Programare cu premeditare

Cătălin Frâncu

Prefață

Aici voi spune ceva introductiv.

- Problemele sînt ordonate după dificultate.
- Pentru Codeforces și Kilonova aveți nevoie de un cont pentru a vedea sursele.

Cuprins

Partea I

Structuri de date pe vectori

Următoarele capitole tratează structuri de date care pot procesa anumite operații pe vectori în timp mai bun decît $\mathcal{O}(N)$. Ocazional aceste structuri se aplică și matricilor.

Subiectele de ONI / baraj ONI / lot din anii trecuți abundă în probleme rezolvabile cu astfel de structuri:

- 3dist (baraj ONI 2022)
- 6 de Pentagrame (lot 2024)
- Babel (baraj ONI 2025)
- Balama (baraj ONI 2024)
- Bisortare (ONI 2021)
- Circuit (lot 2025)
- Emacs (baraj ONI 2021)
- Erinaceida (lot 2022)
- Guguștiuc (baraj ONI 2022)
- Împiedicat (baraj ONI 2023)
- Lupușor (ONI 2022)
- Medwalk (lot 2025)
- Perm (baraj ONI 2024)
- Piezișă (baraj ONI 2022)
- Subiectul III (lot 2024)
- Șirbun (baraj ONI 2023)
- Trapez (lot 2025)

Pare o idee bună să le învățăm și să le stăpînim bine. 🙂 Concret, vom studia trei structuri:

1. arbori de intervale;
2. arbori indexați binar;
3. descompunere în radical.

Vom exemplifica structurile și vom face benchmarks pe două probleme didactice. Apoi vom vedea, prin probleme, cum putem extinde aceleași structuri pentru nevoi mai complicate.

Varianta 1 (actualizări punctuale, interogări pe interval): Se dă un vector de N elemente întregi și Q operații de două tipuri:

1. $\langle 1, x, val \rangle$: Adaugă val pe poziția x a vectorului.
2. $\langle 2, x, y \rangle$: Calculează suma pozițiilor de la x la y inclusiv.

Varianta 2 (actualizări pe interval, interogări pe interval): Similar, dar operația 1 este pe interval:

1. $\langle 1, x, y, val \rangle$: Adaugă val pe pozițiile de la x la y inclusiv.
2. $\langle 2, x, y \rangle$: Calculează suma pozițiilor de la x la y inclusiv.

Vom menționa ocazional și **Varianta 3 (actualizări pe interval, interogări punctuale)**:

1. $\langle 1, x, y, val \rangle$: Adaugă val pe pozițiile de la x la y inclusiv.

2. $\langle 2, x \rangle$: Returnează valoarea poziției x.

Toate implementările mele sînt disponibile pe GitHub.

Capitolul 1

Arbori de intervale

Arborii de intervale[1] (AINT) sînt o structură foarte puternică și flexibilă. Ușurința implementării depinde de natura operațiilor pe care dorim să le admitem.

1.1 Reprezentare

Ca multe alte structuri (heap-uri, AIB, păduri disjuncte), arborii de intervale se reprezintă pe un simplu vector. Ei sînt arbori doar la nivel logic, în sensul că fiecare poziție din vector are o altă poziție drept părinte.

Pentru început, să presupunem că vectorul dat are $n = 2^k$ elemente. Atunci vectorul necesar S are $2n$ elemente, în care cele n elemente date sînt stocate începînd cu poziția n. Apoi,

- Cele $n/2$ elemente anterioare stochează valori agregate (sume, minime, xor etc.) pentru perechi de valori din vectorul dat.
- Cele $n/4$ elemente anterioare stochează valori agregate pentru grupe de 4 valori din vectorul dat.
- ...
- Elementul $S[1]$ stochează valoarea agregată a întregului vector.
- Valoarea $S[0]$ rămîne nefolosită.

Iată un exemplu pentru $n = 16$. Datele de la intrare se regăsesc pe pozițiile 16-31.

[1] Există o inversiune între nomenclatura internațională și cea românească. Internațional, structura pe care o învățăm astăzi se numește segment tree, iar interval tree este o structură diferită, care stochează colecții de intervale. Cîțiva ani am înotat împotriva curentului și am fost (posibil) singurul român care se referea la această structură ca „arbori de segmente". În acest curs am adoptat și eu denumirea încetățenită.

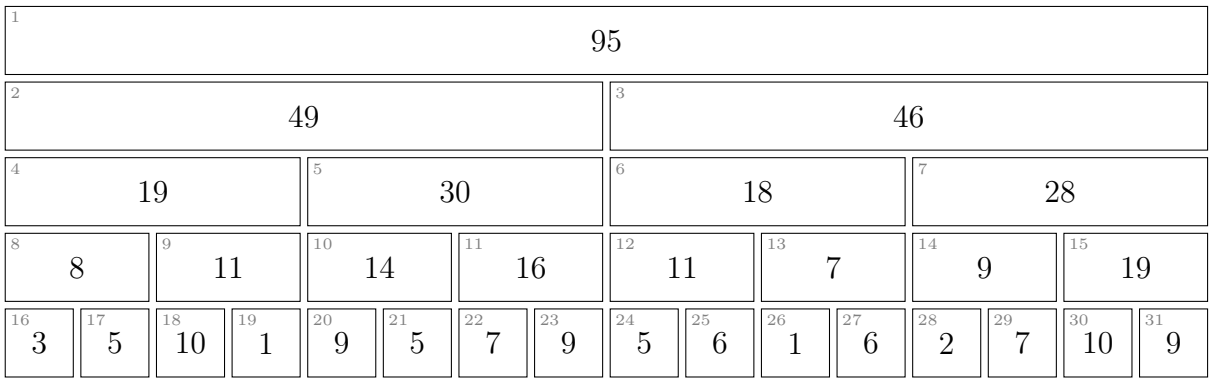


Figura 1.1: Un arbore de intervale cu 16 frunze și 15 noduri interne. Valorile din fiecare celulă reprezintă suma din frunzele subîntinse de acea celulă. Cu cifre mici este notat indicele fiecărei celule.

Facem cîteva observații preliminare:

- Fiii unui nod i sînt $2i$ și $2i + 1$.
- Părintele lui i este $\lfloor i/2 \rfloor$.
- Toți fiii stîngi au numere pare și toți fiii drepți au numere impare.

De exemplu, fiii lui 6 sînt 12 și 13, iar fiii acestora sînt respectiv 24-25 și 26-27. Aceasta corespunde cu intenția noastră ca 6 stocheze informații agregate (suma) despre nodurile 24-27.

După cum vom vedea în secțiunea următoare, arborii de intervale obțin timpi logaritmici pentru operații, deoarece numărul de niveluri este $\log n$.

1.1.1 Memoria necesară

În această formă, structura necesită $2n$ memorie pentru n elemente dacă n este putere a lui 2 sau foarte aproape. De exemplu, pentru $n = 1024$, sînt necesare 2048 de celule. Dar, dacă n depășește cu puțin o putere a lui 2, atunci el trebuie rotunjit în sus. Pentru $n = 1025$, baza arborelui necesită 2048 de celule, iar arborele în întregime necesită 4096 de celule. De aceea spunem că, în cel mai rău caz, arborele poate ajunge la $4n$ celule ocupate în cel mai rău caz.

În realitate, necesarul este doar de $3n$ cu puțină atenție la alocare. Pentru $n = 1025$, alocăm 2048 de celule pentru nivelurile superioare ale arborelui, dar putem aloca fix 1025 pentru bază (nu 2048). Totalul este circa $3n$.

Pentru a calcula următoare putere a lui 2, putem folosi bucla naivă:

```
int p = 1;
while (p < n) {
  p *= 2;
}
n = p;
```

Sau o buclă care folosește *bit hacks*:

```
while (n & (n - 1)) {
  n += n & -n;
}
```

Mai concis, putem folosi funcția `__builtin_clz(x)`, care ne spune cu cîte zerouri începe numărul `x`:

```
n = 1 << (32 - __builtin_clz(n - 1));
```

1.1.2 Reprezentări alternative

Există și reprezentări mai compacte, care ocupă exact $2n-1$ noduri, adică strictul necesar teoretic. Iată un exemplu pentru un vector cu 6 noduri.

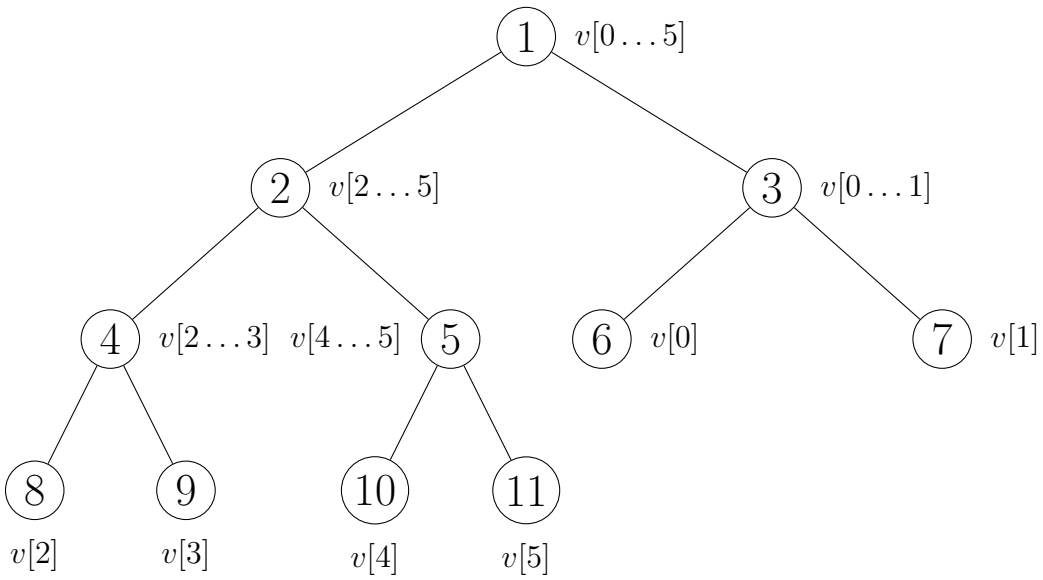


Figura 1.2: Reprezentarea arborilor de intervale cu exact $2n-1$ noduri.

Vedem că frunzele (adică vectorul dat, $v[0] \dots v[5]$) se află pe pozițiile consecutive 6-11. În schimb, această reprezentare pare mai greu de vizualizat și încalcă o abstracție importantă: frunzele nu mai sînt la același nivel. Structura se pretează la operațiile de actualizare și interogare, dar nu sînt sigur că se pretează și la restul operațiilor pe care le discutăm în secțiunile următoare. De aceea prefer să folosesc și să predau structura rotunjită la 2^k noduri.

1.2 Operații elementare

1.2.1 Actualizarea punctuală

Nu uitați că poziția i din datele de intrare este stocată efectiv în $s[n+i]$. Apoi, cînd elementul aflat pe poziția i primește valoarea val, toate nodurile care acoperă poziția i trebuie recalculate:

```
void set(int pos, int val) {
```

```
  pos += n;
  s[pos] = val;
  for (pos /= 2; pos; pos /= 2) {
    s[pos] = s[2 * pos] + s[2 * pos + 1];
  }
}
```

Dacă nu ni se dă noua valoare absolută, ci variația δ față de valoarea anterioară, atunci codul este chiar mai simplu, căci toți strămoșii poziției se modifică tot cu δ:

```
void add(int pos, int delta) {
  for (pos += n; pos; pos /= 2) {
    s[pos] += delta;
  }
}
```

Apropo de *clean code*: Remarcați că am denumit funcțiile `set` și `add`, nu le-am denumit pe ambele `update`. Astfel am evidențiat diferența dintre ele.

1.2.2 Construcția în $\mathcal{O}(n \log n)$

O variantă de construcție este să invocăm funcția set de mai sus pentru fiecare valoare de la intrare. Complexitatea va fi $\mathcal{O}(n \log n)$.

1.2.3 Construcția în $\mathcal{O}(n)$

Putem reduce timpul de construcție dacă doar inserăm valorile frunzelor, fără a le propaga la strămoși. La final calculăm foarte simplu nodurile interne, în ordine descrescătoare.

```
void build() {
  for (int i = n - 1; i >= 1; i--) {
    s[i] = s[2 * i] + s[2 * i + 1];
  }
}
```

1.2.4 Calculul sumei pe interval

Să calculăm suma pe intervalul original [2, 12], care corespunde intervalului [18, 28] din reprezentarea internă. Ideea este să descompunem acest interval într-un număr logaritmic de segmente, mai exact [18,19], [20,23], [24,27] și [28,28]. Avantajul descompunerii este că avem deja calculate sumele acestor intervale, respectiv în nodurile 9, 5, 6 și 28.

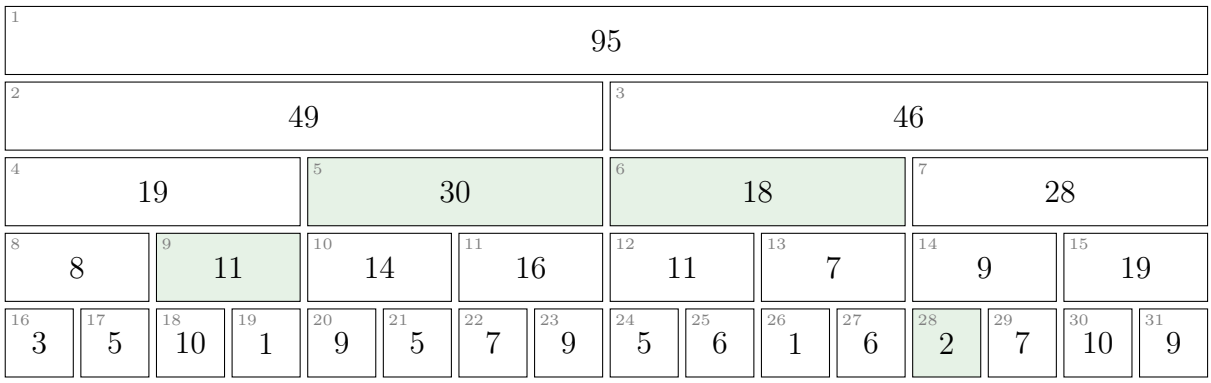


Figura 1.3: Suma intervalului [18, 28] este egală cu suma valorilor nodurilor 9, 5, 6 și 28.

Pornim cu doi pointeri l și r din capetele interogării date. Apoi procedăm astfel:

- Dacă l este fiu stîng, putem aștepta ca să includem un strămoș al său, care va include și alte poziții utile. În schimb, dacă l este fiu drept, trebuie să îl includem în sumă, căci orice strămoș al său va include și elemente inutile din stînga lui l. Apoi avansăm l spre dreapta.
- Printr-un raționament similar, dacă r este fiu stîng, includem valoarea sa în sumă și avansăm r spre stînga.
- Urcăm pe nivelul următor prin înjumătățirea lui l și r.
- Continuăm cît timp $l \leq r$.

Astfel, vom selecta cel mult două intervale de pe fiecare nivel al arborelui și vom restrînge corespunzător intervalul dat, pînă cînd îl reducem la zero. De aici rezultă complexitatea logaritmică.

```
long long query(int l, int r) { // [l, r] închis
  long long sum = 0;

  l += n;
  r += n;

  while (l <= r)  {
    if (l & 1) {
      sum += s[l++];
    }
    l >>= 1;

    if (!(r & 1)) {
      sum += s[r--];
    }
    r >>= 1;
  }

  return sum;
}
```

Clarificare: la ultimul nivel, dacă $l = r$, atunci $s[l]$ va fi selectat exact o dată, fie datorită lui l,

fie datorită lui r, după cum poziția este impară sau pară.

1.2.5 Căutarea unei sume parțiale

Ca și la AIB-uri, dacă toate valorile sînt pozitive are sens întrebarea: pe ce poziție suma parțială atinge valoarea P? Pentru simplitate, recomand să adăugați o santinelă de valoarea infinită pe poziția n. Aceasta garantează că suma parțială se atinge întotdeauna, iar dacă răspunsul este n, atunci de fapt suma parțială nu există în vectorul fără santinelă.

```
int search(int sum) {
  int pos = 1;

  while (pos < n) {
    pos *= 2;
    if (sum > s[pos]) {
      sum -= s[pos++];
    }
  }

  return pos - n;
}
```

1.2.6 Căutarea într-un arbore de maxime

Dat fiind un vector v cu n elemente, ni se cere să răspundem la interogări de tipul $\langle pos, val \rangle$ cu semnificația: găsiți cea mai mică poziție $i > pos$ pe care se află o valoare $v[i] > val$. În secțiunea următoare vom vedea problemele Points și Împiedicat care au această nevoie.

Pentru rezolvare, să construim peste acest vector un arbore de intervale de maxime. Fiecare nod stochează maximul dintre cei doi fii ai săi. Ca urmare, fiecare nod stochează maximul dintre frunzele pe care le subîntinde. Atunci soluția constă din doi pași:

- Mergi la dreapta și în sus, similar pointerului l din operația de sumă pe interval prezentată anterior. Oprește-te cînd ajungi la un nod cu o valoare $> val$. Știm că acest nod subîntinde cel puțin o frunză de valoare $> val$.
- Din acest nod, coboară în fiul care are la rîndul său o valoare $> val$. Dacă ambii fii au această proprietate, coboară în fiul stîng. Oprește-te cînd ajungi la o frunză.

Pentru a simplifica codul, putem adăuga o santinelă infinită la finalul vectorului, ca să ne asigurăm că problema are soluție.

```
int find_first_after(int pos, int val) {
  pos += n + 1;

  while (v[pos] <= val) {
    if (pos & 1) {
```

```
      pos++;
    } else {
      pos >>= 1;
    }
  }

  while (pos < n) {
    pos = (v[2 * pos] > val) ? (2 * pos) : (2 * pos + 1);
  }

  return pos - n;
}
```

Am inclus acest algoritm, deși este rar întîlnit în pratică, pentru a ilustra flexibilitatea uriașă a arborilor de intervale.

1.2.7 Adaptarea la alte tipuri de operații

Aceeași structură de date poate răspunde la multe alte feluri de actualizări și interogări. Nu detaliem aici, vom studia probleme. Ce este important este să ne dăm seama ce stocăm în fiecare nod și cum combină părintele informațiile din cei doi fii.

1.2.8 Implementarea recursivă

Există și o implementare recursivă, pe care nu o vom discuta acum (o menționez doar ca să o fac de rîs). O vom discuta mai tîrziu în acest capitol. Este păcat că mulți elevi învață și stăpînesc doar acea implementare, pe care o aplică și cînd nu este nevoie de ea, deși implementarea iterativă de mai sus este de 2-3 ori mai rapidă. Implementarea iterativă ar trebui să fie implementarea voastră de referință oricînd este suficientă.

Exemplu: din implementarea iterativă rezultă imediat că:

1. Complexitatea este $\mathcal{O}(\log n)$, întrucît l și r urcă exact un nivel la fiecare iterație.
2. De pe fiecare nivel selectăm cel mult două intervale.

Vă urez succes să demonstrați aceste lucruri în implementarea recursivă. 😈

1.3 Probleme

1.3.1 Problema Xenia and Bit Operations (Codeforces)

enunț • sursă

Problema este simplisimă. O includ doar ca exemplu de arbore care face operații diferite pe niveluri diferite.

1.3.2 Problema Distinct Characters Queries (Codeforces)

enunț • sursă

Există diverse abordări pentru această problemă. Una este să construim un AIB sau un AINT pentru fiecare caracter, cu memorie totală $\mathcal{O}(\Sigma n)$ (tradițional Σ denotă mărimea alfabetului). Fiecare structură reține pozițiile pe care apare un caracter. Modificările sînt simple: debifăm poziția în AIB-ul corespunzător vechiului caracter și o marcăm în AIB-ul noului caracter. Pentru interogări, verificăm pentru fiecare din cele 26 de caractere dacă suma pe intervalul dat este non-zero. Rezultă o complexitate de $\mathcal{O}(\Sigma q \log n)$.

Dar iată și o soluție mai elegantă, care reduce complexitatea la $\mathcal{O}(q \log n)$, folosind paralelismul nativ pe 32 de biți al procesorului. Vom folosi 26 de biți din fiecare întreg, cîte unul pentru fiecare caracter. Într-o frunză care stochează litera 'f' vom seta pe 1 doar cel de-al șaselea bit, așadar valoarea întreagă va fi `000...000100000`. Apoi, un nod intern va stoca OR-ul pe biți al frunzelor din intervalul acoperit. Acest gen de informație se numește **mască de biți** (engl. *bitmask*).

Ce semnifică acest OR pe biți? Fiecare dintre biți va fi 1 dacă și numai dacă litera corespunzătoare apare cel puțin o dată în intervalul acoperit. Să observăm că bitul 6 va fi 1 indiferent dacă intervalul conține un caracter 'f' sau multiple caractere 'f'. Rezultă că fiecare mască va avea atîția biți setați (biți 1) cîte caractere distincte există în interval.

