât timp x & 2^pozi =
             0 execută
    pozi ← pozi + 1
  sfârșit cât timp
  x ← x - 2^pozi
  pozi ← pozi + 1
sfârșit cât timp
s_4 ← s
scrie 'Suma elemente-
 lor dreptunghiului es-
 te ', s_1 - s_2 - s_3 + s_4
sfârșit dacă
scrie 'Introduceți codul
                operației:'
citește cod
sfârșit cât timp
sfârșit algoritm
```

Acesta este, așadar, algoritmul eficient pentru rezolvarea problemei.

P050315: Concurs

Problema se reduce la a determina succesiv, cea mai îndepărtată pereche de puncte. Vom efectua, în total, N astfel de determinări. După fiecare determinare vom elimina punctele care formează cea mai îndepărtată pereche și vom efectua următoarea determinare pentru punctele rămase. Ne vom opri în momentul în care sunt determinate toate cele N distanțe.

Așadar, pentru a rezolva această problemă va trebui să aplicăm de N ori algoritmul de determinare a celei mai îndepărtate perechi de puncte. Acest algoritm va fi prezentat în cele ce urmează.

Algoritmul ineficient

O rezolvarea ineficientă constă în determinarea distanțelor dintre toate perechile de puncte (vor fi $2 \cdot N \cdot (2 \cdot N - 1) / 2$ astfel de perechi) și alegerea, de data aceasta, a celei mai mari dintre ele.

Așadar, problema se reduce la o simplă determinare a unei valori maxime dintre $2 \cdot N \cdot (2 \cdot N - 1) / 2$ valori care sunt calculate.

Evident, ordinul de complexitate al acestui algoritm este $O(N^2)$.

Algoritmul eficient

Pentru început, este ușor de observat faptul că cele două puncte se află, obligatoriu, pe înfășurătoarea convexă a mulțimii de puncte.

Vom demonstra această afirmație prin metoda reducerii la absurd. Să presupunem că unul dintre cele două puncte nu se află pe înfășurătoarea convexă, așadar se află în interiorul acesteia. Vom nota celălalt punct prin P, iar acest punct prin Q. Dacă vom prelungi segmentul care unește punctele P și Q, atunci acesta va intersecta o latură a înfășurătorii convexe sau va trece printr-un alt punct al acesteia.

În acest din urmă caz se observă imediat că distanța de la P la Q este mai mică decât distanța de la P la punctul respectiv, ca urmare punctele P și Q nu se pot afla la distanța maximă.

Pentru cel de-al doilea caz vom considera că latura intersectată are ca extremități punctele R și S, iar prelungirea segmentului care unește punctele P și Q intersectează latura respectivă în punctul T.

Pentru început se observă faptul că lungimea segmentului care unește punctele P și T este mai mare decât cea care unește punctele P și Q.

Vom considera acum triunghiul PRS (determinat de vârfurile P, R și S). Punctul T se va afla pe latura RS a acestui triunghi. Există o teoremă care afirmă că cel puțin una dintre laturile PR și PS are lungimea mai mare decât segmentul PT.

Ca urmare una dintre laturile triunghiului va fi mai lungă decât segmentul PT care, la rândul său, este mai lung decât segmentul PQ. Datorită faptului că această latură are ca extremități două vârfuri ale înfășurătorii convexe, deducem imediat că punctele P și Q nu se pot afla la distanța maximă.

În concluzie, primul pas necesar pentru a determina punctele aflate la distanța maximă este determinarea unei înfășurători convexe.

O variantă care poate fi utilizată în continuare este considerarea succesivă a vârfurilor înfășurătorii și determinarea, printr-o metodă similară cu căutarea binară, a celui mai îndepărtat vârf față de vârful considerat.

Se va considera nodul din stânga (sau dreapta) nodului ales. Atât timp cât distanța crește, la fiecare pas k, vom "sări" la al 2^k-lea nod. În momentul în care distanța începe să scadă, vom reveni la nodul anterior și vom începe să înjumătățim "distanța" saltului.

