

OLIMPIADA NAȚIONALĂ DE INFORMATICĂ 2026, ETAPA NAȚIONALĂ
CLASELE 11-12
DESCRIEREA SOLUȚIILOR

COMISIA ȘTIINȚIFICĂ

PROBLEMA 1: EPRUBETE

Propusă de: prof. Panaete Adrian, Colegiul Național “A.T. Laurian”, Botoșani

În descrierea soluției, vom nota capacitățile eprubetelor cu $c[1], \dots, c[N]$. Simularea directă a procesului nu este posibilă decât pentru punctaj parțial, deoarece timpul total de umplere poate fi foarte mare. În schimb, vom descrie structural cum se desfășoară umplerea și vom răspunde la fiecare întrebare printr-o procedură recursivă.

Pentru fiecare eprubetă i , definim `pic_per_unitate[i]` ca numărul total de picături care trebuie create pentru a obține o singură picătură în eprubeta i .

Dacă definim `dep[i]` ca fiind eprubetele de care depinde eprubeta i , avem imediat relația:

$$\text{pic_per_unitate}[i] = 1 + \sum_{j \in \text{dep}[i]} \text{pic_per_unitate}[j].$$

Într-adevăr, pentru a crea o picătură în i , trebuie creată chiar acea picătură, iar în plus trebuie compensate toate picăturile consumate din eprubetele de care depinde i .

Definim și `timp_umplere_prefix[i]` ca numărul de secunde după care eprubetele $1, 2, \dots, i$ se umplu pentru prima oară toate simultan. Atunci avem:

$$\text{timp_umplere_prefix}[i] = \text{timp_umplere_prefix}[i - 1] + c[i] \cdot \text{pic_per_unitate}[i].$$

Într-adevăr, după ce primele $i - 1$ eprubete sunt pline, pentru a umple complet eprubeta i mai trebuie puse în ea exact $c[i]$ picături, iar fiecare astfel de picătură “costă” în total `pic_per_unitate[i]` secunde.

Vom rezolva o problemă puțin mai generală. Pentru un prefix $[1..i]$ și pentru o valoare c_i , vrem să determinăm în ce eprubetă se pune o picătură la o anumită secundă, dacă lucrăm doar cu eprubetele $1, 2, \dots, i$, iar capacitățile sunt

$$c[1], c[2], \dots, c[i - 1], c_i.$$

Cu alte cuvinte, ultima capacitate este revizuită.

Această generalizare este exact cea de care avem nevoie în recursie. Fixăm deci un apel recursiv pe prefixul $[1..i]$, în care ultima eprubetă are capacitatea revizuită c_i .

Comportamentul algoritmului de umplere se descompune în *pași* astfel:

- Pasul 0: umplem primele $i - 1$ eprubete. Acest lucru durează

$$T = \text{timp_umplere_prefix}[i - 1]$$

secunde.

- Apoi începem să punem picături în eprubeta i , una câte una, până când fie eprubeta i se umple, fie se golește complet măcar una dintre eprubetele din care se consumă substanță pentru a obține substanța i . Dacă notăm

$$\text{cap_min_dep} = \min_{j \in \text{dep}[i]} c[j],$$

atunci putem pune exact `cap_min_dep` picături consecutive în eprubeta i .

- După aceea trebuie reumplute eprubetele dependente ale lui i .
- Se repetă alternativ ultimii doi pași de câte ori este nevoie.

Notăm

$$\alpha = \text{cap_min_dep} \cdot \sum_{j \in \text{dep}[i]} \text{pic_per_unitate}[j].$$

Aceasta este durata necesară pentru a reumple toate eprubetele dependente după ce am pus cap_min_dep picături în eprubeta i .

Așadar, după pasul 0, procesul continuă în blocuri repetitive:

- punem cap_min_dep picături în eprubeta i ;
- reumplem eprubetele dependente, în α secunde.

Vom numi *fază* o astfel de pereche de doi pași consecutivi.

După

$$K = \left\lfloor \frac{c_i}{\text{cap_min_dep}} \right\rfloor$$

faze complete, în eprubeta i vor fi puse $K \cdot \text{cap_min_dep}$ picături.

Dacă c_i este divizibil cu cap_min_dep , acestea sunt toate fazele. Altfel, mai există o fază suplimentară, în care se mai pun doar $c_i \bmod \text{cap_min_dep}$ picături în eprubeta i , după care se reumple din nou eprubetele dependente. Cu alte cuvinte, ultima fază poate fi specială.

Pentru a afla unde se încadrează secunda cerută, procedăm după cum urmează.

Mai întâi verificăm dacă secunda se află în pasul 0. Mai exact, dacă $\text{secunda} \leq T$, atunci reluăm căutarea pe prefixul format din primele $i - 1$ eprubete.

În caz contrar, secunda se află după pasul 0, deci într-una dintre fazele descrise mai sus.

