DESCRIEREA SOLUȚIILOR, OLIMPIADA NAȚIONALĂ DE INFORMATICĂ,
BARAJ SENIORI

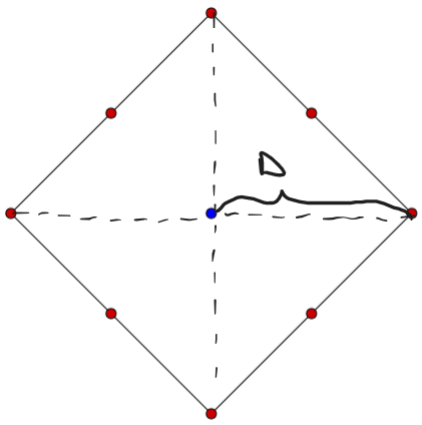
3DIST

Propusă de: stud. Bogdan-Ioan Popa, Universitatea din București
asist. doctorand Andrei-Costin Constantinescu, ETH Zürich

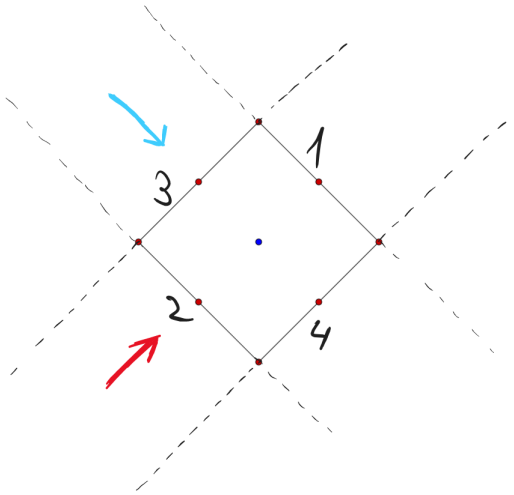
Vom clasifica fiecare punct i în funcție de valoarea $d(i)$. Un triplet (i, j, k) este valid doar dacă $d(i) = d(j) = d(k)$ (i.e. sunt în aceeași clasă), deci are sens să rezolvăm problema independent pentru fiecare clasă de puncte (o mică observație care nu era neapărat necesară pentru rezolvarea problemei este aceea că ne interesează doar punctele care au $d(i)$ par). Pentru a putea calcula $d(i)$ pentru fiecare i de la 1 la N se propune următoarea idee de rezolvare:

- Se sortează punctele în ordine lexicografică;
- Pentru fiecare punct i de la 1 la N vom vrea să determinăm care este distanța minimă către un punct $j < i$. Un astfel de punct va contribui la distanță cu $-X_j$ și $\pm Y_j$ în funcție de cum se compară cu Y_i. Astfel, distanța minimă către un astfel de punct va fi egală cu cea mai mică valoare dintre $X_i + Y_i + \min(-X_j - Y_j \mid j < i, Y_j <= Y_i)$ și $X_i - Y_i + \min(-X_j + Y_j \mid j < i, Y_j > Y_i)$. Pentru a afla acele minime se pot folosi doi arbori de intervale sau doi arbori indexați binar, după o normalizare a valorilor Y;
- Se procedează asemănător pentru a determina distanța minimă către puncte cu $j > i$, de data aceasta parcurgând punctele în ordine inversă.

Acum că avem calculate distanțele d, putem trece la a număra tripletele. După cum am spus și mai sus, vom rezolva independent problema pentru fiecare clasă de puncte. Să presupunem că suntem în clasa D de puncte. Pentru fiecare punct i din această clasă vom determina din câte triplete face parte. Se observă că punctele cu care acesta poate să facă triplet se află pe perimetrul rombului centrat în i care are diagonale de lungime egală cu $2D$ și laturile paralele cu diagonalele reperului cartezian, așa cum se poate vedea mai jos.



În plus, vor exista maxim 8 puncte care se vor afla pe perimetru. Pentru a le putea determina optim, vom „*glisa*" rombul în direcția indicată de săgeata albastră pentru a putea determina punctele de pe laturile 1 și 2, apoi în direcția indicată de săgeata roșie pentru a determina punctele de pe laturile 3 și 4 așa cum se vede în figura de mai jos.



