OLIMPIADA NAȚIONALĂ DE INFORMATICĂ, ETAPA NAȚIONALĂ
BARAJ SENIORI
DESCRIEREA SOLUȚIILOR

COMISIA ȘTIINȚIFICĂ

Problema 1: Liars

Propusă de: Eugenie Daniel Posdărăscu, Youni

O să implementăm o dinamică pe arbore care o să calculeze numărul de soluții pentru un subarbore. Spre deosebire de majoritatea dinamicilor pe arbore, de data aceasta o să trebuiască să ținem o stare care să specifice și tipologia nodului părinte (pentru a ști câți din fiii unui nod sunt mincinoși, este necesar să știm dacă părintele acelui nod este mincinos sau nu).

Prin urmare avem următoarea dinamică:

$$\text{dp}[\text{node}][\text{node}_{\text{node}}][\text{state}_{\text{father}}] = \begin{pmatrix} \text{numărul de configurații distincte considerând subarborele nodului node} \\ \text{știind că starea lui node este state}_{\text{node}} \text{ (0/1 dacă este mincinos sau nu),} \\ \text{iar starea părintelui lui node este state}_{\text{father}} \text{ (asemănător, 0/1 dacă este mincinos sau nu).} \end{pmatrix}$$

În funcție de $\text{state}_{\text{father}}$ și de node (răspunsul lui node la întrebarea lui Zoli), putem deduce, dacă node este sincer, câți din fiii lui sunt mincinoși (notăm P din aceștia), iar dacă node este mincinos putem deduce câți din fiii lui sunt ok să fie mincinoși (oricâți dar nu un număr anume, notăm tot P). Prin urmare, dacă notăm cu T numărul de fii a lui node, rămâne să calculăm în câte moduri putem selecta K din acești T fii să fie mincinoși pentru a avea o soluție validă.

Dacă vrem ca un fiu X să fie mincinos o să ne uitam la $\text{dp}[X][0][\text{state}_{\text{node}}]$ (node este tatăl lui X deci starea tatălui lui X este starea lui node), respectiv $\text{dp}[X][1][\text{state}_{\text{node}}]$ dacă vrem să selectăm că e sincer. Pentru a combina rezultate o să implementăm o altă dinamică de tip rucsac în cadrul acestor fii:

$$\text{Rucsac}[i][j] = \begin{pmatrix} \text{numărul de configurații distincte astfel încât din primii } i \\ \text{fii a nodului node,} \\ j \text{ din aceștia sunt mincinoși.} \end{pmatrix}$$

Evident, luăm cele 2 cazuri în care al i-lea este fie sincer, fie mincinos: $\text{Rucsac}[i][j] = \text{Rucsac}[i-1][j] \cdot \text{dp}[\text{fiu}[i]][1][\text{state}_{\text{node}}]$ (din primii $i-1$ fii avem tot j mincinoși deoarece al i-lea este sincer) + $\text{Rucsac}[i-1][j-1] \cdot \text{dp}[\text{fiu}[i]][0][\text{state}_{\text{node}}]$ (din primii $i-1$ fii avem $j-1$ mincinoși deoarece al i-lea este mincinos)

După ce am calculat acest rucsac, rămâne să numărul soluțiile în $\text{dp}[\text{node}]$:

$$\text{dp}[\text{node}][0][\text{state}_{\text{father}}] = \begin{pmatrix} \text{Rucsac}[T][Q], \text{ unde } Q \text{ poate să ia orice valoare} \\ \text{dar } \textit{nu } P \text{ (deoarece node este mincinos)} \end{pmatrix}$$

$$\text{dp}[\text{node}][1][\text{state}_{\text{father}}] = \text{Rucsac}[T][P]$$

Pentru a realiza și reconstituirea, o să ținem o dinamică asemănătoare cu dp, doar că în loc să calculăm numărul total de soluții, o să ținem doar 0/1 dacă există soluție sau nu. Pe baza acestei dinamici și a deciziilor pe care le luăm, putem după să reconstituim o soluție.