Facem actualizări în acest arbore înlocuind masca din frunză și propagînd valoarea spre strămoși cu operația OR. Pentru a răspunde la interogări,

- colectăm cele $\mathcal{O}(\log n)$ măști care compun interogarea;
- le combinăm cu OR;
- numărăm biții din rezultat, de exemplu cu funcția `__builtin_popcount`.

1.3.3 Problema K-query (SPOJ)

enunț • surse

Problema fiind offline, este destul de natural să ordonăm interogările. Sper să vă obișnuiți și voi să luați în calcul această posibilitate.

Ordonarea după capătul stîng sau drept nu pare să ducă nicăieri. Exemplu: ordonăm interogările după capătul drept dr. Atunci, după ce adăugăm elementul $a[dr]$ la structura noastră (oricare ar fi ea), trebuie să răspundem la interogări de tipul: cîte numere $> k$ există începînd cu poziția st? Eu nu am reușit să găsesc o structură echilibrată care să răspundă la întrebări. Poate voi reușiți?

În schimb, ordonarea descrescătoare după valoare duce la o soluție relativ directă. Pentru o interogare (st, dr, k), marcăm (cu 1) într-o structură de date toate pozițiile elementelor mai mari decît k. Apoi numărăm valorile 1 din intervalul $[st, dr]$.

Pentru a găsi rapid toate elementele mai mari decît k (care nu au fost deja inserate în structură), rezultă că trebuie să sortăm și vectorul în ordine descrescătoare, reținînd și poziția originală a

fiecărei valori.

În fapt, putem implementa această soluție chiar și cu un AIB. Sursa este identică cu cea bază pe arbori de intervale cu excepția `struct`-ului. În acest caz, timpii de rulare sînt aproape egali, dar în general vă recomand să folosiți AIB unde se poate.

1.3.4 Problema Sereja and Brackets (Codeforces)

enunț • sursă

Iată și o problemă pentru a cărei rezolvare este mai puțin clar că ne ajunge un arbore de intervale. Vom construi un arbore în care nodurile stochează valori mai complexe care se combină după reguli speciale.

Să considerăm o subsecvență contiguă. Din ce constă ea? Dintr-un subșir (pe sărite) care este bine format, plus niște paranteze deschise neîmperecheate, plus niște paranteze închise neîmperecheate. De exemplu, în subșirul `))) ( () ( () ( () (` am evidențiat cu bold cele 6 caractere bine formate. Rămîn 4 paranteze deschise și 3 închise. Să notăm aceste cantități cu f (lungimea subșirului bine format), d (surplusul de paranteze deschise) și i (surplusul de paranteze închise).

Cum combinăm două subsecvențe adiacente (f_1, d_1, i_1) și (f_2, d_2, i_2)? Clar putem concatena porțiunile bine formate. Dar mai mult, putem prelua și $\min(d_1, i_2)$ perechi dintre surplusurile de paranteze deschise, respectiv închise. Șirul rezultat va fi bine format. Ne putem convinge de asta eliminînd porțiunile bine formate f_1 și f_2, ca și cînd ele nu ar exista. Dacă nu sînteți convinși, puteți apela la o definiție echivalentă pentru un șir de paranteze bine format: pentru orice prefix, diferența dintre numărul de paranteze deschise și închise este pozitivă.

Rezultă că intervalul concatenat va avea parametrii:

- $f = f_1 + f_2 + 2\min(d_1, i_2)$
- $d = d_1 + d_2 - \min(d_1, i_2)$
- $i = i_1 + i_2 - \min(d_1, i_2)$

Construcția arborelui se face ca de obicei, combinînd fiii doi cîte doi. La interogare este nevoie de puțină atenție pentru a colecta și combina intervalele în ordinea corectă (de la stînga la dreapta). Ne bazăm pe observația că operația de compunere nu este comutativă, dar este asociativă.

1.3.5 Problema Copying Data (Codeforces)

enunț • sursă

Aici întîlnim o formă complementară a arborilor de intervale: actualizări pe interval și interogări punctuale (*range update, point query*). Mecanismul necesar folosește o reprezentare puțin diferită. O problemă foarte similară este Range Update Queries (CSES).

(Cei dintre voi care stăpînesc arborii de intervale cu propagare *lazy* vor fi tentați să se repeadă la aceia: Pe fiecare nod ținem informația *lazy* că segmentul din b a fost suprascris cu un segment

din a începînd de la o poziție p (sau cu o deplasare $\pm p$, cum preferați). La actualizări, propagăm informația la fii după nevoie. La interogare, propagăm informația pînă în frunza cerută, pentru a afla de unde provine. Dar nu este nevoie de aceste complicații.)

Să pornim de la observația de bun simț: Dacă o copiere acoperă o poziție, atunci la descompunerea sa în intervale, unul dintre acele intervale va fi strămoș al poziției poziția (*duh!*).

Ne vom folosi și de numerele de ordine ale interogărilor, care vor funcționa ca niște momente de timp între 1 și q. Acum, să construim un arbore de intervale care, pentru o operație de copiere (x, y, k):

- Descompune intervalul $[y, y + k - 1]$ prin metoda obișnuită.
- Notează pe fiecare interval momentul t și diferența $x - y$.

Dacă ulterior o altă copiere va acoperi unul dintre aceste intervale, vom nota acolo momentul t' și diferența $x' - y'$. Atunci ultimul moment (și, implicit, ultima proveniență) a suprascrierii unei poziții este dată de cel mai mare moment de timp **dintre toți strămoșii poziției**.

1.3.6 Problema PHF (FMI No Stress 2013)

enunț • sursă

Problema ne cere să simulăm un șir de meciuri de piatră-hîrtie-foarfecă de tip „cîștigătorul la masă" și să admitem actualizări punctuale pe acest șir. Deoarece nu ne permitem o simulare în $\mathcal{O}(n)$ pentru fiecare din cele q actualizări, vom căuta să accelerăm simularea la $\mathcal{O}(\log n)$.

Caracterul de pe fiecare poziție, să-i spunem X, este un meci între X și cîștigătorul meciului de pe poziția anterioară. Echivalent, X este o funcție definită pe mulțimea $\{P, H, F\}$ cu valori tot în $\{P, H, F\}$, unde $X(c)$ este chiar rezultatul unui meci între X și c. De exemplu, P este funcția:

$$\begin{cases} P(P) &= P \\ P(H) &= H \\ P(F) &= P \end{cases}$$

Atunci o înșiruire de caractere este o compunere de funcții. De exemplu, dintr-un șir de intrare de patru caractere, numite generic $XYZT$, îl tratăm pe X ca argument, iar rezultatul final este $T(Z(Y(X)))$ sau $(T \circ Z \circ Y)(X)$.

Orice funcție are nevoie de un argument. 🙂 De aceea, tratăm separat primul caracter, iar pe celelalte $n-1$ le punem într-o structură. (O altă abordare este să definim primul caracter ca pe o funcție care returnează acel caracter independent de intrarea fictivă). Această structură trebuie să mențină rezultatul compunerii caracterelor, cu modificări. Vom folosi un arbore de intervale unde informația dintr-un nod este funcția compusă a intervalului subîntins. Reprezentăm aceste funcții prin tabelul complet (trei valori). Tabelele frunzelor le definim manual, iar tabelul unui nod intern este compunerea tabelelor celor doi fii. Tabelul rădăcinii este ceea ce ne interesează:

compunerea pozițiilor $2 \dots n$ din șir, adică o funcție pe care o vom aplica primului caracter din șir.

Implementarea mea rotunjește numărul de noduri la o putere a lui 2. De aceea la dreapta vom avea și noduri vide, pe care le tratăm ca pe funcții identice ($X(c) = X$).

1.3.7 Problema Points (Codeforces)

enunț • sursă

Problema are rating de 2800 pentru că se compune din multe blocuri, dar niciunul nu este de speriat, căci sîntem deja versați în arbori de intervale. 😎 Aș zice că problema ar fi grea la un baraj ONI sau ușoară la lot.

Ca să putem construi un arbore de intervale, în primul rînd normalizăm coordonatele x. Păstrăm și o tabelă cu valorile originale, căci pe acelea trebuie să le afișăm.

Am putea reformula întrebarea pentru operația `find x y`: dintre toate punctele cu $x' > x$, există vreunul cu $y' > y$? Ne gîndim că am putea folosi un AINT de maxime, indexat după x, cu valori din y, cu interogarea: „Caută maximul pe intervalul $[x + 1, n)$ și spune-mi dacă este mai mare decît y”.

Dar astfel aflăm doar dacă există un punct. Ca să-l găsim, întrebarea corectă este: „dă-mi cea mai din stînga poziție după x pe care maximul depășește y”. Din fericire, putem face asta cu același AINT maxime, așa cum am explicat în secțiunea de teorie:

1. Pornind de la prima poziție validă (în cazul nostru, $x + 1$), mergem în sus și spre dreapta, spre intervale tot mai mari, pînă cînd găsim o poziție de valoare $> y$. Ca să evităm cazurile particulare, adăugăm la finalul vectorului o santinelă de valoare infinită.
2. De la această poziție, coborîm în timp ce menținem în vizor valoarea $> y$. Dacă putem coborî în orice direcție, preferăm stînga.

Astfel putem gestiona operațiile de adăugare (cînd maximul pentru un x fixat poate doar să crească). Următoarea întrebare este cum gestionăm ștergerile. Cea mai directă soluție este să menținem cîte un `set` STL pentru fiecare coordonată x. Suma mărimilor acestor seturi nu va depăși n. Cu metoda `rbegin()` putem afla noul maxim după inserări și ștergeri.

Ultima întrebare, odată ce stabilim că răspunsul pentru `find x y` este la abscisa x', este: care dintre punctele cu această abscisă este răspunsul? Folosim același set și metoda `upper_bound()` pentru a afla cel mai mic y' strict mai mare decît y.

Complexitatea soluției este $\mathcal{O}(n \log n)$, atît pentru normalizarea inițială cît și pentru procesarea operațiilor. Fiecare operație necesită o căutare în set și o căutare sau actualizare în AINT.

1.3.8 Problema Medwalk (Lot 2025)

enunț • sursă

Problema admite și o soluție diferită, mult mai rapidă, bazată pe AIB-uri 2D, dar iată o soluție care folosește doar arbori de intervale.

Din enunț putem defini forma drumului: el va merge pe linia de sus a unor coloane, apoi va folosi ambele linii de pe o coloană c pentru a coborî, apoi va merge pe linia de jos a coloanelor rămase. Acum, să presupunem că avem un oracol care, pentru orice interogare, ne spune coloana c. Atunci vom muta restul coloanelor fie în stînga, fie în dreapta lui c, pentru a folosi valoarea de sus sau de jos, oricare este mai mică.

Cu alte cuvinte, mulțimea de valori de pe drumul care minimizează medianul constă din

- minimele de pe toate coloanele;
- minimul dintre maximele de pe coloane.

Răspunsul la fiecare interogare este elementul median al acestei mulțimi. Logica pentru a afla a k-a valoare este relativ simplă și implică trei valori: al k-lea minim, al $k-1$-lea minim și minimul maximelor. De aici înainte, putem abstractiza matricea ca doi vectori, unul cu minimele perechilor și altul cu maximele. De exemplu, cînd o coloană se modifică din (3, 6) în (3, 2), atunci minimul se modifică din 3 în 2, iar maximul din 6 în 3.

De aceea, avem nevoie de două structuri independente:

- O structură pentru maxime, care să admită actualizări punctuale și interogare de minim pe interval.
- O structură pentru minime, care să admită actualizări punctuale și interogări de al k-lea element pe interval.

Pentru prima structură, ochiul nostru de-acum experimentat ne spune că putem folosi un simplu AINT. Dar pentru a doua? Am găsit pe StackOverflow o idee bine explicată, pe care o reiau.

Vom folosi un arbore de intervale **pe valori**. Așadar, nu indexăm pozițiile conform cu pozițiile din vector, ci cu valorile existente în vector. Fiecare frunză din aint, corespunzătoare unei valori v, reține o colecție ordonată (un `set`, în esență) cu pozițiile pe care apare valoarea v. Fiecare nod intern reține reuniunea colecțiilor fiilor săi. Cu alte cuvinte, dacă un nod subîntinde valorile $[l, r]$, colecția sa va enumera toate pozițiile pe care apar valori între l și r.

Remarcăm că memoria necesară este $\mathcal{O}(n \log V_{max})$, deoarece aint-ul conține V_{max} valori, deci are înălțime $\log V_{max}$, iar fiecare poziție din vectorul original va fi enumerată în $\log V_{max}$ colecții.

Pentru actualizare, trebuie să ștergem poziția modificată din lista vechii valori minime și din listele tuturor strămoșilor. Apoi inserăm poziția în listele noii valori minime. De exemplu, dacă minimul coloanei 100 se modifică din 30 în 20, atunci de la poziția 30 din aint și din toți strămoșii eliminăm elementul 100 din colecție. Apoi la poziția 20 în aint și în toți strămoșii inserăm elementul 100.

Rămîne să descriem interogările. Pentru a afla al k-lea minim dintr-un interval de coloane $[l, r]$, pornim din rădăcina arborelui de intervale (luînd așadar în calcul toate valorile de la 0 la V_{max}). Consultăm fiul stîng (valorile $1 \ldots V_{max}/2$) și ne întrebăm: cîte apariții au aceste valori pe poziții din $[l, r]$? Putem răspunde la această întrebare printr-o diferență, reducînd întrebarea la forma:

cîte apariții au aceste valori pe pozițiile $0 \dots r$? Așadar, trebuie numărate elementele mai mici sau egale cu r din setul rădăcinii. Set-ul simplu din STL nu poate gestiona această întrebare, dar putem folosi un set extins din PB/DS. Nu detaliem acum, dar ne vom reîntîlni cu acest tip de date.

Dacă numărul de valori între 0 și $V_{max}/2$ care apar pe poziții între l și r este $\geq k$, atunci acolo se va afla și al k-lea element, deci coborîm în fiul stîng. Altfel coborîm în fiul drept.

Complexitatea algoritmului este $\mathcal{O}((n+q) \log n \log V_{max})$. De exemplu, fiecare interogare coboară $\log V_{max}$ niveluri, iar la fiecare nivel face o căutare într-un set de $\mathcal{O}(n)$ elemente în timp $\mathcal{O}(\log n)$.

Bibliografie

[1] CS Academy, *Segment Trees*, URL: https://csacademy.com/lesson/segment_trees.

[2] CP Algorithms, *Segment Tree*, URL: https://cp-algorithms.com/data_structures/segment_tree.html.

Capitolul 2

Arbori de intervale cu propagare *lazy*

2.1 Operații pe interval

Să reluăm exemplul din capitolul trecut și să spunem acum că dorim să adăugăm 100 pe intervalul [2, 12], corespunzător nodurilor [12, 28] din arbore.

Ca să nu facem efort $\mathcal{O}(n)$, vom descompune intervalul ca mai înainte și vom nota informația „+100" în nodurile 9, 5, 6 și 28, cu semnificația că valoarea reală a tuturor frunzelor de sub aceste noduri a crescut cu 100.

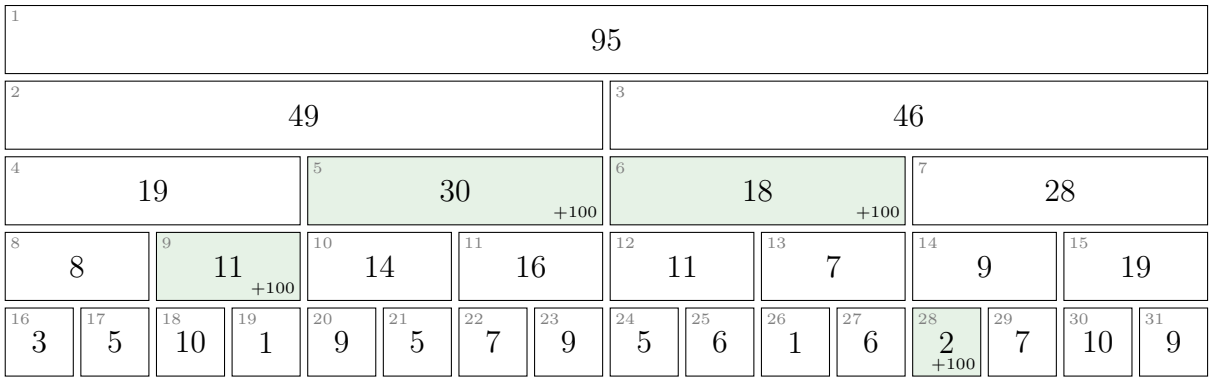


Figura 2.1: Pentru a adăuga 100 pe intervalul [18, 28], notăm valoarea *lazy* 100 pe nodurile 9, 5, 6 și 28.

Aceasta este o **informație *lazy***: o informație care stă într-un nod intern și care trebuie propagată tuturor frunzelor subîntinse de acel nod. Totuși, amînăm efortul acestei propagări pînă cînd el devine strict necesar; tocmai de aceea se numește **propagare *lazy***. (Mulți elevi denumesc întreaga structură „AINT cu *lazy*", dar asta este... lene.)

Evaluarea *lazy* este un concept des întîlnit:

- Memoizarea unor valori într-un vector / matrice, cu speranța că nu va fi nevoie să calculăm tabelul complet.

- Amînarea evaluării lui `y` în expresia booleană `x || y`, cu speranța că `x` va fi evaluat ca adevărat, iar `y` va deveni irelevant.
- Inițializarea unei componente costisitoare dintr-un program doar cînd devine necesară (o conexiune la baza de date, o zonă a hărții dintr-un joc).

Așadar, definim un al doilea vector numit `lazy` și executăm `lazy[x] += 100` pe pozițiile 9, 5, 6 și 28.

Motivul pentru care treaba se complică este următorul. Dacă acum primim o interogare de sumă pe intervalul [25,29]? Nu putem să însumăm, ca de obicei, pozițiile 25, 13 și 14, căci pierdem din vedere că unele dintre noduri au (cîte) +100. Sigur, putem lua asta în calcul, dar trebuie să clarificăm operațiile, altfel efortul poate deveni $\mathcal{O}(n)$.

În primul rînd, introducem două funcții noi (le puteți include în alte funcții, dar pentru claritate le puteți declara de sine stătătoare):

- `push()`, care propagă informația *lazy* de la un nod la fiii săi;
- `pull()`, care combină în părinte informația din cei doi fii după o actualizare.

```
void push(int node, int size) {
  s[node] += lazy[node] * size;
  lazy[2 * node] += lazy[node];
  lazy[2 * node + 1] += lazy[node];
  lazy[node] = 0;
}

void pull(int node, int size) {
  s[node] =
    s[2 * node] + lazy[2 * node] * size / 2 +
    s[2 * node + 1] + lazy[2 * node + 1] * size / 2;
}
```

Pentru problema dată (dar nu pentru toate problemele), codul are nevoie să știe numărul de frunze subîntinse (`size`).

Să presupunem acum că dorim să calculăm suma intervalului [2, 12] și că este posibil să avem niște sume *lazy* în multe alte noduri. Știm că codul descompune interogările în intervale mai scurte și nu urcă mai sus de acestea. Dacă există valori *lazy* mai sus (să zicem în rădăcină), codul nu va afla de ele. De aceea, în pregătirea interogării, trebuie să vizităm toți strămoșii intervalului și să propagăm în jos (*push*) informația *lazy*. Dar, dacă ne gîndim, lista completă a acestor strămoși constă doar din strămoșii capetelor de interval! Pentru intervalul [18,28], este nevoie să propagăm în jos informația *lazy* din strămoșii lui 18 (adică 1, 2, 4 și 9) și ai lui 28 (adică 1, 3, 7 și 14).

Dacă apelăm push din acești strămoși, de sus în jos, avem garanția că informația pe intervalele dorite este la zi. Vă rămîne vouă ca experiment de gîndire să demonstrați că, după operațiile *push*, nu va mai exista informație *lazy* în niciun strămoș al niciunei poziții din interogare.