Dacă am ajuns la pasul l, acum, la fiecare, pas vom descrește cu 1 puterea. Dacă la un anumit pas, puterea este p, vom efectua un salt la al 2^p-lea nod. Dacă distanța descrește vom reveni, iar dacă nu, vom continua de la vârful respectiv. În final, vom obține vârful aflat la distanța maximă față de nodul considerat.

După considerarea tuturor vârfurilor, vom obține distanța maximă între două puncte din mulțimea dată.

Algoritmul descris funcționează corect deoarece, parcurgând nodurile într-un anumit sens distanța începe să crească până la un moment dat, după care ea începe să scadă.

O a doua variantă constă în alegerea unui vârf P_1 al înfășurătorii și parcurgerea acesteia (începând cu punctul din stânga sau din dreapta sa) atâta timp cât distanța crește.

Vom nota punctul respectiv prin P. În continuare vom trece la punctul P_2 (cel aflat pe înfășurătoare imediat lângă punctul P_1, în funcție de sensul ales) și vom continua parcurgerea.

Poate fi ușor observat faptul că nu are sens să mai verificăm toate punctele parcurse la pasul anterior (cele până la P) deoarece distanțele obținute vor fi mai mici.

Așadar, vom continua de la punctul P atâta timp cât distanța va crește. Vom trece apoi la punctul P_3 și vom repeta în continuare procedeul până în momentul în care am luat în considerare toate punctele de pe înfășurătoare.

Evident, în final vom cunoaște distanța maximă dintre două puncte de pe înfășurătoare și, implicit, distanța maximă dintre două puncte ale mulțimii date.

Analiza complexității

Este cunoscut faptul că operația de determinarea a înfășurătorii convexe are ordinul de complexitate $O(N \cdot \log N)$, dacă utilizăm scanarea *Graham* sau $O(N \cdot h)$, dacă utilizăm potrivirea *Jarvis*.

În situația în care, după determinarea înfășurătorii, vom folosi metoda căutării binare, distanța maximă va fi determinată într-un timp de ordinul $O(h \cdot \log h)$, datorită faptului că vom efectua h căutări binare, fiecare având ordinul de complexitate $O(\log h)$.

În cazul în care vom utiliza cea de-a doua metodă, aceasta va avea ordinul de complexitate $O(h)$, deoarece se poate observa faptul că prin avansări se va parcurge de cel mult două ori înfășurătoarea.

Ca urmare, ordinul de complexitate al algoritmului de determinare a unei perechi de puncte aflate la distanță maximă este același cu cel al algoritmului folosit pentru determinarea înfășurătorii convexe.

P050316: Zid magic

Problema se reduce la a determina înfășurătoarea convexă a punctelor care reprezintă coordonatele locuințelor elfilor.

Datorită faptului că este posibil ca toate punctele să se afle pe această înfășurătoare convexă trebuie utilizată scanarea *Graham*, deoarece potrivirea *Jarvis* ar putea avea ordinul de complexitate $O(N^2)$, iar limita de timp admisă pentru execuția programului va fi depășită.

După determinarea înfășurătorii convexe vom scrie în fișierul de ieșire punctele care o determină, în ordine trigonometrică.

Analiza complexității

Citirea datelor de intrare implică citirea celor N perechi de coordonate ale locuințelor elfilor, așadar ordinul de complexitate al operației de citire a datelor de intrare este $O(N)$.

Pentru cele N puncte vom determina înfășurătoarea convexă folosind scanarea *Graham*, operație al cărei ordin de complexitate este $O(N \cdot \log N)$.

Scrierea datelor în fișierul de ieșire se realizează în timp liniar deoarece vom scrie cel mult N numere. Așadar, ordinul de complexitate al acestei operații este $O(N)$.