Problema 2: Sándor

Propusă de: stud. Alexandru Ispir, Universitatea București

În problemă se consideră șirul inițial $v_1, v_2, \ldots, v_n$ pe care rulăm algoritmul lui Sándor. La finalul algoritmului, doar o submulțime din acest șir a fost adăugată la sumă, fie aceasta $p_1 > p_2 > \cdots > p_k$, pentru care $\sum_{i=1}^{k} p \leq G$. Pentru a rezolva cerința unde $T = 1$ o soluție *forță brută* este eliminarea pe

rând a fiecărui element din șir și aplicarea algoritmului. În continuare vom încerca să reducem numărul de iterații ale algoritmului lui Sándor.

O primă observație care trebuie făcută este că doar eliminarea unui element din subșirul p va influența rezultatul, iar pentru oricare altă tăietură algoritmul va returna în continuare suma S. Astfel reducem soluția la $|p|$ iterații.

Pentru a optimiza în continuare vom face următoarea notare. Comprimăm șirul v în intervale de numere egale de tipul $[x_i, \text{fr}_i]$ unde X_i este valoarea intervalului, iar $[\text{fr}_i]$ este numărul de valori egale cu x_i, iar $x_1 > x_2 > \cdots > x_k$. În cazul în care mai multe elemente din subșirul p sunt egale, eliminarea acestora va rezulta în același șir, deci fiecare valoare distinctă trebuie considerată o singură dată. Întrucât $\sum_{i=1}^{k} p \leq G$ înseamnă că șirul p nu conține mai mult de aproximativ $\sqrt{G}$ valori distincte.

Momentan avem o complexitate $O(N\sqrt{G})$, dar putem reduce complexitatea recalculării răspunsului algoritmului lui Sándor folosindu-ne de scrierea comprimată a șirului după următorul algoritm:

(1) next$[i]$ = indicele poziției primului număr $\leq i$
(2) Cât timp mai putem selecta elemente:
(3) poz = next$[G]$, noua valoare adăugată la sumă
(4) Fie x_{poz} și fr_{poz} valorile intervalului unde se află numărul dorit.
(5) $G = G - \min(\text{fr}_{\text{poz}}, G/x_{\text{poz}}) \cdot x_{\text{poz}}$, adăugăm cât de multe valori x_{poz} putem

Acest algoritm face maxim $\sqrt{G}$ pași din aceleași motive enumerate mai sus (aprox. numărul de elemente distincte selectate de algoritm).

Astfel pentru $T = 1$ am obținut o soluție în complexitate $O(G)$. În continuare vom extinde această soluție pentru $T = 2$.

Pentru $T = 2$, fie prima valoare eliminată p_i. Vom simula algoritmul rapid exact ca la cerința $T = 1$, dar vom păstra un alt șir $m_1, m_2, m_3, \ldots, m_\ell$, șir al valorilor prin care trece algoritmul după eliminarea lui p_i.

În continuare, cum știm că numărul de valori distincte din șirul $m_1, m_2, \ldots, m_\ell$ este tot mai mic decât $\sqrt{G}$, ce vom face este să iterăm prin acest șir asupra celei de-a doua valori eliminate, fie aceste m_j.

Atenție, valoarea m_j trebuie să fie $\leq p_i$, pentru că altfel eliminarea ei ar putea influența apariția lui p_i în șirul de valori.

Ultimul pas, după eliminarea lui p_i și m_j din șirul inițial, este să rulăm algoritmul din nou pentru a vedea ce sumă obține la final. Contribuția acestei sume la rezultat va fi $\text{fr}[p_i] \times \text{fr}[m_j]$ dacă $p_i \neq m_j$, sau $\text{C}(\text{fr}[p_i], 2)$ pentru $p_i = m_j$. Complexitatea finală fiind $G \times \sqrt{G}$.