Lungimea unei faze complete este $\text{cap_min_dep} + \alpha$. Prin urmare, numărul de faze complete încheiate înaintea secunde noastre este

$$\text{faze_inainte} = \left\lfloor \frac{\text{secunda} - T - 1}{\text{cap_min_dep} + \alpha} \right\rfloor.$$

La începutul fazei în care se află secunda noastră, în eprubeta i mai trebuie puse

$$\text{cap_inceput_faza} = c_i - \text{faze_inainte} \cdot \text{cap_min_dep}$$

picături.

Așadar, în primul pas al fazei curente vom pune în eprubeta i

$$Z = \min(\text{cap_min_dep}, \text{cap_inceput_faza})$$

picături.

Dacă secunda noastră se află în acest prim pas al fazei, atunci răspunsul este chiar i .

Rămâne cazul în care secunda se află în al doilea pas al fazei, adică în reumplerea eprubetelor dependente ale lui i . Fie $j_1 < j_2 < \dots < j_k$ eprubetele din $\text{dep}[i]$. După ce am pus cele Z picături în eprubeta i , fiecare dintre eprubetele $j_1, \dots, j_k$ a pierdut exact Z picături. Din regula problemei, se alege mereu prima eprubetă neplină. De aceea, reumplerea acestor eprubete se face în *subpași*, în ordinea:

$$[1..j_1], [1..j_2], \dots, [1..j_k].$$

Mai precis, în subpasul ℓ se execută exact algoritmul de umplere pe prefixul $[1..j_\ell]$ până când toate eprubetele din acest prefix devin din nou pline.

Durata subpasului ℓ este

$$X(\ell) = Z \cdot \text{pic_per_unitate}[j_\ell],$$

deoarece trebuie puse exact Z picături în eprubeta j_ℓ , iar o picătură în j_ℓ costă în total $\text{pic_per_unitate}[j_\ell]$ secunde.

Definim sumele parțiale

$$Y(0) = 0, \quad Y(\ell) = X(1) + X(2) + \dots + X(\ell).$$

Dacă notăm cu sfarsit_pas_1 ultima secundă a primului pas al fazei curente, atunci subpasul ℓ ocupă exact intervalul de secunde

$$\text{sfarsit_pas_1} + Y(\ell - 1) + 1, \dots, \text{sfarsit_pas_1} + Y(\ell).$$

Parcurgem acești subpași și găsim primul ℓ pentru care secunda cerută se află în intervalul de mai sus. Odată determinat acest ℓ , apare ideea cheie din rezolvare (care este și motivul pentru care rezolvăm problema mai generală), descrisă în cele ce urmează.

Știm că secunda noastră se află în subpasul ℓ , adică în algoritmul care reumple prefixul $[1..j_\ell]$. La începutul acestui subalgoritm:

- primele $j_\ell - 1$ eprubete sunt pline;
- eprubeta j_ℓ are exact $c[j_\ell] - Z$ picături.

Pe parcursul subpasului, eprubeta j_ℓ va avea mereu cel puțin $c[j_\ell] - Z$ picături. Putem deci privi situația echivalent astfel:

- eprubeta j_ℓ este inițial goală;
- capacitatea ei este revizuită la Z .

Mai mult, putem privi întregul subpas ca pornind chiar din configurația în care toate eprubetele din prefixul $[1..j_\ell]$ sunt goale, doar că atunci secunda relevantă nu mai este aceeași. Dacă notăm cu `secunda_prim` poziția secunde noastre în interiorul subpasului curent, atunci în configurația standardizată de mai sus trebuie să căutăm secunda

$$\text{secunda_prim} + \text{timp_umplere_prefix}[j_\ell - 1].$$

Motivul este că, pornind din această configurație standardizată, după exact $\text{timp_umplere_prefix}[j_\ell - 1]$ secunde ajungem în configurația reală de la începutul subpasului.

Prin urmare, dacă secunda noastră se află în subpasul ℓ , reducem problema recursiv la:

- prefixul $[1..j_\ell]$;
- capacitatea finală revizuită la Z ;
- secunda `secunda_prim + timp_umplere_prefix[jℓ - 1]`.

Moment la care putem să apelăm recursiv.

În concluzie, după precalculearea tablourilor `pic_per_unitate` și `timp_umplere_prefix`, fiecare întrebare se rezolvă prin această funcție recursivă:

- dacă secunda este în pasul 0, coborâm la prefixul $[1..i - 1]$;
- altfel identificăm faza curentă;
- dacă secunda este în primul pas al fazei, răspunsul este i ;
- altfel identificăm subpasul și apelăm recursiv aceeași procedură pe un prefix mai mic, cu capacitatea ultimei eprubete schimbată la Z .

Cum la fiecare apel recursiv prefixul se micșorează, procedeul se termină. Complexitatea totală a soluției este $O(N \cdot Q)$.