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

P050317: Locuințe

Problema se reduce la a verifica, pentru fiecare dintre cărări (drepte) dacă punctele corespunzătoare căsuțelor elfilor cu haine deschise la culoare se află de o parte a dreptei și punctele corespunzătoare căsuțelor elfilor cu haine închise la culoare se află de cealaltă parte a dreptei.

Pentru aceasta vom determina ecuațiile dreptelor și vom înlocui variabilele x și y ale acestora cu coor-

donatele căsuțelor elfilor. Pentru căsuțele elfilor cu haine deschise la culoare va trebui ca rezultatul să aibă un anumit semn, iar pentru căsuțele elfilor cu haine închise la culoare acesta va trebui să aibă semn opus.

O variantă este să determinăm semnul pentru căsuța primului elf cu haine deschise la culoare. Apoi vom determina semnele pentru căsuțele tuturor celorlalți elfi cu haine deschise la culoare.

Dacă apare un semn diferit putem trage imediat concluzia că această cărare nu este validă. Evident, dacă obținem o valoare 0, dreapta corespunzătoare cărării va trece prin punctul corespunzător căsuței unui elf, deci nici în această situație cărarea nu este validă.

Dacă nu am identificat nici o situație care să ducă la concluzia că o cărare nu este validă, putem fi siguri că toți elfii cu haine deschise la culoare au căsuțele de aceeași parte a dreptei.

În continuare, vom determina semnele pentru căsuțele elfilor cu haine închise la culoare. Dacă apare valoarea 0 sau același semn ca și pentru elfii cu haine deschise la culoare, rezultă imediat că acea cărare nu este validă.

Dacă nu am identificat nici acum o situație care să ducă la concluzia că acea cărare nu este validă, putem fi siguri că toți elfii cu haine închise la culoare au căsuțele de cealaltă parte a dreptei.

Ca urmare, în acest moment putem concluziona că respectiva cărare este validă.

Analiza complexității

Citirea datelor de intrare implică citirea celor m perechi de coordonate ale căsuțelor elfilor cu haine deschise la culoare, a celor n perechi de coordonate ale căsuțelor elfilor cu haine închise la culoare, precum și coordonatele perechilor de puncte care determină cele k drepte corespunzătoare cărărilor propuse de sfetnici. Așadar, ordinul de complexitate al operației de citire a datelor de intrare este $O(m + n + k)$.

Pentru fiecare dintre cele k cărări vom verifica, pe baza coordonatelor celor două puncte date, ecuațiile dreptelor corespunzătoare. Pentru o cărare, ordinul de complexitate al acestei operații este $O(1)$.

În cel mai defavorabil caz, se verifică semnele corespunzătoare tuturor căsuțelor elfilor cu haine deschise la culoare și tuturor căsuțelor elfilor cu haine închise la culoare. Așadar, pentru o dreaptă vom verifica semnele pentru $m + n$ căsuțe, operație al cărei ordin de complexitate este $O(m + n)$.

După verificare, vom scrie în fișierul de ieșire mesajul corespunzător, operație al cărei ordin de complexitate este $O(1)$.

Rezultă imediat că operația de verificare a validității unei poteci are ordinul de complexitate $O(1) + O(m + n) + O(1) = O(m + n)$. În total vom efectua k astfel de verificări, ordinul de complexitate al întregi operații fiind $O(k) \cdot O(m + n) = O(k \cdot (m + n))$.

Scrierea datelor în fișierul de ieșire se realizează pe parcursul verificărilor, deci nu se consumă timp suplimentar pentru această operație.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(m + n + k) + O(k \cdot (m + n)) = O(k \cdot (m + n))$.

P050318: Cercuri

În cazul în care considerăm că cercurile sunt vârfurile unui graf orientat, problema se reduce la alegerea unei configurații a arcelor, astfel încât fiecare nod să aibă un grad exterior dat (numărul săgeților care pleacă din cercul corespunzător) și un grad interior dat (numărul săgeților care ajung la cercul corespunzător).

