Košice, Slovacia

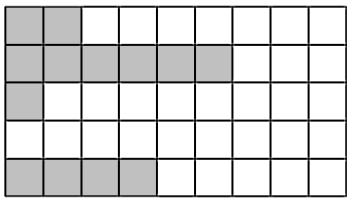
CEOI 2002

În continuare vă vom prezentăm soluțiile oficiale ale celor șase probleme propuse spre rezolvare la ediția din acest an a Olimpiadei de Informatică a Europei Centrale.

P060207: Bugs Integrated, Inc.

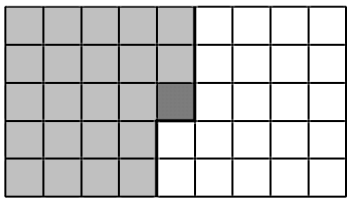
În prezentarea acestei soluții vom considera că valorile coordonatelor orizontale variază între 0 și N - 1, iar cele ale coordonatelor verticale între 0 și M - 1.

Fie $B = (b_0, b_1, \ldots, b_{M-1})$ un vector de numere întregi (numit *margine*). Definim mulțimea mărginită $S(B)$ ca fiind formată din toate celulele de coordonate $[x, y]$ astfel încât $x \le b_y$. Cu alte cuvinte, mulțimea marginii este formată din toate celulele aflate la stânga marginii. De obicei, nu se va face distincție între o margine și mulțimea sa.



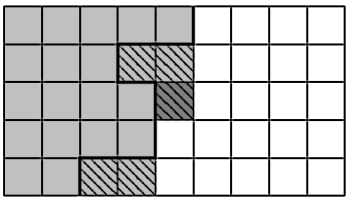
Mulțimea marginii B = (1, 5, 0, -7, 3) pentru N = 9 și M = 5

Pentru fiecare pereche $[i, j]$ ($0 \le i < N$, $0 \le j < M$) definim $[i, j]$-***marginea*** ca fiind $B[i, j] = (b_0, b_1, \ldots, b_{M-1})$, astfel încât $b_0 = b_1 = \ldots = b_j = i$ și $b_{j+1} = b_{j+2} = \ldots = b_{M-1} = i - 1$. Mulțimea $S(B[i, j])$ va fi numită $[i, j]$-***bază***. Cu alte cuvinte, $S(B[i, j])$ este mulțimea celulelor care se află la stânga celulei de coordonate $[i, j]$ sau pe aceeași coloană, dar deasupra celulei $[i, j]$. Celula $[i, j]$ este inclusă și ea în $[i, j]$-*bază*.



[4, 2]-marginea B[4, 2] = (4, 4, 4, 3, 3) și baza sa

Considerăm o bază cu marginea $B[i, j]$. Fie P un vector $(p_0, p_1, \ldots, p_{M-1})$ ale cărui elemente sunt cuprinse în mulțimea {0, 1, 2}. Un astfel de vector este numit ***profil***. Vom nota prin $B - P$ vectorul $(b_0 - p_0, \ldots, b_{M-1} - p_{M-1})$. Pentru *baza* considerată vom defini *profilul* P ca fiind mulțimea $S(B[i, j] - P)$. Vom nota cu 0 *profilul* pentru care toate valorile p_i sunt nule și cu e_j *profilul* pentru care avem $p_j = 1$, iar restul valorilor sunt nule.



Profilul P=(0, 2, 1, 0, 2) al [4, 2]-bazei

Un *profil* poate fi privit ca un număr cuprins între 0 și $3^M - 1$, scris în baza 3.

Putem privi placa de silicon ca fiind mulțimea pătrățelelor valide (care nu sunt defecte). Vom nota această mulțime prin G. Așadar, va trebui să determinăm numărul maxim de *chip*-uri care pot fi tăiate dintr-o placă G.

Ideea care stă la baza rezolvării este folosirea programării dinamice; modalitatea în care este aplicată această metodă va fi descrisă în continuare.

Pentru fiecare *bază* $B[i, j]$ și fiecare *profil* P calculăm valoarea $A[i, j, P]$ care reprezintă numărul maxim de *chip*-uri care pot fi tăiate din mulțimea $G \cap S(B[i, j] - P)$. Se observă că avem $G \cap S(B[N - 1, M - 1] - 0) = G$, așadar soluția problemei va fi dată de valoarea $A[N - 1, M - 1, 0]$.

Secțiunea din placă ce corespunde unei mulțimi $G \cap S(B[0, j])$ este prea subțire pentru a se putea tăia vreun *chip* din ea, motiv pentru care vom avea $A[0, j, P] = 0$ pentru orice valori j și P.

În continuare vom lua în considerare toate bazele $B[i, j]$ parcurgând placa de la stânga la dreapta și de sus în jos. Pentru fiecare bază vom lua în considerare toate *profilurile* posibile.

Considerăm o *bază* $B[i, j]$ și un *profil* P; vom lua în considerare două situații: $p_j > 0$ și $p_j = 0$.

Dacă $p_j > 0$, atunci $G \cap S(B[i, j] - P) = G \cap S(B' - P')$, unde $B' = S(B[i, j])$ este *baza* luată în considerare la pasul

anterior, și $P' = P - e_j$. Datorită egalității anterioare, obținem $A[i, j, P] = A[i', j', P']$.