Astfel obținem o funcție foarte similară cu cea din capitolul trecut

```
void push_path(int node, int size) {
  if (node) {
    push_path(node / 2, size * 2);
    push(node, size);
  }
}

long long query(int l, int r) {
  long long sum = 0;
  int size = 1;

  l += n;
  r += n;
  push_path(l / 2, 2); // pornim din părinte
  push_path(r / 2, 2);

  while (l <= r)  {
    if (l & 1) {
      sum += s[l] + lazy[l] * size;
      l++;
    }
    l >>= 1;

    if (!(r & 1)) {
      sum += s[r] + lazy[r] * size;
      r--;
    }
    r >>= 1;
    size <<= 1;
  }

  return sum;
}
```

Dacă viteza este crucială, putem scrie și o funcție `push_path` cu circa 10% mai rapidă, iterativă, folosind operații pe biți. Să considerăm nodul $22 = 10110_{(2)}$. Strămoșii lui sînt 1, 2, 5 și 11 care au respectiv reprezentările binare 1, 10, 101 și 1011, care sînt fix prefixele lui 10110! Deci îl vom deplasa pe 10110 la dreapta cu 4, 3, 2 și respectiv 1 bit pentru a-i obține strămoșii.

```
void push_path(int node) {
  int bits = 31 - __builtin_clz(n);
  for (int b = bits, size = n; b; b--, size >>= 1) {
    int x = node >> b;
    push(x, size);
  }
}

// Acum primul apel este chiar din frunză:
```

```
...
push_path(l);
push_path(r);
...
```

Actualizările sînt foarte similare. Apelăm `pull()` după terminarea actualizărilor, deoarece trebuie să lăsăm arborele într-o stare coerentă și trebuie să preluăm orice modificare de la fii.

```
void pull_path(int node) {
  for (int x = node / 2, size = 2; x; x /= 2, size <<= 1) {
    pull(x, size);
  }
}

void update(int l, int r, int delta) {
  l += n;
  r += n;
  int orig_l = l, orig_r = r;
  push_path(l / 2, 2);
  push_path(r / 2, 2);

  while (l <= r)  {
    if (l & 1) {
      lazy[l++] += delta;
    }
    l >>= 1;

    if (!(r & 1)) {
      lazy[r--] += delta;
    }
    r >>= 1;
  }

  pull_path(orig_l);
  pull_path(orig_r);
}
```

Iată și o altă implementare care combină bucla `while` principală cu funcția `pull_path`.

Notă: În această implementare, valoarea *lazy* se aplică întregului subarbore, inclusiv nodului însuși. În implementarea de pe CP Algorithms, valoarea *lazy* se aplică subarborelui fără nodul însuși. Oricare dintre formulări este acceptabilă, cîtă vreme o folosiți consecvent.

Notă: În practică, cîmpurile `lazy` și `s` merită încapsulate într-un `struct`. Datorită localității acceselor la memorie, diferența de viteză este notabilă (circa 25%). Aici le-am lăsat separate pentru concizie.

2.2 Implementare recursivă (actualizări punctuale)

Lecția trecută am spus că există și o implementare recursivă. Să o examinăm acum (mulți o știți deja).

```
void update(int node, int pl, int pr, int pos, int delta) {
  if (pr - pl == 1) {
    s[node] += delta;
  } else {
    int mid = (pl + pr) >> 1;
    if (pos < mid) {
      update(2 * node, pl, mid, pos, delta);
    } else {
      update(2 * node + 1, mid, pr, pos, delta);
    }
    s[node] = s[2 * node] + s[2 * node + 1];
  }
}
```

Metoda recursivă cară după ea 5 parametri:

- `node`: nodul curent din arbore (aka poziția în vector);
- `pl`, `pr`: intervalul din vectorul inițial acoperit de `node`. Eu am optat pentru implementarea cu `pl` inclusiv și `pr` exclusiv. Dacă preferați intervale închise, este OK.
- `pos`, `delta`: poziția de modificat și valoarea de adăugat/scăzut.

Vedem că funcția coboară recursiv în fiul stîng sau fiul drept, după caz. Un exemplu de apel ar fi:

```
update(1, 0, n, some_pos, some_val);
```

Mai interesant, iată și implementarea funcției de interogare (sumă pe interval):

```
long long query(int node, int pl, int pr, int l, int r) {
  if (l >= r) {
    return 0;
  } else if ((l == pl) && (r == pr)) {
    return s[node];
  } else {
    int mid = (pl + pr) >> 1;

    return
      query(node * 2, pl, mid, l, min(r, mid)) +
      query(node * 2 + 1, mid, pr, max(l, mid), r);
  }
}
```

Regăsim trei din aceiași parametri, `node`, `pl` și `pr`. În plus,

- `l`, `r`: Intervalul [închis, deschis) pe care dorim să calculăm suma.

Funcția se reapelează pe cei doi fii, restrîngînd corespunzător intervalul $[l, r)$. Iată o imagine care arată arborele de apeluri pentru calculul sumei pe intervalul [3, 10):

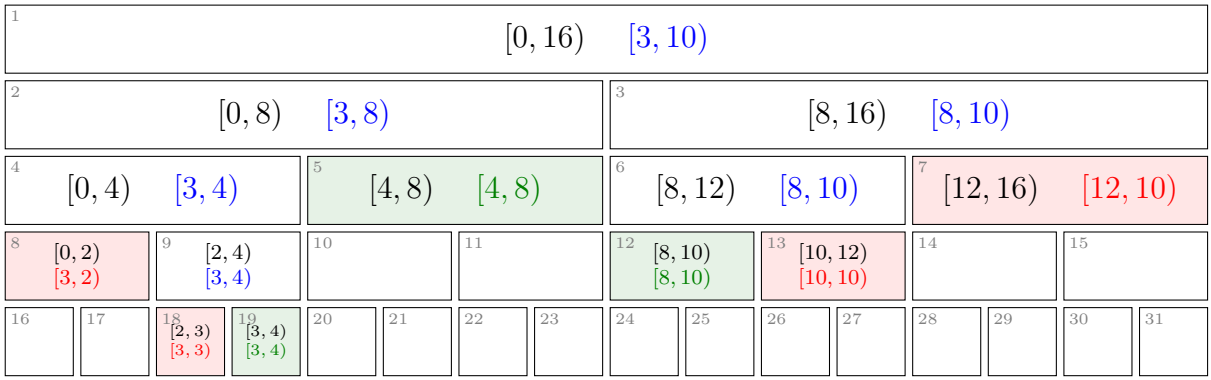


Figura 2.2: Arborele de apeluri în funcția de actualizare recursivă pe intervalul $[3, 10)$.

Am ilustrat cu albastru nodurile care au nevoie să-și apeleze descendenții, cu verde nodurile selectate integral, iar cu roșu nodurile eliminate.

Complexitatea rămîne $\mathcal{O}(\log n)$, deși funcția se reapelează pentru ambii fii. De ce?

Discutăm implementarea recursivă pentru că ea plutește prin supa culturală și vreau să puteți citi cod scris astfel. Dar ea este un exemplu de dopaj, de implementare repetată *mot à mot* indiferent de nevoile problemei. Implementarea recursivă este de 2-3 ori mai lentă decît cea iterativă pentru actualizări punctuale. Presupun că există două motive:

Implementarea recursivă cară după ea 5-6 parametri la fiecare apel, care trebuie copiați, puși/scoși de pe stivă etc. Implementarea iterativă folosește doar 3 variabile.

Implementarea recursivă este nevoită să pornească din rădăcină, să coboare pînă la frunze, apoi să revină din recursivitate. Implementarea iterativă se oprește imediat ce termină de descompus intervalul $[l, r]$.

La varianta cu propagare *lazy* diferența de timp aproape dispare, pentru că ambele implementări trebuie să urce pînă la rădăcină.

Nu vă năpustiți la implementarea recursivă dacă nu este nevoie. Rezistați tentației de a fi leneși, de a învăța o singură structură de date, pe care să o pictați indiferent de situație! Trebuie să aspirați la mai mult de atît, dacă este să vă meritați locul în lot.

2.3 Implementare recursivă (actualizări pe interval)

În această implementare, observăm cum:

- apelăm `push` înainte de reapelarea recursivă, pentru a-i garanta fiecărui nod că deasupra sa nu mai există informații lazy;

- apelăm `pull` după revenirea din recursivitate, ca să lăsăm arborele într-o stare coerentă.

```
long long query(int node, int pl, int pr, int l, int r) {
  if (l >= r) {
    return 0;
  } else if ((l == pl) && (r == pr)) {
    return s[node] + lazy[node] * (r - l);
  } else {
    push(node, pr - pl);
    int mid = (pl + pr) >> 1;
    return
      query(node * 2, pl, mid, l, min(r, mid)) +
      query(node * 2 + 1, mid, pr, max(l, mid), r);
  }
}

void update(int node, int pl, int pr, int l, int r, int delta) {
  if (l >= r) {
    return;
  } else if ((l == pl) && (r == pr)) {
    lazy[node] += delta;
  } else {

    push(node, pr - pl);
    int mid = (pl + pr) >> 1, child_size = (pr - pl) >> 1;
    update(2 * node, pl, mid, l, min(r, mid), delta);
    update(2 * node + 1, mid, pr, max(l, mid), r, delta);
    pull(node, child_size);

  }
}

void process_ops() {
  for (int i = 0; i < num_queries; i++) {
    if (q[i].t == OP_UPDATE) {
      update(1, 0, n, q[i].l - 1, q[i].r, q[i].val);
    } else {
      answer[num_answers++] = query(1, 0, n, q[i].l - 1, q[i].r);
    }
  }
}
```

2.4 Arta proiectării unui arbore de intervale

Atunci cînd primim o problemă și avem de proiectat o structură de date, cum știm dacă un arbore de intervale se potrivește scopului? În general, trebuie să urmărim trei lucruri.

În primul rînd, pentru a putea procesa rapid actualizările pe interval, dorim să facem efort $\mathcal{O}(1)$

în fiecare interval elementar din descompunere. De aceea, în general informația *lazy* stochează fix ce primim de la operațiile de *update*: o cantitate de adăugat sau de atribuit, un bit de semn, un bit care arată că intervalul curent trebuie inversat etc.

Este nevoie de atenție la compunerea actualizărilor. Dacă avem două cantități de adăugat pe același interval, cîmpul *lazy* va reține suma cantităților. Dacă avem două atribuiri succesive pe același interval, cîmpul *lazy* o va reține doar pe ultima.

În al doilea rînd, pentru a putea procesa rapid interogările pe interval, dorim să facem efort $\mathcal{O}(1)$ în fiecare interval din descompunere. De aceea, informația propriu-zisă din fiecare nod trebuie să includă valorile pe care le cer operațiile de *query*. Acestea sînt un punct de pornire, dar uneori nu sînt suficiente, ci este nevoie să menținem mai multe valori din care să le putem alege pe cele cerute.

În sfîrșit, în al treilea rînd trebuie să verificăm că avem tot ce ne trebuie pentru a compune două intervale alăturate la interogare, pentru a recalcula un părinte din cei doi fii ai săi (operația *pull*) și pentru a propaga informația *lazy* de la părinte la fii (operația *push*).

O regulă de aur este că, la începutul și la sfîrșitul fiecărei funcții, arborele trebuie să fie într-o stare **coerentă**. Aceasta înseamnă că trebuie să definim un **contract**, o promisiune despre care este structura logică a fiecărui nod. Vă recomand chiar să notați acest contract într-un comentariu de una-două fraze, la începutul codului pentru AINT. Apoi, scrieți codul astfel încît, la intrarea și la ieșirea din orice funcție, toate nodurile arborelui să respecte acel contract.

De exemplu, dacă informația *lazy* este o cantitate de adăugat pe tot subarborele, atunci operația *push* trebuie neapărat să se încheie prin a pune pe 0 valoarea *lazy* din părinte. În niciun caz nu trebuie să încheiem operația *push* lăsînd aceeași valoare *lazy* în părinte și în fii.

2.5 Probleme

2.5.1 Problema Polynomial Queries (CSES)

enunț • surse

Problema seamănă mult cu cea discutată la teorie, dar pe intervale nu mai adăugăm constante, ci progresii aritmetice. Așadar, pare natural să reținem exact această informație *lazy*: în fiecare nod reținem că în fiecare frunză acoperită de acel nod trebuie să adăugăm cîte un termen al unei progresii cu un anumit prim element și pasul (deocamdată) 1. De exemplu, dacă în figura 2.1 facem o actualizare pe intervalul [18, 28], atunci în nodul 5 notăm progresia cu primul termen 3 și pasul 1. Informația *lazy* este o pereche $\langle 3, 1 \rangle$.

Trebuie tratate atent diversele cazuri care iau naștere. Dacă două progresii acoperă același interval, vor lua naștere progresii cu pas mai mare decît 1. Să luăm un exemplu:

- Progresia cu primul termen 5 și pasul 3, așadar 5, 8, 11, 14, ...
- Progresia cu primul termen 2 și pasul 7, așadar 2, 9, 16, 23, ...

- După însumare dorim să avem termenii 7, 17, 27, 37, ...
- Rezultă că suma este și ea o progresie cu primul termen 7 și pasul 10. Cu alte cuvinte, informațiile *lazy* se pot compune ușor: $\langle 5, 3\rangle + \langle 2, 7\rangle = \langle 7, 10\rangle$.

La propagarea în jos a informației *lazy*, în cei doi fii vom adăuga progresii cu același pas. În fiul drept, primul termen trebuie calculat, dar este ușor. Dacă într-un nod care acoperă 16 elemente avem o progresie cu primul element 3 și pasul 5, atunci fiul drept va începe cu al nouălea termen al progresiei:

$$3 + 5 \cdot (16/2) = 43$$

La operațiile de adăugare, pe toate intervalele din descompunere vom aduna progresii cu pasul 1, dar primul element diferă pentru fiecare interval (la fel, nu este greu de calculat).

Contractul pe care l-am ales pentru implementarea iterativă este:

- `first` și `step` înseamnă că pe nodurile din intervalul acoperit trebuie adăugate valorile `first`, `first + step`, `first + 2 * step`, ...
- Valoarea `s` din fiecare nod **nu** include și suma progresiei dată de `<first, step>` din acel nod.

2.5.2 Problema Nezzar and Binary String (Codeforces)

enunț • sursă

Problema este relativ directă odată ce „ne prindem” că trebuie să procesăm operațiile în ordine inversă. Știm șirul final f și fie $[l, r]$ ultimul interval inspectat de Nanako. În momentul inspecției, șirul curent trebuia să fie identic cu f pe pozițiile $[1, l) \cup (r, n]$, căci pe acelea nu le putem modifica. Pe pozițiile $[l, r]$ trebuiau să fie doar biți 0 sau doar biți 1. Care dintre ele? Știm că la ultima modificare am modificat strict mai puțin de jumătate din biți. Să notăm cu z numărul de zerouri și cu u numărul de unu de pe pozițiile $[l, r]$ din f. Iau naștere trei cazuri:

1. Dacă $z > u$, înseamnă că la pasul anterior $[l, r]$ conținea doar 0.
2. Dacă $z < u$, înseamnă că la pasul anterior $[l, r]$ conținea doar 1.
3. Dacă $z = u$, problema nu are soluție, căci nu putem opera modificarea necesară.

Astfel, toate operațiile sînt forțate, mergînd înapoi în timp. Răspunsul este `YES` doar dacă putem procesa toate operațiile, iar la final ajungem la șirul s.

Rezultă că, pentru a efectua efectiv operațiile, avem nevoie de un arbore de segmente cu valori de 0 și 1 în frunze, cu funcții de sumă pe interval (pentru a stabili majoritatea) și de atribuire pe interval (pentru a face *undo* la o operație).

2.5.3 Problema Simple (infO(1)Cup 2019)

enunț • sursă

Să aplicăm regula menționată ca să proiectăm un arbore de intervale pentru această problemă.

- În cîmpul *lazy* vom stoca valorile primite la update, așadar cantitățile de adăugat pe tot subarborele.
- În cîmpurile propriu-zise vom stoca valorile necesare pentru interogări, așadar minimul par pe interval și maximul impar pe interval.
- Ce altceva ne mai trebuie ca să putem menține informația la actualizări? Să observăm că o cantitate *lazy* impară schimă paritatea valorilor pe întregul interval. Noul minim par este fostul minim impar, plus cantitatea *lazy*. De acea, vom introduce încă două cantități: maximul par și minimul impar.

Ca de obicei, este important ca la implementare să alegem dacă valoarea *lazy* este deja inclusă în nodul curent și mai trebuie aplicată doar la subarbore sau dacă ea trebuie aplicată inclusiv nodului curent. Eu am ales prima variantă. Ambele sînt bune, cîtă vreme codul respectă alegerea făcută.

Pe unele intervale nu vor exista valori pare sau impare. Dacă facem cazuri speciale pentru toate acele situații, vom avea undeva între 5 și 10 `if`-uri de presărat prin cod. O abordare mai simplă este să notăm în acele noduri $+\infty$ pentru a arăta că nu există minime și $-\infty$ pentru a arăta că nu există maxime. Apoi lăsăm aceste valori să se combine fără să mai tratăm cazuri particulare. La final, știm că orice valori definite vor fi între 1 și $2 \cdot 20^9 + 2 \cdot 20^5 \cdot 2 \cdot 20^9$, conform limitelor din enunț. Valorile din afara acestui interval le interpretăm ca fiind nedefinite.

În cod am încercat să separ funcțiile specifice nodului de funcțiile specifice arborelui.

2.5.4 Problema Balama (Baraj ONI 2024)

enunț • surse

Vă veți întîlni des cu probleme unde soluția devine simplă dacă analizăm informațiile în altă ordine. În cazul de față, în loc să luăm în calcul liniile (care sînt subsecvențe ordonate), să analizăm coloanele.

Care va fi răspunsul pe ultima coloană? Desigur, va fi maximul din vector. Mult mai interesantă este întrebarea: care va fi răspunsul pe penultima coloană? Va fi cel mai mare element care este vreodată (în cel puțin o fereastră) **al doilea maxim**.

Exemplu: fie maximul global 1.000 și fie al doilea maxim global 999. Dacă 999 stă foarte departe de 1.000, la distanță de cel puțin k, atunci 999 va fi maxim în toate ferestrele de lățime k care îl conțin. Cu alte cuvinte, 999 se va regăsi doar pe ultima coloană în matricea B (și va fi mascat de 1.000). Să spunem că următoarele valori din șir, în ordine descrescătoare, sînt 998, 997 și 996 și toate se află la distanță mare unele de altele. Niciunul dintre ele nu va apărea pe penultima coloană în B.

În schimb, să spunem că următoarea valoare ca mărime, 995, se află aproape (la distanță $< k$) de o valoare anterioară, cum ar fi 997. Atunci există o fereastră în care 995 și 997 coexistă, deci 995 va fi al doilea maxim din acea fereastră și va apărea pe penultima coloană în B. Cum alte valori mai mari nu au această proprietate, 995 este răspunsul pe penultima poziție a soluției.

(Amănunt esențial 😉: 995 nu poate fi și al treilea maxim. Dacă exista o fereastră de lățime k care îl cuprindea pe 995 și alte două valori anterioare, atunci înainte să ajungem la 995 una dintre acele două valori anterioare ar fi fost al doilea maxim).

Astfel, putem considera elemente în ordine descrescătoare și, pentru fiecare element x ne întrebăm: cîte elemente văzute anterior conține fiecare dintre ferestrele care îl conțin pe x (cel mult k la număr)? Dacă o astfel de fereastră are e elemente, și dacă a $e+1$-a valoare din soluție (numărînd de la dreapta) este încă necunoscută, atunci pune x pe poziția $e+1$ a soluției.

Exemplu: Dacă considerăm elementul 900 și constatăm că într-una din ferestrele care îl conțin pe 900 existau deja alte 5 valori, atunci în acea fereastră 900 este al 6-lea element. Dacă a 6-a poziție din soluție este încă neocupată, scriem 900 acolo, acesta fiind maximul posibil.

Implementarea sună fioros, dar nu este! În realitate avem nevoie de o structură cu două operații:

- Incrementează pozițiile de la st la dr, pentru a arăta că în ferestrele de la $[st, st+k-1]$ și pînă la $[dr, dr+k-1]$ avem cîte un element în plus.
- Află maximul de pe pozițiile de la st la dr.

Operația a doua ne este suficientă deoarece ferestrele nu vor ajunge brusc la 6 elemente. Vor apărea mai întîi ferestre cu 1, 2, 3, 4, 5 elemente. Cu alte cuvinte, soluția se completează de la dreapta spre stînga.

Vom implementa un AINT de maxime în care informația *lazy* din fiecare nod este valoarea de adăugat pe fiecare poziție din intervalul acoperit.

Capitolul 3

Arbori indexați binar

Arborii indexați binar (AIB), numiți și arbori Fenwick, iar în engleză *binary indexed trees (BIT)*, servesc ca și arborii de intervale tot la rezolvarea în $\mathcal{O}(\log n)$ a unor operații pe vectori. Ei sînt mai puțin flexibili și universali decît arborii de intervale. Nu toate problemele rezolvabile cu AINT pot fi rezolvate și cu AIB. Dar acolo unde se potrivesc, AIB-urile sînt ușor de codat și sînt de 2-3 ori mai rapide decît arborii de intervale.

3.1 *Benchmarks*

Dacă se potrivesc mai multe structuri, contează pe care o alegem? Ca să alegem în cunoștință de cauză, iată niște măsurători de viteză (*benchmarks*). Le-am făcut în 2025 pe un procesor AMD Ryzen 7 4700U, la acea vreme comparabil cu evaluatoarele de la Kilonova și Codeforces.

Am măsurat timpii de rulare pentru diverse implementări ale problemei în ambele variante (actualizări punctuale sau pe interval).

- arbori indexați binar, $\mathcal{O}(\log n)$ per operație;
- arbori de segmente iterativi, $\mathcal{O}(\log n)$ per operație;
- arbori de segmente recursivi, $\mathcal{O}(\log n)$ per operație;
- descompunere în radical, $\mathcal{O}(\sqrt{n})$ și cel mult două împărțiri per operație;
- descompunere în radical, $\mathcal{O}(\sqrt{n})$ și $\mathcal{O}(\sqrt{n})$ împărțiri per operație.