Având determinate cele maxim 8 puncte candidate pentru a face triplet cu punctul i, putem încerca toate combinațiile posibile și să creștem un contor. Răspunsul va fi valoarea contorului împărțită la 3. Complexitatea temporală a soluției este $\mathcal{O}(N \log N)$.

Piezișă

Propusă de: stud. Ioan Popescu, Universitatea Politehnica din București
stud. George-Alexandru Râpeanu, Universitatea Babeș-Bolyai

O primă observație pe care o putem face este aceea că dacă ne uităm la șirul de xoruri pe prefixe, un query (x, y) se reduce la un query de tipul: găsiți distanța dintre cele cele mai apropriate valori egale din șirul prefixe de sume xor, unde un element din pereche se află la stânga lui x iar alt element se află la dreapta lui y. Formal, șirul de xor pe prefixe este definit prin: $s_i = a_0 \oplus a_1 \oplus a_2 \ldots \oplus a_i$. Cu $\oplus$ s-a notat operația de xor pe biți.

Soluție $\mathcal{O}(distincte \cdot (N + Q))$ (30 de puncte)

O soluție care rezolvă primele 3 subtaskuri este urmatoarea: Încercăm pentru fiecare valoare *distinctă* din șirul de xor pe prefixe să vedem care ar fi răspunsul optim folosind aceasta valoare pentru fiecare query. Să zicem ca am fixat o astfel de valoare val. Se pot precalcula pentru fiecare poziție i din șir, valorile $prev_i$ si $next_i$, unde $prev_i$ reprezintă cea mai mare poziție j din șirul s, cu proprietatea că $j \leq i$ și $s_j = val$. Similar, $next_i$ reprezintă cea mai mica poziție j din șirul s, cu proprietatea că $j \geq i$ și $s_j = val$. Cu ajutorul acestora, segmentul de lungime minimă care incepe și se termină in val, care conține un query (x, y) este $[prev_x, next_y]$ (dacă ambele există). Astfel, răspunsul pentru un query va fi lungimea minima a tuturor segmentelor de acest fel.

Soluție $\mathcal{O}\left(\frac{QN}{C} + Q \log N + NC \log N\right)$ (70 de puncte)

Ce putem face este să împărțim valorile numerelor în două: cele care au frecvența mai mare decât o valoare aleasă de noi, C, și cele care au frecvența mai mică decât C. Pentru a rezolva un query, putem să ne uităm la valorile care au frecvența mai mare și cele care au frecvența mai mică separat.

Cele mai mari sunt în număr de maxim $\frac{N}{C}$, deci putem afla răspunsul pentru fiecare query în $\mathcal{O}\left(\frac{N}{C}\right)$, folosind un algoritm similar cu cel descris in soluția anterioara. Astfel, complexitatea finală în acest caz este $\mathcal{O}\left(\frac{QN}{C}\right)$.

Pentru valorile mici, putem să sortăm queryurile crescător după capătul dreapta. Să zicem ca ne aflăm la un query (x, y). Facem notația $next_i = j$ unde $i < y$, și $j \geq y$, iar $s_j = s_i$. Pentru a răspunde la un query, trebuie să aflăm $\min_{i<x}(next_i - i)$. Atunci când trecem la alt query, pentru că avem queryurile sortate după capătul dreapta, y poate doar să crească. Incrementăm capătul dreapta cu 1 până ajunge la y curent. Fiecare iterație ne modifică $next_i$ pentru maxim

C poziții. Cum capătul dreapta se modifică de maxim N ori, numărul de modificări este maxim NC. Pentru a face rapid operațiile de update și query, putem folosi un arbore de intervale, complexitatea pentru updateuri și queryuri fiind $\mathcal{O}(Q \log N + NC \log N)$.

Alegerea lui C optim este lăsată la latitudinea cititorului.

Soluție $\mathcal{O}\left(\frac{QN}{C} + Q\left(B + \frac{N}{B}\right) + NC\right)$ (100 de puncte)

Soluția pentru valorile cu o frecvență mare nu trebuie îmbunătățită, ce ne deranjează pe noi este acel $NC \log N$. Ce putem face mai bine? De C nu prea putem scăpa, de N nici atât, deci $\log N$ este ce ne încurcă pe noi. Avem o structură care răspunde la queryuri și updateuri în $\mathcal{O}(\log(N))$, dar numărul de queryuri este mult mai mic decât numărul de updateuri. Ce structură putem folosi care să ne răspundă rapid la queryuri, și mai important, să facă updateurile în $\mathcal{O}(1)$? O descompunere în radical!