Dacă $p_j = 0$, există trei posibilități de a obține numărul maxim de *chip*-uri care pot fi tăiate din porțiunea $G \cap S(B - P)$ a plăcii:

- nu tăiem nici un *chip* care să aibă colțul din dreapta-jos în poziția $[i, j]$;
- tăiem un *chip* orizontal având colțul din dreapta-jos în pozițiile $[i, j]$;
- tăiem un *chip* vertical care să aibă colțul din dreapta-jos în poziția $[i, j]$.

Pentru primul caz, numărul maxim de *chip*-uri este dat de $A[i', j', P]$, unde $S(B[i', j'])$ este *baza* luată în considerare la pasul anterior.

Pentru al doilea caz, numărul de *chip*-uri este dat de $A[i'', j'', P + 2 \cdot e_j + 2 \cdot e_{j-1}]$, unde $S(B[i'', j''])$ este *baza* luată în considerare cu doi pași înainte.

Pentru al treilea caz, numărul maxim de *chip*-uri este dat de $A[i''', j''', P + e_j + e_{j-1} + e_{j-2}]$, unde $S(B[i''', j'''])$ este *baza* luată în considerare cu trei pași înainte.

Este evident că $A[i, j, P]$ va fi maximul acestor trei valori (al doilea și al treilea caz vor fi luate în considerare numai dacă *chip*-ul corespunzător poate fi tăiat în această poziție - nici unul dintre cele șase pătrățele pe care ar trebui să le ocupe *chip*-ul nu poate fi defect).

Testarea posibilităților de a tăia un *chip* vertical sau orizontal în diferite cazuri este o operație destul de simplă. Un *chip* orizontal poate fi tăiat dacă pătrățelele pe care le ocupă nu sunt defecte, avem $i \geq 1$, $j \geq 2$ și $p_j = p_{j-1} = 0$. În mod similar, un *chip* vertical va putea fi tăiat dacă pătrățelele pe care le ocupă nu sunt defecte, avem $i \geq 1$, $j \geq 2$ și $p_j = p_{j-1} = p_{j-2} = 0$. În final vom afișa valoarea $A[M - 1, N - 1, 0]$.

Se observă că nu trebuie reținute informații decât pentru ultimele trei *baze* considerate anterior și pentru *baza* curentă.

Analiza complexității

Operația de citire a datelor referitoare la o placă are ordinul de complexitate $O(M \cdot N)$ deoarece pot exista cel mult $M \cdot N$ pătrățele defecte.

Pentru fiecare element $A[i, j, P]$, timpul necesar calculării valorii sale este ***constant*** deoarece implică doar testarea validității unui număr finit de pătrățele și a unor condiții simple referitoare la i, j, p_j, p_{j-1} și p_{j-2}.

Fiecare *profil* este un vector format din M elemente care pot avea, fiecare trei valori distincte. Ca urmare, vom avea 3^M *profiluri* distincte, deci 3^M valori posibile pentru P. Vor fi $M \cdot N$ valori posibile pentru combinațiile i și j, deci tabloul tridimensional A va avea $M \cdot N \cdot 3^M$ elemente. Fiindcă fiecare element este determinat în timp constant, ordinul de complexitate al operației de determinare a numărului maxim de *chip*-uri care trebuie tăiate este $O(M \cdot N \cdot 3^M)$.

Datele de ieșire constau într-un singur număr, deci ordinul de complexitate al operației de scriere a acestora este $O(1)$.

În concluzie, pentru o anumită placă, ordinul de complexitate al algoritmului de rezolvare este $O(M \cdot N) + O(M \cdot N \cdot 3^M) + O(1) = O(M \cdot N \cdot 3^M)$.

Deoarece sunt D plăci, algoritmul va fi aplicat de D ori, dar valorile M și N nu vor fi aceleași, deci nu putem exprima simplu ordinul total de complexitate.

http://cs.science.upjs.sk/ceoi/documents/bugs_sou.zip

P060208: Batalionul Cuceritorului

Pentru început vom încerca să estimăm cât de bună este o poziție. Dacă avem doar doi soldați și aceștia se află pe aceeași treaptă, atunci la runda următoare vom avea cel mult un soldat, dar care se află cu o treaptă mai sus. Așadar, putem spune că un soldat aflat pe treapta S este echivalent (are aceeași valoare) cu doi soldați aflați pe treapta $S + 1$.

Vom considera că un soldat aflat pe treapta S are valoarea 2^{N-S}. Valoarea unei poziții va fi dată de suma valorilor tuturor soldaților. Se observă că toate pozițiile în care cuceritorul a câștigat au cel puțin valoarea 2^{N-1} deoarece există un soldat pe prima treaptă.

Vom demonstra acum că în cazul în care valoarea unei poziții este mai mică decât 2^{N-1} și comandantul joacă optim, atunci cuceritorul va pierde. Dacă valoarea este mai mică decât 2^{N-1}, cuceritorul nu a câștigat încă. La o anumită rundă, el își va împărți soldații într-un mod oarecare. Valoarea unuia dintre cele două grupuri va fi obligatoriu mai mică decât 2^{N-2}. Comandantul va alege ca acest grup să rămână, iar celălalt să plece. Când un soldat urcă o treaptă, valoarea sa se dublează; ca urmare, valoarea noii poziții va fi mai mică decât $2 \cdot 2^{N-2} = 2^{N-1}$ și numărul soldaților va scădea. Există un număr finit de soldați, deci jocul se va termina după un număr finit de runde. Valoarea pozițiilor va fi întotdeauna mai mică decât 2^{N-1}, deci cuceritorul nu va putea câștiga; până la urmă, datorită faptului că numărul soldaților scade în continuu, nu va mai rămâne nici un soldat, deci cuceritorul va pierde.