Precizez că vom discuta descompunerea în radical abia în capitolul următor, dar pare un moment bun să privim aceste *benchmarks*.

Am ales limitele $n = q = 500.000$ pentru ambele variante ale problemei. Pentru unele programe contează cîte dintre operații sînt interogări și cîte sînt actualizări. Pentru aceste situații, am măsurat doi timpi, notați astfel:

- 250u/250q: există cîte 250.000 de operații din fiecare tip;
- 100u/400q: există 100.000 de actualizări și 400.000 de interogări.

Toate testele sînt pe `long long` și 90% dintre intervale au lungime peste $n/2$. Toți timpii măsoară

strict partea de procesare (excluzînd citirea și scrierea).

3.1.1 Varianta 1 (*point update, range query*)

structură	**timp**
arbore indexat binar	23 ms
arbore de intervale iterativ (250u/250q)	46 ms
arbore de intervale iterativ (100u/400q)	55 ms
arbore de intervale recursiv (250u/250q)	116 ms
arbore de intervale recursiv (100u/400q)	130 ms
descompunere în radical (250u/250q)	113 ms
descompunere în radical (100u/400q)	177 ms
descompunere în radical cu împărțiri (250u/250q)	763 ms
descompunere în radical cu împărțiri (100u/400q)	1.219 ms

Tabela 3.1: Timpii de rulare pentru sume pe interval și actualizări punctuale.

3.1.2 Varianta 2 (*range update, range query*)

structură	**timp**
arbore indexat binar (250u/250q)	66 ms
arbore indexat binar (100u/400q)	58 ms
arbore de intervale iterativ (250u/250q)	183 ms
arbore de intervale iterativ (100u/400q)	175 ms
arbore de intervale recursiv (250u/250q)	211 ms
arbore de intervale recursiv (100u/400q)	203 ms
descompunere în radical (250u/250q)	235 ms
descompunere în radical (100u/400q)	274 ms

Tabela 3.2: Timpii de rulare pentru sume pe interval și actualizări pe interval.

3.1.3 Concluzii

Reținem că:

- Arborii indexați binar sînt de departe cei mai rapizi.
- Pentru actualizări punctuale, arborii de intervale iterativi sînt de două ori mai rapizi decît cei recursivi.
- Descompunerea în radical ține binișor pasul cu arborii de segmente. Din experiență, aceasta este o particularitate a problemei alese. Pentru alte probleme diferența poate fi mai mare.
- Împărțirile îngreunează enorm descompunerea în radical.

3.2 Actualizări punctuale și interogări pe interval

3.3 Reprezentare

AIB-ul descompune informația în felul următor: Poziția k din vector stochează suma ferestrei de p elemente care se termină la poziția k, unde p este cea mai mare putere a lui 2 care îl divide pe k. De exemplu, pentru $k = 40$, $p = 8$. Așadar, pe poziția 40 AIB-ul va stoca suma celor 8 valori de pe pozițiile $[33 \dots 40]$.

Iată un exemplu care arată sus valorile pe care dorim să le reținem, iar jos valorile concrete pe care ajunge să le stocheze vectorul. Subliniez că AIB-ul nu folosește memorie suplimentară, ci doar stochează diferit informația în același vector. Suspectez că și de aici provine eficiența lui în raport cu arborii de intervale.

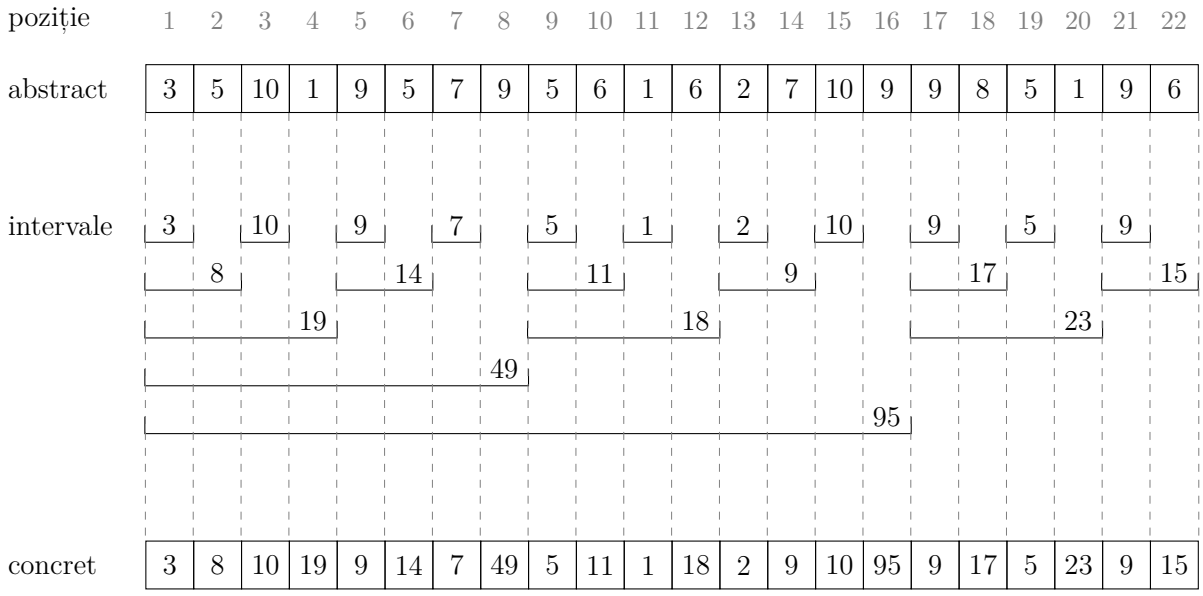


Figura 3.1: Un arbore indexat binar cu 22 de poziții. Vectorul de sus este cel abstract, iar vectorul de jos este cel stocat concret în memorie. Fiecare interval arată pozițiile a căror sumă o notăm în capătul din dreapta al intervalului.

3.4 Operația de interogare (suma unui interval)

AIB-urile tratează interogările pe un interval oarecare $[x, y]$ prin diferența a două interogări pe prefix, $[1, y]$ și $[1, x-1]$. Pentru a răspunde la o interogare pe prefix, de exemplu suma pe intervalul $[1, 21]$, descompunem acel prefix în intervale dintre cele stocate în AIB, respectiv $[1, 16]$, $[17, 20]$ și $[21, 21]$. Odată ce includem o poziție x și tot intervalul pe care îl acoperă ea, pentru a ajunge la următoarea poziție de însumat trebuie, prin definiție, să scădem cea mai mare putere a lui 2 care îl divide pe x. Rezultă codul:

```
struct fenwick_tree {
  int v[MAX_N + 1]; // indexare de la 1
```

```
  int prefix_sum(int pos) {
    int s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }

  int range_sum(int from, int to) {
    return prefix_sum(to) - prefix_sum(from - 1);
  }
};
```

Expresia `pos &= pos - 1` elimină cel mai din dreapta bit de 1 dintr-un număr; de exemplu, din $20 = 10100_{(2)}$ ea obține $16 = 10000_{(2)}$. Pe cazul general,

```
pos             = abc...xyz1000...000
pos - 1         = abc...xyz0111...111
pos & (pos - 1) = abc...xyz0000...000
```

De aici rezultă și complexitatea $\mathcal{O}(\log n)$, căci reprezentarea oricărei poziții în baza 2 are cel mult $\log n$ biți de 1.

3.5 Operația de actualizare (adăugare pe poziție)

La actualizarea pe o poziție, trebuie actualizate toate intervalele care conțin acea poziție. De exemplu, la actualizarea poziției 11 trebuie actualizate intervalele $[11, 11]$, $[9, 12]$ și $[1, 16]$, așadar trebuie recalculate pozițiile 11, 12 și 16 din AIB. Această parte pare magică: de ce pozițiile 13, 14 și 15 nu trebuie actualizate? Dar ne putem convinge că, cît ne îndepărtăm de poziția inițială (11), ne interesează doar pozițiile responsabile de intervale suficient de mari încît să acopere poziția 11.

```
void add(int pos, int val) {
  do {
    v[pos] += val;
    pos += pos & -pos;
  } while (pos <= n);
}
```

Pentru a înțelege expresia `pos & -pos`, să facem o scurtă digresiune. Numerele cu semn sînt reprezentate în calculator în complement față de 2. Aceasta înseamnă că, pentru a reprezenta un număr negativ,

- Reprezentăm întîi numărul pozitiv (valoarea absolută).

- Îi inversăm toți biții.
- Adăugăm 1.

De exemplu, pentru a îl reprezenta pe -20 procedăm astfel:

- Îl reprezentăm pe +20: `000...00010100`.
- Îi inversăm toți biții: `111...11101011`.
- Adăugăm 1: `111...11101100`.

Această reprezentare are două avantaje:

1. Putem folosi aceleași circuite logice pentru operații pe numere cu sau fără semn.
2. Reprezentările lui $+0$ și -0 sînt identice, `000...000`. În complement față de 1 există două reprezentări, `000...000` și `111...111`, ceea ce este straniu.

Revenind, observăm acum că `20 & -20` este `000...00000100`, adică formula `pos & -pos` izolează ultimul bit, adică mărimea intervalului subîntins de `pos`. Prin adăugarea acestei cantități la `pos`, obținem intervalul imediat următor care include poziția `pos`.

Dacă din orice motiv această expresie vă scapă din memorie, puteți inventa pe loc formule echivalente, de exemplu:

```
pos = (pos | (pos - 1)) + 1;
```

Complexitatea este tot $\mathcal{O}(\log n)$ deoarece la fiecare pas eliminăm cel puțin un bit 1 din reprezentarea binară a lui pos.

3.6 Construcția în $\mathcal{O}(n)$

Dat fiind un vector-sursă `src`, este tentant să construim arborele într-un al doilea vector apelînd de n ori rutina de adăugare:

```
void build(int* src) {
  for (int i = 1; i <= n; i++) {
    add(i, src[i]);
  }
}
```

Această metodă cere timp $\mathcal{O}(n \log n)$. Există însă o metodă care refolosește vectorul și rulează în $\mathcal{O}(n)$:

```
void build() {
  for (int i = 1; i <= n; i++) {
    int j = i + (i & -i);
    if (j <= n) {
      v[j] += v[i];
    }
```

```
  }
}
```

Explicație: fiecare element `v[i]` este propagat doar la următorul element `j` care include poziția `i`. Este treaba acelui segment să propage adaosul și mai departe. Pentru scenariul relativ comun în care construim AIB-ul, apoi facem n interogări și n actualizări, construcția în $\mathcal{O}(n)$ reduce costul de rulare cu circa 20%.

Paradoxal, tocmai fiindcă este atît de rapid, costul de funcționare al unui AIB este adesea înecat de costul altor operații (în special intrarea/ieșirea). De aceea optimizările sînt greu de observat. Dar *the hacker spirit* ne obligă să folosim oricum soluția inteligentă.

3.7 Găsirea unei valori punctuale

Pentru a găsi valoarea pe o singură poziție `k`, o putem calcula în $\mathcal{O}(\log n)$ ca pe `sum(k) - sum(k - 1)`. Sau, desigur, putem păstra o copie a vectorului real. Dar există și o implementare în $\mathcal{O}(1)$ amortizat.

Să considerăm poziția $k = 60$. Dacă o calculăm prin diferența sumelor parțiale, obținem

```
val(60) = sum(60) - sum(59) = (v[60]          + v[56] + v[48] + v[32]) -
                              (v[59] + v[58] + v[56] + v[48] + v[32])
                            = v[60] - (v[59] + v[58])
```

Se vede că, de la poziția 56 încolo, sumele de intervale se anulează în cele două paranteze. Nu este o coincidență. Fie:

```
k     = bbb...bbb10000
k - 1 = bbb...bbb01111
```

Unde `b` sînt niște biți oarecare, iar poziția k se termină într-un bit 1 urmat de cîțiva (posibil 0) biți de 0. Atunci, în calculul sumelor parțiale, pozițiile k și $k-1$ vor elimina biți de la coadă pînă cînd vor ajunge la strămoșul comun, care este

```
str   = bbb...bbb00000
```

De aceea, codul este:

```
int get_value_at(int pos) {
  int result = v[pos];
  int ancestor = pos & (pos - 1);
  pos--;
  while (pos != ancestor) {
    result -= v[pos];
    pos &= pos - 1;
  }
  return result;
```

```
}
```

Acest cod pare tot logaritmic. În realitate, jumătate din valorile din AIB (cele de pe poziții impare) stochează chiar valoarea în acel punct, deci bucla din `get_value_at()` va face 0 iterații. Un sfert din valorile din AIB vor face o iterație, o optime dintre ele vor face două iterații. În general, pentru o poziție k, funcția `get_value_at()` va face atîtea iterații cîte zerouri are la coadă reprezentarea binară a lui k. Media acestei valori este 1 (așadar constantă) dacă distribuția lui k este uniformă și aleatorie.

3.8 Căutarea binară a unei sume parțiale

Dacă vectorul (abstract) are doar valori non-negative, atunci sumele parțiale sînt nedescrescătoare și are sens întrebarea: Care este prima poziție pe care se atinge suma parțială S?

Putem face o căutare binară naivă: examinăm suma parțială la poziția $n/2$, apoi la una dintre pozițiile $n/4$ sau $3n/4$ după caz, etc. Dar fiecare dintre aceste interogări durează $\mathcal{O}(\log n)$, deci complexitatea totală a algoritmului va fi $\mathcal{O}(\log^2 n)$. Dar iată și o metodă în $\mathcal{O}(\log n)$, similară cu căutarea binară prin „metoda Mihai Pătrașcu” (ca să adoptăm nomenclatura din supa culturală olimpică).

Fie p cea mai mare putere a lui 2 cel mult egală cu n. Pentru exemplul inițial, $n = 22$, deci $p = 16$. Observația-cheie este că putem afla suma parțială pe pozițiile $[1 \ldots p]$ printr-o singură operație: ea este exact `v[p]`! După cum `v[p] < S` sau `v[p] > S`, ne îndreptăm atenția către pozițiile din stînga sau din dreapta lui p. Următoarea interogare o vom face la `v[p / 2]` sau la `v[3 * p / 2]`.

În practică, invariantul este: `pos` reprezintă cea mai mare poziție cunoscută pe care suma parțială **nu** atinge valoarea S. La final, funcția returnează `pos + 1`. De asemenea, ținem cont că nu întotdeauna putem avansa la dreapta (nu putem depăși valoarea n).

```
int bin_search(int sum) {
  int pos = 0;

  for (int interval = max_p2; interval; interval >>= 1) {
    if ((pos + interval <= n) && (v[pos + interval] < sum)) {
      sum -= v[pos + interval];
      pos += interval;
    }
  }

  return pos + 1;
}
```

Exemplu: pentru AIB-ul din figura 3.1 și suma parțială 72, algoritmul funcționează astfel:

- Verifică poziția 16. Suma este 95, prea mare.

- Verifică poziția 8. Suma este 49. Așadar, căutăm suma parțială 72 - 49 = 23 începînd dincolo de poziția 8.
- Verifică poziția 12. Suma este 18. Așadar, căutăm suma parțială 23 - 18 = 5 începînd dincolo de poziția 12.
- Verifică poziția 14. Suma este 9, prea mare.
- Verifică poziția 13. Suma este 2. Așadar, căutăm suma parțială 5 - 2 = 3 începînd dincolo de poziția 13.
- Răspunsul este poziția 14.

Aici, `max_p2` este cea mai mare putere a lui 2 care nu depășește n. O putem afla naiv, de exemplu astfel:

```
max_p2 = n;
while (max_p2 & (max_p2 - 1)) {
  max_p2 &= max_p2 - 1;
}
```

, sau într-o singură linie cu funcția `__builtin_clz(n)`, care returnează numărul de zerouri la stînga lui n:

```
max_p2 = 1 << (31 - __builtin_clz(n));
```

Corolar: putem folosi căutarea binară pentru a afla prima poziție cu o valoare nenulă într-un AIB. Aceasta este poziția pe care suma parțială atinge valoarea 1. Similar putem afla ultima poziție cu o valoare nenulă. Mai trebuie doar să menținem și suma elementelor din AIB, ceea ce cere două linii de cod.

Corolar: dacă AIB-ul ține valori de 0 și 1, putem folosi căutarea binară pentru a afla poziția celui de-al k-lea bit 1 (în engleză această valoare se numește *k-th order statistic*). Ea este fix poziția pe care suma parțială atinge valoarea k. Multe probleme de permutări, care necesită evidența elementelor văzute / nevăzute, se încadrează aici (exemplu: codificarea / decodificarea permutărilor).

3.9 Alte operații decît adunarea

Arborii Fenwick pot gestiona și alte operații, cîtă vreme ele sînt **inversabile**. Reamintesc că **suma** pe intervalul $[l \ldots r]$ se calculează ca **diferența** sumelor pe intervalele $[1 \ldots r]$ și $[1 \ldots l-1]$. Deci operația inversă (scăderea) trebuie să fie definită. Exemplu: operația xor, operația de înmulțire modulo un număr prim etc.

Deoarece operația *max* nu este inversabilă, AIB-urile nu suportă, pe cazul general, operațiile:

1. actualizare punctuală;
2. maxim pe interval.

Totuși, putem folosi AIB-uri pentru interogări de maxim dacă următoarele condiții sînt adevărate:

1. Toate interogările sînt pe prefix (capătul stînga este întotdeauna 1).
   - Sau toate interogările sînt pe sufix, caz în care putem reflecta toți indicii față de n.
2. Prin actualizare, valorile pot doar să crească.

Temă de gîndire: De ce este necesar ca toate valorile să crească? Ce poate să meargă prost dacă într-un AIB de maxime valorile pot să și scadă?

Raționamente similare putem aplica pentru operațiile *min*, *and* și *or*. Cum reformulăm condiția 2 pentru aceste operații?

Un exemplu mai extravagant este operația *or* pe `bitset`-uri, vezi problema Erinaceida.

3.10 Probleme

3.10.1 Problema The Permutation Game Again (SPOJ)

enunț • sursă

Problema ne cere să aflăm **rangul** unei permutări (engl. *rank*). Acesta este numărul de ordine al permutării în lista ordonată lexicografic a tuturor permutărilor mulțimii $\{1, 2, \ldots, n\}$.

Echivalent, trebuie să răspundem eficient la întrebarea: cîte permutări vin înaintea celei date în lista permutărilor?

Să considerăm un exemplu. Dacă primul element al permutării este 9, atunci toate permutările care încep cu $1, 2, \ldots, 8$ o vor preceda în listă. Există $8(n-1)!$ astfel de permutări.

Similar, dacă al doilea element este 3, atunci toate permutările care încep cu 91 sau 92 o vor preceda în listă. Există $2(n-2)!$ astfel de permutări.

Dar dacă al treilea element este 7? Acum trebuie să ținem cont de faptul că pe 3 l-am văzut deja. Trebuie să socotim permutările care încep cu $931, 932, 934, 935, 936$. Există $5(n-3)!$ astfel de permutări.

Cu alte cuvinte, trebuie să răspundem eficient la întrebarea: cîte elemente mai mici decît cel curent am văzut în prefixul dinaintea elementului curent? Putem gestiona această informație cu un AIB de 0 și 1. Cînd procesăm un element de valoare x, adunăm 1 pe poziția x în AIB. Astfel, suma parțială din AIB pe o poziție y ne va arăta cîte elemente mai mici decît y am procesat pînă în prezent.

Pentru un plus de eficiență, sursa nu reține întreaga permutare, ci doar citește cîte un element, îl ia în calcul la rang, îl bifează în AIB, apoi îl aruncă.

3.10.2 Problema Multiset (Codeforces)

enunț • sursă

„Aproape” putem rezolva problema cu un singur vector de frecvențe. Dar avem nevoie să găsim eficient al k-lea element ca să-l putem șterge. Un vector de frecvențe ne dă inserări în $\mathcal{O}(1)$, dar ștergeri în $\mathcal{O}(n)$.

De aceea, înlocuim vectorul cu un AIB în care pe poziția x notăm frecvența lui x în multiset. Astfel putem căuta al k-lea element reformulînd definiția: al k-lea element este poziția p pe care suma parțială atinge sau depășește valoarea k.

3.10.3 Problema Hanoi Factory (Codeforces)

enunț • sursă

Pare natural să sortăm inelele descrescător după diametrul exterior. Ce facem la egalitate? Toate inelele de același diametru exterior pot fi stivuite, caz în care îl vom prefera deasupra pe cel cu diametrul interior minim, ca să ne maximizăm șansele de a putea pune alt inel deasupra lui. Așadar, ca departajare, sortăm inelele descrescător după diametrul interior.