Pentru simplitate o să avem bucketuri de mărime B. Queryurile sunt simple, au complexitate $\mathcal{O}\left(B + \frac{N}{B}\right)$, dar cum facem updateuri în $\mathcal{O}(1)$? Pentru a putea face updateuri rapid, trebuie să avem următoarea proprietate adevărată: valorile $next_i$ pot doar să scadă. Pentru a respecta această proprietate, sortăm queryurile descrescător după capătul dreapta.

Alegerea lui B și C optimi este lăsată la latitudinea cititorului.

Soluție $\mathcal{O}\left(Q \log Q + QB + \frac{N^2}{B}\right)$ (100 de puncte)

O soluție alternativă poate fi găsită utilizând o modificare a (1). Similar ca și în algoritm, vom sorta queryurile prima data după $\left\lfloor \frac{x}{B} \right\rfloor$. În caz de egalitate, vom sorta *descrescător* după y. Este important să facem asta, deoarece, astfel, capătul drept doar va scoate elemente din Mo, o operație care e mai ușoara decât introducerea lor (din motive similare cu cele din soluția anterioară). Acum, pentru capătul stâng, ca să evitam adăugarea elementelor cu capătul stâng, la început de fiecare query el va fi setat la începutul bucketului curent. În momentul în care acesta trebuie ajustat, el va șterge din Mo toate elementele de la începutul bucketului până la $x - 1$, și va ține minte schimbările într-o stivă. Pentru a fi pregătit pentru următorul query, vom da `undo` folosindu-ne de această stivă.

Soluție $\mathcal{O}\left(QB + \left(\frac{N}{B}\right)^2\right)$ (100 de puncte)

O altă soluție care poate să obțină punctaj maxim este următoarea: împărțim șirul s în bucketuri de B elemente. Acum ne propunem să precalculăm $p(i, j)$ = lungimea subsegmentului de lungime minimă care are suma xor egală cu 0, care are capătul stâng într-unul dintre bucketurile $0, 1, \ldots, i$, iar capătul drept într-unul dintre bucketurile $j, j+1, \ldots$. Asta poate fi realizat relativ simplu. Acum, pentru un query (x, y) avem următoarele cazuri: segmentul optim are unul dintre capete în bucketul lui x sau în bucketul lui y, sau în niciunul. Răspunsul pentru cazul în care segmentul optim nu are capătul nici în bucketul lui x, nici în bucketul lui y este deja acoperit, noi putând să ne uităm direct în $p(bucket(x) - 1, bucket(y) + 1)$. Vom mai analiza doar cazul în care segmentul optim are capătul stâng în bucketul lui x, cazul în care are capătul drept în bucketul lui y fiind analog. Pentru acest caz, vom sorta queryurile descrescător după y, și vom ține pentru pozițiile mai mari decât y-ul curent șirul $first_i$, care ține cea mai mică poziție dintre cele procesate pe care se află valoarea i. Acum, putem doar să iterăm prin pozițiile $bucket(x) \cdot B$, ..., x și să ne folosim de $first_i$ pentru a găsi noile segmente.

Portocal

Propusă de: stud. Alexandra Maria Udriștoiu, Universitatea din București

Vom calcula următoarea programare dinamică pe arbore:

$d(i, 0)$ = numărul minim de șiruri formate din valorile de pe lanțuri de la nodul i la nodurile din subarborele lui i care sunt mai mici lexicografic decât șirul $S_{niv_i} \ldots S_k$ și nici unul dintre șirurile formate nu este egal cu $S_{niv_i} \ldots S_k$.

$d(i, 1)$ = numărul minim de șiruri formate din valorile de pe lanțuri de la nodul i la nodurile din subarborele lui i care sunt mai mici lexicografic decât șirul $S_{niv_i} \ldots S_k$ și cel puțin unul dintre șirurile formate nu este egal cu $S_{niv_i} \ldots S_k$.

$num(i, 0)$ = numărul de moduri de a completa valorile lipsă din subarborele nodului i pentru care se obține $d(i, 0)$.

