Olimpiada Națională de Informatică, Etapa Națională
Proba 1 de baraj
Descrierea Soluțiilor

Comisia Științifică

26 aprilie 2024

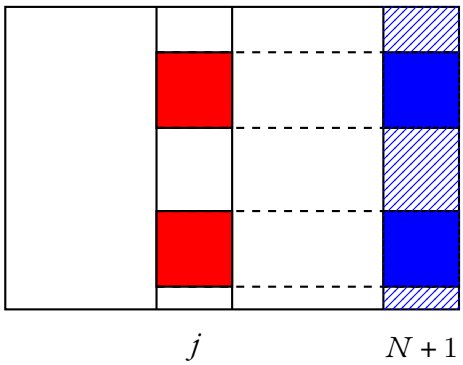
Problema 1: Graba

Vom codifica o instrucțiune de forma `if(`x_i `== y) return z` ca tripletul (i, y, z). Setul $\{1, \ldots, n\}$ îl vom nota cu $[n]$.

Observație 1. *Dacă există soluție, atunci există un $j \in [N]$, y, z astfel încât, pentru oricare $i \in [M]$ cu $A_{ij} = y$ avem că $A_{i,N+1} = z$.*

Demonstrație. Să considerăm orice soluție validă la problemă. Aceasta trebuie să înceapă cu un triplet (j, y, z) oarecare (exceptând cazul trivial când matricea din input nu are niciun element). Observăm că pentru ca soluția să fie corectă, oricare input care are $A_{ij} = y$ trebuie să aibă $A_{i,N+1} = z$ — căci altfel, soluția ar afișa z pentru inputul i, în loc de $A_{i,N+1} \neq z$. □

Grafic, valorile din matricea A arată așa; cu culoarea roșie este marcat elementul y, și cu culoarea albastră este marcat elementul z. Zona hașurată cu albastru conține elemente ce pot sau nu fi z.



Observație 2. *Dacă există un $j \in [N]$, y, z astfel încât, pentru oricare $i \in [M]$ cu $A_{ij} = y$ avem că $A_{i,N+1} = z$, atunci o soluție validă este următoarea: prima operație este (j, y, z); restul operațiilor sunt acelea ce ar fi afișate pentru matricea A', care conține* doar rândurile i din A cu $A_{ij} \neq y$.

Demonstrație. Oricare dintre inputurile i cu $A_{ij} = y$ sunt rezolvate corect de prima operație; restul sunt rezolvate corect de restul operațiilor (căci am presupus ca operațiile rămase sunt o soluție validă pentru submatricea A'). □

Așadar, rezultă că următoarea este o soluție corectă (dar ineficientă) pentru problemă:

1. Cât timp matricea A nu este goală, repetăm următorii pasi.
2. Căutăm un $j \in [N]$, y, z astfel încât, pentru oricare $i \in M$ cu $A_{ij} = y$ avem că $A_{i,N+1} = z$.
3. Dacă acest element nu există, atunci conform Observației 2, nu există soluție, deci putem afișa -1.
4. Dacă acest element există, aplicăm operația (j, y, z), eliminăm toate liniile i cu $A_{ij} = y$, și ne întoarcem la primul pas.

O soluție $O(NM)$ implementată naiv cu hash-uri (folosind de exemplu `unordered_map` din STL) ar trebui să obțină 81 sau 100 de puncte în funcție de detaliile implementării. Aceste structuri de date au complexitate teoretică foarte bună, dar în practică se comportă mai slab decât structuri mai primitive care sunt stocate într-un mod mai compact în memorie.

Vom descrie acum o metodă eficientă de a implementa acest algoritm. Vom crea un arbore înrădăcinat cu următoarea structură:

1. Arborele are o rădăcină.
2. Copiii rădăcinii reprezintă coloanele matricii.
3. Să considerăm un nod ce reprezintă coloana j. Acest nod are un copil pentru fiecare valoare $x = A_{ij}$.
4. Să considerăm un nod ce reprezintă valoarea x, copilul nodului ce reprezintă coloana j. Acesta are un copil pentru fiecare valoare $y = A_{i,N+1}$, pentru acele valori i unde $A_{ij} = x$.
5. Pentru un nod ce reprezintă valoarea y, copilul nodului ce reprezintă valoarea x, copilul nodului ce reprezintă coloana j, vom avea un copil pentru fiecare i unde $A_{ij} = x$, $A_{i,N+1} = y$.