Acum orice turn valid va fi un subșir din șirul sortat, pe sărite, dar fără reordonare. Și atunci putem defini relativ ușor o recurență calculabilă în $\mathcal{O}(n^2)$. Fie H_i înălțimea maximă a unui turn care are în vîrf inelul i. Atunci inelul aflat imediat sub i, fie el j, respectă condițiile $j < i$ și $in_j < out_i$. Așadar,

$$H_i = h_i + \max_{j<i,\ in_j<out_i} H_j$$

Pentru a reduce complexitatea la $\mathcal{O}(n \log n)$, procesăm inelele de la stînga la dreapta. Atunci $j < i$ este întotdeauna respectată și trebuie doar să răspundem la întrebarea: dintre toate inelele cu $in_j <$ un x dat (unde $x = out_i$), care este valoarea maximă pentru H_j? Putem răspunde la întrebare cu un AIB de maxime, indexat după diametrele interioare, pe care îl interogăm despre maximul pe pozițiile $[1 \dots out_i - 1]$. După calcularea lui H_i, optimizăm maximul din AIB pe poziția in_i cu valoarea H_i.

Diametrele pot fi mari, dar le putem normaliza în intervalul $[1 \dots 2n]$.

Există și o soluție mai ingenioasă, cu o stivă ordonată, care nu face obiectul acestui capitol.

3.10.4 Problema Subsequences (Codeforces)

enunț • sursă

Problema pare abordabilă cu programare dinamică. Să căutăm întîi formula de recurență, care nu este dificilă. Fie $C_{l,i}$ numărul de subsecvențe crescătoare de lungime l terminate pe poziția i. Atunci:

$$C_{l,i} = \sum_{j<i,\ a_j<a_i} C_{l-1,j}$$

Implementarea în $\mathcal{O}(kn^2)$ este așadar directă. Cum procedăm să reducem calculul sumei de la $\mathcal{O}(n)$ la $\mathcal{O}(\log n)$? Observăm aici un mecanism pe care îl vom regăsi și la alte probleme. Pare că dorim o interogare bidimensională (suma valorilor $C_{l-1,j}$ pe poziții unde $j < i$ și simultan $a_j < a_i$). În realitate, însă, putem reduce interogarea la una unidimensională.

Să inserăm într-un AIB valorile $C_{l-1,1} \dots C_{l-1,i-1}$. Atunci condiția $j < i$ este automat satisfăcută și dorim suma valorilor pe pozițiile unde $a_j < a_i$. De aceea, vom indexa AIB-ul nu după j, ci după a_j. Cu alte cuvinte, vom scrie $C_{l-1,j}$ nu la poziția j, ci la poziția a_j. Desigur, după ce calculăm $C_{l,i}$ adăugăm și $C_{l-1,i}$ la AIB.

Ca fapt divers, puteam face reducerea la interogări unidimensionale pe dos: iterăm prin elemente în ordinea crescătoare a valorilor, astfel încît condiția $a_j < a_i$ să fie automat satisfăcută. Atunci AIB-ul ar fi fost indexat după pozițiile propriu-zise, deci $C_{l-1,j}$ ar fi stat chiar la poziția a_j.

Noua complexitate este $\mathcal{O}(kn \log n)$: pentru fiecare dintre cele $k \times n$ valori ale lui C facem o interogare și o actualizare în AIB. Ca optimizare de memorie, ne este suficientă o singură linie din matricea C.

3.10.5 Problema D-query (SPOJ)

enunț • sursă

La prima vedere AIB-urile ne sînt inutile aici, pentru că funcția „numărul de elemente distincte" nu poate fi calculată prin diferențe de intervale: dacă în intervalul $[1, 10]$ avem 5 elemente distincte, iar în $[1, 20]$ avem 8 elemente distincte, nu știm destule despre numărul de elemente distincte din $[11, 20]$.

Dar, ca la multe alte probleme, varianta offline (în care primim de la început toate interogările) este considerabil mai simplă decît varianta online (în care trebuie să răspundem la o interogare înainte de a o primi pe următoarea).

Pare natural să sortăm interogările și să le scanăm cumva, dar cum? Să fixăm o poziție r și să considerăm toate intervalele care se termină la r. Dacă un interval $[l, r]$ conține o valoare x, atunci o poate conține o dată sau de multiple ori, dar în mod sigur va include **cea mai din dreapta** apariție a lui x înainte de r. Și atunci, pentru o poziție fixată r, dorim să bifăm toate pozițiile $i \in [1, r]$ pentru care nu există o altă poziție $j \in [i + 1, r]$ cu $v[i] = v[j]$.

De exemplu, pentru $r = 8$ și prefixul (1 1 7 6 1 2 6 2), dorim să stocăm bifele (0 0 1 0 1 0 1 1), pentru a indica cele mai din dreapta apariții ale lui 1, 2, 6 și 7. Atunci răspunsul la interogarea $[5, 8]$ va fi tocmai numărul de bife (adică suma) din intervalul $[5, 8]$, pentru că astfel ne asigurăm că numărăm exact o apariție, ultima, a fiecărei valori din interval.

AIB-ul de bife este ușor de actualizat. Dacă, de exemplu, următorul element din vector este 7, trebuie să ștergem bifa de la ultima apariție a lui 7 (poziția 3) și să o aplicăm pe poziția 9. Avem nevoie de un vector cu poziția ultimei apariții a fiecărei valori (dacă există), ceea ce este fezabil deoarece valorile nu depășesc 1.000.000.

La final, reordonăm interogările conform ordinii inițiale și afișăm răspunsurile.

3.10.6 Problema Magic Board (CodeChef)

enunț • sursă

Pare că avem de-a face cu o matrice binară uriașă, dar secretul este să reținem separat informații despre linii și despre coloane. Observația-cheie este că operațiile `Set` modifică doar linii și coloane întregi.

Putem reformula o interogare de tipul `RowQuery i` astfel. Dacă ultima resetare a liniei i a fost la momentul de timp t (operația cu numărul t) și la valoarea 0, atunci cîte coloane au fost modificate din 0 în 1 după momentul t? Dacă au fost k coloane modificate, răspunsul la interogare este $n-k$.

Similar, dacă ultima resetare a liniei i a fost la momentul de timp t (operația cu numărul t) și la valoarea 1, și dacă ulterior k coloane au fost modificate din 1 în 0, atunci răspunsul la interogare este chiar k.

Astfel, trebuie să răspundem la întrebări de tipul: cîte modificări există la valoarea v la timpi $> t$? De aceea, vom ține două AIB-uri pe coloane, indexate după timp (adică după numărul operației), în care stocăm timpul ultimei resetări a fiecărei coloane în 0 și respectiv în 1.

De asemenea, cînd primim o interogare, trebuie să știm timpul ultimei modificări a acelei linii sau coloane, ceea ce putem stoca naiv: un vector de perechi (timp, valoare).

Informațiile pe linii și pe coloane sînt perfect simetrice. Vom studia codul ca să vedem cum putem elimina duplicarea codului. Asta doar dacă evitarea duplicării codului este importantă pentru noi. 😉

3.10.7 Problema Ball (Codeforces)

enunț • sursă

Să abstractizăm problema: date fiind n puncte în spațiu, cîte dintre ele sînt dominate de un alt punct? Spunem că un punct (x, y, z) domină un punct (x', y', z') dacă $x > x'$, $y > y'$ și $z > z'$.

Ca și la problema Subsequences, un prim pas este să reducem interogările tridimensionale la interogări bidimensionale. Să sortăm punctele descrescător după z. Dacă toate z-urile ar fi diferite, atunci am ști că toate punctele procesate anterior au z-ul mai mare decît toate cele viitoare. Dat fiind că z-urile pot fi egale, trebuie să ne adaptăm. Este suficient să procesăm punctele în grupuri cu același z. Pentru toate punctele dintr-un grup calculăm întîi răspunsul și abia apoi le adăugăm la structura de date (oricare ar fi ea).

Astfel, problema pentru punctul $p(p_x, p_y, p_z)$ devine: există vreun punct văzut anterior care să aibă și x-ul și y-ul mai mare? Interogarea pare bidimensională: trebuie să aflăm dacă există vreun punct în cadranul de la (p_x, p_y) la (∞, ∞). Iar coordonatele sînt prea mari pentru un AIB 2D.

Aici intervine ultimul artificiu. Dorim să reducem problema la o interogare unidimensională pe sufix: dintre toate punctele văzute anterior, considerîndu-le doar pe cele cu $x > p_x$, există vreunul cu $y > p_y$? Putem reformula această întrebare ca pe una de maxim: Dă-mi maximul lui y pe domeniul $[p_x, \infty)$. Dacă acest maxim este $> p_y$, atunci există un punct care îl domină pe p, altfel nu.

Putem răspunde la aceste întrebări cu un AIB de maxime, cu două observații:

Trebuie să normalizăm coordonatele x la intervalul $[1, n]$. Nu ne interesează valorile exacte, ci doar relațiile între ele.

AIB-ul de maxime știe să calculeze doar maxime pe prefix, nu pe sufix. Dar putem să reflectăm toate valorile x normalizate față de n pentru a transforma interogările pe sufix în interogări pe prefix.

Observație tangențială: Întotdeauna estimați mărimea fișierului de intrare! În acest caz, avem 1,5 milioane de numere pe 9 cifre plus spații, deci circa 15 MB. Timpul de rulare este efectiv dominat de citire. Sursa inclusă a rulat în 1.100 ms. O a doua sursă, cu citire rapidă, a rulat în 170 ms.

3.10.8 Problema Medwalk, revizitată (Lot 2025)

enunț • sursă

Am discutat această problemă și în capitolul de arbori de intervale. Am găsit o soluție bazată pe un AINT de seturi, construit peste valorile din matrice. Să vedem acum una de 5 ori mai rapidă, probabil cea pe care a dorit-o comisia.

Primele observații se mențin. Decuplăm (conceptual) matricea în doi vectori, unul cu minimele și unul cu maximele de pe fiecare coloană. Pentru a minimiza medianul (scorul unui interval $[l, r]$), căutăm un drum minim lexicografic. Acesta va consta din minimele de pe intervalul $[l, r]$ și din minimul maximelor. Medianul acestei mulțimi va fi una din trei valori posibile (logica deciziei este simplă):

- fie minimul maximelor;
- fie medianul minimelor;
- fie elementul anterior medianului minimelor.

Pentru cazul (1), minimul maximelor îl menținem într-un aint simplu, cu actualizări punctuale și cu interogări de minim pe interval. Pentru cazurile (2) și (3) trebuie să răspundem la interogări de al k-lea element pe interval. Aici introducem următoarea soluție, constînd dintr-un AIB offline 2D construit peste minimele din matrice.

Dorim să căutăm binar al k-lea element. Bunăoară, prima dată ne întrebăm: este el mai mare decît $V_{max}/2$? Dacă da, îl căutăm între $V_{max}/2$ și V_{max}. Dacă nu, îl căutăm între 1 și $V_{max}/2$. În general, pentru plaja de valori curentă, $[x, y]$, vom calcula mijlocul $m = (x+y)/2$ și îi vom adresa structurii de date întrebarea: cîte valori între x și m apar la intrare pe poziții între l și r? Dacă

răspunsul este $\geq k$, atunci al k-lea element are valoarea între x și m. Altfel, el are valoarea între m și y.

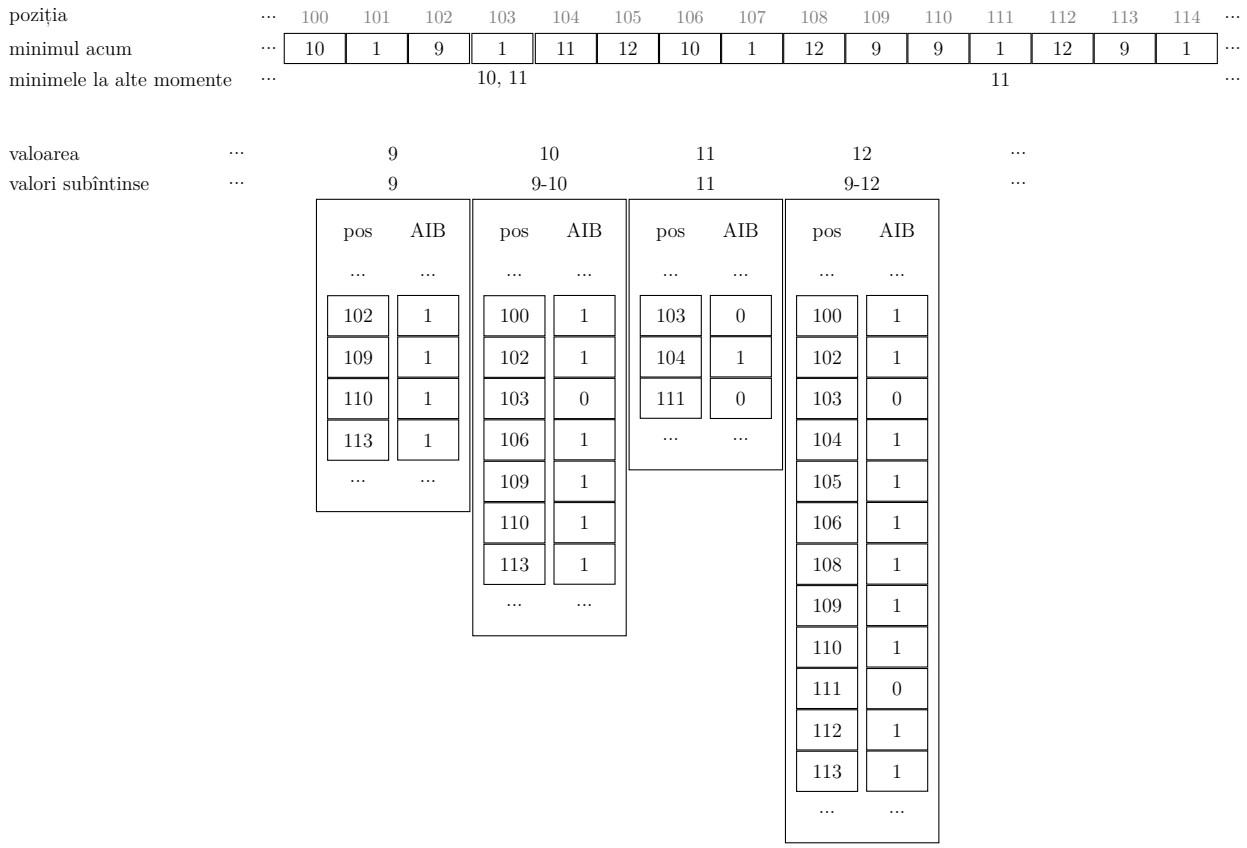
Să încercăm o structură naivă: o matrice binară uriașă A de dimensiuni $n \times V_{max}$, unde $A_{p,v}$ reține 1 dacă valoarea v apare în matrice la poziția p, 0 altfel. Atunci răspunsul la întrebarea „cîte valori între v_1 și v_2 apar pe poziții între l și r?" este suma dreptunghiului $[l, r] \times [v_1, v_2]$. Sună bine, dar memoria nu ne permite această matrice.

De aceea, vom comprima fiecare coloană într-un AIB pornind de la următoarea observație. Coloana v va reține 1, oricînd pe durata întregului program, doar pe liniile p cu proprietatea că poziția p în vectorul de minime va avea, cel puțin după o operație, valoarea v. De aceea, în primă fază simulăm operațiile și colectăm toate minimele posibile pe toate pozițiile, care vor fi cel mult $n + q$. (De aici provine partea de *offline* din nume.) Pe fiecare coloană v stocăm:

- O listă a pozițiilor posibile pentru valori 1, ordonate crescător.
- Un AIB de 0/1 construit peste această listă.

De exemplu, dacă valoarea 9 este măcar o dată minimă pe pozițiile 102, 109, 110, 113, atunci pe coloana 9 stocăm vectorul $[102, 109, 110, 113]$ și un AIB de 0/1 cu 4 poziții disponibile.

Dar facem mai mult decît atît, căci v-am promis un AIB 2D. 🙂 Vectorul de coloane este el însuși un AIB. Coloana v nu reține doar pozițiile aparițiilor lui v, ci ale oricărei valori din $(v - p, v]$, unde p este cea mai mare putere a lui 2 mai mică sau egală cu v. În particular, cele 4 valori 102, 109, 110, 113 exemplificate mai sus nu apar doar în lista coloanei 9, ci și în listele coloanelor $10, 12, 16, 32, 64, \ldots$.



Să observăm că fiecare poziție apare în lista unui număr logaritmic de coloane. De aceea, suma lungimilor tuturor listelor (și a tuturor AIB-urilor) este $\mathcal{O}(n \log V_{max})$.

Pe această structură putem răspunde în timp $\mathcal{O}(n \log n \log V_{max})$ la interogări de al k-lea element.

3.11 Interogări punctuale și actualizări pe interval

Putem privi actualizările pe interval ca pe un fel de „Șmen al lui Mars online". În șmenul lui Mars prelucrăm operațiile „adaugă x pe pozițiile $[l \dots r]$" adăugînd x pe poziția l și $-x$ pe poziția $r+1$. Deosebirea este că în Șmenul lui Mars interogările vin doar la sfîrșit, după toate actualizările, pe cînd în varianta online interogările pot veni și pe parcurs.

AIB-ul poate fi adaptat foarte ușor la această nevoie:

1. Actualizare: adaugă x pe poziția l și $-x$ pe poziția $r+1$.
2. Interogare pe poziția k: calculează suma prefixului $[1 \dots k]$.

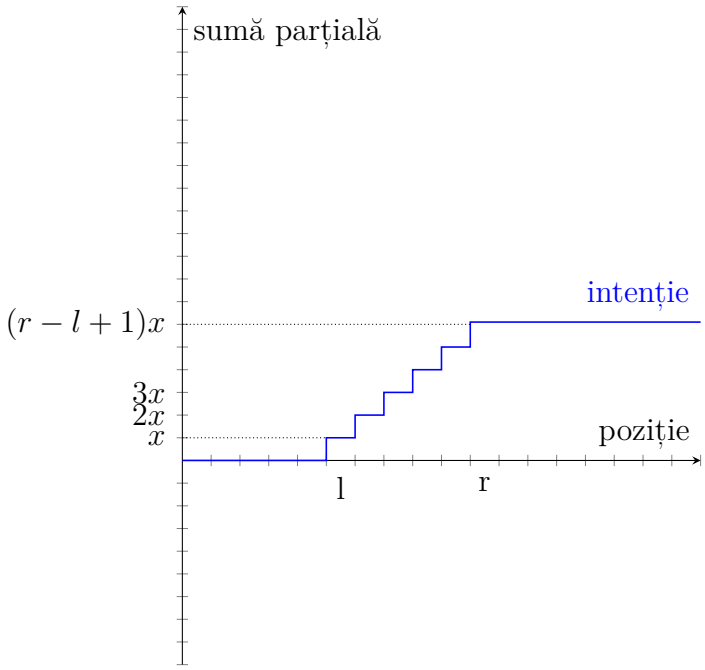
Aceasta funcționează deoarece, la interogarea pe poziția k,

1. Dacă $k < l$, atunci suma prefixului $[1 \dots k]$ nu va include pozițiile l și $r+1$.
2. Dacă $k > r$, atunci suma prefixului $[1 \dots k]$ va include pozițiile l și $r+1$, care se vor anula reciproc. Dorim aceasta, deoarece $k \notin [l, r]$, deci variația intervalului $[l, r]$ nu trebuie să afecteze poziția k.
3. Dacă $l \leq k \leq r$, atunci suma prefixului $[1 \dots k]$ va include doar poziția l, nu și poziția $r+1$, deci poziția k va fi afectată de variația pe intervalul $[l, r]$.

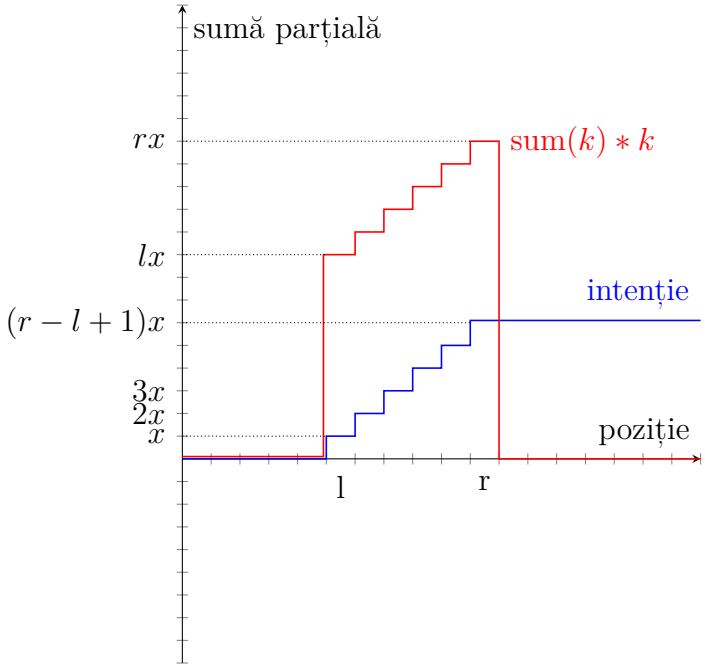
3.12 Interogări și actualizări pe interval

Să rezolvăm și această versiune, doar de amorul artei. Nu cred că discuția de mai jos se aplică la altceva decît la sume (la xor-uri, de pildă).