Pentru aceasta vom crea o rețea de transport care va conține $2 \cdot n + 2$ noduri. Două dintre acestea vor fi sursa și destinația. n dintre noduri vor fi legate de sursă prin arce ale căror capacități vor fi egale cu gradele interioare ale nodurilor.

Celelalte n noduri vor fi legate de destinație prin arce ale căror capacități vor fi egale cu gradele exterioare ale nodurilor.

De la fiecare dintre primele n noduri vor pleca arce de capacitate 1 spre fiecare dintre celelalte n noduri.

În această rețea vom determina fluxul maxim. Dacă acesta va fi m, atunci săgețile pot fi desenate (graful poate fi construit). Arcele rețelei de transport pe care există flux vor indica modul în care vor fi desenate săgețile (configurația arcelor).

Analiza complexității

Datele de intrare constau din $2 \cdot n + 2$ numere, deci ordinul de complexitate al operației de citire a acestora va fi $O(n)$.

Pentru crearea rețelei de transport va trebui, mai întâi să construim cele $2 \cdot n + 2$ noduri ale acesteia, operație al cărui ordin de complexitate este $O(n)$. Pentru a lega primele n noduri de sursă avem nevoie de un timp de ordinul $O(n)$; același ordin are și timpul necesar legării celorlalte n noduri de destinație.

Fiecare dintre primele n noduri va fi legat de fiecare dintre ultimele n (cu o singură excepție - nu va exista un arc de la nodul i la nodul $n + i$, deoarece o săgeată trebuie să pornească de la un cerc și să ajungă la un alt cerc); ordinul de complexitate al operației pentru un nod este $O(n)$, deci pentru toate nodurile obținem un timp de ordinul $O(n^2)$. Așadar, ordinul de complexitate al operației de construire a rețelei de transport este $O(n) + O(n) + O(n) + O(n^2) = O(n^2)$.

Vom observa că numărul arcelor rețelei de transport va fi $n + n + n \cdot (n - 1) = n \cdot (n + 1)$. În această rețea va trebui să determinăm un flux maxim.

Datorită faptului că știm că acesta va fi cel mult m, putem folosi algoritmul *Ford-Fulkerson* al cărui ordin de complexitate va fi $O(m \cdot (n + n \cdot (n + 1))) = O(m \cdot n^2)$, deoarece fluxul maxim va fi m, iar numărul de muchii va fi $n \cdot (n + 1)$.

Dacă am alege algoritmul *Edmonds-Karp*, am avea, teoretic, ordinul de complexitate $O((n \cdot (n + 1) + n) \cdot n \cdot (n + 1) \cdot n) = O(n^5)$. Totuși, datorită parcurgerii în lățime, fiecare drum are, de fapt, lungimea 3 în

foarte multe situații (marea majoritate), deci ordinul real de complexitate în cazul mediu va fi, de fapt, $O(n^3)$.

După determinarea fluxului va trebui doar să verificăm arcele care conțin flux nenul și să scriem datele în fișierul de ieșire. Ordinul de complexitate al acestei operații este $O(n^2)$.

În concluzie, algoritmul de rezolvare a acestei probleme pentru cazul mediu are ordinul de complexitate $O(n) + O(n^2) + O(m \cdot n^2) + O(n^2) = O(m \cdot n^2)$ dacă fluxul este determinat cu ajutorul algoritmului *Ford-Fulkerson* și $O(n) + O(n^2) + O(n^3) + O(n^2) = O(n^3)$ dacă se utilizează algoritmul *Edmonds-Karp*.

P050319: Ordine

Poienile din *Pădurea Aurie* pot fi considerate a fi nodurile unui graf neorientat ale cărui muchii sunt date de drumurile dintre acestea.

Astfel, problema se reduce la identificarea unui ciclu într-un graf. Pentru aceasta putem utiliza un algoritm de determinare a ciclurilor și ne vom opri în momentul în care este identificat primul ciclu. Enunțul problemei ne asigură că graful va conține întotdeauna cel puțin un ciclu.