Fie t oricare nod de pe nivelul 3 cu exact un singur copil. Lanțul de la rădăcină la (unicul său) copil reprezintă tripletul (j, y, z) pe care îl vrem; mai mult frunzele din subarborele lui reprezintă acele linii din matrice ce vrem să le eliminăm. Așadar, menținem acest arbore, eliminând frunzele ce corespund liniilor eliminate. Atunci când eliminăm o frunză, eliminăm și părinții săi recursiv dacă este nevoie; putem observa atunci când un nod de pe nivelul 3 are exact un fiu, și să îl adăugăm la acel moment la o coadă, marcând-ul pentru eliminare ulterioară. Complexitatea soluției este $O(NM)$.

Implementare alternativă

Pe fiecare coloană j și pentru fiecare valoare distinctă de pe acea coloană, creăm o listă dublu înlănțuită. Concret, fiecare poziție $A_{i,j}$ conține indicii liniilor anterioară i' și următoare i'' pentru care $A_{i,j} = A_{i',j} = A_{i'',j}$. Acum, dacă există o listă de valori x pe coloana j pentru care toate valorile corespunzătoare y de pe coloana $N + 1$ sînt egale, atunci putem emite condiția

```
if (x_j == x) return y;
```

Apoi putem șterge din evidență toate liniile din lista găsită. Numim o astfel de listă **bună**. Vom menține o coadă de liste bune. Inițializăm coada cu listele care sînt bune de la început. Pe măsură ce eliminăm linii, din liste vor începe să dispară elemente. Unele liste, care inițial corespundeau la valori y multiple, pot deveni bune prin ștergerea elementelor, caz în care le adăugăm la coadă. Dacă coada devine goală înainte să eliminăm toate liniile, atunci testul nu are soluție.

Pentru a menține eficient informația dacă o listă este bună, stocăm numărul de **discrepanțe** între valorile y corespunzătoare. Mai exact, pentru fiecare listă stocăm numărul de perechi de poziții consecutive care corespund la valori y diferite. Numărul de discrepanțe poate scădea cu 1 la dispariția unui element din listă și putem întreține valoarea curentă în $O(1)$. Cînd numărul de discrepanțe ajunge la 0, înseamnă că lista a devenit bună.

Restul implementării cere o gestiune corectă a pointerilor. Bunăoară, este posibil ca o listă să se afle în coada de liste bune, dar să devină vidă înainte să-i vină rîndul (prin eliminarea tuturor liniilor pe care le acoperea).

Problema 2: Perm

Soluția 1 (2 puncte)

Numărul minim de interschimbări pentru un interval $[x, y]$ în acest caz este egal cu $y - x$, deoarece mereu putem obține ciclul $x + 1, x + 2, \ldots, y, x$ în $y - x$ interschimbări.

Soluția 2 (11 puncte)

Numărul minim de interschimbări pentru un interval $[x, y]$ de care avem nevoie este suma dintre:

- numărul de cicli complet incluși în intervalul $[x, y] - 1$, sau 0 dacă nu avem niciun ciclu
- numărul de elemente strict mai mici decât x din intervalul $[x, y]$
- numărul de elemente strict mai mari decât y din intervalul $[x, y]$.

De ce? Presupunând că am avea doi cicli, am avea nevoie doar de o singură interschimbare pentru a obține un ciclu. Astfel pentru a uni K cicli între ei, am avea nevoie de $K - 1$ interschimbări. Atunci când avem un element mai mic decât x, respectiv mai mare decât y, e clar că avem nevoie de o interschimbare ca să îl scoatem din interval. Observăm că mereu putem alege aceste interschimbări astfel încât cu elementele din afară să obținem un singur ciclu.

În acest subtask este suficient să găsim pentru intervalul specificat numărul de cicli și elemente care nu sunt în intervalul $[x, y]$ în $O(N)$, aceasta fiind și complexitatea dorită.

Soluția 3 (9 puncte)

Generăm toate combinațiile de interschimbări posibile.

Complexitate $O(Q \cdot 2^{N \cdot (N-1)/2})$

Soluția 4 (18 puncte)

Toate query-urile fiind pe prefixe putem să le sortăm crescător după y, și să ținem o `Pădure de mulțimi disjuncte` pentru a contoriza numărul de ciclii. Numărul de elemente mai mari decât y sunt ușor de aflat.