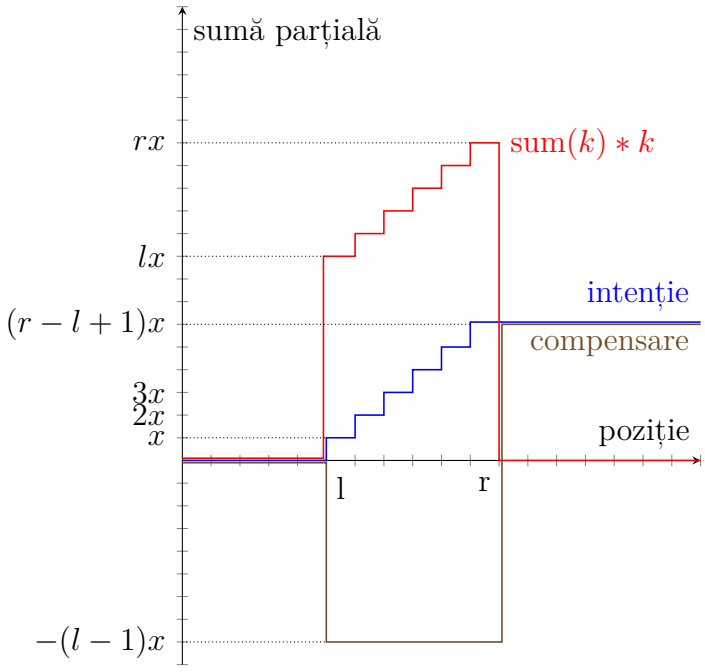
Avînd în vedere că AIB-urile se preocupă în special de sume parțiale, să examinăm grafic sumele parțiale după operația de adăugare a lui x pe intervalul $[l, r]$. Pe pozițiile $[l \dots r]$ ia naștere o funcție scară: dorim ca sumele parțiale să crească cu x, $2x$, ..., $(r-l+1)x$.



Intuitiv, deoarece suma parțială crește cu poziția, sîntem tentați să returnăm, la poziția k, valoarea $\text{sum}(k) \times k$. Pentru ca după r suma parțială să se oprească din crescut, adăugăm x la poziția l și scădem x la poziția $r+1$, similar cu șmenul lui Mars. Doar că atunci funcția $\text{sum}(k) \times k$ are graficul:



Într-adevăr, observăm că suma parțială la stînga lui l este 0, iar la dreapta lui r este $x - x = 0$. Totuși, pare că cele două grafice nu au nicio legătură! Dar nu este chiar așa. Observăm că diferența între cele două funcții arată relativ simplu:



Pentru a corecta al doilea grafic ca să arate ca primul, trebuie să menținem un al doilea AIB în care:

- să scădem pe pozițiile $l, \ldots, r$ valoarea $(l-1)x$;
- să adăugăm pe pozițiile $r+1, \ldots, n$ valoarea $(r-l+1)x$.

În termeni de sume parțiale, trebuie:

- să scădem pe poziția l valoarea $(l-1)x$;
- să adăugăm pe poziția $r+1$ valoarea rx.

Astfel ia naștere codul care, dacă nu-l înțelegem, poate părea foarte ezoteric:

```
struct fenwick_tree_2 {
  fenwick_tree v, w;

  void from_array(long long* src, int n) {
    v.n = n;
    w.from_array(src, n);
  }

  void range_add(int l, int r, long long val) {
    v.add(l, val);
    v.add(r + 1, -val);
    w.add(l, -val * (l - 1));
    w.add(r + 1, val * r);
  }

  long long prefix_sum(int pos) {
    return v.prefix_sum(pos) * pos + w.prefix_sum(pos);
  }
```

```
  long long range_sum(int l, int r) {
    return prefix_sum(r) - prefix_sum(l - 1);
  }
};
```

3.13 Arbori indexați binar 2D

Putem extinde arborii indexați binar la matrice, cu costuri $\mathcal{O}(\log^2 n)$ pentru operații. Să considerăm următoarele operații (*point update, rectangle query*):

- `1 row col val`: Adaugă val la coordonatele (row, col)
- `2 row1 col1 row2 col2`: Calculează suma dreptunghiului (row_1, col_1) - (row_2, col_2) inclusiv.

3.13.1 Structura informației

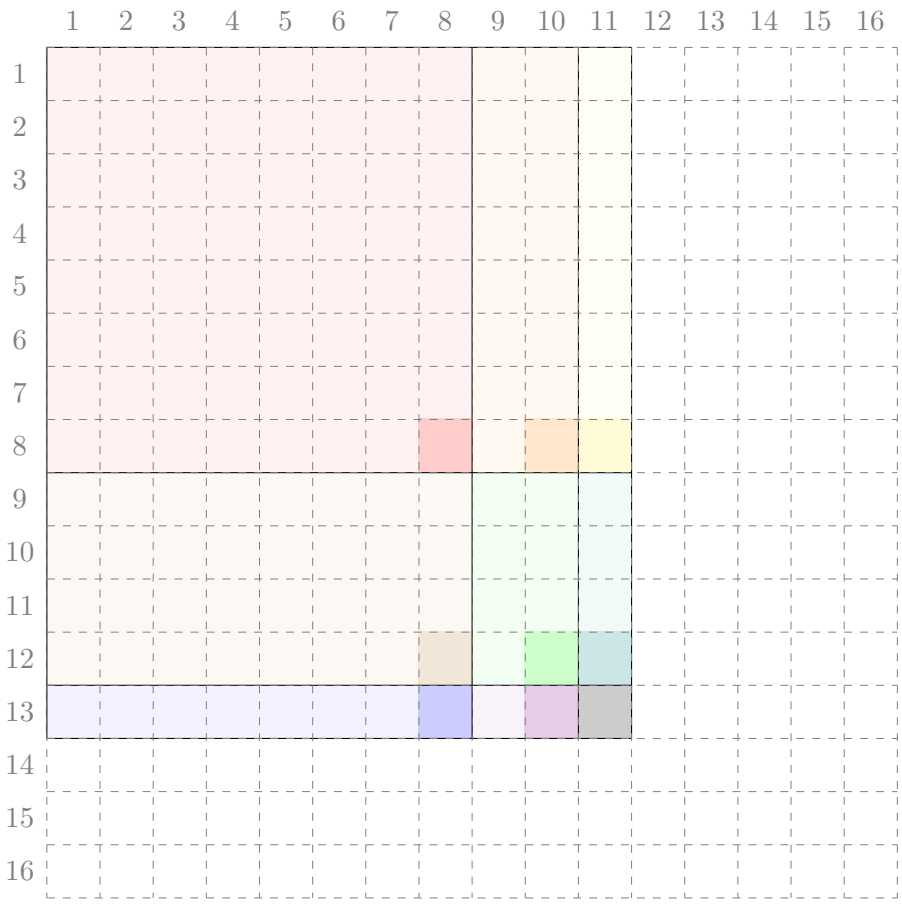
Așa cum arborele 1D modifică structura informației din vector, arborele 2D modifică structura informației din matrice. El stochează anumite sume parțiale. Mai exact, arborele stochează la poziția (r, c) suma elementelor din submatricea (originală) de dimensiuni $p \times q$ cu colțul dreapta-jos la coordonatele (r, c), unde

- p este cea mai mare putere a lui 2 care îl divide pe r
- q este cea mai mare putere a lui 2 care îl divide pe c.

De exemplu, la poziția 40, 60 arborele stochează suma elementelor din matricea de dimensiuni 8×4 cuprinsă între liniile 33 și 40 și coloanele 57 și 60.

3.13.2 Calculul sumei dintr-un dreptunghi

Putem folosi această structură pentru a calcula suma dreptunghiurilor de forma $(1, 1) - (r, c)$. Iată o figură pentru $r = 13$ și $c = 11$.



Dreptunghiul de dimensiune 13×11 se compune din 3×3 dreptunghiuri cu laturile puteri ale lui 2. Trebuie să însumăm valorile din colțurile jos-dreapta ale acestor dreptunghiuri (desenate mai întunecat). Codul este:

```
int prefix_sum(int row, int col) {
  int s = 0;

  for (int r = row; r; r &= r - 1) {
    for (int c = col; c; c &= c - 1) {
      s += mat[r][c];
    }
  }

  return s;
}
```

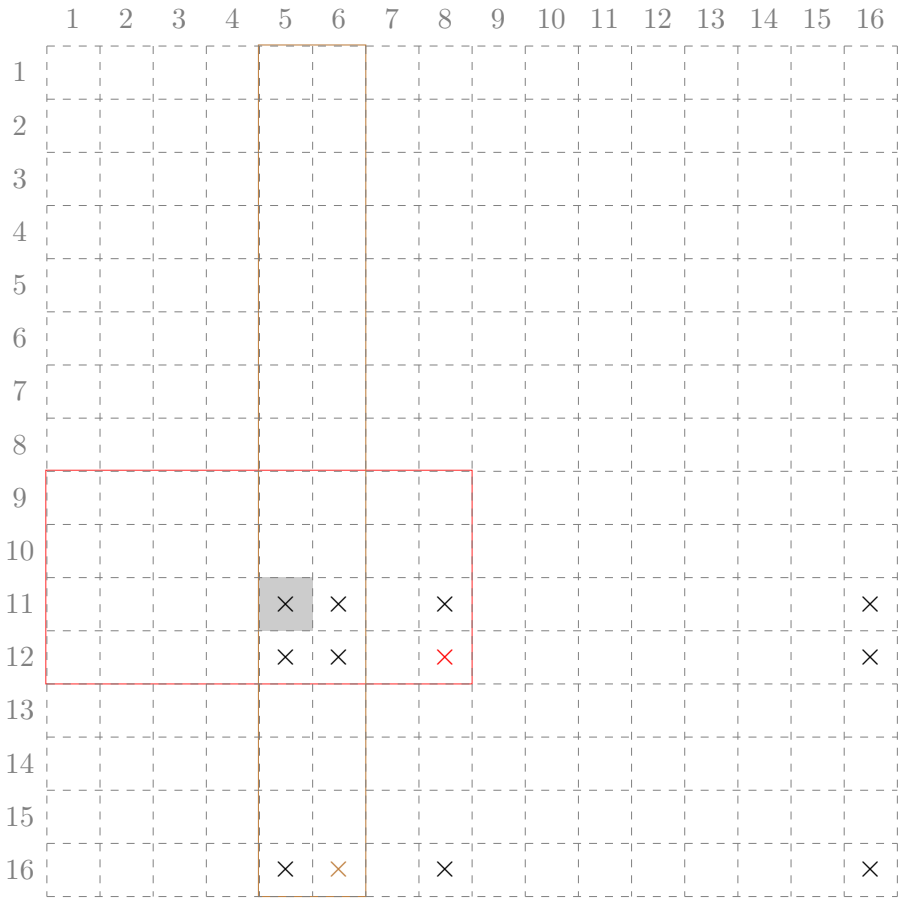
Desigur, putem calcula prin includeri și excluderi suma unui dreptunghi arbitrar:

```
int rectangle_sum(int row1, int col1, int row2, int col2) {
  return
    + prefix_sum(row2, col2)
    - prefix_sum(row2, col1 - 1)
    - prefix_sum(row1 - 1, col2)
    + prefix_sum(row1 - 1, col1 - 1);
```

```
}
```

3.13.3 Actualizări punctuale

Ca și la varianta 1D, cînd modificăm valoarea de la coordonatele (r, c) trebuie să modificăm corespunzător toate dreptunghiurile care includ acele coordonate. Iată un exemplu pentru linia 11, coloana 5:



Se observă că trebuie actualizate liniile 11, 12 și 16, adică exact cele care ar include poziția 11 într-un AIB unidimensional. Similar, trebuie actualizate coloanele 5, 6, 8 și 16, adică exact cele care ar include poziția 5. Am evidențiat cu roșu și cu auriu două dintre aceste coordonate, $(12, 8)$ și $(16, 6)$, și dreptunghiurile aferente lor, pentru a evidenția că ele includ celula $(11, 5)$.

Codul pentru actualizare este:

```
void add(int row, int col, int val) {
  for (int r = row; r <= n; r += r & -r) {
    for (int c = col; c <= n; c += c & -c) {
      mat[r][c] += val;
    }
  }
}
```

3.13.4 Construcția în $\mathcal{O}(n^2)$

Și acest arbore poate fi construit in-place, refolosind matricea dată la intrare și în timp $\mathcal{O}(1)$ per element.

În varianta unidimensională, trebuia să propagăm valoarea de la poziția x la poziția $x + (x \,\&\, -x)$. În varianta bidimensională am vrea să procedăm astfel:

- Fie $r' = r + (r \,\&\, -r)$ și $c' = c + (c \,\&\, -c)$.
- Propagăm valoarea de la poziția (r, c) la poziția (r, c'). De acolo, ea se va propaga automat la coloanele următoare.
- Propagăm valoarea de la poziția (r, c) la poziția (r', c). De acolo, ea se va propaga automat la liniile următoare, iar pe fiecare linie pe coloanele următoare.

Dar apare o problemă: valoarea de la (r, c) se va propaga la (r', c') de două ori, pe căi diferite! Din fericire, este suficient să o scădem o dată. Iată codul:

```
void build() {
  for (int r = 1; r <= n; r++) {
    for (int c = 1; c <= n; c++) {
      int next_r = r + (r & -r);
      int next_c = c + (c & -c);

      if (next_r <= n) {
        mat[next_r][c] += mat[r][c];
      }

      if (next_c <= n) {
        mat[r][next_c] += mat[r][c];
      }

      if ((next_r <= n) && (next_c <= n)) {
        mat[next_r][next_c] -= mat[r][c];
      }
    }
  }
}
```

Capitolul 4

Descompunere în radical

Pe lîngă faptul că oferă complexitate optimă pentru unele probleme, metoda oferă și un substitut bun pentru algoritmi în $\mathcal{O}(n \log n)$ și în special $\mathcal{O}(n \log^2 n)$, în cazul în care nu găsim ideea. Descompunerea în radical este, în experiența mea, mult mai ușor de implementat.

4.1 Actualizări punctuale

Revenim la problema inițială: actualizări punctuale, sume pe interval. Împărțim vectorul în blocuri de lungime $k = \sqrt{n}$. Așadar, vor exista $\lceil n/k \rceil$ blocuri (engl. *blocks* sau *buckets*). Pentru fiecare bloc, menținem o informație suplimentară: suma elementelor din acel bloc.

Memorie suplimentară: $\mathcal{O}(\sqrt{n})$.

```
int v[MAX_N];
int b[MAX_BUCKETS];
int bs, nb;

// Se poate implementa și cu împărțiri pentru concizie (sînt doar n).
void init_buckets() {
  nb = sqrt(n + 1);
  bs = n / nb + 1;

  for (int i = 0; i < nb; i++) {
    int bucket_start = i * bs;
    for (int j = 0; j < bs; j++) {
      b[i] += v[j + bucket_start];
    }
  }
}

int array_sum(int* v, int l, int r) {
  int sum = 0;
  while (l < r) {
    sum += v[l++];
```

```
  }
  return sum;
}

int fragment_sum(int l, int r) {
  return array_sum(v, l, r);
}

int bucket_sum(int l, int r) {
  return array_sum(b, l, r);
}

int range_sum(int l, int r) { // [l, r)
  int bl = l / bs, br = r / bs;
  if (bl == br) {
    return fragment_sum(l, r);
  } else {
    return
      // capete
      fragment_sum(l, (bl + 1) * bs) +
      fragment_sum(br * bs, r) +
      // blocuri complete
      bucket_sum(bl + 1, br);
  }
}

void point_add(int pos, int val) {
  v[pos] += val;
  b[pos / bs] += val;
}
```

Vă recomand să lucrați pe intervale închise la stînga, deschise la dreapta, ca să simplificați aritmetica.

Încheiem secțiunea cu observația că funcția `range_sum` are complexitatea $\mathcal{O}(k + n/k)$: Ea iterează naiv prin blocurile acoperite parțial, așadar $\mathcal{O}(k)$, și iterează rapid prin blocurile acoperite complet, așadar $\mathcal{O}(n/k)$. Alegem $k = \sqrt{n}$ ca să minimizăm acea sumă, dar vom vedea în unele exemple că pot lua naștere și alte sume care duc la valori diferite pentru k.

4.2 Actualizări pe interval

Folosim o informație suplimentară care amintește de informația propagată *lazy* din arborii de segmente. Pentru problema dată (sume pe interval), am denumit această informație `bdelta`. Ea are semnificația: `bdelta[j]` este o valoare care trebuie adăugată la fiecare element din blocul `j`. Așadar, valoarea reală a unui element `i` din blocul `j` este `v[i] + bdelta[j]`.

```
int fragment_sum(int l, int r, int bucket) {
```

```
  return
    (r - l) * bdelta[bucket] +
    array_sum(v, l, r);
}

int bucket_sum(int l, int r) {
  return
    array_sum(bsum, l, r) +
    bs * array_sum(bdelta, l, r);
}

int range_sum(int l, int r) {
  int bl = l / bs, br = r / bs;
  if (bl == br) {
    return fragment_sum(l, r, bl);
  } else {
    return
      // capete
      fragment_sum(l, (bl + 1) * bs, bl) +
      fragment_sum(br * bs, r, br) +
      // blocuri complete
      bucket_sum(bl + 1, br);
  }
}

void array_add(long long* v, int l, int r, int val) {
  while (l < r) {
    v[l++] += val;
  }
}

void fragment_add(int l, int r, int bucket, int val) {
  bsum[bucket] += (r - l) * val;
  array_add(v, l, r, val);
}

void range_add(int l, int r, int val) {
  int bl = l / bs, br = r / bs;
  if (bl == br) {
    fragment_add(l, r, bl, val);
  } else {
    // capete
    fragment_add(l, (bl + 1) * bs, bl, val);
    fragment_add(br * bs, r, br, val);

    // blocuri complete
    array_add(bdelta, bl + 1, br, val);
  }
}
```

Paranteză: conceptual, aint-urile fac același lucru cu descompunerea în radical, deci multe concepte se translatează între cele două, cum ar fi propagarea *lazy*. Marea inovație a arborilor de intervale este că garantează descompunerea în $\mathcal{O}(\log n)$ blocuri.

4.3 Optimizări

4.3.1 Evitați împărțirile!

Codul anterior face doar două împărțiri per operație. În general, aveți nevoie strict de o împărțire pentru fiecare indice primit ca parametru. Evitați stilul de mai jos, care face $\mathcal{O}(\sqrt{n})$ împărțiri (sau operații modulo) per operație. El poate fi **de cîteva ori mai lent** (vedeți *benchmark*-urile).

```
int bl = l / bs, br = r / bs;
int sum = 0;

// stînga
do {
  sum += v[l++];
} while (l % bs);

// dreapta
while (r % bs) {
  sum += v[--r];
}

// blocuri complete
for (int i = bl + 1; i < br; i++) {
  sum += b[i];
}

return sum;
```

4.3.2 Alegerea mărimii blocurilor

Am auzit că mărimea blocului merită declarată constantă, deoarece compilatorul va optimiza împărțirile. Am experimentat, dar diferențele nu sînt semnificative. Am încercat chiar să declar mărimea o putere a lui 2, pentru ca împărțirile să fie ieftine. Codul rezultat a fost mai lent! Cred că motivul este că se dezechilibrează acea funcție $k + n/k$ pe care descompunerea în radical o optimizează.

Diferența de viteză este vizibilă cînd codul face multe împărțiri. Cînd codul face $\mathcal{O}(1)$ împărțiri per operație, nu mai contează.

Are sens să declarați mărimea constantă dacă vi se pare codul mai simplu (evitați cîteva calcule la inițializare). În acest caz, vă recomand să puneți constanta mică (3-4) cînd lucrați local și să-i

dați valoarea reală cînd trimiteți sursa. Altfel, dacă testați local pe teste mici și mărimea blocului 300, nu veți testa decît codul cu interogări și actualizări într-un singur bloc.

4.3.3 Alegerea mărimii blocurilor pentru operații inegale

Dacă programul vostru depășește timpul, merită să variați mărimea blocurilor în jurul lui $\sqrt{n}$ ca să vedeți dacă se schimbă ceva. Fie b numărul de blocuri și fie l lungimea unui bloc. Operațiile pot depinde doar de una dintre aceste variabile sau pot depinde de ambele, dar în mod inegal. Exemplu: dacă constanta pentru cele două capete de segment (de lungime medie $l/2$) este mult mai mare decît constanta pentru blocurile întregi, merită redus l-ul puțin.