PROBLEMA 2: LOOP

Propusă de: stud. Todoran Alexandru-Raul, Harvard University

Subtask 1: $N \cdot M \leq 12$ (**8 puncte**). Pentru acest subtask, o abordare bazată pe metoda backtracking este suficientă. Vom genera toate drumurile care nu trec de două ori prin aceeași celulă din punctul (ℓ_1, c_1) până în punctul (ℓ_2, c_2) . Pentru un drum determinat, trebuie să verificăm fiecare perioadă posibilă a traseului. Numărul total de drumuri este $O(4^{N \cdot M})$, deci complexitatea finală este $O(4^{N \cdot M} \cdot N \cdot M)$.

Subtask 2: Se garantează că $S_p = \emptyset$ oricare ar fi $p > 1$ și $5 \leq N, M$ (**7 puncte**). Pentru acest subtask, știm că nu există drumuri periodice. Prin urmare, este suficient să numărăm în câte moduri se poate ajunge din punctul de start până la punctul de final cu număr minim de pași. Această problemă se poate rezolva folosind BFS. Pe lângă distanța minimă la fiecare celulă, vom reține și numărul de moduri de a ajunge în celula respectivă cu lungime minimă.

Subtask 3: Nu există celule blocate și $5 \leq N, M$ (19 puncte). Pentru acest subtask, este nevoie să facem observații despre care sunt valorile p care pot fi soluție. După prima repetiție, vom ajunge în punctul $(\ell_1 + \Delta_x, c_1 + \Delta_y)$. După a doua repetiție, vom ajunge în punctul $(\ell_1 + 2 \cdot \Delta_x, c_1 + 2 \cdot \Delta_y)$. Trebuie ca la final să ajungem în punctul (ℓ_2, c_2) . Dar, (ℓ_2, c_2) trebuie să fie egal cu $(\ell_1 + (p-1) \cdot \Delta_x, c_1 + (p-1) \cdot \Delta_y)$. Deducem de aici că p trebuie să fie divizor al $|\ell_1 - \ell_2|$ și divizor al $|c_1 - c_2|$. De aici, putem deduce că avem un număr mic de valori ale lui p pe care trebuie să le testăm. Vom fixa fiecare p care poate să fie parte din răspuns. Acum, trebuie să numărăm câte drumuri de lungime minimă există de la (ℓ_1, c_1) până la $(\ell_1 + \frac{\ell_2 - \ell_1}{p}, c_1 + \frac{c_2 - c_1}{p})$. Din nou, putem rezolva această problemă cu BFS, exact ca la subtask-ul anterior. Alternativa este să folosim o formulă pentru numărul de drumuri, care este $\binom{\frac{|\ell_2 - \ell_1|}{p} + \frac{|c_2 - c_1|}{p} - 1}{\frac{|c_2 - c_1|}{p}}$.

Subtask 4: $N \cdot M \leq 2500$ (29 puncte). Pentru acest subtask, trebuie să luăm în considerare și obstacolele din matrice. În rest, soluția va rămâne la fel ca la subtask-ul anterior. Exact ca la subtask-ul anterior, vom calcula numărul de drumuri de lungime minimă de la punctul de start, până la primul punct intermediar (adică până la punctul la care ajungem după ce executăm pentru prima dată secvența de instrucțiuni). Observăm că pentru a putea folosi celula (i, j) în acest traseu, celulele (i, j) , $(i + \Delta_x, j + \Delta_y)$, ..., $(i + (p-1) \cdot \Delta_x, j + (p-1) \cdot \Delta_y)$ trebuie să fie libere. Pentru fiecare celulă (i, j) , putem verifica în $O(p)$ dacă această celulă poate fi parte dintr-un traseu periodic. Complexitatea finală devine $O(N \cdot M \cdot \min(N, M) \cdot D)$, unde D este numărul de valori posibile ale lui p .

Subtask 5: $N, M \geq 5$ (37 puncte). Pentru acest ultim subtask, singura diferență este cum verificăm dacă putem folosi celula (i, j) într-un traseu cu perioadă p . Este suficient să calculăm $n_{i,j}$ - cât de mulți pași (cu Δ_x pe direcția x și Δ_y pe direcția y) până când întâlnim un obstacol. Putem calcula această tabelă folosind programare dinamică în $O(N \cdot M)$ folosind relația $n_{i,j} = 1 + n_{i+\Delta_x, j+\Delta_y}$ dacă celula (i, j) nu este blocată, respectiv 0 dacă este blocată. Complexitatea finală este $O(N \cdot M \cdot D)$.

PROBLEMA 3: WIND

Propusă de: stud. Roșu Adrian-Andrei, Universitatea Națională de Știință și Tehnologie Politehnica București

În continuare toate operațiile de modulo vor fi ignorate, deoarece dacă folosim `unsigned int` pe toate operațiile cu mere, atunci operația de mod 2^{32} va fi făcută automat.