După identificarea ciclului vom scrie în fișierul de ieșire numerele de ordine ale nodurilor care formează ciclul identificat.

Analiza complexității

Citirea datelor de intrare implică citirea extremităților celor m muchii ale grafului, operație al cărei ordin de complexitate este $O(m)$. Pe parcursul citirii vom crea și reprezentarea grafului, operație care nu consumă timp suplimentar.

În continuare va trebui să identificăm un ciclu, operație care implică, în cel mai defavorabil caz, efectuarea unei parcurgeri complete în adâncime a grafului. Aceasta are ordinul de complexitate $O(m + n)$.

Datele de ieșire constau în numerele de ordine ale nodurilor care formează un ciclu. Ciclul poate fi format, în cel mai defavorabil caz, din toate cele n noduri ale grafului, deci operația de scriere a rezultatului în fișierul de ieșire are ordinul de complexitate $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(m) + O(m + n) + O(n) = O(m + n)$.

P050320: Strategie

Cele n obiective strategice vor reprezenta nodurile unui graf neorientat ale cărui muchii vor fi date de cele m cărări

Astfel, problema se reduce la determinarea unei partiționări în cicluri disjuncte pentru graful construit. Numărul ciclurile disjuncte va fi întotdeauna $m - n + 1$. Pe măsură ce identificăm aceste cicluri le vom scrie în fișierul de ieșire.

Pentru a partiționa un graf în cicluri disjuncte va trebui, mai întâi, să determinăm un arbore parțial. Apoi vom lua în considerare toate muchiile care nu fac parte din acest arbore și vom închide ciclurile corespunzătoare.

Cea mai convenabilă modalitate este folosirea arborelui *DF*, deoarece operația de închidere a ciclurilor este mai simplă. Toate muchiile care nu fac parte din arborele *DF* vor fi muchii de întoarcere. În momentul în care se va identifica o astfel de muchie, va fi determinat și ciclul pe care îl închide.

Analiza complexității

Citirea datelor de intrare implică citirea extremităților celor m muchii ale grafului, operație al cărei ordin de complexitate este $O(m)$. Pe parcursul citirii vom crea și reprezentarea grafului, operație care nu consumă timp suplimentar.

În continuare va trebui să identificăm ciclurile disjuncte ale grafului, operație al cărei ordin de complexitate este $O(m \cdot n - n^2)$. Pe măsura identificării acestora, ele vor fi scrise în fișierul de ieșire, deci generarea fișierului de ieșire nu va consuma timp suplimentar.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(m) + O(m \cdot n - n^2) = O(m \cdot n - n^2)$.

P050321: Zvonuri

Putem privi elfii ca fiind nodurile unui graf orientat în care există o muchie de la un nod la altul, dacă un zvon este comunicat de către elevul corespunzător primului nod la nodul corespunzător celui de-al doilea nod.

În aceste condiții problema se reduce la determinarea numărului componentelor tare-conexe ale unui graf orientat.

Datorită numărului mare de muchii, vom păstra graful sub forma unei liste a succesorilor. Pentru fiecare nod vom cunoaște poziția la care încep succesorii săi în această listă. Evident, poziția la care se termină succesorii unui nod este dată de poziția la care încep succesorii următorului nod (cel care are numărul de ordine mai mare cu 1). Folosind această reprezentare a grafului, vom putea aplica fără dificultăți deosebite oricare dintre algoritmii rapizi de determinare a componentelor tare-conexe.

Pe parcursul determinării acestor componente le vom număra. Evident, datorită numărului relativ mare de noduri, algoritmul simplu (cel care are ordinul de complexitate $O(n^3)$) nu va putea fi folosit. Totuși, algoritmul *plus-minus*, care rulează în timp pătratic poate fi utilizat.

Analiza complexității

Citirea datelor de intrare implică citirea succesorilor nodurilor grafului (care sunt folosiți pentru crearea listei celor m arce), așadar ordinul de complexitate al acestui algoritm este $O(m)$.