Problema 3: Tornade

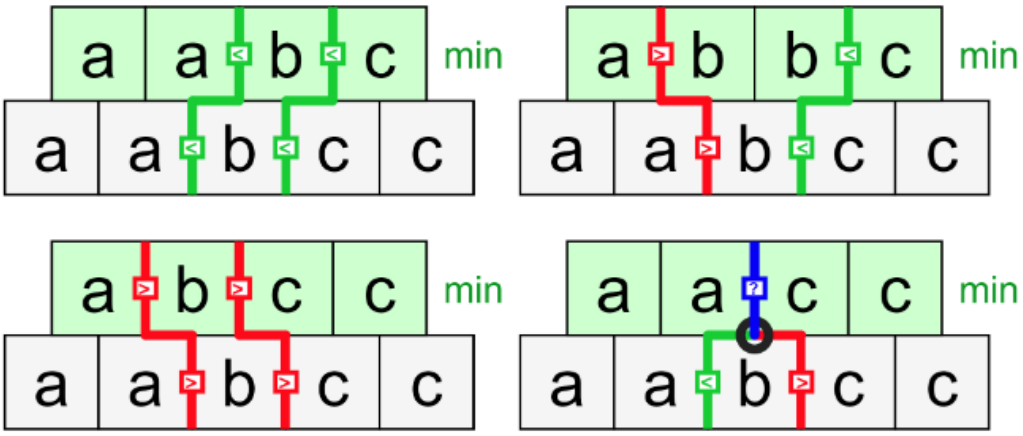
Propusă de: stud. Alexandru-Raul Todoran, Harvard University

Să desenăm un separator colorat între orice două blocuri consecutive cu numere diferite. Separatorul va fi *verde* dacă numărul din stânga este mai mic decât cel din dreapta și *roșu* în caz contrar.

Acum, să presupunem că avem 5 blocuri consecutive cu numerele a, a, b, c, c respectiv. Presupunem că nivelul de deasupra este de tip *min* și analizăm următoarele cazuri:

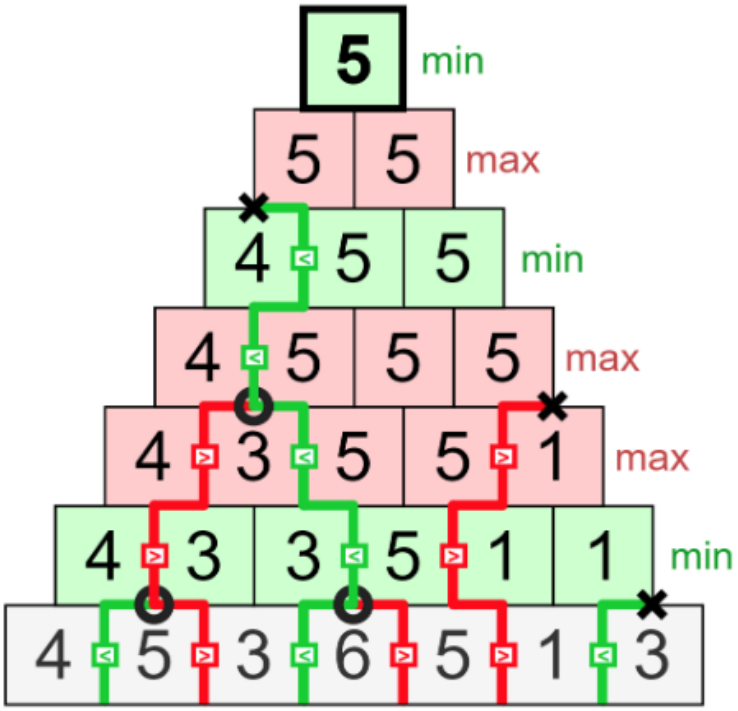
(1) $a < b < c$,
(2) $a > b > c$,
(3) $a < b > c$,
(4) $a > b < c$.

Pentru fiecare caz, configurația rândului de deasupra este ilustrată mai jos:



Observăm că separatorul *verde* se deplasează întotdeauna spre dreapta, iar separatorul *roșu* se deplasează întotdeauna spre stânga. De asemenea, dacă se întâlnesc, aceștia „se luptă”, rezultând un separator care poate fi fie verde, fie roșu (în funcție de ordinea dintre a și c). Putem ghici că pe un rând de tip *max*, separatorul **roșu** se deplasează spre dreapta, iar cel verde spre stânga. Dacă adăugăm o etichetă în dreapta fiecărui rând, pare că separatorii „vor să meargă spre culoarea lor”, fugind în același timp de culoarea opusă.

Mai jos este un exemplu complet. Analizând mișcarea separatorilor, putem să-i grupăm în „șerpi” care se târăsc în sus pe piramidă, fiind șterși atunci când se luptă sau când se lovesc de un perete.



Bun, această reprezentare este interesantă, dar cum ne ajută să rezolvăm problema? Ei bine, ideea-cheie este că orice rând poate fi determinat din setul de separatori dacă știm și numerele din stânga și dreapta pe care aceștia le separă. Mai mult, toți „șerpii” de aceeași culoare se deplasează sincron în aceeași direcție la fiecare pas.