Subtask 1: $M, Q, K \leq 2000$ (7 puncte). Deoarece limitele sunt mici în acest subtask, cea mai simplă și directă abordare este simularea pas cu pas a tuturor operațiilor. Pentru evenimentele de primul tip, schimbăm direcția vântului direct în memorie într-un timp constant de $O(1)$. Când întâlnim o actualizare a merelor de tipul doi, parcurgem pur și simplu submatricea indicată și creștem cantitatea de mere din fiecare celulă cu valoarea dată, operație care durează $O(N \cdot M)$. În cele din urmă, pentru interogările de tipul trei, simulăm efectiv cele K mutări succesive ale lui Gigel și adunăm merele colectate pe parcurs, ceea ce ne conduce la o complexitate de $O(K)$ pentru fiecare interogare completă.

Subtask 2: $M \leq 200$ și $Q \leq 2000$ (9 puncte). Aici dispăre restricția asupra lui K , așadar simularea manuală a tuturor pașilor nu mai este suficient de rapidă. Cheia rezolvării stă în a analiza proprietățile traseului pe care Gigel îl parcurge.

Observație 1. Deoarece din fiecare celulă vântul suflă într-o singură direcție precisă, drumul lui Gigel din orice celulă de start (i, j) este mereu unic. Graful format de direcțiile vântului este unul funcțional, așadar deplasarea poate fi privită ca o funcție matematică de forma $f : \mathbb{N}^3 \rightarrow \mathbb{N}$, unde $f(i, j, k)$ reprezintă răspunsul pentru o operație de tipul 3 cu parametrii respectivi.

Observație 2. Într-un astfel de graf, orice drum pornit dintr-o celulă poate avea doar două topologii posibile. Fie este un drum complet **aciclic** care părăsește matricea în cel mult $N \cdot M$

pași, fie are o traiectorie de tip R_0 (de la litera grecească ρ), fiind compus dintr-o porțiune liniară inițială (coada) urmată de un ciclu care se repetă la nesfârșit (bucula).

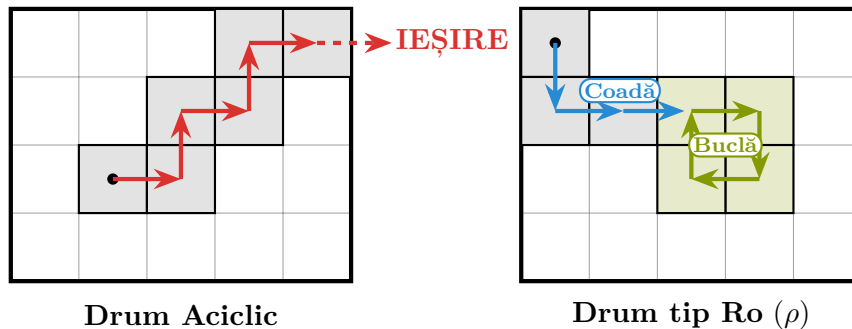


FIGURE 1. Vizualizarea celor două topologii posibile ale traseului lui Gigel.

Cunoscând acest comportament, algoritmul devine clar: marcăm pozițiile parcurse într-o matrice de vizitare pe măsură ce avansăm. Când Gigel ajunge într-o celulă pe care am mai vizitat-o, știm sigur că am descoperit un ciclu, lucru care se va întâmpla garantat după cel mult $N \cdot M$ mutări. Odată ce am închis această buclă, calculăm rapid de câte ori o va parcurge Gigel complet în pașii rămași din K . Înmulțim numărul de mere din ciclu cu acest număr de repetări, iar pentru restul de pași (care nu pot forma un ciclu complet) reluăm simularea naivă, celulă cu celulă. Datorită acestei abordări, complexitatea scade la cel mult $\mathcal{O}(N \cdot M)$ pe interogare.

Subtask 3: $N = 1$ (11 puncte). Faptul că matricea se reduce la o singură linie schimbă radical dinamica mișcărilor. Dacă la un moment dat Gigel ajunge într-o celulă cu vânt orientat înspre Nord sau Sud, el iese imediat de pe hartă, iar interogarea returnează valoarea -1. Prin urmare, singurele bucle infinite care se pot forma structural sunt ciclurile între celule adiacente cu direcțiile Est și Vest (EV). Pentru a răspunde rapid la interogări, nu mai este nevoie să simulăm, ci doar să aflăm care este primul caracter diferit de direcția celulei de start pe care îl va întâlni Gigel pe direcția sa de deplasare. Acest lucru se poate implementa extrem de eficient menținând permanent patru structuri de date de tip `std::set`, câte una pentru fiecare direcție cardinală, căutând apoi rapid intersecțiile cu ajutorul funcției `std::set.lower_bound()`. Deoarece dimensiunea M poate fi mare, actualizările cantităților de mere pe intervale extinse nu mai pot fi făcute prin parcurgere liniară, ci necesită un arbore de intervale cu lazy propagation. Această structură permite efectuarea modificărilor în timp logaritm, obținând o complexitate finală de $\mathcal{O}(Q \cdot \log M)$.