Acum vom arăta că în cazul în care valoarea unei poziții este cel puțin egală cu 2^{N-1} și cuceritorul joacă optim, atunci el va câștiga. Cuceritorul va trebui să își împartă soldații în două grupuri a căror valoare este cel puțin egală cu 2^{N-2}. Indiferent de alegerea comandantului, noua poziție (după ce soldații rămași urcă o treaptă) va avea valoarea cel puțin egală cu 2^{N-1} și numărul soldaților va scădea. Numărul soldaților este finit, deci jocul se va încheia după un număr finit de mutări. Datorită faptului că valoarea pozițiilor nu va scădea niciodată sub 2^{N-1}, înseamnă că și valoarea ultimei poziții va fi cel puțin 2^{N-1}. Aceasta înseam-

nă că cel puțin un soldat se află pe prima treaptă, deci cuceritorul a câștigat.

Mai trebuie să arătăm că există întotdeauna posibilitatea de a împărți soldații în două grupuri a căror valoare să fie cel puțin egală cu 2^{N-2}. Considerăm șirul $a_1, ..., a_M$ astfel încât $N - 2 \geq a_1 \geq a_2 \geq ... \geq a_M \geq 0$, $M \geq 2$ și $\sum_{i=1}^{M} 2^{a_i} \geq 2^{N-2}$; vom arăta că există o valoare $k \leq M$ astfel încât $\sum_{i=1}^{k} 2^{a_i} \geq 2^{N-2}$. Pentru a demonstra această afirmație vom folosi metoda inducției matematice. Pentru $M = 2$ avem $a_1 = N - 2$ și orice valoare a_2 cuprinsă între 0 și $N - 2$ sau $a_1 = a_2 = N - 3$. În primul caz alegem $k = 1$, iar în al doilea $k = 2$ pentru a obține $\sum_{i=1}^{k} 2^{a_i} \geq 2^{N-2}$. Așadar, ipoteza de inducție este demonstrată. Pentru $M > 2$, dacă $\sum_{i=1}^{M} 2^{a_i} = 2^{N-2}$ alegem $k = M$. Dacă avem $M > 2$ și $\sum_{i=1}^{M} 2^{a_i} > 2^{N-2}$, atunci, datorită faptului că 2^{N-2} și $\sum_{i=1}^{M} 2^{a_i}$ sunt multiplii ai lui 2^{a_M}, avem $\sum_{i=1}^{M} 2^{a_i} \geq 2^{N-2} + 2^{a_M}$, deci $\sum_{i=1}^{M-1} 2^{a_i} \geq 2^{N-2}$. În caz de egalitate ne oprim, iar în caz de inegalitate aplicăm aceeași metodă pentru $M - 1$. Vom ajunge fie în situația $M = 2$ (corespunzătoare ipotezei de inducție), fie ne vom opri deoarece am obținut egalitatea. Așadar afirmația este demonstrată.

Rezultă că, dacă valoarea totală a soldaților este cel puțin egală cu 2^{N-1}, vom putea alege primii soldați (cei de pe cele mai înalte trepte) a căror sumă a valorilor este exact 2^{N-2}. Cuceritorul va putea împărți soldații în două grupuri cu valori cel puțin egale cu 2^{N-2} (primul format din acești soldați, iar celălalt din cei rămași), deci până la urmă va câștiga jocul.

Pentru că pot fi până la 2000 de trepte, vom fi nevoiți să simulăm operații aritmetice cu numere mari sau să folosim diferite artificii pentru a le evita (valoarea unui soldat va putea atinge 2^{1999}, iar cea a unei poziții $1000000000 \cdot 2^{1999} \approx 2^{30} \cdot 2^{1999} = 2^{2029}$).

Dacă suntem într-o poziție câștigătoare, vom alege soldați de pe treptele 1, 2, ... până în momentul în care valoarea lor totală ajunge la 2^{N-2}. Dacă suntem într-o situație necâștigătoare vom alege un grup vid de soldați și vom pierde imediat.

Pentru a mări viteza algoritmului putem precalcula, la începutul jocului, anumite valori. Fie s_i numărul soldaților de pe cea dea i-a treaptă. Vom precalcula sumele $T_k = \sum_{i=1}^{k} s_i \cdot 2^{N-i}$. Numerotăm soldații de 1 la M în ordinea în care se află pe trepte la începutul jocului. Dacă folosim algoritmul prezentat, vom putea împărți soldații astfel încât fiecare grup să fie format din soldați cu numere de ordine consecutive. Vom păstra numărul curent de soldați de pe fiecare treaptă, precum și două referințe la primul și ultimul soldat din grupul curent. Se observă că nu va trebui să mutăm soldații de pe o treaptă pe alta, deoarece știm că după r runde, un soldat se va afla cu r trepte mai sus față de poziția lui inițială.

Folosind valorile precalculate, vom găsi foarte repede valoarea unui grup de soldați cu numere de ordine consecutive. Pentru a determina poziția în care va fi împărțit grupul se poate folosi căutarea binară.

Analiza complexității

Datorită faptului că numărul de cifre al numerelor mari este limitat, putem considera că ordinul de complexitate al operațiilor cu astfel de numere este $O(1)$.