Pentru a determina pozițiile de început ale succesorilor nodurilor, este necesară o parcurgere a vectorului care conține numerele succesorilor, ordinul de complexitate al acestei operații fiind $O(n)$.

Pentru a determina șirul succesorilor este necesară o nouă parcurgere a muchiilor; ca urmare, ordinul de complexitate al acestei operații este $O(m)$.

În continuare vom presupune că am folosit algoritmul eficient de determinare a componentelor tare-conexe. Așadar acestea sunt identificate într-un timp de ordinul $O(m + n)$.

Datele de ieșire constau într-un singur număr, deci operația de scriere a rezultatului în fișierul de ieșire are ordinul de complexitate $O(1)$.

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

P050322: Plan

Satele elfilor pot fi privite ca fiind nodurile unui graf ale cărui muchii sunt date de drumurile dintre sate. Costurile muchiilor reprezintă costurile necesare construirii unui zid magic pe cărările dintre două sate.

În aceste condiții problema se reduce la determinarea unui arbore parțial de cost minim și a celui de-al doilea arbore parțial de cost minim într-un graf neorientat. Prin al doilea arbore parțial minim înțelegem arborele cu cel mai mic cost care este diferit de arborele parțial de cost minim.

Pentru a determina arborele parțial de cost minim putem folosi unul dintre algoritmii clasici: *Kruskal* sau *Prim*. Datorită faptului că graful este specificat prin lista muchiilor, poate fi preferat algoritmului lui *Kruskal*.

Pentru a utiliza algoritmul lui *Kruskal*, muchiile trebuie sortate în funcție de costul lor. Datorită faptului că sunt cel mult 1000 de muchii, nu trebuie neapărat folosit un algoritm performant de sortare, fiind suficient unul care funcționează în timp pătratic.

După construirea listei ordonate a muchiilor, arborele parțial de cost minim se determină folosind algoritmul amintit. Vom crea un vector de indici în care un element va indica numărul de ordine (în lista sortată) al unei muchii care face parte din arborele parțial de cost minim.

Vom demonstra în continuare că al doilea arbore parțial de cost minim diferă printr-o singură muchie de arborele parțial de cost minim. În acest scop vom utiliza metoda reducerii la absurd. Vom presupune că acest arbore diferă prin două sau mai multe muchii. Așadar, au fost eliminate cel puțin două muchii care au fost înlocuite cu altele.

Evident, costurile muchiilor adăugate sunt mai mari sau egale cu costurile muchiilor eliminate deoarece, în caz contrar ele ar fi făcut parte din arborele parțial de cost minim.

Reintroducând în arbore una dintre muchiile eliminate și eliminând-o pe cea cu care aceasta a fost înlocuită vom obține un arbore parțial minim care are un cost mai mic sau egal. În plus, acest arbore diferă de arborele parțial de cost minim datorită prezenței celeilalte muchii nou introduse.

Ca urmare, am obținut un arbore care are un cost mai mic sau egal decât "al doilea arbore parțial de cost minim" și un cost mai mare sau egal decât cel al arborelui parțial de cost minim. În concluzie, ipoteza este falsă, deci al doilea arbore parțial de cost minim diferă printr-o singură muchie de arborele parțial de cost minim.

Pentru a determina al doilea arbore parțial de cost minim vom considera că oricare dintre muchiile care nu fac parte din arborele parțial de cost minim este o muchie candidată pentru inserare.

Prin inserarea unei astfel de muchii se creează un ciclu; așadar, una dintre muchiile din acest ciclu trebuie eliminată pentru a păstra proprietatea de arbore. Pentru ca arborele obținut să aibă un cost cât mai mic vom elimina acea muchie care are cea mai mică diferență de cost față de muchia introdusă.

Pentru a regăsi soluția vom păstra diferența minimă obținută la un moment dat, muchia adăugată în momentul în care a fost determinată această diferență minimă și muchia eliminată în momentul respectiv.