Subtask 4: $N \leq 10$ și **Gigel nu se poate deplasa în sus (14 puncte)**. Această cerință funcționează ca o extindere naturală a ideii de la subtask-ul anterior. Neavând deloc direcția Nord pe hartă, singurele cicluri valide rămân tot cele de tip Est-Vest, care se desfășoară strict orizontal. Acest detaliu ne permite să tratăm fiecare linie a matricei ca fiind o entitate independentă de celelalte. Practic, vom rula algoritmul optimizat cu `std::set`-uri pe linia curentă pe care ne aflăm. În momentul în care Gigel întâlnește un vânt orientat spre Sud, el coboară automat pe linia imediat următoare, moment în care reluăm căutarea de pe noua poziție. Cum putem coborî de cel mult N ori înainte de a ieși din matrice, arhitectura generală a soluției va consta într-o colecție de $4 \cdot N$ set-uri pentru direcții și N Arbori de Intervale, câte unul asociat fiecărei linii în parte (sau unul în funcție de implementare). Complexitatea se scalează proporțional, devenind $\mathcal{O}(Q \cdot N \cdot \log M)$.

Subtask 5: $N \leq 10$ și există doar interogări (11 puncte). Lipsa totală a operațiilor de actualizare implică faptul că matricea direcțiilor și cantitățile de mere rămân complet statice. Când rutele nu se schimbă dinamic, putem optimiza masiv simulările deplasărilor lungi folosind tehnica *Binary Lifting*. Vom construi o dinamică tridimensională unde starea $dp[step][i][j]$

este definită ca un tuplu de forma $\{i_{nou}, j_{nou}, mere_colectate\}$, reprezentând locația finală a lui Gigel și numărul de mere strânse după exact 2^{step} pași, pornind din (i, j) .

Pentru a construi recurența, ne bazăm pe principiul că un drum lung se poate sparge perfect în două jumătăți egale. Dacă prima jumătate ne duce într-o stare intermediară (i_{mid}, j_{mid}) care se află încă în matrice, continuăm parcurgerea pentru a obține starea finală. Obținem astfel următoarea recurență matematică:

$$dp[step][i][j] = \begin{cases} \{i_{final}, j_{final}, mere_{mid} + mere_{final}\}, & \text{dacă } (i_{mid}, j_{mid}) \text{ este validă} \\ \{-1, -1, 0\}, & \text{altfel (iese din matrice)} \end{cases}$$

Pentru a răspunde concret la interogări, descompunem numărul total de pași K în baza 2 și compunem drumul final din “salturi” exponențiale, folosind doar biții setați. Complexitatea pentru această cerință este $\mathcal{O}(N \cdot M \cdot \log K + Q \cdot \log K)$.

Subtask 6: $N \leq 10$ și $Q \leq 30\,000$ (27 puncte). Pentru a putea aborda probleme fără limite la nivel structural trebuie să facem două observații esențiale:

Observație 3. Dacă împărțim matricea în blocuri de $N \times B$, unde B este o constantă și precalculăm pentru fiecare poziție (i, j) din acest bloc, dacă am porni să zburăm din această celulă fără limitări asupra lui K :

- (1) Coordonatele de ieșire din bloc (i', j') .
- (2) Numărul de pași K' făcuți până să ieșim din bloc.
- (3) Numărul de mere colectate pe parcurs.

, atunci putem “sări” din bloc în bloc în complexitate de $\mathcal{O}(1)$.

Observație 4. Dacă avem de a face cu un drum R_0 , atunci nu este nevoie să găsim exact punctul în care se închide ciclul prima dată (punctul de intersecție dintre coadă și buclă). Dacă observăm că se închide ciclul în orice altă poziție din buclă, atunci toate calculele matematice de la subtask-ul 2 rămân adevărate.

Avem așadar următoarea strategie: precalculăm pentru fiecare bloc cele trei informații esențiale, iar când avem un eveniment de tip 1, schimbăm direcția celei în memorie și reconstruim toate aceste valori. Putem proceda astfel: pornim de la o celulă (i, j) și simulăm brut zborul lui Gigel prin blocul respectiv. Ținem minte într-o stivă valorile curente ale pașilor făcuți, respectiv merelor colectate pentru fiecare celulă parcursă. La final, folosindu-ne de răspunsul pentru celula inițială, putem calcula foarte ușor răspunsul și pentru toate celulele atinse pe parcurs. Trebuie, desigur, să avem grijă în cazul ciclurilor. Dacă ajungem într-o celulă pe care deja am mai procesat-o, atunci putem să ne oprim, întrucât știm exact din acel punct, cum se comportă zborul nostru. Întrucât fiecare celulă este parcursă exact o singură dată, obținem complexitatea de $\mathcal{O}(N \cdot B)$.