$num(i, 1)$ = numărul de moduri de a completa valorile lipsă din subarborele nodului i pentru care se obține $d(i, 1)$.

Cu niv_i s-a notat nivelul nodului i în arbore. Fie w_i numărul de noduri din subarborele lui i și $nval_i$ numărul de noduri din subarbore ce nu au asociate o valoare, excluzând nodul i.

Considerăm cazul general, când $niv_i < K$. Considerăm $fiu_1 \dots fiu_G$ fii nodului i în arbore. Pentru a calcula $d(i, 0)$ și $num(i, 0)$, vom considera următoarele 3 cazuri:

- $val_i > S_{niv_i}$. Toate șirurile formate vor fi mai mari lexicografic decât $S_{niv_i}, \dots, S_k$. Așadar, $d(i, 0) = 0$ și $num(i, 0) = M^{nval_i} \cdot$ *numărul de moduri de a avea* $val_i > S_{niv_i}$.
- $val_i < S_{niv_i}$. Toate șirurile formate vor fi mai mici lexicografic decât $S_{niv_i}, \dots, S_k$. Așadar, $d(i, 0) = w_i$ și $num(i, 0) = M^{nval_i} \cdot$ *numărul de moduri de a avea* $val_i < S_{niv_i}$.
- $val_i = S_{niv_i}$, dar nu vrem să avem niciun șir egal cu $S_{niv_i}, \dots, S_k$. Atunci, $d(i, 0) = 1 + d(fiu_1, 0) + \dots + d(fiu_G, 0)$ și $num(i, 0) = num(fiu_1, 0) \cdot \dots \cdot num(fiu_G, 0)$.

Dintre aceste 3 cazuri, le vom considera doar pe cele pe care le putem obține pentru nodul i (dacă val_i este completată, avem unul singur, iar dacă $val_i = -1$, trebuie să vedem dacă există valori mai mici, respectiv mai mari, decât S_{niv_i}). Dintre valorile $d(i, 0)$ obținute, cea finală va fi cea mai mică dintre ele. În cazul în care se obține $d(i, 0)$ minim pentru mai multe cazuri, $num(i, 0)$ va fi suma numărului de moduri pentru acele cazuri.

Fie $d(i, 0/1)$ minim dintre $d(i, 0)$ și $d(i, 1)$ (nu ne interesează dacă $S_{niv_i} \dots S_k$ apare printre șirurile formate). Similar, $num(i, 0/1)$ va fi numărul de moduri de a completa valorile lipsă din subarbore pentru a obține $d(i, 0/1)$.

Pentru a calcula $d(i, 1)$ trebuie luat în considerare doar cazul când val_i este egal cu S_{niv_i} și $S_{niv_{i+1}} \dots S_k$ apare cel puțin unul dintre fii. Pentru a nu număra un mod de mai multe ori, vom fixa fiu_j primul fiu în care apare șirul. Astfel, $d(i, 1)$ va fi minim din:

$1 + d(fiu_1, 0) + \dots + d(fiu_{j-1}, 0) + d(fiu_j, 1) + d(fiu_{j+1}, 0/1) + \dots + d(fiu_G, 0/1)$

Iar $num(i, 1)$ va fi sumă pentru valorile j care dau $d(i, 1)$ minim, din:

$num(fiu_1, 0) + \dots + num(fiu_{j-1}, 0) + num(fiu_j, 1) + num(fiu_{j+1}, 0/1) + \dots + num(fiu_G, 0/1)$.

Pentru a avea complexitate finală $\mathcal{O}(N)$, trebuie să ținem sume parțiale pentru a calcula cele două valori de mai sus.

Când $niv_i = K$, pentru a calcula $d(i, 0)$ se consideră doar primele două cazuri. $d(i, 1)$ va fi egal cu $d(fiu_1, 0) + \dots + d(fiu_G, 0)$ și $num(i, 1) = num(fiu_1, 0) \cdot \dots \cdot num(fiu_G, 0)$, dacă val_i poate fi S_K.

Dacă $niv_i > K$, putem considera că toate șirurile formate vor fi mai mari, deci $d(i, 0) = 0$ și $num(i, 1)$ va fi numărul de moduri de a completa valorile din subarborele lui i.