Vom folosi o structură de date (precum un set) pentru a stoca șerpii. Pentru fiecare șarpe avem următoarele informații, valabile permanent:

- Poziția sa inițială (pe primul nivel).
- Culoarea sa (verde sau roșu).
- Numerele din stânga și dreapta pe care le separă.

Observăm că nu păstrăm poziția curentă a unui șarpe, deoarece vrem ca informația sa să rămână statică. Vom determina unde se află un șarpe pe un anumit nivel folosind două variabile: $\text{shift}_{\text{min}}$ și $\text{shift}_{\text{max}}$. Pentru un rând de tip *min*, scădem $\text{shift}_{\text{max}}$ cu 1, iar pentru un rând de tip *max*, scădem $\text{shift}_{\text{min}}$ cu 1.

Acum, vrem să putem detecta când doi șerpi se luptă, deci vom folosi o structură de date care stochează informații despre toate perechile de șerpi consecutivi:

- Poziția inițială a șarpelui din stânga.
- Poziția inițială a șarpelui din dreapta.

În orice moment, distanța dintre doi șerpi consecutivi este dată de expresia:$(\text{init}_r + \text{shift}_{\text{col}_r}) - (\text{init}_\ell + \text{shift}_{\text{col}_\ell})$. Pozițiile inițiale sunt constante, iar diferența $(\text{shift}_{\text{col}_r} - \text{shift}_{\text{col}_\ell})$ poate fi și ea făcută constantă dacă folosim două seturi: unul pentru perechi de tip (min, max), altul pentru perechi de tip (max, min). Astfel, putem păstra perechile sortate crescător după diferența $(\text{init}_r - \text{init}_\ell)$.

Acum suntem gata pentru soluție:

- Parcurgem nivelele de jos în sus.
- Actualizăm valorile shift_{min} și shift_{max} în funcție de culoarea nivelului.
- Cât timp există o pereche de șerpi care se luptă, eliminăm șarpele care pierde și actualizăm structurile de date. Nu uităm nici de șerpii care se lovesc de perete.
- Când ștergem ultimul șarpe, trebuie să verificăm ce număr va ajunge în vârful piramidei (în acel moment, sunt doar două posibilități).

Faptul că la fiecare luptă numărul total de șerpi scade cu 1 ne asigură că avem o complexitate amortizată de $O(n)$, însă folosim și o structură de date care are operații în $O(\log n)$.

Complexitate temporală: $O(n \log n)$.

Problema 4: Cifre

Propusă de: Tamio-Vesa Nakajima

În problema această, se dă o matrice de cifre A_{ij} de N pe N și două șiruri J_t, P_t de lungime K. Două entități, denumite T și C, mișcă un pion, inițial la poziția (i_0, j_0) — la a t-a mutare, dacă pionul este la poziția (i, j), îl mișcă într-un pătrat de mărime $2P_t + 1$ cu centrul la (i, j) entitatea J_t. La finalul procesului, se concatenează cifrele prin care am trecut, și rezultatul procesului este numărul rezultat R. Entitatea T mută în așa fel încât R să fie minim, entitatea C în așa fel încât R să fie maxim. Se cere să se afișeze R modulo 666 013 pentru fiecare (i_0, j_0) inițial.

Prima idee din această problemă este să ținem $v[i][j][t]$, numărul R rezultat dacă se simulează procesul începând de la pasul t și poziția (i, j). Pentru a calcula această valoare, trebuie să găsim valoarea minimă/maximă dintr-un pătrat cu centrul în (i, j). Calcularea cu forță brută a tuturor acestor valori are complexitatea $O(N^4K^2)$. Vom descrie pas cu pas cum optimizăm această abordare.

În primul rând, optimizăm găsirea minimului/maximului. Vom presupune că este cunoscută metoda de a găsi minimele/maximele pe subsecvențe de lungime dată într-o structură lineara. (Pentru mai multe detalii vedeți problema deque de pe infoarena.) Pentru a găsi minimele/maximele pentru toate *pătratele* dintr-o matrice, trebuie efectiv calculate minimele/maximele pe subsecvențe de lungime constantă pe *linii*, apoi minimele/maximele de lungime constantă *a maximelor deja calculate*, pe *coloane*. Această metodă face $O(N^2)$ comparații, deci ne duce actual la o complexitate de $O(N^2K^2)$.