Pentru interogări, trebuie să mai facem o observație:

Observație 5. Valorile celulelor de ieșire dintr-un anumit bloc, dacă Gigel nu iese din matrice, aparțin coloanei de început sau sfârșit a altui bloc. Cum avem $\frac{M}{B}$ blocuri, atunci avem maxim $\frac{2 \cdot N \cdot M}{B}$ de puncte în care putem să ne aflăm după un salt. Deci, conform principiului cutiei lui Dirichlet, dacă sărim de $\frac{2 \cdot N \cdot M}{B} + 1$ ori, atunci sigur suntem blocați într-un ciclu, deci putem să aplicăm aritmetica modulară.

Făcând această observație, simularea zborului va fi următoarea: Plecăm din celula de start, (i, j) și sărim din bloc în bloc având grijă să identificăm eventualele cicluri. Sărim cât timp numărul de pași necesari este $\leq K$. Dacă parcurgem de două ori aceeași graniță, procesăm ciclul format. Când nu mai putem sări integral peste un bloc, putem simula naiv restul pașilor. Obținem complexitatea de $\mathcal{O}(\frac{N \cdot M}{B} + N \cdot B)$, de unde observăm că, de fapt, soluția se comportă cel mai bine când $B = \sqrt{M}$, deci obținem complexitatea finală de $\mathcal{O}(N \cdot \sqrt{M})$.

Pentru a rezolva complet subtaskul, trebuie să gestionăm și actualizările merelor. Spre deosebire de subtask-urile precedente, dacă am folosi un arbore de intervale cu lazy propagation, când ar trebui să recalculăm informațiile pentru un bloc sau pentru a simula brut ultimii pași conținuți într-un bloc, am obține complexitatea de $\mathcal{O}(N \cdot \sqrt{M} \cdot \log M)$, lucru care ar duce la ieșirea soluției din limitele de timp. Pentru a putea implementa eficient și această actualizare vom mai face o observație:

Observație 6. *Dacă știm numărul de pași făcuți de Gigel într-un bloc, numărul de mere cât și o valoare `lazy_bloc`, care reprezintă suma merelor care au fost adăugate la toate celulele din acest bloc, atunci numărul final de mere va fi dat de formula:*

$$\text{mere_final} = \text{mere_initial} + \text{pasi_facuti} \cdot \text{lazy_bloc}.$$

Așadar, când vom avea o actualizare de tipul 2, vom itera prin toate blocurile care sunt complet conținute în intervalul $[l, r]$ și vom crește valoarea lui `lazy_bloc` cu `val` pentru fiecare dintre aceste blocuri în complexitate de $\mathcal{O}(\sqrt{M})$. Acum trebuie să rezolvăm o ultimă problemă: intervalele de actualizare nu se vor alinia perfect cu "granițele" blocurilor noastre. În acest caz, vom avea următoarea structură pentru interval:

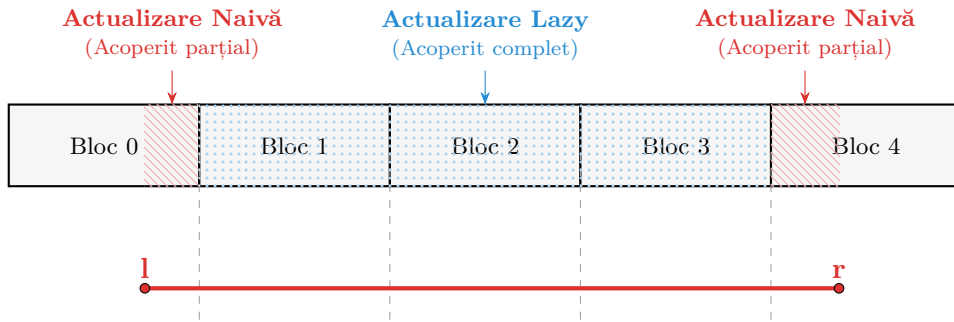


FIGURE 2. Structura unei actualizări de tip 2 pe intervalul $[l, r]$.

Observăm că putem avea maxim *două* blocuri care nu sunt incluse complet. Putem parcurge naiv toate celulele care mai trebuie actualizate și să le actualizăm manual, apoi reconstruim cele 3 valori pentru ambele blocuri. Deci, și pentru actualizările de tipul doi, atingem complexitatea maximă de $\mathcal{O}(N \cdot \sqrt{M})$.

Subtask 7 & 8: $N \leq 10$, respectiv $N \leq 25$ (100 puncte). Pentru a rezolva complet problema, trebuie să extindem soluția de mai sus. Observăm că toate cele 3 tipuri de evenimente sunt clasice pentru un arbore de intervale. Pentru a putea menține ideea de la soluția precedentă, pentru fiecare nod din arbore, care corespunde unui interval $[l, r]$, vom reține exact cele 3 informații de mai sus, anume: pentru fiecare linie de pe coloanele l și r vom calcula:

- (1) Coordonatele de ieșire din bloc (i', j') .
- (2) Numărul de pași K' făcuți până să ieșim din nodul din arbore.
- (3) Numărul de mere colectate pe parcurs.