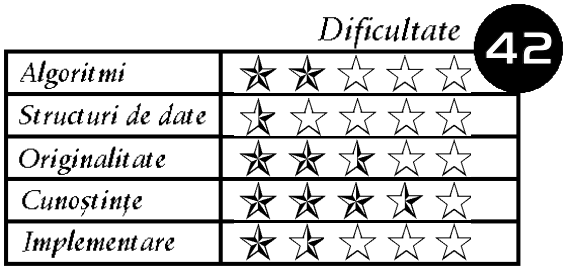
Preluarea datelor de intrare constă în completarea de către bibliotecă a unui tablou cu N elemente, deci se realizează în timp ***liniar***.

Numărul de runde este cel mult $N - 1$, deoarece soldații de pe ultima treaptă vor ajunge pe prima după $N - 1$ runde. La fiecare rundă vom realiza o căutare binară care are ordinul de complexitate $O(\log N)$ și vom completa un tablou cu N elemente, operație realizabilă în timp ***liniar***. Apelul funcției `step` implică efectuarea unor operații în timp ***liniar***; funcția apelându-se o dată pentru fiecare rundă. Așadar, operațiile realizate pentru fiecare rundă au ordinul de complexitate $O(\log N) + O(N) + O(N) = O(N)$. Pentru toate rundele ordinul de complexitate devine $O(N) \cdot O(N) = O(N^2)$.

Precalcularea valorilor T_k se realizează în timp ***liniar*** deoarece fiecare valoare T_k se obține folosind valoarea T_{k-1} și efectuând o adunare și o înmulțire: $T_k = T_{k-1} + s_k \cdot 2^{N-k}$.

Datele de ieșire sunt create de funcția `step`, deci scrierea lor nu implică operații suplimentare.

Așadar, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N) + O(N^2) + O(N) = O(N^2)$.



http://cs.science.upjs.sk/ceoi/documents/conquer_sou.zip

P060209: Un gărduleț decorativ

Pentru rezolvarea acestei probleme vom aplica metoda programării dinamice.

Fie $U_{N,i}$ numărul permutărilor care descriu un gard în care primul element al permutării este i, iar al doilea este mai mare decât primul. O astfel de permutare va fi numită ***crescătoare***. Analog, fie $D_{N,i}$ numărul permutărilor care descriu un gard în care primul element al permutării este i, iar al doilea este mai mic decât primul. O astfel de permutare este numită ***descrescătoare***.

Deoarece N este cuprins între 1 și 20, putem precalcula valorile U și D pentru toate combinațiile de valori i și N.

Formulele de calcul sunt următoarele:

- $D_{N,1} = 0$
- $D_{N,i+1} = \sum_{k=1}^{i} U_{N-1,k} = D_{N,i} + U_{N-1,i}$
- $U_{N,i} = D_{N,N+1-i}$

Prima formulă este evidentă pentru că primul element este 1, iar al doilea nu poate fi mai mic decât acesta.

Cea de-a doua formulă se obține astfel: se consideră toate permutările descrescătoare de lungime N care încep cu $i + 1$. Al doilea element k al unei astfel de permutări va fi cuprins între 1 și i. Dacă eliminăm din permutare primul element ($i + 1$) și descreștem cu 1 toate elementele mai mari decât i, obținem o permutare crescătoare care începe cu k. Așadar există o corespondență biunivocă între mulțimea permutărilor crescătoare de lungime N care încep cu i și mulțimea permutărilor crescătoare care încep cu o valoare mai mică decât i. Ca urmare, cele două mulțimi au același număr de elemente.

Așadar avem:

$$D_{N,i+1} = \sum_{k=1}^{i} U_{N-1,k} = \sum_{k=1}^{i-1} U_{N-1,i} + U_{N-1,i} = D_{N,i} + U_{N-1,i}.$$

Cea de-a treia relație este obținută foarte ușor dacă punem în corespondență o permutare $(p_1, \ldots, p_N)$ cu permutarea $(N + 1 - p_1, \ldots, N + 1 - p_N)$. Există o corespondență biunivocă între mulțimea permutărilor crescătoare care încep cu i și cea a permutărilor descrescătoare care încep cu $N + 1 - i$. Ca urmare, cele două mulțimi au același număr de elemente.

Vom folosi valorile U și D pentru a determina gărdulețul cu numărul de catalog C.

Este evident că există $U_{N,i} + D_{N,i}$ permutări care încep cu i. Primul element al permutării se obține foarte simplu, fiind cel mai mic număr pentru care $\sum_{i=1}^{a_1} (D_{N,i} + U_{N,i}) \geq C$.

După determinarea acestui element vom calcula valoarea a_2. Vom scădea cu 1 valorile tuturor elementelor din permutarea pe care o căutăm care sunt mai mari decât a_1. Restul permutării va descrie un gărdulеț cu N - 1 elemente. Știm că dacă primul element al noii permutări este mai mic decât a_1, va fi descrescătoare, iar dacă este mai mare decât a_1, va fi crescătoare. De asemenea, știm că trebuie să căutăm cea de-a C_1-a astfel de permutare unde:

$$C_1 = C - \sum_{i=1}^{a_1-1} (D_{N,i} + U_{N,i}).$$

Pentru x cuprins între 1 și a_1 - 1 există $U_{N-1,x}$ astfel de permutări care încep cu x, iar pentru $x > a_1$ există $D_{N-1,x}$ permutări care încep cu x. Vom determina următorul element y al permutării într-un mod similar determinării primului element. Dacă obținem $y < a_1$ vom alege $a_2 = y$, iar dacă avem $y > a_1$, vom alege $a_2 = y + 1$.