Iată un exemplu mai interesant. Să spunem că nu știm să rezolvăm corect una dintre operații și găsim doar o implementare în $\mathcal{O}(b+l^2)$. Pare catastrofic: dacă alegem $b = l = \sqrt{n}$, avem de fapt o soluție în $\mathcal{O}(n)$. 🤔

Dar se poate asimptotic mai bine. Să alegem $b = n^{2/3}$, caz în care $l = n/b = n^{1/3}$. Dacă $n = 100.000$, atunci $b = 2.174$ și $l = 46$. O soluție în $\mathcal{O}(\sqrt{n})$ per operație ar face circa 600 de operații, pe cînd a noastră va face circa 4.300 de operații. Desigur, diferența este notabilă, dar, cu o implementare eficientă, avem șanse să „ținem aproape".

Pauză de matematică: mai exact, noi vrem să găsim minimul funcției

$$f(x) = x^2 + \frac{n}{x}$$

Funcția își atinge minimul cînd derivata este zero, iar derivata este

$$f'(x) = 2x - \frac{n}{x^2} = \frac{2x^3 - n}{x^2}$$

Rezultă că l minim este $\sqrt[3]{n/2} \approx 37$, iar funcția va face sub 4.100 de operații.

4.4 Probleme

4.4.1 Problema Mexitate (ONI 2018 clasa a 9-a)

enunț • surse

Această problemă nu mi se pare nici în ruptul capului de clasa a 9-a. Posibil de baraj juniori.

Am notat cu m numărul de linii deoarece m vine înaintea lui n în alfabet, după cum și k vine înaintea lui l în alfabet.

Să luăm în calcul soluția naivă: calculăm mex-ul matricei din colțul stînga-sus, apoi translatăm în diverse feluri matricea pentru a recalcula incremental mex-urile celorlalte ferestre. De exemplu, putem urma un traseu șerpuit: translatăm matricea la dreapta pînă la capăt, apoi o dată în jos, apoi în stînga pînă la capăt etc.

La fiecare translație, menținem într-o structură S informații despre frecvența elementelor din matrice. Ștergem din structură elementele care ies din fereastră și le inserăm pe cele care intră în fereastră. Atunci complexitatea translatărilor va fi $\mathcal{O}(mnk + ml)$. Ca să minimizăm efortul, rotim sau transpunem matricea cînd $k > l$. Astfel, $k \leq l$ și complexitatea translatărilor va fi $\mathcal{O}(mn\sqrt{mn})$.

Odată ce am adus raționamentul pînă aici, eu am și trimis o sursă, cu structura S implementată naiv ca vector de frecvențe. Astfel m-am asigurat că restul programului funcționează, căci el are suficient de multă logică (matrice de dimensiuni arbitrare, transpoziții, șerpuire...). Deoarece am gîndit totul modular, am programat structura `frequency_tracker` să expună funcțiile `add`, `remove` și `mex`. Pentru versiunea optimizată, doar am rescris acele funcții.

Pentru punctaj maxim, ce ne dorim de la structura S? Dorim să facem $\mathcal{O}(mn\sqrt{mn})$ inserări și ștergeri și $\mathcal{O}(mn)$ calcule de mex. Rezultă că ne permitem $\mathcal{O}(\sqrt{mn})$ per apel de mex, dar avem nevoie de $\mathcal{O}(1)$ pe inserare și ștergere.

Atunci putem folosi descompunerea în radical. Stocăm vectorul naiv de frecvențe. În plus, în fiecare bloc stocăm numărul de valori nenule. Inițial această valoare este 0. Ea crește cu 1 ori de cîte ori frecvența unui element din bloc crește de la 0 la 1 și, invers, scade cu 1 ori de cîte ori frecvența unui element din bloc scade de la 1 la 0. Funcția mex caută din bloc în bloc pînă cînd găsește un bloc cu cel puțin o valoare nulă, apoi caută naiv prin acel bloc.

4.4.2 Problema Give Away (SPOJ)

enunț • surse

Problema are limită de timp mare (1-2 secunde, iar conform paginii *status*, chiar peste 6 secunde). Acesta este un indiciu bun că o soluție în $O((n+q)\sqrt{n})$ este arhisuficientă. Doar că pare mai rău de atît! Să presupunem că ținem pe fiecare bloc o structură S (nu știm încă ce). Atunci:

- Sigur putem procesa actualizările în $O(\sqrt{n})$, chiar și naiv la o adică.
- La căutare, pare simplu să procesăm naiv blocurile acoperite parțial, în $O(\sqrt{n})$.
- Dar ce facem cu blocurile acoperite complet? Dacă folosim orice structură echilibrată, introducem un factor logaritmic în plus pentru căutarea lui c.

Rezultă o complexitate de $\mathcal{O}(q\sqrt{n}\log n)$. Dar în 6 secunde, este acceptabil.

Mărimea blocurilor

Dacă am vrea să optimizăm suma $k + n/k$, am alege $k = \sqrt{n} \approx 707$. Dar noi dorim să optimizăm suma $k + n/k \log k$. Ochiometric log-ul va fi undeva între 7 și 12, deci este important să-l mărim pe k. Atunci $\log k$ va crește lent, dar n/k va scădea rapid. Din cîteva încercări pe hîrtie aflăm că valoarea optimă pentru k este 2.000 sau 3.000. Într-adevăr, timpii de execuție pe acolo ating optimul.

Detalii de implementare

Așadar, dorim o structură S care să admită inserări, ștergeri, și numărarea elementelor mai mari sau egale cu o valoare dată. Eu am izolat această structură într-un `struct` și am scris o primă implementare naivă (S este doar un vector). Cu doar 5 linii de cod în plus, sursa naivă mă ajută să verific corectitudinea restului programului.

A doua încercare a fost cu structuri de date din STL, care trece în 3 secunde. Dar este nevoie să memorați papagalicește două noțiuni (le vom relua în capitolele viitoare):

1. `set`-ul simplu nu este suficient, căci el poate să caute valoarea c, dar nu și să numere elementele mai mari sau egale cu c. Este nevoie de structuri cu statistici de ordine (PBDS).
2. Blocurile pot conține valori egale, deci ne trebuie un multiset. Multisetul PBDS este o încropeală, iar ștergerea trebuie rescrisă.

Dar a treia încercare este elementară și trece în 1.3 secunde: pe fiecare bloc ținem vectorul original (ca să știm ce element înlocuim) și o copie sortată. Putem căuta binar în acea copie, iar la inserare/ștergere o actualizăm prin deplasări naive.

4.4.3 Problema Holes (Codeforces)

enunț • sursă

Îmi place această problemă pentru că este nestandard. Nu facem efectiv împărțirea în blocuri și nu stocăm informații agregate pe fiecare bloc. În loc de aceasta, fiecare poziție *pos* reține informații ca să își accelereze trecerea prin blocul său:

- care este ultima destinație din același bloc pe care o vizitează o bilă pornind de la *pos*;
- cîte salturi face bila pînă la acea destinație.

Atunci operațiile sînt:

- La interogare, urmărim traseul bilei din bloc în bloc, în timp $\mathcal{O}(\sqrt{n})$.
- La actualizare, trebuie să recalculăm poziția modificată și toate pozițiile din stînga ei, din același bloc, tot în $\mathcal{O}(\sqrt{n})$.

Sursa mea se apropie de limita de timp (700 ms din 1000 ms). Am încercat următoarea optimizare care cred că ajută. Am ales o mărime mai mare pentru blocuri, 1.000 în loc de cea teoretică ($\sqrt{100.000} \approx 316$). Astfel accelerăm interogările, care traversează mai puține blocuri, în defavoarea actualizărilor, care au de recalculat mai multe poziții. Dar actualizările operează foarte local, pe 1.000 de poziții vecine și vor beneficia de cache. În schimb, interogările sar de colo-colo prin vector.

Remarc și că cele mai rapide soluții ajung la 122 ms. Ele folosesc *link-cut trees* pentru a obține $\mathcal{O}(n \log n)$. Putem percepe structura ca fiind arborescentă: părintele unei poziții este poziția unde sare bila. Atunci actualizările schimbă structura arborelui, ancorînd poziția modificată de un alt părinte. De aici (cred) decurge soluția cu *link-cut trees*.

4.4.4 Problema Piezișă (Baraj ONI 2022)

enunț • surse

Un element comun tuturor soluțiilor este: renumerotăm vectorul de la 1. Acum, dacă xorul pe $[l, r]$ este 0, atunci xorurile pe $[0, l-1]$ și $[0, r]$ sînt egale. Așadar, calculăm xorurile parțiale și le normalizăm, ca să fie indexabile. Acum răspunsul la orice interogare $[l, r]$ este o pereche (x, y) cu $0 \leq x < l$, $r \leq y \leq n$ și $v[x] = v[y]$.

Brute force de 100p

Menționez pentru completitudine că următorul *brute force* optimizat ia 100p: Răspundem la interogări online, imediat ce le primim. Fie o interogare $[l, r]$. Căutăm binar ultima apariție a lui $v[r]$ pe o poziție anterioară lui l (pentru aceasta, colectăm în prealabil listele de poziții pentru fiecare valoare distinctă din v). Fie această poziție q. Atunci avem o soluție de mărime r–q, posibil ∞ dacă elementul $v[r]$ nu apare înainte de poziția l. Repetăm aceeași întrebare la pozițiile $r+1, r+2, \ldots$. Menținem minimul răspunsurilor la aceste căutări, fie el m. Ne oprim la poziția $l+m$ (sau, desigur, la n), deoarece dincolo de ea am putea primi doar răspunsuri mai mari decît optimul curent. Afișăm răspunsul m.

Metoda 1

Fără sortarea interogărilor nu am găsit nicio soluție. Așadar, să sortăm interogările după capătul stîng și să răspundem la ele baleind l de la 1 la n. Acum, pentru o interogare $[l, r]$, dorim ca pentru fiecare poziție $r' \geq r$ să aflăm dacă valoarea $v[r']$ există pe vreo poziție $l' \leq l$. Aceasta este mult prea scump, deci dorim cumva să răspundem la întrebări pentru mai multe poziții simultan.

Să descompunem vectorul în bucăți de mărime $\sqrt{n}$. Pentru o interogare $[l, r]$, fie blocurile celor două capete bl și br. Atunci răspunsul poate avea capătul stîng fie în blocul bl (în stînga lui l), fie într-un bloc anterior. Similar pentru capătul drept. Cazul care ne încurcă este cel în care ambele capete sînt în blocurile bl, respectiv br. Vrem să evităm să comparăm naiv aceste $\mathcal{O}(\sqrt{n} \times \sqrt{n}) = \mathcal{O}(n)$ posibilități.

Astfel ne vine ideea să calculăm o singură informație pentru toate pozițiile din stînga lui l. Fie $left[x]$ ultima poziție (înaintea lui l) pe care apare valoarea x. Evident, putem menține vectorul $left$ în $\mathcal{O}(1)$ pe măsură ce l avansează.

Capătul drept al interogării poate varia oricum, și aici intervine descompunerea în radical. Fie $best[b]$ cea mai mică soluție cunoscută pînă în prezent între orice valoare din blocul b și orice valoare din stînga lui l. Putem menține vectorul $best$ în $\mathcal{O}(1)$ per bloc pe măsură ce l avansează (așadar efort $\mathcal{O}(n\sqrt{n})$ efort global). Pentru aceasta, trebuie să cunoaștem, pentru elementul în curs de procesare $v[l]$, cea mai din stînga apariție a sa în fiecare bloc următor. Putem precalcula această informație la început, într-o manieră similară cu colectarea listelor de apariții a fiecărei valori. Vezi vectorul `ptr` și funcția `preprocess_values`.

Cu aceste informații, răspunsul pentru interogarea curentă $[l, r]$ este minimul dintre:

- Valorile *best* ale blocurilor mai mari decît br.
- Diferențele $r' - left[v[r']]$ pentru elementele din blocul br începînd cu poziția r.

Codul este lung, cam urît și cam lent, dar ia 100p.

Metoda 2

Citind surse mai rapide decît a mea, am găsit-o pe aceasta, care folosește o metodă mai simplă. Principala diferență este că sortează interogările diferit: crescător după blocul lui l, iar la egalitate descrescător după r. Această sortare amintește de algoritmul lui Mo, pe care îl vom discuta în curînd.

Vectorul $left$ are aceeași definiție ca mai înainte, dar include doar elementele dinaintea blocului bl.

La procesarea unui bloc, avem avantajul că toate r-urile scad. De aceea, și la dreapta putem ține un vector $right$ similar cu $left$: $right[x]$ indică prima apariție a lui x pe o poziție mai mare decît r-ul curent. Atenție, vectorul $right$ trebuie golit și recalculat pentru fiecare bloc bl. Acest efort este $\mathcal{O}(n\sqrt{n})$, deci acceptabil.

Pe măsură ce r scade, menținem și valoarea minimă m a oricărei perechi $right[x] - left[x]$. Deoarece r doar scade, intervalele doar scad, deci m poate doar să scadă.

Acum, pentru $[l, r]$, răspunsul poate fi:

- chiar m, dacă soluția are capătul stîng înaintea blocului bl;
- altfel, minimul dintre $right[x] - x$ pentru toate valorile x din blocul bl, din stînga lui l.

Implementarea este conceptual mai simplă și de peste două ori mai rapidă!

4.5 Descompunere după operații

Iată acum o tehnică înrudită. Proiectăm o structură relativ naivă pentru procesarea operațiilor. Prin „naiv” înțelegem, de exemplu, inserarea într-un vector prin deplasarea elementelor, fără a folosi structuri de date echilibrate.

Facem aceste operații naive cîtă vreme ele se încadrează în $\mathcal{O}(\sqrt{n})$ sau $\mathcal{O}(\sqrt{q})$ per operație.

Periodic, trebuie să intervenim pentru ca structura naivă să nu degenereze în timp și să nu ajungă la $\mathcal{O}(n)$ sau $\mathcal{O}(q)$. De aceea, aproximativ o dată la $\sqrt{q}$ operații iterăm prin structură și o „consolidăm” (ce înseamnă asta depinde de la problemă la problemă). Această consolidare poate dura $\mathcal{O}(n)$, pentru ca efortul total al consolidărilor să fie $\mathcal{O}(n\sqrt{q})$. Această limită de timp face consolidările să fie facile, în general.

Să studiem o problemă concretă.

4.6 Probleme

4.6.1 Problema Serega and Fun (Codeforces)

enunț • surse

Trebuie să procesăm eficient operațiile:

1. Rotește circular la dreapta, cu o poziție, un interval $[l, r]$.
2. Raportează frecvența unei valori k într-un interval $[l, r]$.

Iată întîi soluția „clasică", cu descompunere în blocuri de mărime egală. Soluția relativ directă. În fiecare bloc putem menține:

1. O listă a elementelor.
2. Informații despre frecvență (`map` sau vector simplu).

Atunci, la rotire, trebuie să:

1. Rotim naiv blocurile acoperite parțial.
2. Rotim eficient blocurile acoperite complet. La rotire, adăugăm la începutul listei elementul care ne parvine de la blocul anterior și trimitem ultimul element din listă în blocul următor.

La interogare, trebuie să:

1. Consultăm element cu element blocurile acoperite parțial.
2. Consultăm vectorii de frecvență ai blocurilor acoperite complet.

Discuție despre implementarea cu structuri STL

Putem rezolva problema și cu structuri ca `deque`, `map` sau `unordered_map`, dar este nevoie de o implementare atentă ca să nu depășim timpul și memoria.

În particular, mare atenție la operatorul `[]`! Este tentant să îl folosim ca să aflăm frecvența unui element. Dacă elementul nu există în `map`, operatorul va returna 0, ceea ce este corect. Dar **operatorul inserează elementul** dacă nu exista deja.

Ce înseamnă asta? În mod normal, suma mărimilor tabelelor hash din toate blocurile va fi n. Dar, dacă pentru fiecare din cele q interogări noi căutăm o valoare inexistentă în fiecare dintre cele $\sqrt{n}$ blocuri, și dacă toate acele valori sînt inserate, suma mărimilor tabelelor va ajunge la $q\sqrt{n}$. Necesarul de memorie va crește enorm. Este obligatoriu să folosim iteratori pentru căutarea sau decrementarea frecvențelor, pentru ca mărimea tabelelor să rămînă $\mathcal{O}(\sqrt{n})$.

Detalii de implementare

Prime sursă stochează lista de elemente din fiecare bloc într-un `deque` din STL. A doua sursă folosește un simplu buffer circular: un vector de mărime `BUCKET_SIZE` împreună cu un pointer la primul element din listă. Atunci putem face rotirea completă în $\mathcal{O}(1)$ mutînd pointerul de start

cu un element spre stînga. Pe poziția noului element de start scriem valoarea provenită din blocul anterior.

O altă diferență este că prima sursă stochează frecvențele din fiecare bloc într-o tabelă hash, pe cînd a doua folosește un vector de frecvențe, căci elementele au valori de cel mult n. Necesarul de memorie crește la $\mathcal{O}(n \times numBuckets)$. Din acest motiv, merită să experimentăm cu mai puține blocuri de dimensiune mai mare. Într-adevăr, pentru blocuri de mărime $2\sqrt{n} \approx 640$, timpul scade la jumătate față de $\sqrt{n}$.

Mai remarcăm că frecvența într-un singur bloc încape pe tipul `short`, nu este nevoie de `int`. Doar cu această modificare am redus timpul de rulare cu 20%. Am experimentat și cu `unsigned char`, dar atunci frecvența maximă stocabilă ar fi 256, deci mărimea maximă a unui bloc ar trebui să fie 256, iar timpul se înrăutățește.

Soluție cu descompunere după operații

Iată acum și rezolvarea în care lăsăm blocurile să fluctueze ca lungime. La rotire, extragem efectiv elementul de pe poziția r din blocul său și îl inserăm la indicele corect în blocul poziției l. Blocurile dintre l și r rămîn nemodificate. Ca fapt divers, complexitatea rotirii depinde doar de mărimea blocului, nu și de numărul de blocuri.

Cu timpul, lungimile blocurilor pot degenera, ceea ce poate încetini operațiile: dacă un bloc devine foarte mare, atunci inserarea, ștergerea și numărarea de elemente din acel bloc vor fi lente. De aceea, periodic refacem structura blocurilor:

1. Colectăm toate blocurile într-un vector. Acesta este chiar conținutul real al vectorului.
2. Redistribuim elementele în blocuri de mărime egală.

Putem face redistribuirea fie periodic (la fiecare $\mathcal{O}(\sqrt{q})$ operații), fie la nevoie, cînd în momentul inserării detectăm că unul dintre blocuri a atins un anumit prag, să zicem dublu față de lungimea inițială. În ambele cazuri, complexitatea rămîne $\mathcal{O}(q\sqrt{n})$.

Această implementare este relativ elementară și se mișcă de circa două ori mai repede decît precedenta!

4.7 Procesări diferite înainte și după $\sqrt{n}$

Următoarele două secțiuni teoretice prezintă două tehnici care ating timp $\mathcal{O}(n\sqrt{n})$ din motive matematice. Tehnicile nu duc la descompunere în radical „convențională”, cu blocuri, dar nu știu unde altundeva să le încadrez.

Prima tehnică pornește de la observațiile:

1. Între 1 și $\sqrt{n}$ există $\sqrt{n}$ valori[1].
2. Valorile de forma $\lfloor n/k \rfloor$, cu $k > \sqrt{n}$, iau doar $\mathcal{O}(\sqrt{n})$ valori distincte.

[1] From the Department of Redundancy Department.

4.8 Probleme

4.8.1 Problema Time to Raid Cowavans (Codeforces)

enunț • sursă

În foarte multe cuvinte, problema ne dă un vector cu n elemente și q interogări $\langle prim, pas \rangle$. Răspunsul fiecărei interogări este suma valorilor de pe pozițiile care formează progresia cu primul termen $prim$ și pasul pas.

Desigur, codul naiv care însumează progresiile este lent:

```
long long naive_prog_sum(int first, int step) {
  long long sum = 0;
  for (int i = first; i <= n; i += step) {
    sum += w[i];
  }
  return sum;
}
```

Dar... nu chiar atît de lent! Dacă pasul este cel puțin $\sqrt{n}$, progresia va avea cel mult $\sqrt{n}$ termeni. Rămîne să tratăm progresiile cu pasul mic (așadar $1, 2, \ldots, \sqrt{n}$). Ne permitem să facem o preprocesare în $\mathcal{O}(n)$ pentru fiecare din acești pași. Pentru un pas pas, preprocesăm efectiv $prep[i]$ = răspunsul la progresia cu primul termen i și pasul pas. Apoi putem răspunde la interogările pentru acel pas în $\mathcal{O}(1)$.

```
void preprocess(int step) {
  for (int i = n; (i > n - step) && (i >= 1); i--) {
    prep[i] = w[i];
  }
  for (int i = n - step; i >= 1; i--) {
    prep[i] = w[i] + prep[i + step];
  }
}
```

Complexitatea totală este:

- $\mathcal{O}(q\sqrt{n})$ pentru progresiile cu pas mare (calculate naiv);
- $\mathcal{O}(n\sqrt{n} + q)$ pentru preprocesarea tuturor răspunsurilor pentru pași mici.