Vom construi acest arbore *bottom-up*. După ce calculăm răspunsul din frunze (pe care îl putem face în complexitate de $\mathcal{O}(N)$ printr-o simplă simulare), vom parcurge lanțul recursivității și vom combina subarborii corespunzători. Funcția de combinare a doi subarbori reprezintă cea mai complicată parte a implementării acestei soluții. Dacă simulăm efectiv mersul lui Gigel, atunci în cel mai rău caz, pentru fiecare linie vom avea $2 \cdot N$ "sărituri" între subarbori. Definim

o săritură ca fiind o muchie dusă între coloana mid și $mid + 1$ a matricii, sau invers, unde mid reprezintă coloana din mijlocul intervalului, adică chiar frontiera fiilor pe care încercăm să-i combinăm, care ne forțează să schimbăm la fiecare pas subarboarele analizat.

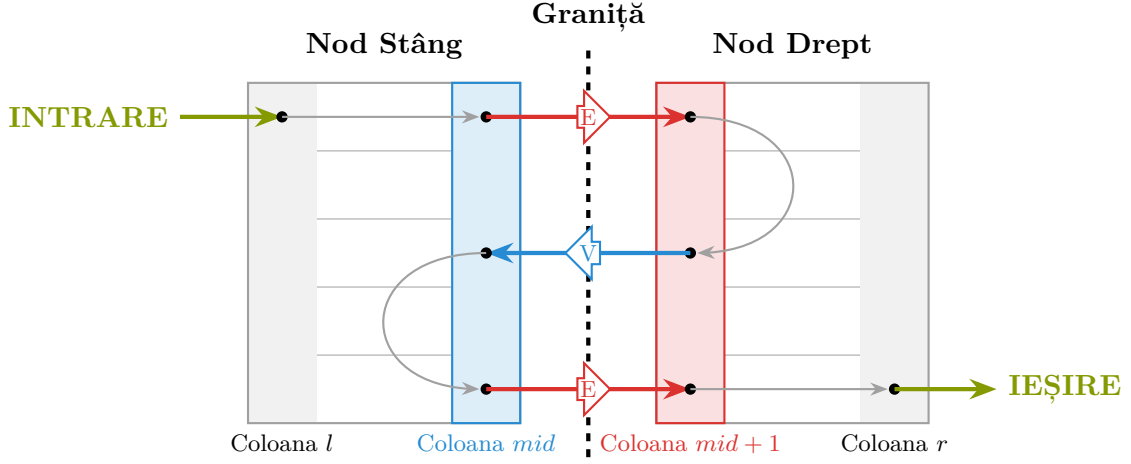


FIGURE 3. Exemplu de traseu parcurs de Gigel între doi subarbori vecini, în acest caz avem 3 "sărituri".

Practic, aceste sărituri reprezintă întoarcerea repetată în același subarbor din care am plecat (Gigel își schimbă direcția frecvent de la E la V și invers).

Acest număr maxim de $2 \cdot N$ este dictat tot de principiul cutiei lui Dirichlet (dacă ar fi fost $2 \cdot N + 1$, atunci sigur am fi închis un ciclu). Obținem o complexitate de $\mathcal{O}(N^2)$ pentru combinarea a doi subarbori. Însă, ne putem folosi de observația 1, și, similar cu subtask-ul anterior, folosim o stivă și memoizăm răspunsul pentru toate liniile și coloanele vizitate. Cu ajutorul memoizării, putem reduce complexitatea la $\mathcal{O}(N)$. De aici tragem concluzia că dacă avem o actualizare de tipul 1, atunci schimbăm în memorie valoarea vântului și facem o actualizare tipică pe un arbore de intervale în complexitate de $\mathcal{O}(N \cdot \log M)$. Deoarece folosim aceeași structură de date pentru ambele actualizări, în acest caz suprapunerile vor fi mereu perfecte. Putem ține, pe lângă informațiile deja stocate, în fiecare nod, și lazy propagation. Când avem actualizare de tipul doi, procedăm ca la un arbore de intervale normal și propagăm adăugarea lui val . Când propagăm trebuie să aplicăm formula de la observația 6 pentru a actualiza corect numărul de mere colectate din fiecare coloană. La final de apel recursiv, combinăm din nou subarborii. Desigur, complexitatea rămâne tot $\mathcal{O}(N \cdot \log M)$.