Acum cunoaștem primele două elemente ale permutării pe care o căutăm și știm dacă aceasta este crescătoare sau descrescătoare. De asemenea cunoaștem numărul de ordine C_2 al permutării de pe ultimele N - 2 poziții. Următoarele elemente ale permutării date vor fi determinate folosind aceeași metodă.

Presupunem că știm care sunt primele k ($k \geq 2$) elemente ale permutării pe care o căutăm și numărul de ordine C_k al permutării de pe ultimele N - k poziții. Ultimele două elemente cunoscute arată dacă permutarea de lungime N - k este crescătoare sau descrescătoare. Vom determina primul element al acestei permutări luând în considerare toate permutările valide de N - k elemente care încep cu x până depășim valoare C_k. Calculăm valoarea C_{k+1} prin scăderea din C_k a numărului permutărilor care încep cu o valoare mai mică decât elementul a_k determinat.

Pentru simplitate, vom lucra la fiecare pas k cu o permutare de N - k elemente cuprinse între 0 și N - k - 1. Un element i dintr-o astfel de permutare va corespunde în permutarea finală celui de-al i-lea element care nu a fost folosit anterior (pe pozițiile anterioare ale permutării căutate).

Analiza complexității

Datele de intrare constau într-un singur număr, deci ordinul de complexitate al operației de citire a acestora este $O(1)$.

Precalcularea valorilor U și D implică un timp ***constant*** pentru fiecare valoare $U_{N,i}$ și $D_{N,i}$. În total sunt N^2 valori $U_{N,i}$ și N^2 valori $D_{N,i}$, deci ordinul de complexitate al operației de determinare a acestora este $O(N^2) + O(N^2) = O(N^2)$.

Determinarea unui element al permutării căutate se realizează în timp ***liniar*** deoarece implică luarea în considerare a cel mult N valori pentru acesta. În total sunt determinate N elemente, deci determinarea permutării căutate are ordinul de complexitate $O(N) \cdot O(N) = O(N^2)$.

Operația de afișare a datelor de ieșire constă în scrierea celor N elemente ale permutării, deci are ordinul de complexitate $O(N)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme pentru un gărdulеț este $O(1) + O(N^2) + O(N^2) + O(N) = O(N^2)$.

Deoarece sunt k gărdulețe, algoritmul va fi aplicat de k ori, dar pentru valori diferite ale lui N, deci ordinul total de complexitate nu poate fi exprimat printr-o expresie simplă.

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

http://cs.science.upjs.sk/ceoi/documents/fence_sou.zip

P060210: Autostrada și cei șapte pitici

Pentru început vom observa faptul că o autostradă poate fi construită dacă și numai dacă nu intersectează înfășurătoarea convexă a punctelor care reprezintă casele piticilor. Dacă este intersectată înfășurătoarea convexă, atunci ea va fi

intersectată în două dintre muchii, deoarece autostrăzile nu pot trece prin punctele care reprezintă casele. Dacă autostrada intersectează o muchie (latură) a înfășurătorii convexe, atunci cele două case din extremitățile laturii se vor afla de o parte și de alta a autostrăzii, deci aceasta nu poate fi construită.

După citirea coordonatelor claselor, se construiește înfășurătoarea convexă a acestora, folosind unul dintre algoritmii clasici de realizare a acestei operații.

În continuare vom citi datele referitoare la o linie și va trebui să decidem dacă intersectează un poligon convex. Pentru a găsi un algoritm eficient care să realizeze această operație vom presupune că linia are o direcție și, potrivit acestei direcții, există o cea mai din stânga și o cea mai din dreapta casă. Este evident că linia va intersecta poligonul dacă și numai dacă cele două case (puncte) se află în semiplane diferite determinate pe linie.

În continuare vom prezenta un algoritm eficient care determină cel mai din stânga și cel mai din dreapta punct corespunzător unei direcții.

Să presupunem că știm care este cel mai din stânga și cel mai din dreapta punct pentru o anumită direcție. Dacă schimbăm direcția cu un unghi foarte mic, punctele vor rămâne aceleași.

Totuși, după mai multe schimbări de direcție, vor apărea modificări. Vom încerca să determinăm când are loc o astfel de modificare. Considerăm două linii care respectă direcția considerată și care trec prin cel mai din dreapta și cel mai din stânga punct. Când rotim direcția, cele două puncte nu se vor modifica decât în momentul în care una dintre laturile înfășurătorii convexe se va afla pe una dintre cele două linii. În acest moment celălalt punct care determină latura respectivă va deveni cel mai din dreapta sau cel mai din stânga punct. Așadar, cele două puncte căutate se modifică doar dacă direcția este aceeași cu cea a uneia dintre laturile înfășurătorii convexe.

Dacă înfășurătoarea convexă are M laturi, atunci mulțimea direcțiilor va fi împărțită în M intervale, fiecărui interval corespunzându-i o pereche de puncte (un cel mai din dreapta și un cel mai din stânga punct). Pentru fiecare dreaptă vom determina direcția folosind o căutare binară și vom verifica dacă punctele corespunzătoare intervalului direcției sunt sau nu în același semiplan determinat de dreaptă.

Analiza complexității

Datele de intrare constau în N puncte și M linii, deci ordinul de complexitate al operației de citire a acestora este $O(M) + O(N) = \boldsymbol{O(M + N)}$.