Pentru a determina ciclul format prin adăugarea unei muchii vom crea o listă de părinți ai nodurilor din arborele parțial de cost minim. Pentru fiecare nod, cu excepția nodului 1 (pe care îl vom considera ca fiind rădăcina arborelui) vom păstra numărul de ordine al părinților nodurilor în arborele parțial de cost minim care are rădăcina în nodul 1.

Lista părinților poate fi determinată folosind următorul algoritm: cât timp nu au fost stabiliți părinții tuturor nodurilor, se parcurge lista muchiilor arborelui parțial de cost minim; în momentul în care este identificată o muchie pentru care se cunoaște părintele uneia dintre extremități și nu se cunoaște părintele celeilalte extremități se poate afirma că părintele acestei a doua extremități este prima extremitate.

Cunoscând lista părinților, se poate determina un ciclu format prin adăugarea unei muchii, dacă se parcurg traseele de la extremitățile muchiilor spre rădăcină.

După determinarea acestor două trasee se elimină porțiunile comune (acestea nu fac parte din ciclu).

Muchiile de pe cele două trasee (după eliminarea porțiunii comune) sunt muchiile care, împreună cu muchia adăugată, formează un ciclu.

În acest moment se poate verifica dacă prin eliminarea uneia dintre muchiile de pe trasee și adăugarea muchiei considerate se obține o diferență mai mică decât minimul din acel moment.

Pentru a obține mai rapid costul unei muchii va trebui să avem la dispoziție o matrice a costurilor. Aceasta poate fi creată pe măsura citirii datelor de intrare.

În final vom cunoaște muchiile care fac parte din primul arbore parțial minim, muchia care este eliminată în vederea obținerii celui de-al doilea arbore de cost minim și muchia care trebuie inserată în vederea obținerii acestui al doilea arbore de cost minim.

Este cunoscut faptul că numărul de muchii dintr-un arbore parțial al unui graf cu n noduri este n - 1. Pentru a scrie în fișierul de ieșire muchiile arborelui parțial de cost minim va trebui să parcurgem doar lista muchiilor și să scriem extremitățile corespunzătoare.

Pentru a scrie în fișierul de ieșire muchiile celui de-al doilea arbore parțial de cost minim vom scrie mai întâi extremitățile muchiei adăugate.

În continuare vom parcurge din nou lista muchiilor și vom scrie extremitățile corespunzătoare numai

dacă acestea nu sunt cele ale muchiei eliminate.

Analiza complexității

Citirea datelor de intrare implică citirea celor m muchii ale grafului, așadar ordinul de complexitate al acestui algoritm este $O(m)$. În paralel cu citirea se realizează și crearea matricei costurilor, ordinul de complexitate al acestei operații fiind tot $O(m)$.

Algoritmul de sortare a muchiilor grafului are ordinul de complexitate $O(m^2)$ deoarece nu este necesară folosirea unui algoritm mai performant.

Algoritmul de determinare a muchiilor care fac parte din arborele parțial minim implică o parcurgere a listei sortate a muchiilor și, la fiecare pas, actualizarea claselor de echivalență corespunzătoare nodurilor grafului. Ca urmare, ordinul de complexitate al acestei operații este $O(m \cdot n)$.

Pentru determinarea listei părinților va trebui parcursă de mai multe ori lista muchiilor. Așadar, fiecare parcurgere are ordinul de complexitate $O(m)$. Datorită faptului că avem n noduri și la fiecare parcurgere se va determina cel puțin părintele unui nod, ordinul de complexitate al operației de determinare a listei părinților este $O(m \cdot n)$.

Pentru a determina cel de-al doilea arbore parțial de cost minim va trebui să luăm în considerare toate cele $m - n + 1$ muchii care nu fac parte din arborele parțial de cost minim. Pentru fiecare astfel de muchie vom determina traseele până la rădăcină (nodul 1).