Trebuie să optimizăm acum cât costă să facem o comparație. Ca soluție parțială care ia câteva puncte în plus, observăm că putem *compacta* câte 17 cifre într-un singur număr în format `long long`, îmbunătățind constanta considerabil. În continuare vom optimiza comparația la $O(1)$, ducând la o soluție în complexitate $O(N^2K)$.

Observăm că, în calculul minimelor/maximelor pentru $v[\star][\star][t]$ nu ne folosim efectiv de *valorile* lui $v[\star][\star][t+1]$ — ci doar de (i) ordinea lor relativă, respectiv (ii) valoarea lor modulo 666 013. Așadar putem optimiza complexitatea ținând doar $\text{ord}[\star][t]$, ordinea sortată a lui $v[\star][\star][t]$, $\text{id}[i][j][t]$, al câtelea element este $v[i][j][t]$ în această ordine, și $\text{vmod}[i][j][t] = v[i][j][t] \bmod 666\,013$. Cu doar aceste informații putem calcula minimele/maximele folosind $\text{id}[\star][\star][t]$; apoi recalculăm ord și id folosind ordinea asta sortată în mod asemănător cu radix sort. În final ne dă complexitatea $O(N^2K)$.

Problema 5: Babel

Propusă de: Matteo Verzotti, Tamio-Vesa Nakajima

Rezolvăm mai întâi problema în varianta fără actualizări. În varianta sa cea mai simplă, problema este un exemplu clasic de recursivitate: Avem toate discurile plasate în ordine descrescătoare $(n, n-1, \ldots, 1)$ pe tija 1 și trebuie să mutăm toate discurile pe tija 2.

Definim move_stack(i) ca fiind numărul de mutări necesare pentru a muta un *stack* complet de discuri $(i, i-1, \ldots, 1)$ pe o altă tijă.

Pentru a efectua această operație, trebuie, mai întâi, făcut spațiu pentru discul i să fie mutat. Așadar, ordinea operațiilor este următoarea:

- Mutat *stack-ul* $i-1$ pe o tijă auxiliară,
- Mutat discul i pe tija destinație,
- Mutat *stack-ul* $i-1$ peste discul i.

Rezultă formula:

$$\text{move\_stack}(i) = 2 \cdot \text{move\_stack}(i-1) + 1.$$

Cum move_stack$(1) = 1$, rezultă formula generală move_stack$(i) = 2^i - 1$.

Acum, pentru a rezolva cazul în care distribuția inițială a discurilor este aleatoare, trebuie să regândim recursivitatea. Fie $t[i]$ tija pe care se află discul i, $t[i] \in \{1, 2, 3\}$. Definim funcția SOLVE recursiv conform Algoritmului 1.

Algorithm 1 Recursiv

```
function SOLVE(i, tija_dest)
    if i = 1 then
        return SOLVE(i − 1, tija_dest)
    end if
    return SOLVE(i − 1, 6 − tija_dest − t[i]) + 2^(i−1);
end function
```

Observăm că suma rezultată este, de fapt, o sumă de puteri ale lui 2. Cu alte cuvinte, în timpul parcurgerii, dacă discul i nu este acolo unde trebuie să fie, adună la suma valoarea 2^{i-1}. Deci funcția solve se poate rescrie iterativ conform Algoritmului 2.

Algorithm 2 Iterativ

```
function SOLVE(i, tija_dest)
    suma ← 0
    for i ← n, n − 1, . . . , 1 do
        suma ← suma · 2
        if tija_dest ≠ t[i] then
            suma ← suma + 1
        end if
    end for
    return suma;
end function
```

Pentru subtask-ul cu complexitate $O(N \cdot Q)$, soluția prezentată este suficientă: actualizăm vectorul t în $O(N)$ la fiecare query și calculăm suma în $O(N)$.

Pentru soluția de 100 de puncte, trebuie făcută următoarea observație: Atunci când parcurgem discurile de la n la 1, observăm că fiecare poziție a discului $t[i]$ are un efect („acțiune") asupra sumei, în funcție și de tija de destinație.