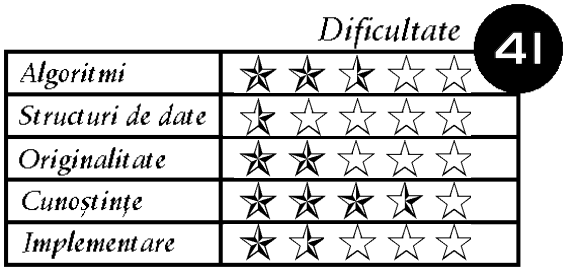
Determinarea înfășurătorii convexe folosind un algoritm eficient are ordinul de complexitate $\boldsymbol{O(N \cdot \log N)}$.

Înfășurătoarea convexă va conține cel mult N elemente, deci vom determina $O(N)$ intervale ale direcțiilor. Determinarea unui interval se realizează în timp ***constant***, deci ordinul de complexitate al operației de determinare a tuturor intervalelor este $\boldsymbol{O(N)}$.

Căutarea binară necesită un timp ***logaritmic***, și, datorită faptului că realizăm M căutări, verificarea posibilităților de construire a tuturor autostrăzilor are ordinul de complexitate $\boldsymbol{O(M \cdot \log N)}$.

Pentru fiecare autostradă vom scrie un singur mesaj, deci ordinul de complexitate al operației de scriere a datelor de ieșire este $\boldsymbol{O(M)}$.

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



http://cs.science.upjs.sk/ceoi/documents/dwarfs-sou.zip

P060211: Gărzile regale

Pentru simplitate, vom considera castelul ca fiind înconjurat de ziduri. Vom numi ***segment de linie*** o porțiune continuă a unei linii care nu conține nici un zid și nu poate fi extins. Adică este delimitată la stânga și la dreapta de ziduri.

Similar, vom numi ***segment de coloană*** o porțiune continuă a unei coloane care nu conține nici un zid și care nu poate fi extinsă. Este evident că fiecare segment de linie sau coloană poate conține cel mult un gardian.

Construim un graf bipartit în care vârfurile uneia dintre partiții corespund *segmentelor de linie* iar vârfurile celeilalte partiții corespund *segmentelor de coloană*. Între două noduri din partiții diferite va exista o muchie dacă și numai dacă segmentele respective se intersectează și punctul de intersecție nu conține o capcană.

Cu alte cuvinte, o muchie corespunde pătrățelelor în care se pot afla gardieni.

Considerăm acum o amplasare validă a gardienilor. Muchiile grafului bipartit care corespund unor poziții în care se află gardieni formează un cuplaj al grafului (fiecare *segment de linie* sau *coloană* conține cel mult un gardian, deci fiecare vârf poate fi incident cu cel mult o muchie aleasă). Pe de altă parte, fiecare cuplaj al grafului bipartit corespunde unei amplasări valide a gardienilor.

Așadar, pentru a amplasa cât mai mulți gardieni, este suficient să determinăm un cuplaj maxim în graful bipartit construit. Pentru determinarea cuplajului poate fi folosit unul dintre algoritmii clasici destinați acestui scop (vezi ***CLR***[1] p. 515).

După determinarea cuplajului vom scrie în fișierul de ieșire numărul muchiilor care îl formează (adică numărul maxim al gardienilor), precum și pozițiile corespunzătoare acestor muchii.

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

Analiza complexității

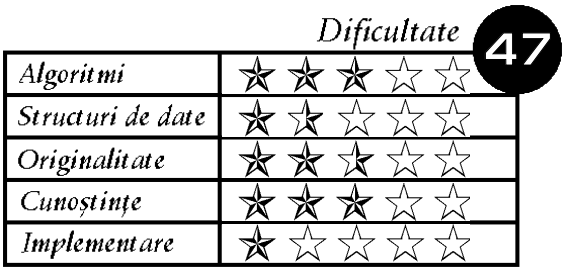
Castelul este codificat printr-o matrice cu M linii și N coloane, deci operația de citire a datelor de intrare are ordinul de complexitate $O(M \cdot N)$.

Construirea grafului bipartit implică traversarea matricei, deci ordinul de complexitate al acestei operații este $O(M \cdot N)$.

Algoritmul de determinare a cuplajului maxim are ordinul de complexitate $O(|V| \cdot (|V| \cdot |E|))$, unde $|V|$ este numărul de vârfuri, iar $|E|$ este numărul de muchii. Graful construit are $O(M \cdot N)$ vârfuri și $O(M \cdot N)$ muchii, deci în acest caz ordinul de complexitate al algoritmului va fi $O(M^2 \cdot N^2)$.

Cuplajul va conține $O(M \cdot N)$ muchii, deci acesta va fi ordinul de complexitate al operației de scriere al datelor de ieșire.

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



http://cs.science.upjs.sk/ceoi/documents/guards-sou.zip

P060212: Petrecere aniversară

Vom considera că numele prietenilor lui *John* sunt variabile logice. Dacă o variabilă este adevărată, înseamnă că *John* ar trebui să invite persoane corespunzătoare, iar dacă este falsă, persoana nu va fi invitată. De asemenea, cererile primite de *John* pot fi privite ca expresii logice. Sarcina noastră este de a atribui valori variabilelor astfel încât prin evaluarea tuturor formulelor să se obțină valoarea *adevărat*.