Un astfel de traseu va conține cel mult $n - 1$ noduri, deci ordinul de complexitate al operației de determinare a unui traseu este $O(n)$. Cele două trasee conțin cel mult $n - 1$ noduri, deci operația de eliminare a porțiunii comune are ordinul de complexitate tot $O(n)$.

Considerarea muchiilor care fac parte din ciclu are același ordin de complexitate $O(n)$ deoarece un ciclu este format din cel mult n muchii. În concluzie, operația de determinare a celui de-al doilea arbore parțial de cost minim are ordinul de complexitate $O(m - n + 1) \cdot (O(n) + O(n) + O(n)) = O(m - n) \cdot O(n) = O(m \cdot n - n^2)$.

Scrierea datelor în fișierul de ieșire implică două traversări a listei muchiilor arborelui parțial de cost minim, deci are ordinul de complexitate $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(m) + O(m^2) + O(m \cdot n) + O(m \cdot n) + O(m \cdot n - n^2) + O(n) = O(m \cdot (m + n))$.

P050323: Mesaj

Satele elfilor pot fi privite ca fiind nodurile unui graf neorientat în care muchiile reprezintă cărările dintre sate.

În aceste condiții problema se reduce la determinarea, pentru fiecare nod, a distanței față de nodul identificat prin 1 (cel care reprezintă cel mai important sat).

Distanța dintre două noduri este dată de numărul muchiilor din care este format cel mai scurt drum dintre cele două noduri.

Pentru a determina aceste distanțe vom folosi algoritmul de parcurgere în lățime (***B**readth **F**irst - **BF***) a unui graf. La fiecare pas vom determina, pentru toate nodurile aflate pe nivelul curent, nodurile de pe nivelul următor.

Nodurile de pe nivelul următor sunt acele noduri care au legături directe cu cel puțin un nod de pe nivelul curent și nu se află pe un nivel anterior.

Este evident faptul că toate nodurile de pe un anumit nivel se vor afla la aceeași distanță față de nodul identificat prin 1.

Așadar, distanța față de acest nod va fi dată de nivelul pe care se află nodul respectiv în urma parcurgerii în lățime.

După determinarea acestor distanțe vom scrie rezultatele în fișierul de ieșire.

Analiza complexității

Citirea datelor de intrare implică citirea celor m muchii ale grafului, așadar ordinul de complexitate al acestui algoritm este $O(m)$. În paralel cu citirea se realizează crearea structurii de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot $O(m)$.

Algoritmul de parcurgere în lățime a unui graf are ordinul de complexitate $O(m)$, deci acesta este ordinul de complexitate al operației de determinare a distanțelor la care se află nodurile față de nodul identificat prin 1.

Scrierea datelor în fișierul de ieșire implică scrierea a $n - 1$ valori, așadar ordinul de complexitate al acestei operații este $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(m) + O(m) + O(m) + O(n) = O(m + n)$.

P050324: Atac

Satele elfilor pot fi privite ca fiind nodurile unui graf neorientat în care muchiile reprezintă cărările dintre sate. Un sat poate fi părăsit în siguranță doar dacă nu este nod critic în graf.

Vom utiliza algoritmul de determinare a nodurilor critice și, în momentul detectării unui astfel de nod, vom marca acest nod ca fiind critic.

În final, vom număra nodurile necritice și vom scrie numerele de ordine ale acestora.

Analiza complexității

Citirea datelor de intrare implică citirea celor m muchii ale grafului, așadar ordinul de complexitate al acestui algoritm este $O(m)$. În paralel cu citirea se realizează crearea structurii de date în care se memorează graful, ordinul de complexitate al acestei operații fiind tot $O(m)$.

Algoritmul de determinare a nodurilor critice ale unui graf are ordinul de complexitate $O(m + n)$.

Datorită faptului că un graf poate conține cel mult n noduri critice, ordinul de complexitate al operației de scriere a datelor de ieșire are ordinul de complexitate $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(m) + O(m) + O(m + n) + O(n) = O(m + n)$.