Complexitatea finală va rămâne $\mathcal{O}(N)$.

Subtaskurile 1 și 4 pot fi rezolvate prin metoda backtracking.

Pentru subtaskul 2, observăm că, după ce am fixat un nod i pentru care vrem să se obțină un șir egal cu S și am completat corespunzător valorile de pe lanțul de la rădăcină la i, pentru ca acest șir să apară cât mai repede în ordine lexicografică, nodurilor rămase fără valori le putem da tuturor valoarea M. Așadar, vom încerca toate posibilitățile de a alege nodul i, vom completa valorile, și vom număra cu ajutorul unei parcurgeri DFS câte șiruri ar fi mai mici decât nodul S pentru această completare, alegând minimul. Complexitatea acestei soluții este $\mathcal{O}(N^2)$.

Miyuki vrea să împăturească arborigami

Propusă de: Vlad-Alexandru Gavrilă-Ionescu, SEPI

Prima observație pe care trebuie să o facem este că o operație este validă doar dacă nodurile a_i și b_i împăturite în cadrul operației sunt fie vecine, fie au un vecin comun. Altfel, operația introduce un ciclu în arbore. Mai observăm și că, dacă există vreo muchie între două noduri A și B din arborele original pentru care niciunul dintre A și B nu a făcut parte dintr-o operație de împăturire, atunci arborele obținut în urma tuturor operațiilor de împăturire nu are cum să fie stea. Astfel, problema se reduce la a afla un subset minim de noduri S astfel încât fiecare

muchie să aibă cel puțin unul din capete într-un nod ce aparține lui S. Acest subset S se numește acoperirea minimă cu noduri a arborelui(2).

Pentru a implementa acest lucru, putem face o parcurgere în adâncime pe arborele original, și să aplicăm următoarea strategie greedy: după ce am parcurs toți subarborii aferenți unui nod x, îl vom selecta pe x ca făcând parte din subsetul S dacă și numai dacă cel puțin unul dintre fii direcți ai lui x nu aparține lui S (pentru a evita situația descrisă în a doua observație). Soluții alternative care folosesc programarea dinamică sau algoritmul de cuplaj în graf bipartit sunt de asemenea suficiente pentru a obține punctaj maxim.

Pentru a reconstitui, pe baza setului S, operațiile care duc la împăturirea arborelui inițial într-unul stea, este suficient să luăm nodurile din S în ordinea adâncimii lor în arborele inițial, și să le unim la fiecare pas cu cel mai de sus nod aflat în arbore la momentul respectiv. Acesta este, la prima operație, cel mai de sus nod aflat în arborele inițial, iar apoi, nodul creat în urma precedentei operații de împăturire. Observăm că, aplicând această strategie, operațiile respectă mereu prima observație. Complexitatea finală a acestei soluții este $\mathcal{O}(N)$.

Subtask 1 (10 puncte)

Soluția parțială pentru primul subtask presupune selectarea subsetului S cu backtracking în $\mathcal{O}(2^N)$ și apoi simularea operațiilor în $\mathcal{O}(N^2)$ pentru a ne asigura că ele sunt valide, pentru o complexitate totală de $\mathcal{O}(2^N N^2)$.

Subtask 2 (20 de puncte)

Al doilea subtask admite orice algoritm polinomial pentru determinarea setului S (de exemplu, programare dinamică în $\mathcal{O}(N^2)$).

Subtask 3 (10 puncte)

Pentru al treilea subtask, observăm că arborele dat este un lanț, prin urmare putem selecta setul S ca fiind format din toate nodurile cu indice par.

Guguștiuc

Propusă de: stud. Matei Tinca, Vrije Universiteit Amsterdam

Pentru primul subtask, putem simula efectiv toate operațiile în complexitate $\mathcal{O}(N^2)$.

Pentru următoarele două subtaskuri, pentru fiecare interval ne putem menține un set în care stocăm toate intervalele rămase. Atunci când vine o operație de `split` luăm intervalele care îl conțin pe x și le împărțim în câte două intervale. Complexitatea soluției este $\mathcal{O}(QN \log N)$.