General, suma este transformată astfel:

$$\text{suma} = a \cdot \text{suma} + b$$

unde $a = 2$, iar b depinde de tija_dest.

Mai precis:

$$b[\text{tija\_dest}] = \begin{cases} 1, & \text{dacă } t[i] \neq \text{tija\_dest}, \\ 0, & \text{altfel.} \end{cases}$$

De asemenea, tija de destinație se transformă astfel:

$$\text{tija\_dest} \leftarrow \begin{cases} 6 - t[i] - \text{tija\_dest}, & \text{dacă } t[i] \neq \text{tija\_dest}, \\ \text{tija\_dest}, & \text{altfel.} \end{cases} \tag{1}$$

Astfel, transformăm perechea (suma, tija_dest) astfel:

$$(\text{suma}, \text{tija\_dest}) \rightarrow (a \cdot \text{suma} + b[\text{tija\_dest}], f[\text{tija\_dest}]),$$

unde funcția f este dată de (1). Notăm cu F_i transformarea aplicata asupra perechii (suma, tija_dest), de către discul i. Atunci cand parcurgem toate discurile comportamentul poate fi modelat ca o compunere de transformări:

$$(\text{suma}, \sim) = F_1(F_2(F_3(\ldots F_n(0, \text{tija\_dest}) \ldots)))$$

altfel scris:

$$F_{1,n} = F_1 \circ F_2 \circ \ldots \circ F_n$$

Este important de observat că putem calcula $F_{x,y}$ în $O(1)$, unde

$$F_{x,y} = F_x \circ F_{x+1} \circ \cdots \circ F_y,$$

dacă avem un interval $[x, y]$ de lungime len $= y - x + 1$, și $t[i] =$ tija pentru orice $i \in [x, y]$, astfel. Fie $F_{x,y} = (a, b, f)$, atunci:

$$a = 2^{\text{len}}$$

$$f(j) = \begin{cases} j, & \text{dacă } j = \text{tija sau len este par} \\ 6 - j - \text{tija}, & \text{altfel} \end{cases}$$

$$b(j) = \begin{cases} j, & \text{dacă } j = \text{tija sau len este par} \\ 6 - j - \text{tija}, & \text{altfel} \end{cases}$$

Compunerile de transformări pot fi stocate într-un arbore de intervale, permițând actualizări *lazy* pe intervale în $O(\log N)$. Construcția arborelui de intervale: fiecare nod va stoca transformarea echivalentă pentru intervalul asociat lui. Cu alte cuvinte, nodul asociat intervalului $[x, y]$, va stoca transformarea $F_{x,y}$. În funcția de update, se propagă lazy și se apelează recursiv până se ajunge la un interval inclus în cel de update, unde se atribuie nodului respectiv în $O(1)$, transformarea echivalentă descrisă mai sus, pentru interval de tije egale. La întoarcere, se îmbină transformările astfel: fie $F_{\text{st}} = (a, b, f)$ și $F_{\text{dr}} = (a', b', f')$ transformările aferente nodurilor din st, respectiv dreapta. Atunci, rezultatul compoziției este (a'', b'', f''), dat de:

$$a'' = a \cdot a',$$
$$b''(i) = a \cdot b'(i) + b(f'(i)),$$
$$f''(i) = f(f'(i)).$$

Problema 6: Memes

Propusă de: Costin-Andrei Oncescu

Se observă că prin operațiile descrise în enunț, se poate obține orice subșir de lungime N format din numerele șirului v, în care păstrăm ordinea relativă a elementelor. Astfel, vom considera dinamica

$$d[i][j] = \begin{pmatrix} \text{numărul de secvențe de lungime } i \text{ ce se pot obține folosind } v[1], ..., v[j], \\ \text{păstrând ordinea lor relativă, care se termină cu } v[j]. \end{pmatrix}$$

Recurența este:

$$d[i][j] = \sum_{k=\text{last}[j]+1}^{j} d[i-1][k],$$

unde $\text{last}[j]$ este ultima apariție a lui $v[j]$ la o poziție mai mică decât j.

Astfel, obținem soluția în $O(N^2\Sigma)$ timp și $O(N^2)$ memorie, unde Σ este numărul de numere distincte din șir. Făcând observația că pentru a calcula $d[i]$ folosim doar $d[i-1]$, putem păstra doar ultima linie a matricei pentru a reduce memoria la $O(N)$.