De fapt, aceasta este una dintre cele mai cunoscute probleme de algoritmică, ea fiind cunoscută sub numele de problema ***SAT***, prescurtarea de la satisfiabilitate (vezi ***CLR*** p. 805). Pe cazul general, se știe că această problemă este *NP*-completă, deci nu se cunoaște nici un algoritm polinomial de rezolvare a acesteia.

Câteva dintre cele zece fișiere de intrare sunt destul de mari, deci fișierele de ieșire corecte nu pot fi determinate dacă se folosește un algoritm exponențial. Cu toate acestea, cele zece fișiere folosite pentru testare sunt cunoscute și au anumite particularități care permit determinarea fișierelor de ieșire corecte. Pot fi folosite orice mijloace pentru determinarea fișierelor de ieșire; totuși, pentru unele dintre ele, este necesară scrierea unor programe.

În cele ce urmează, literele A, B, … vor fi folosite pentru a nota expresii logice arbitrare, nu doar variabile. Vom folosi semnul minus ('-') pentru a nota negația logică a unei expresii; așadar, vom putea nota anumite expresii logice care nu sunt permise de enunțul problemei (de exemplu $-(A|B)$). Se poate observa foarte ușor că semnul `'&'` reprezintă operatorul de conjuncție (*ȘI logic*), semnul `'|'` reprezintă operatorul de disjuncție (*SAU logic*), iar semnul `'=>'` reprezintă implicația logică.

Se observă că expresiile $(A \Rightarrow B)$, $(-A \mid B)$ și $(-B \Rightarrow -A)$ sunt echivalente. Ca urmare, expresia $(A \Rightarrow -A)$ este adevărată dacă și numai dacă expresia A este falsă. De asemenea, formula $(A \Rightarrow (B \Rightarrow C))$ este echivalentă cu $((A \,\&\, B) \Rightarrow C)$. Vom avea nevoie și de legile lui ***Morgan***:

- $-(A_1 \,\&\, \ldots \,\&\, A_m)$ este echivalentă cu $(-A_1 \mid \ldots \mid -A_m)$;
- $-(A_1 \mid \ldots \mid A_m)$ este echivalentă cu $(-A_1 \,\&\, \ldots \,\&\, -A_m)$.

Din afirmațiile anterioare rezultă că următoarele expresii sunt echivalente:

- $(A_1 \Rightarrow (A_2 \Rightarrow (\ldots (A_m \Rightarrow (B_1 \mid \ldots \mid B_n)) \ldots)))$;
- $((A_1 \,\&\, \ldots \,\&\, A_m) \Rightarrow (B_1 \mid \ldots \mid B_n))$;
- $((-(A_1 \,\&\, \ldots \,\&\, A_m)) \mid (B_1 \mid \ldots \mid B_n))$;
- $(-A_1 \mid \ldots \mid -A_m \mid B_1 \mid \ldots \mid B_n)$.

Vom numi variabilele și negațiile acestora folosind termenul comun *literal*. Spunem că o expresie este în ***formă normală conjunctivă*** (***FNC***), dacă este o conjuncție între mai multe formule logice și fiecare dintre aceste formule logice este o disjuncție a unor literali. De exemplu, formula $((A \mid B) \,\&\, (B \mid -C \mid C \mid C) \,\&\, -D \,\&\, (A \mid C))$ este în *FNC*. Se poate demonstra matematic faptul că orice expresie logică este echivalentă cu o expresie în *FNC*. Aceste observații ne vor permite să rescriem anumite fișiere de intrare pentru ca expresiile corespunzătoare să fie în *FNC*.

Se observă că denumirile variabilelor din fișierele 2 - 10 sunt *b*, *c*, *d*, …, *i*, *j*, *ba*, *bb* etc. Dacă înlocuim a *i*-a literă a alfabetului cu a *i*-a cifră (începând numerotarea cu cifra 0) obținem variabilele 1, 2, 3, …, 8, 9, 10, 11 etc. În plus, negația este reprezentată de semnul minus, deci negația unei variabile va fi dată de numărul negativ obținut prin adăugarea semnului '-' în fața numărului care reprezintă variabila.

PARTY1.IN

Primul fișier nu respectă convenția de notare a variabilelor din celelalte 9 și cea mai simplă modalitate de obținere a fișierului de ieșire corect este rezolvarea manuală a acestui caz. Sunt doar șapte expresii și cinci variabile, deci nu va fi prea dificilă găsirea soluției.

PARTY2.IN - PARTY4.IN

Aproape toate expresiile din aceste trei fișiere de intrare au forma $(lit_1 \mid lit_2 \mid \ldots \mid lit_k)$, unde lit_x este un literal. Acest tip de fișier este destul de convenabil deoarece înseamnă că cel puțin un literal de pe fiecare linie trebuie să fie *adevărat*. Expresiile care nu respectă această regulă vor fi pur și simplu rescrise folosind formulele descrise anterior. Observăm că nu mai avem nevoie de caracterele `'('`, `')'` sau `'|'`, deci le putem elimina. Fișierul de intrare va fi privit ca fiind o conjuncție de linii și, după rescrierea liniilor care nu respectă regula, expresia va fi în *FNC*.

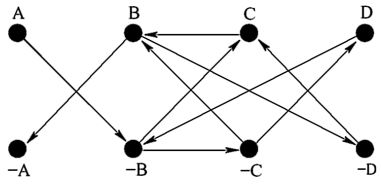
Aceste fișiere de intrare sunt relativ mici, deci pot fi utilizați algoritmi *backtracking* pentru a găsi soluțiile. Folosind câteva optimizări simple, soluțiile vor fi determinate foarte repede.