Complexitate $O(N \cdot \log N)$

Soluția 5 (80 puncte)

Putem folosi algoritmul lui Mo. Împărțim fictiv permutarea în blocuri de lățime $\sqrt{N}$. Ordonăm interogările după blocul lui x, iar ca departajare după y (pentru un bonus de viteză, putem ordona y-ii crescător în blocurile impare și descrescător în blocurile pare). Acum menținem informații despre intervalul curent $[x, y]$, informații pe care le putem actualiza în $O(1)$ când extindem sau contractăm intervalul curent cu o poziție. Atunci efortul total al actualizărilor este $O((Q + N)\sqrt{N})$.

Informațiile necesare sînt (1) numărul de cicluri conținute în intervalul curent și (2) numărul de lanțuri conținute în intervalul curent. Pentru (1), precalculăm pentru fiecare ciclu extremele sale stînga s și dreapta d și, într-un vector global v asignăm $v[s] = d$ și $v[d] = s$. Cu vectorul v putem determina dacă la extinderea sau la contractarea intervalului curent înglobăm sau pierdem un ciclu.

Pentru (2), exprimăm numărul de lanțuri ca numărul de poziții i pentru care $p[i]$ este în afara intervalului curent, deoarece fiecare astfel de poziție indică un capăt de lanț. Fie z poziția care intră sau iese din interval. Trebuie să vedem dacă $p[z]$ este sau nu în interval și dacă poziția t pentru care $p[t] = z$ este sau nu în interval. Iau naștere cîteva **if**-uri simple.

Complexitate $O((N + Q) \cdot \sqrt{N})$

Soluția 6 (100 puncte)

Ordonăm query-urile crescător după x.

Pentru a număra ciclii incluși într-un interval $[x, y]$, putem privi un ciclu care are cel mai din stânga element pe poziția a și cel mai din dreapta element pe poziția b ca pe un interval $[a, b]$. Astfel reducem problema la: avem maxim N intervale și trebuie să răspundem la Q query-uri de forma $[x, y]$ - câte intervale sunt incluse în $[x, y]$. Pentru a face asta, spunem că un interval $[a, b]$ este activ cât timp x-ul dintr-un query este mai mic sau egal decât a, deci pentru un query trebuie doar să numărăm câte intervale active au capătul dreapta mai mic sau egal decât y, ceea ce putem face ușor cu un `Arbore de intervale` sau un `Arbore indexat binar`. Începem cu toate intervalele active, și, evident, ne folosim de faptul că query-urile sunt sortate crescător după x pentru a dezactiva intervalele o singură dată.

Pentru a număra elementele care nu au valori incluse în intervalul $[x, y]$, putem trata cazul în care numărăm elementele mai mici decât x, celălalt fiind echivalent. Similar cu soluția pentru ciclii, spunem că o valoare este activă dacă este mai mică decât x. Fiecare query $[x, y]$ se reduce la a afla câte valori active sunt în acest interval, ceea ce putem afla cu un `Arbore de intervale` sau un `Arbore indexat binar`. Începem cu toate valorile inactive, iar cum x poate doar să crească, activăm mereu valorile mai mici decât x care nu au fost încă active.

Complexitate $O((Q + N) \cdot \log N)$

Problema 3: Redpanda

În problema aceasta ni se dă un arbore A cu n noduri, cu rădăcină, și ni se cere să eliminăm apoi să adăugăm alternativ muchii astfel încât la final să ne dea un arbore cu adâncimea maxim k. (Adâncimea este definită ca lungimea lanțului cel mai lung de la rădăcină la o frunză — aceasta este mai mică cu 1 decât nivelul definit în problemă.) Se dorește găsirea numărului minim de operații.

Mai întâi definim o noțiune utilă. Pentru un arbore X, definim $\text{centru}(X)$ ca fiind un vârf al lui X care minimizează cea mai mare distanță la un nod oarecare din X. În alte cuvinte,

$$\text{centru}(X) = \arg\min_x \max_y \text{dist}(x, y).$$

Nodul $\text{centru}(X)$ este nodul cel mai apropiat de centrul unui lanț diametral din arbore. (Adică, dacă vârfurile (a, b) determină diametrul arborelui, atunci $\text{centru}(X)$ este fie la mijlocul lanțului dintre ele, sau unul dintre nodurile cât mai aproape de acel mijloc dacă lanțul este de lungime impară. În cazul impar, nodul ales este arbitrar.) Observăm că putem găsi $\text{centru}(X)$ în timp linear folosind 3 DFS-uri: două pentru a găsi diametrul, și al treilea pentru a găsi nodul din mijlocul lanțului diametral.