Acum nu mai trebuie decât să gestionăm eficient interogările. Observăm că dacă suntem la coloana curentă y , atunci dacă găsim un interval din arborele de intervale de forma $[l, y]$ sau $[y, r]$ și numărul de pași K rămas ne permite, atunci putem sări întreg intervalul în $\mathcal{O}(1)$. Vom numi în continuare aceste intervale, *intervale aliniate*. Pentru a obține primul interval aliniat, pornim din rădăcină, și mergem doar în noduri care îl conțin complet pe y , propagând val în caz de nevoie. Acum începem să sărim în $\mathcal{O}(1)$ peste intervale, cât timp $K \leq \text{pasi_facuti_in_subarbor}$. Când K devine mai mic, știm sigur că Gigel rămâne blocat în nodul curent și începem să coborâm recursiv pentru a găsi exact celula unde se oprește. Întrucât între doi frați este prezent fenomenul de "sărituri", trebuie să observăm, că putem avea maxim $2 \cdot N$ sărituri fără a închide un ciclu. Așadar trebuie să forțăm algoritmul să rămână fixat într-un nod până ce se termină toate "săriturile". Acest lucru cuplat cu detecția ciclurilor, reduce complexitatea la $\mathcal{O}(N \cdot \log M)$.

Lemă: Vom coborî în subarbor **doar** pentru prima aliniere sau doar când K nu mai este suficient de mare, formând astfel un lanț de lungime $2 \cdot \log M$.

Demonstrație: Observăm că dacă suntem într-un nod aliniat $[l, r]$ și avem un K suficient de mare, atunci din el putem ieși fie în coloana $l - 1$, fie în $r + 1$. Exact una dintre ele este

”fratele” nodului la care ajungem în $\mathcal{O}(1)$ (noi suntem în părinte și apelăm acest copil pentru care obținem răspunsul în $\mathcal{O}(1)$, deci nu mai coborâm) și simulăm ”săriturile” în $\mathcal{O}(N)$. În celălalt caz, trebuie să **urcăm** pentru a obține un nod aliniat. Cel mai apropiat caz este în părinte, deci, chiar și worst-case, noi urcăm cel puțin 1 nivel înspre rădăcină. Când nu mai avem un K suficient de mare coborâm din nou pentru a finaliza parcurgerea. Cum pentru fiecare nod parcurs putem simula în $\mathcal{O}(N)$ ”săriturile” și, de asemenea, trebuie să îl și propagăm pe *val*, tot în $\mathcal{O}(N)$, am demonstrat complexitatea pentru interogare ca fiind de ordinul $\mathcal{O}(N \cdot \log M)$.

În final, complexitatea optimă pentru această problemă este $\mathcal{O}(N \cdot M + Q \cdot N \cdot \log M)$. Subtask-ul 7 are scopul de a departaja soluțiile cu împărțire în bucăți de radical optimizate suplimentar, cât și cele cu arbori de intervale cu constante mari de implementare sau complexități de $\mathcal{O}(N^2)$ pentru combinarea a doi subarbori sau $\mathcal{O}(N \cdot \log^2 M)$ pentru interogări.

ECHIPA

Problemele pentru această etapă au fost pregătite de:

- Szabó Zoltan, Inspectoratul Școlar Județean, Târgu-Mureș
- Panaete Adrian, Colegiul Național “A.T. Laurian”, Botoșani
- Constantinescu Andrei-Costin, ETH, Zürich, Elveția
- Bogdan Vlad-Mihai, Universitatea București
- Roșu Adrian-Andrei, Universitatea Națională de Știință și Tehnologie Politehnica, București
- Todoran Alexandru-Raul, Harvard University, Cambridge, Massachusetts
- Badea Lucian-Andrei, TU Delft, Olanda
- Verzotti Matteo-Alexandru, Universitatea București
- Ariciu Toma, Universitatea Națională de Știință și Tehnologie Politehnica, București
- Spătărel Dan-Constantin, București
- Ciucu Mihai, CS Academy, București
- Posdarascu Eugenie-Daniel, Youni, București
- Stănescu Matei-Octavian, Universitatea Națională de Știință și Tehnologie Politehnica, București
- Prosie Radu-Teodor, Universitatea București
- Ilie Luca-Mihai, École Polytechnique, Paris, Franța
- Haba Matei-Ionuț, Universitatea Națională de Știință și Tehnologie Politehnica, București
- Marciuc Andrei, Universitatea Națională de Știință și Tehnologie Politehnica, București
- Moca Andrei, Universitatea “Babeș-Bolyai”, Cluj-Napoca
- Nuță Ștefan-Alexandru, Universitatea Națională de Știință și Tehnologie Politehnica, București
- Taschina Luca-Marian, Universitatea “Babeș-Bolyai”, Cluj-Napoca
- Turcu Andrei-Cristian, Universitatea București
- Gheorghies Petruț-Rareș, Universitatea Națională de Știință și Tehnologie Politehnica, București
- Ionescu Matei, Universitatea “Alexandru Ioan Cuza”, Iași
- Ciuleanu Vlad, ETH, Zürich, Elveția