O observație importantă pentru găsirea soluției este faptul că dacă avem două operații de `split` la pozițiile x și y și după avem o operație de `skip` la o poziție z, atunci din toate intervalele care îl conțin pe z se va șterge intervalul (x, y). Putem să reformulăm problema astfel: se dau N intervale și Q updateuri definite prin triplete de numere (x, y, z), iar un update înseamnă că pentru fiecare interval care îl conține pe z se șterge intervalul (x, y).

Pentru a ne aduce problema la noua formă, putem simula operațiile inițiale pe intervalul $(1, \infty)$, folosind soluția de la subtaskul anterior. Când avem o operație de `skip`, vedem primul `split` din stânga punctului în care se aplică operația și la fel și pe partea dreaptă. Astfel, ne putem scoate tripletele de numere (x, y, z).

O proprietate importantă a acestor triplete ce poate fi dedusă din modul de construcție este faptul că intervalele (x, y) specifice tripletelor sunt fie disjuncte, fie incluse unul în altul.

În această formă a problemei, tripletele devin updateuri, iar cele N intervale inițiale devin queryuri: pentru un interval (x, y), vrem să vedem, după ce aplicăm toate updateurile, cu cât contribuie la răspuns intervalul respectiv.

Pentru cazul în care există mai multe triplete care au același z, putem lua în considerare doar primul triplet găsit (și anume cel care are intervalul cel mai mare).

Pe această formă, putem obține soluții pentru restul subtaskurilor.

Ne putem menține un arbore de intervale, unde pentru un segment de lungime 1, adică de forma $(x, x + 1)$ reținem dacă acel segment a fost acoperit. Ca și detalii de implementare, acest

arbore de intervale va menține minimul pe un interval și de câte ori apare acesta, iar când vrem să marcăm faptul că am acoperit intervalul (x, y), atunci adăugăm 1 pe intervalul $(x, y-1)$.

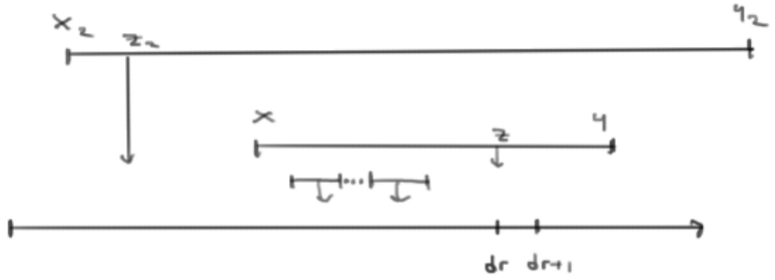
Pentru un interval din cele N inițiale, pentru a vedea cât rămâne din acesta după ce aplicăm operațiile definite de triplete, luăm toate tripletele cu z care aparține intervalului respectiv și acoperim în arborele de intervale fiecare interval asociat tripletelor. Lungimile rămase vor fi numărul de apariții ale lui 0 pe intervalul $x, y-1$.

Pentru următoarele două subtaskuri, presupunem că am rezolvat intervalul (x, y) și vrem să rezolvăm alt interval. Putem să ne mișcăm pointerii x și y până când se potrivesc cu cei ai intervalului pe care vrem să îl rezolvăm. Mișcarea unui pointer înseamnă incrementarea sau decrementarea acestuia, iar după fiecare modificare se activează sau dezactivează tripletele care intră sau ies din acoperirea intervalului (x, y). Dacă rezolvăm intervalele în ordinea din input, rezolvăm subtaskul 4. Dacă folosim Algoritmul lui Mo, atunci rezolvăm și subtaskul 5. Complexitățile sunt $\mathcal{O}(val_{max} \log N)$, respectiv $\mathcal{O}(N\sqrt{N})$.

Pentru a rezolva toată problema, putem încerca următoarea abordare: iterăm cu capătul dreapta al unui interval și vedem cum se schimbă răspunsul în funcție de capătul stâng. Presupunând că suntem la pasul dr, ne calculăm șirul res_{st} ca fiind rezultatul queryului (st, dr). Dacă avem șirul calculat pentru pasul dr, trebuie să vedem cum se modifică res_{st} dacă trecem de la dr la $dr+1$.