PARTY5.IN - PARTY9.IN

Aceste fișiere de intrare sunt prea mari pentru ca un algoritm exponențial să ruleze într-un timp rezonabil. Totuși, studiind cu atenție fișierele, observăm că fiecare expresie (cu excepția câtorva din fișierul `PARTY9.IN`) conține doar doi literari care formează o implicație sau o disjuncție.

Putem rescrie fiecare expresie astfel încât să devină o implicație. De exemplu (A | B) devine ($-A$ => B). În continuare construim un graf orientat ale cărui vârfuri vor fi literalii care apar în expresie (variabilele și negațiile lor), iar arcele vor reprezenta implicațiile. Semnificația unui arc va fi următoarea: dacă vârful din care pornește are valoarea *adevărat*, atunci și vârful la care ajunge trebuie să aibă valoarea *adevărat*.

Din expresia ($-A$ => B) deducem că există arc de la vârful corespunzător literalului -A la cel corespunzător literalului B. Expresia este echivalentă cu ($-B$ => A) deci vom avea un arc și de la $-B$ la A. Așadar, fiecare expresie implică apariția a două arce în graf. Se observă că graful este simetric (dacă înlocuim fiecare literal cu negația sa și inversăm sensul arcelor obținem același graf).



Graful pentru formulele: (A => -B), (B | -C), (B | C), (D => -B) și (C | D)

În continuare vom eticheta fiecare vârf cu *adevărat* sau *fals*, astfel încât pentru o variabilă A, exact unul dintre vârfurile A și $-A$ este etichetat cu *adevărat*. De asemenea, dacă un vârf V este etichetat cu *adevărat*, atunci toate vârfurile U pentru care există un drum de la V la U trebuie etichetate cu *adevărat*.

Evident, dacă pentru o anumită variabilă A, vârfurile A și $-A$ se află în aceeași componentă tare-conexă a grafului, nu există nici o etichetare care respectă cele două condiții amintite anterior (unul dintre acești doi literali trebuie să aibă valoarea *adevărat* și, în cazul în care fac parte din aceeași componentă tare-conexă, există un drum între nodurile celor doi literali, deci și celălalt trebuie să aibă valoarea *adevărat*; am ajuns astfel la o contradicție, deci afirmația inițială este falsă; așadar, A și $-A$ nu pot fi în aceeași componentă tare-conexă). În cele ce urmează vom arăta că există soluție în toate celelalte cazuri.

Vom alege o componentă tare-conexă pentru care nu există nici un arc care pornește dintr-un vârf din afara componentei și ajunge într-un vârf din interiorul ei. Vom eticheta toate vârfurile din această componentă C cu *fals*. Datorită simetriei grafului vârfurile care corespund negațiilor literalilor vor forma o componentă tare-conexă C' pentru care nu există nici un arc care pornește dintr-un vârf din interiorul ei și ajunge într-un vârf din afara ei. Vom eticheta toate vârfurile din C' cu *adevărat*. Este evident că etichetarea vârfurilor din C și C' este corectă și nu impune nici o restricție asupra etichetării celorlalte vârfuri ale grafului.

Așadar putem elimina din graf componentele C și C' și aplica același algoritm pentru restul grafului.

Numărul de noduri ale grafului este dublul numărului de variabile; numărul de arce este dublul numărului de expresii; așadar dimensiunea grafului este liniară în raport cu dimensiunea fișierului de intrare. Există un bine cunoscut algoritm liniar de determinare a componentelor tare-conexe ale unui graf. Vom construi apoi un graf al componentelor: fiecare componentă este reprezentată de un nod și există un arc între două noduri i și j, dacă în graful inițial există cel puțin un arc de la un nod din componenta i la un nod din componenta j. Se aplică o sortare topologică asupra grafului componentelor pentru a le alege pe cele care vor fi etichetate cu *fals* (și implicit pe cele ale căror vârfuri vor fi etichetate cu *adevărat*). Întregul algoritm se execută în timp **liniar**.

PARTY10.IN

Acesta este cel mai mare și pare cel mai complicat dintre toate. La o privire mai atentă descoperim că, din contră, soluția poate fi obținută destul de repede. Dacă cercetăm fișierul, observăm că ultimele linii au forma (A => $-A$) sau ($-B$ => B). Din expresiile de primul tip deducem că valoarea variabilei A este *fals*, iar din cele de al doilea tip că valoarea variabilei B este *adevărat*.

Folosind această observație obținem valorile a 997 dintre cele 1000 de variabile. Cele trei valori rămase pot fi determinate folosind primele trei expresii din fișier.

Din enunțul problemei știm că există întotdeauna o soluție. Soluția determinată astfel este singura care verifică expresiile de pe liniile luate în considerare, așadar este singura soluție posibilă. Nu trebuie să verificăm și expresiile de pe celelalte linii deoarece știm că ele trebuie să fie *adevărate* (dacă, totuși, decidem să le verificăm, vom observa că, într-adevăr, toate acestea au valoarea *adevărat* dacă valorile variabilelor sunt determinate folosind primele trei expresii și ultimele 997).

	Dificultate 65
Algoritmi	★★★☆☆
Structuri de date	★★☆☆☆
Originalitate	★★★☆☆
Cunoștințe	★★★★★
Implementare	★★★½☆

http://cs.science.upjs.sk/ceoi/documents/party-sou.zip