Observație 3. *Există o soluție care mereu după ce taie o muchie, separând subarborele A' de rădăcină, ea niciodată nu va tăia o muchie din A' din nou, sau muchia adăugată.*

Demonstrație. Niciodată o soluție optimă nu va elimina o muchie ce a adăugat-o, căci ea ar fi putut adăuga de la început muchia aceea. Altfel, dacă prima operație separă un subarbore A'' de rădăcină, și apoi taie ceva în A'', operațiile pot fi reordonate astfel încât muchia tăiată din A'' să fie tăiată primă. □

Observație 4. *Într-o soluție, dacă se taie o muchie ce separă pe A' e rădăcină, și apoi niciodată nu se taie nimic din A' sau muchia adăugată, atunci este optim să legăm pe $\text{centru}(A')$ de rădăcină. Așadar este necesar și suficient ca diametrul lui A' să fie cel mult $2(k - 1)$.*

Demonstrație. Din definiție, $\text{centru}(A')$ minimizează distanța maximă de la el la oricare nod al lui A' deci este cel mai avantajos nod de care să ne legăm. Mai mult, pentru a avea adâncimea finală cel mult k, distanța de la centru la oricare nod (adică $\lceil \text{diametru}/2 \rceil$) trebuie să fie cel mult $k - 1$; deci diametrul trebuie să fie cel mult $2(k - 1)$. □

Definim $d = 2(k-1)$, diametrul maxim al unei componente separate de o tăietură.

Observație 5. *Considerăm oricare frunză f. Presupunem că f este la adâncime mai mare de k, și că arborele are diametrul mai mare ca d. Fie y primul nod de pe lanțul de la f la rădăcină unde subarborele lui y are diametrul strict mai mare ca d (există căci arborele are diametru $> d$); fie x fiul lui y care îl are pe f în subarbore. Să presupunem că f este cel mai adânc nod din subarborele lui x. Atunci există o soluție care îl taie prima dată pe (x, y).*

Schiță a demonstrației. Considerăm oricare soluție. Dacă această soluție taie o muchie în subarborele lui y, atunci putem să eliminăm acea tăietură, și să tăiem în loc pe (x, y) — subarborele lui y are diametrul cel mult d deci el poate fi separat; mai mult arborele ce rămâne este mai mic decât ar fi fost înainte, deci soluția ce exista deja poate fi folosită acum.

Așadar să presupunem că soluția nu taie nicio muchie în subarborele lui y. Ea trebuie să taie până la urmă o muchie de pe lanțul de la f la rădăcină, căci f este la distanță mai mare de k de rădăcină: fie s nodul mai apropiat de f din această muchie. Mai mult, cum subarborele lui x are diametrul prea mare trebuie să fie măcar o muchie tăiată acolo, fie ea m. (Muchia m nu poate fi în subarborele lui y că am presupus ca nu se taie nicio muchie în acest subarbore.)

Observăm că putem tăia muchia (x, y) în loc de muchia m. Acest fapt rezultă din faptul că frunza f are adâncime *maximă* în subarborele lui x — așadar este „cel mai important" să îl separăm pe el de restul arborelui. Se poate demonstra observând atent de unde poate veni diametrul subarborilor separați. □

Observație 6. *Dacă arborele are adâncimea mai mare ca k, dar are diametrul cel mult d, este necesară și suficientă o singură tăietură.*

Demonstrație. O tăietura este necesară căci arborele are adâncime prea mare. Este suficientă pentru că rădăcina nu poate avea doi fii ce conțin în subarbori frunze mai adânci ca k, căci altfel diametrul ar fi prea mare — deci putem tăia doar muchia de la rădăcină la fiul cu adâncimea maximă. □

Considerăm așadar următorul algoritm.

1. Facem DFS prin arbore.
2. Când suntem într-un nod, vom calcula mai întâi toate tăieturile făcute în subarborelui lui, și apoi vom returna la părintele lui adâncimea arborelui rămas.
3. Sortăm descrescător adâncimile subarborilor rămași ai unui nod.
4. Avem următoarele două cazuri:

   Pentru rădăcină. Tăiem muchiile către toți subarborii cu adâncimea mai mare ca k.

   Pentru oricare alt nod. Cât timp adâncimea primului subarbore este mai mare ca d, sau suma adâncimilor primilor doi subarbori este mai mare ca d, tăiem primul subarbore.

Observăm că acest algoritm va face doar tăieturi sancționate de observațiile precedente. Mai mult, el poate fi implementat și linear, observând că fiii nu trebuie sortați neapărat, ci doar partiționați în $O(1)$ puncte.