Trebuie să vedem intervalul $(dr, dr+1)$ la ce elemente din șir contribuie. Pentru acest caz, observăm că se adaugă 1 la toate elementele de pe pozițiile $z', z'+1, \ldots dr$, unde z' este cel mai mare z al unui triplet în care intervalul acestuia acoperă intervalul $(dr, dr+1)$, deoarece toate intervalele cu capătul stâng până la acel punct nu vor fi afectate de vreun update, iar după acel punct, intervalul $(dr, dr+1)$ va fi șters.

Acuma trebuie ținut cont de cazul în care la poziția dr avem un triplet care afectează șirul.



Pe desenul de deasupra, (x_2, y_2, z_2) este un triplet prin care am trecut deja care are z-ul cât mai mare, iar intervalul lui, (x_2, y_2) îl include în totalitate pe intervalul (x, y). Astfel, putem observa următoarele două modificări: pe intervalul $(x, dr-1)$, elementele lui *res* devin 0, deoarece acestea vor include tripletul (x, y, z), deci toate intervalele cu capătul între x și $dr-1$ vor fi șterse. A doua modificare asupra șirului *res* va fi faptul că în intervalul $(z_2, x-1)$ se va scădea *sub*, unde *sub* este lungimea intervalului (x, dr) din care scădem toate intervalele mai mici care sunt incluse în (x, dr). Practic, pentru toate intervalele cu acele capete, trebuie să ținem cont de faptul că se șterg toate bucățile neacoperite din acel interval.

Aceste modificări se pot implementa folosi un arbore de intervale care suportă următoarele operații:

- `add`(x, y, z) – Pe intervalul (x, y) se adaugă valoarea z;
- `set`$_0(x, y)$ – Intervalul (x, y) se setează pe 0.

Complexitatea acestei soluții va fi $\mathcal{O}((val_{max} + Q + N) \log N)$ și obține scor maxim.

Hoață

Propusă de: asist. doctorand Andrei-Costin Constantinescu, ETH Zürich

Subtask 1 (11 puncte)

Se folosește metoda backtracking. Restricția $g_i \geq 2$ garantează încadrarea soluției în limita de timp.

Subtask 2 (18 puncte)

Se poate proceda după cum urmează: rând pe rând determinăm acțiunile fiecărui hoț, alegând de fiecare dată acțiuni care nu ar declanșa nicio alarmă date fiind acțiunile hoților procesați în prealabil și care ar duce la o captură maximă pentru hoțul curent. Mai exact, pentru fiecare pereche (i, g), unde $1 \leq i \leq N$ și $0 \leq g \leq G$ vom menține o variabilă $x_{i,g}$ reprezentând numărul de hoți care ar mai putea trece prin camera i cu greutate a rucsacului g fără a declanșa alarma. Inițial, vom seta $x_{i,g} = x_i$, iar pe parcurs vom scădea aceste valori după fiecare hoț ale cărui acțiuni au fost decise. Acum, pentru a determina acțiunile optime ale unui hoț, vom face o dinamică $dp_{i,g} =$ care este captura maximă posibilă a hoțului curent până în camera i astfel încât acesta împreună cu hoții anteriori să nu declanșeze alarma iar rucsacul hoțului să fie umplut cu obiecte de greutate totală g, unde vom lua $dp_{i,g} = -\infty$ dacă o alarmă s-ar declanșa indiferent de acțiunile hoțului curent. Această dinamică se calculează pentru $1 \leq i \leq N + 1$ și $0 \leq g \leq G$. Definim cantitățile:

- $A = v_i + dp_{i,g-g_i}$ dacă $i \leq N$ și $g_i \leq g$, iar $A = -\infty$ altfel;
- $B = dp_{i-1,g}$ dacă $i \geq 2$ și $x_{i-1,g} > 0$, iar $B = -\infty$ altfel.

Atunci avem relația de recurență $dp_{i,g} = \max(A, B)$. Inițializarea dinamicii este dată de cazul excepțional $dp_{1,0} = 0$. După calculul dinamicii se poate reconstitui șirul optim de acțiuni ale hoțului curent, după care se trece la hoțul următor. Complexitatea soluției este $\mathcal{O}(NKG)$. Lăsăm ca temă cititorului motivul pentru care algoritmul prezentat este corect (adică de ce este corect sa determinăm acțiunile hoților unul câte unul).