Pentru a reduce complexitatea de timp la $O(N^2)$, calculăm suma $\sum_{k=\text{last}[j]+1}^{j} d[i-1][k]$ folosind sume parțiale.

Soluție alternativă. Vom considera dinamica $d[i][c][j]$ = numărul de secvențe de lungime j ce se pot obține folosind $v[1], ..., v[i]$, care se termină cu numărul c. Recurența va fi următoarea:

$$d[i][c][j] = \begin{cases} d[i-1][c][j] & \text{pentru } 1 \le c \le N, c \ne v[i] \\ \sum_{k=0}^{j-1} \sum_{\substack{l=1 \\ l \ne v[i]}}^{N} d[i-1][l][k] & \text{pentru } c = v[i] \end{cases}$$

Vom calcula odată cu aceasta, $\text{total}[i][j] = \sum_{c=1}^{N} d[i][c][j]$, iar recurența devine:

$$d[i][c][j] = \begin{cases} d[i-1][c][j] & \text{pentru } 1 \le c \le N, c \ne v[i] \\ \sum_{k=0}^{j-1} (\text{total}[i-1][k] - d[i-1][v[i]][k]) & \text{pentru } c = v[i] \end{cases}$$

Având în vedere că

$$\begin{aligned} d[i][v[i]][j] &= \sum_{k=0}^{j-1} (\text{total}[i-1][k] - d[i-1][v[i]][k]) \\ &= \sum_{k=0}^{j-2} (\text{total}[i-1][k] - d[i-1][v[i]][k]) + (\text{total}[i-1][j-1] - d[i-1][v[i]][j-1]) \\ &= d[i][v[i]][j-1] + \text{total}[i-1][j-1] - d[i-1][v[i]][j-1], \end{aligned}$$

recurența se reduce la:

$$d[i][c][j] = \begin{cases} d[i-1][c][j] & \text{pentru } 1 \le c \le N, c \ne v[i] \\ d[i][v[i]][j-1] + \text{total}[i-1][j-1] - d[i-1][v[i]][j-1] & \text{pentru } c = v[i] \end{cases}$$

La fel, pentru $\text{total}[i][j]$, avem $\text{total}[i][j] = \text{total}[i-1][j] - d[i-1][v[i]][j] + d[i][v[i]][j]$.

Cum $d[i][c][j]$ și $\text{total}[i][j]$ depind doar de $d[i-1][c][j]$ și $\text{total}[i-1][j]$, putem menține doar ultimul „strat" al dinamicii în memorie, iar cum la pasul i se modifică doar $d[i][v[i]][j]$, complexitatea de timp este $O(N^2)$.

Echipa

Problemele pentru această etapă au fost pregătite de:

- Ariciu Toma, Universitatea Politehnica, București
- Bogdan Vlad-Mihai, Universitatea București
- Ciortea Liviu, Pexabit, București
- Ciucu Mihai, C.S. Academy, Bucureşti
- Constantinescu Andrei-Costin, ETH Zurich
- Feodorov Andrei, ETH Zurich
- Gavrilă-Ionescu Vlad-Alexandru, Google, Zurich
- Ignat Alex-Matei, Universitatea "Babeş-Bolyai", Cluj-Napoca
- Ispir Alexandru, Universitatea Bucureşti
- Ivan Andrei-Cristian, Universitatea Politehnica, București
- Măgureanu Livia, Universitatea București
- Nakajima Tamio-Vesa, Department of Computer Science, University of Oxford
- Nicoli Marius, Colegiul Național „Frații Buzești", Syncro Soft, Craiova
- Oncescu Costin-Andrei, Harvard University

- Peticaru Alexandru, Universitatea Politehnica, București
- Popa Bogdan Ioan, Universitatea București
- Popescu Adrian Andrei, Universitatea Politehnica, București
- Posdărăscu Eugenie Daniel, Youni, București
- Stănescu Matei-Octavian, Universitatea Politehnica, București
- Szabó Zoltan, Inspectoratul Școlar Județean, Târgu-Mureș
- Todoran Alexandru-Raul, Harvard University
- Tinca Matei, VU Amsterdam
- Verzotti Matteo-Alexandru, Universitatea București