Subtaskurile 3 și 4 (69 de puncte, respectiv, 100 de puncte)

În acest caz, camerele pot avea alarme de valori x_i variate, așa că algoritmul folosit pentru rezolvarea subtaskului 2 nu mai este corect, însă ne putem în continuare folosi de graful de stări al dinamicii pentru a determina simultan acțiunile optime ale tuturor hoților. Mai exact, pentru rezolvarea subtaskurilor 3 și 4 vom folosi tehnica flux maxim de cost minim(3), după cum urmează. Vom avea câte un nod $n_{i,g}$ pentru fiecare $1 \leq i \leq N + 1$ și $0 \leq g \leq G$. Fiecare muchie orientată $a \to b$ din rețeaua de flux va avea o capacitate pozitivă x și un cost nepozitiv y. În acest caz vom nota $a \xrightarrow[y]{x} b$. Acestea fiind spuse, în rețeaua de flux introducem următoarele muchii:

- Pentru $i \leq N$ și $g_i \leq g$ adaugăm muchia $n_{i,g-g_i} \xrightarrow[-v_i]{\infty} n_{i,g}$;
- Pentru $i \geq 2$ adaugăm muchia $n_{i-1,g} \xrightarrow[0]{x_{i-1}} n_{i,g}$.

Sursa se alege nodul $n_{1,0}$ și se introduc introduc în rețea K unități de flux (dacă acest lucru nu este posibil, atunci se va afișa -1) astfel încât costul total al fluxului astfel determinat sa fie minim. Răspunsul va fi atunci -1 înmulțit cu costul fluxului.

În funcție de implementare, acestă abordare poate obține între 69 și 100 de puncte. Pentru 100 de puncte, următoarele optimizări sunt necesare:

- Graful de flux are $(N+1)(G+1) \leq 90\,601$ noduri, deci implementarea nu se poate face cu matrice de adiacență, ci sunt necesare listele de adiacență.
- Găsirea drumurilor minime în graful rezidual trebuie făcută folosind tehnica Dijkstra cu potențiale(4), în locul mai uzualului Bellman-Ford cu coadă(5). Calculul potențialelor se poate face cu programare dinamică, deoarece graful rețelei de flux este aciclic.

Într-o implementare eficientă, complexitatea este $\mathcal{O}(NKG \log(NG))$.

Echipa

Problemele pentru această etapă au fost pregătite de:

- prof. Adrian Panaete – Colegiul Național „A.T. Laurian”, Botoșani
- prof. Ionel-Vasile Piț-Rada – Colegiul Național „Traian”, Drobeta-Turnu Severin
- asist. doctorand Andrei-Costin Constantinescu – ETH, Zürich
- stud. Theodor-Gabriel Tulba-Lecu – Universitatea Politehnica București
- prof. Mihai Bunget – Colegiul National „Tudor Vladimirescu”, Târgu Jiu
- prof. Carmen Popescu – Colegiul National „Gh. Lazăr”, Sibiu
- prof. Dan Pracsiu – Liceul Teoretic „Emil Racoviță”, Vaslui
- specialist IT Vlad-Alexandru Gavrilă-Ionescu – SEPI
- doctorand Tamio-Vesa Nakajima – University of Oxford
- stud. Bogdan-Ioan Popa – Universitatea din București
- stud. Matei Tinca – Vrije Universiteit Amsterdam
- masterand Ioan-Bogdan Iordache – Universitatea din București
- stud. Bogdan Sitaru – University of Oxford
- stud. Stelian Chichirim – Universitatea din București
- stud. Alexandra Maria Udristoiu – Universitatea din București
- inspector Szabo Zoltan – ISJ Mures
- stud. George-Alexandru Râpeanu – Universitatea Babes-Bolyai, Cluj-Napoca
- stud. Ioan Popescu Universitatea – Politehnica București
- stud. Ștefan-Cosmin Dăscalescu – Universitatea din București
- stud. Costin-Andrei Oncescu – University of Oxford
- doctorand Lucian Bicsi – Universitatea din București
- stud. Ioan-Cristian Pop – Universitatea Politehnica București

Referințe

[1] Algoritmul lui Mo
[2] Minimum Vertex Cover
[3] Flux maxim de cost minim
[4] Dijkstra cu potențiale
[5] Bellman-Ford cu coadă