Alte probleme similare:

- Train Maintenance (Codeforces);
- Căsuța (Lot juniori 2025) – atenție, la momentul scrierii acestei secțiuni problema nu are checker.

4.9 Descompunere în valori distincte

Această tehnică pornește de la observația: dacă suma unor numere naturale este n, atunci numerele au $\mathcal{O}(\sqrt{n})$ valori distincte. Demonstrația decurge din inversa sumei Gauss.

4.10 Probleme

4.10.1 Problema Sandor (Baraj ONI 2025)

enunț • sursă

Observăm că algoritmul lui Sandor este un algoritm *greedy* pentru problema rucsacului. Să facem trei experimente de gîndire despre natura acestui algoritm.

În primul rînd, dacă eliminăm un obiect pe care algoritmul oricum nu l-ar selecta (pentru că nu încape), atunci vom obține aceeași sumă ca și cînd nu am elimina nimic. Practic, obiectul nu există pentru algoritm.

Mai interesant, putem generaliza prima observație. Dacă există k obiecte de aceeași greutate și, la acel pas în algoritm, în rucsac încap mai puțin de k din aceste obiecte, atunci pe oricare dintre ele încercăm să-l eliminăm vom obține aceeași greutate ca și cînd nu am elimina nimic. De ce? Dacă, de exemplu, există 5 obiecte identice și doar 3 încap în rucsac, atunci dacă îl eliminăm pe primul algoritmul îl va adăuga pe al 4-lea.

În sfîrșit, deoarece în rucsac punem obiecte cu greutatea totală cel mult g, rezultă că vom pune $\mathcal{O}(\sqrt{g})$ greutăți distincte.

De aceea, merită să stocăm mai degrabă un vector de serii (în sursă le-am numit *runs*) de elemente egale, $\langle val, cnt \rangle$, decît valorile originale. Peste acest vector precalculăm niște *jump pointers*, adică răspunsurile la întrebări de tipul „Cu ce serie să continui algoritmul dacă rucsacul mai are capacitate rămasă c?” Cu această reprezentare, evaluarea algoritmului lui Sandor este elementară și necesită timp $\mathcal{O}(\sqrt{g})$. Mai mult, putem parametriza algoritmul ca să-l putem rula începînd cu oricare dintre serii, nu neapărat cu prima.

Cerința 1

De aici rezultă un algoritm relativ naiv pentru cerința 1. El necesită $\mathcal{O}(\sqrt{g}^2)$, adică $\mathcal{O}(g)$.

Mergînd de la stînga la dreapta prin vector, considerăm fiecare serie $\langle val, cnt \rangle$. Dacă în prezent în rucsac încap mai puțin de cnt obiecte, atunci conform observației din preambul oricum am elimina un element din serie obținem tot soluția inițială (avem cnt astfel de moduri).

Dacă încap toate obiectele, atunci are sens să ne punem întrebarea „dar dacă eliminăm unul, ce obținem?”. Deci punem în rucsac doar $cnt - 1$ obiecte și simulăm algoritmul lui Sandor (naiv) pentru restul vectorului. Apoi revenim la problema originală, punem în rucsac toate cele cnt obiecte și continuăm.

Așadar, facem $\mathcal{O}(\sqrt{g})$ simulări ale algoritmului, pentru o complexitate totală de $\mathcal{O}(g)$. Este important să nu luăm în calcul valorile care nu încap în rucsac, deoarece complexitatea ar crește la $\mathcal{O}(n\sqrt{g})$. Acele valori le sărim folosind *jump pointers*. Doar le contorizăm, prin diferența între n și numărul de valori pe care le-am luat în calcul. Fiecare valoare sărită ne dă un mod de a obține greutatea algoritmului lui Sandor fără eliminări.

Vom parametriza și cerința 1 pentru a o putea rula începînd cu orice serie și cu orice capacitate reziduală a rucsacului, nu neapărat cu prima serie și cu capacitatea g.

Cerința 2

Dacă reușim să facem același gen de trecere prin vector și pentru cerința 2, și să delegăm subprobleme la algoritmul naiv (fără eliminări) și la cerința 1 (o eliminare), atunci vom obține o complexitate de $\mathcal{O}(g\sqrt{g})$. Pentru aceasta, trebuie să analizăm cazurile posibile.

În primul rînd, cîtă vreme din seria curentă nici măcar un obiect nu încape un rucsac, trecem la seria următoare și contorizăm numărul de obiecte ignorate. Fie acest contor *mult*. Să spunem că avem capacitatea $c = 100$ și am ignorat serii de greutate 200, 150 și 130, totalizînd $mult = 10$ obiecte. Atunci avem $C^2_{mult} = 45$ de moduri de a elimina două din aceste obiecte. Rulăm algoritmul lui Sandor naiv și vedem ce sumă obținem. Adăugăm 45 la răspunsul pentru acea sumă.

Dacă măcar un exemplar încape în rucsac, atunci putem încerca să eliminăm un obiect din seriile anterioare și unul din seria curentă. Deci apelăm cerința 1 începînd cu poziția curentă, și îi transmitem că fiecare soluție găsită trebuie socotită de *mult* ori, deoarece există *mult* moduri de a elimina un obiect dinaintea seriei curente.

Dacă seria curentă include cel puțin două obiecte, putem încerca să eliminăm două obiecte din seria curentă, punînd $cnt-2$ în rucsac. Desigur, există C^2_{cnt} moduri de a face asta, iar pentru restul vectorului rulăm Sandor varianta de bază. Îi trimitem algoritmului parametrizat multiplicatorul C^2_{cnt} pentru soluția pe care o va găsi.

Similar, putem elimina doar un obiect din grupa curentă, punînd $cnt-1$ în rucsac, ceea ce putem face în *cnt* moduri distincte. Iar pentru seriile următoare apelăm cerința 1, spunîndu-i că soluțiile găsite trebuie numărate de cîte *cnt* ori.

În sfîrșit, putem să punem toate obiectele în rucsac și să reconsiderăm cerința 2 de la seria următoare.

4.10.2 Problema Puzzle-bile (Lot 2025)

enunț • sursă

Formularea programării dinamice

Vom denumi **fereastră** o serie de celule libere consecutive.

Pare firesc să ne dorim să calculăm valori de următorul tip. Fie $C_{r,c}$ costul pentru a aduce bila pe rîndul r, coloana c. E destul de clar că $C_{r,c}$ va depinde doar de valori din C de pe rîndul $r-1$. Dacă $C_{r,c}$ va depinde de $C_{r-1,c'}$, atunci va trebui să calculăm și costul de a aduce pe rîndul r o fereastră pe intervalul $[c', c]$.

Așadar, să definim și $D_{r,c,l}$ ca fiind costul de a aduce pe rîndul r o fereastră de lățime (cel puțin) l terminată la coloana c. Acum putem defini recurența pentru C:

$$C_{r,c} = \min_{c'=1}^{c}(C_{r-1,c'} + D_{r,c,c-c'+1})$$

Recurența modelează ideea că mergem pe rîndul $r-1$ pînă la coloana c', apoi coborîm pe rîndul r unde am pregătit o fereastră care ne duce pînă la coloana c.

Reducerea complexității

O soluție în $\mathcal{O}(nm^2)$ procedează astfel: pentru fiecare celulă (r, c) considerăm fiecare fereastră de pe rîndul r. Dacă fereastra are lățime l și trebuie deplasată cu p celule pentru a o alinia la dreapta cu coloana c, atunci putem optimiza $C_{r,c}$ cu cantitatea:

$$p + \text{rmq}(C_{r-1,c-l+1}, \ldots, C_{r-1,c})$$

, unde desigur rmq este funcția de minim pe interval. Doar că există $\mathcal{O}(m)$ ferestre pe fiecare rînd, deci complexitatea este prea mare. Aici intervine descompunerea în valori distincte! Există doar $\mathcal{O}(\sqrt{m})$ lățimi distincte de ferestre pe fiecare rînd. De aceea putem itera doar prin aceste lungimi. Pentru o lungime fixată l, nu are sens să testăm decît cele mai apropiate ferestre din stînga, respectiv din dreapta coloanei curente c. De aici rezultă complexitatea totală $\mathcal{O}(nm\sqrt{m})$.

Implementarea nu este deloc simplă datorită următoarei situații pe care o enunțăm, dar nu o detaliem (ea este descrisă cu exemple în cea de-a doua sursă). Cînd aducem o fereastră din stînga coloanei c, decizia este simplă: trebuie să o deplasăm pînă cînd capătul ei drept atinge coloana c. Dar, cînd aducem o fereastră din dreapta coloanei c, avem libertatea să o deplasăm fie pînă cînd capătul ei stîng atinge coloana c, fie mai mult de atît. Depinde ce coloană $c' < c$ ne interesează să „prindem" în recurență. Poate există o valoare c' destul de mică (costul deplasării pînă acolo este mare), dar unde $C_{r-1,c'}$ este foarte mic și justifică costul deplasării. Ia naștere un al doilea tabel RMQ construit nu peste valorile $C_{r-1,c}$, ci peste $C_{r-1,c} - c$.

Cîteva cuvinte despre Arpa's trick

Eu sînt mereu în căutare de noi structuri pentru RMQ. 😂 Îmi displace tabela rară deoarece folosește $\mathcal{O}(n \log n)$ memorie și evit să mă reped la ea. Iată o structură interesantă, numită Arpa's trick. Aparent, și alte nații au șmenurile lor... CP Algorithms are o lecție foarte bună.

Algoritmul răspunde la interogări în ordine crescătoare după capătul drept. Deci putem întîi să sortăm interogările sau să le distribuim în liste după capătul drept. Apoi folosim o pădure de

mulțimi disjuncte, pe care o instanțiem pe măsură ce baleiem poziția capătului drept.

Cînd ajungem la o poziție nouă p unde se află valoarea x, în primul rînd instanțiem o nouă rădăcină în DSF: părintele lui p este p. În plus, p devine părinte și pentru toate rădăcinile anterioare p' care aveau valori $x' > x$. Procedăm astfel deoarece, pentru orice interogări viitoare, valoarea x' nu va fi niciodată RMQ, deoarece x va face și ea parte din orice interogare care îl conține pe x'. Pentru implementarea acestei părți folosim o stivă ordonată.

La interogarea $\text{rmq}[p', p]$, unde p este poziția curentă, răspunsul este rădăcina arborelui lui p'. Într-adevăr, dorim cea mai mică valoare cu care a fost p' unit la orice moment.

Am rezolvat problema Puzzle-bila folosind *Arpa's trick*. Implementarea este dificilă și ineficientă, deoarece colectarea în avans și ordonarea interogărilor de care vom avea nevoie îngreunează codul (ca să folosesc un eufemism). Dar codul pentru *Arpa's trick* în sine este scurt și clar. El funcționează în $\mathcal{O}(\log^* n)$ amortizat cu memorie $\mathcal{O}(n)$.

```
struct arpa_s_trick {
  int val[MAX_N];
  int parent[MAX_N];
  int n;

  // O stivă cu pozițiile care încă nu au un element mai mic la dreapta.
  int st[MAX_N], ss;

  void reset() {
    ss = 0;
    n = 0;
  }

  void append(int x) {
    while (ss && (val[st[ss - 1]] >= x)) {
      parent[st[--ss]] = n;
    }
    st[ss++] = n;
    parent[n] = n;
    val[n++] = x;
  }

  int find(int p) {
    return (parent[p] == p)
      ? p
      : (parent[p] = find(parent[p]));
  }

  int rmq(int p) {
    return val[find(p)];
  }
};
```

Capitolul 5

Algoritmul lui Mo

Algoritmul lui Mo atinge tot complexitatea de $\mathcal{O}((q+n)\sqrt{n})$, ca și metoda descompunerii în radical, dar printr-o metodă destul de diferită. El duce adesea la cod mai simplu, dar necesită ca interogările să fie date în avans (*offline*).

5.1 Algoritmul lui Mo fără actualizări

Algoritmul lui Mo ordonează interogările și le procesează în această ordine:

1. După blocul capătului stîng.
2. La egalitate, după capătul drept.

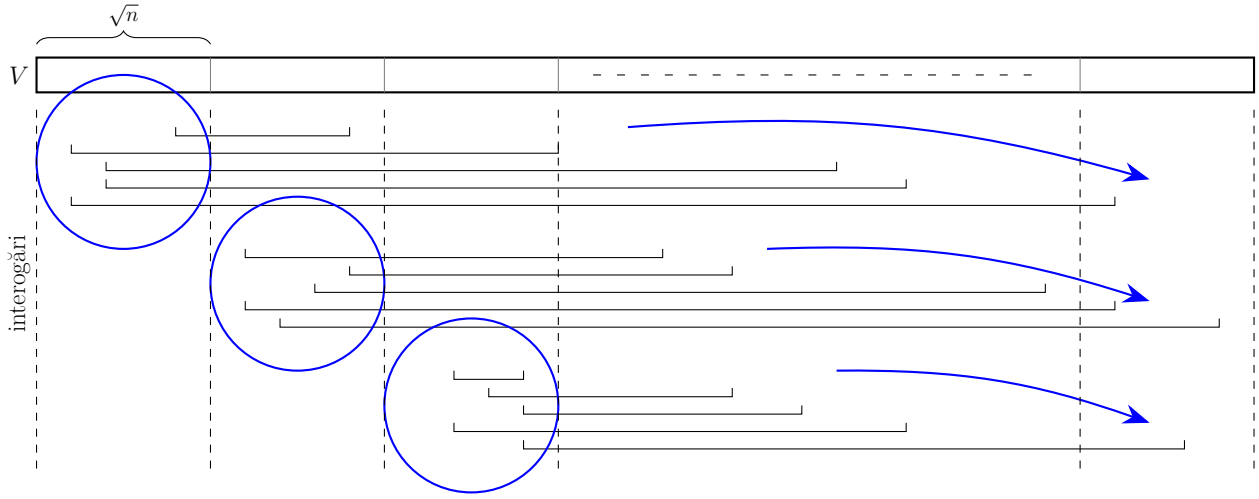


Figura 5.1: Ordonarea interogărilor în algoritmul lui Mo.

Algoritmul lui Mo ține minte informații despre interogarea curentă (răspunsul, dar posibil și alte informații). Pentru a trece la următoarea interogare, algoritmul:

1. Extinde intervalul curent cu cîte un pas, pînă cînd ajunge să includă și următoarea interogare.

2. Restrînge intervalul curent cu cîte un pas, pînă cînd ajunge să coincidă cu următoarea interogare.

Atenție! Pașii trebuie executați în această ordine. Altfel puteți ajunge la intervale negative, din care încercați să eliminați elemente care nu au fost adăugate, cu consecințe neprevăzute.

Vom relua problema D-query (SPOJ) și vom examina implementarea algoritmului lui Mo. Logica este considerabil mai simplă decît la implementarea cu AIB, în $\mathcal{O}(n \log n)$. Surprinzător, timpul de rulare este identic!

5.1.1 Analiză de complexitate

La fiecare interogare, trebuie să deplasăm capetele intervalului curent ca să se suprapună cu interogarea. Care este efortul total?

- Capătul stîng se poate deplasa cu cel mult $\sqrt{n}$ la fiecare interogare, plus n pași pentru toate trecerile între blocuri. Efortul total este de cel mult $q\sqrt{n} + n$ pași.
- Capătul drept se deplasează cu cel mult n pași la dreapta pentru toate interogările dintr-un bloc, apoi cu cel mult n pași înapoi la stînga la trecerea între blocuri. Există $\sqrt{n}$ blocuri, deci efortul total este de cel mult $2n\sqrt{n}$ pași.

Să spunem că includerea sau excluderea unei poziții durează $\mathcal{O}(f(n))$. Pentru problema Dquery efortul este $\mathcal{O}(1)$, dar pentru alte probleme costul poate diferi. Atunci complexitatea totală a algoritmului lui Mo este:

$$\mathcal{O}((q+n)\sqrt{n}f(n))$$

5.1.2 Optimizare de viteză

Nu ne costă aproape nimic (doar un `if` în plus la sortare) ca, în cadrul unui bloc, să ordonăm interogările astfel:

- crescător după capătul drept în blocurile impare;
- descrescător după capătul drept în blocurile impare.

Atunci nu mai plătim costul deplasării spre stînga „în gol” a capătului drept al intervalului curent. Iată o sursă cu această optimizare la problema Dquery. Timpii de rulare nu diferă, dar în general optimizarea poate ajuta.

De amorul artei, strict pentru problema Dquery, am normalizat valorile din vectorul dat. Pot fi cel mult 30.000 de valori distincte, ceea ce înseamnă că vectorul de frecvențe (pe `short`) poate ocupa doar 60 kB, în loc de 2 MB. Din nou, timpii nu diferă.

Aș preciza, tot pentru Dquery, că soluțiile țin foarte bine pasul cu soluția cu AIB (70-75 ms în loc de 60-65).

5.2 Algoritmul lui Mo cu actualizări

Algoritmul lui Mo se pretează și la probleme care au actualizări, cu o complexitate de $\mathcal{O}((q+n)n^{2/3})$. El este în continuare offline, avînd nevoie să citească și să sorteze interogările (și, posibil, să normalizeze valorile).

Esența este să introducem o nouă dimensiune pentru operații, în afară de capetele de interval l și r: timpul t, adică indicele operației (între 1 și q). Structura de date curentă stochează informații despre un interval $[L, R]$ la un moment de timp T, adică despre vectorul cu toate actualizările cu indici mai mici sau egali cu T. Pentru a răspunde la interogarea $[l, r]$ la momentul t, trebuie să:

1. Extindem intervalul curent pînă îl include pe $[l, r]$ (ca și pînă acum).
2. Restrîngem intervalul curent pînă devine egal cu $[l, r]$ (ca și pînă acum).
3. „Avansăm" timpul dacă $T < t$, procesînd actualizările din intervalul $(T, t]$.
4. „Dăm înapoi" timpul dacă $t < T$, inversînd actualizările din intervalul $[t, T)$.

Dacă actualizările efectuate sau inversate se întîmplă în intervalul curent $[l, r]$, actualizăm și structura de date, altfel nu.

5.2.1 Mărimea blocului, ordinea operațiilor, complexitate

Să alegem B blocuri de mărime S. Acum, să sortăm operațiile după blocul lui l, apoi după blocul lui r, apoi după t. Atunci:

Poziția lui l se schimbă cu $\mathcal{O}(S)$ la fiecare operație, plus $\mathcal{O}(n)$ la toate trecerile între blocuri, așadar cu $O(n + qS)$ în total.

Poziția lui r se schimbă cu $\mathcal{O}(S)$ la fiecare operație. În plus, pentru fiecare bloc al lui l (deci de B ori), r va face o trecere prin cele n poziții cu un efort total de $\mathcal{O}(qS + nB)$.

Pentru fiecare pereche de blocuri, t poate trece prin toate cele q operații, cu un efort total de $\mathcal{O}(qB^2)$.

Așadar, complexitatea algoritmului este $\mathcal{O}(qS + nB + qB^2)$. De aceea dorim ca $B^2 \approx S$ și alegem $B \approx n^{1/3}$ și $S \approx n^{2/3}$, pentru o complexitate totală de $\mathcal{O}((q+n)n^{2/3})$.

Anexa A

Cod-sursă

A.1 Arbori de intervale

A.1.1 Problema Xenia and Bit Operations (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_N = 1 << 17;

struct segment_tree {
  int v[2 * MAX_N];
  int n;

  void init(int n) {
    this->n = n;
  }

  void set(int pos, int val) {
    v[pos + n] = val;
  }

  void build() {
    bool is_or = true;
    for (int i = n - 1; i; i--) {
      int l = 2 * i, r = 2 * i + 1;
      v[i] = is_or ? (v[l] | v[r]) : (v[l] ^ v[r]);
      if ((i & (i - 1)) == 0) {
        is_or = !is_or;
      }
    }
  }

  int update(int pos, int val) {
```

```
    pos += n;
    v[pos] = val;

    bool is_or = true;
    for (pos /= 2; pos; pos /= 2) {
      int l = 2 * pos, r = 2 * pos + 1;
      v[pos] = is_or ? (v[l] | v[r]) : (v[l] ^ v[r]);
      is_or = !is_or;
    }
    return v[1];
  }
};

segment_tree st;
int num_queries;

void read_array() {
  int n;
  scanf("%d %d", &n, &num_queries);
  n = 1 << n;
  st.init(n);

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    st.set(i, x);
  }

  st.build();
}

void process_queries() {
  while (num_queries--) {
    int pos, val;
    scanf("%d %d", &pos, &val);
    int root = st.update(pos - 1, val);
    printf("%d\n", root);
  }
}

int main() {
  read_array();
  process_queries();

  return 0;
}
```

A.1.2 Problema Distinct Characters Queries (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>
#include <string.h>

const int MAX_N = 1 << 17;
const int T_UPDATE = 1;

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

struct segment_tree {
  int v[2 * MAX_N];
  int n;

  void init(char* s) {
    int len = strlen(s);
    this->n = next_power_of_2(len);

    for (int i = 0; i < len; i++) {
      v[i + this->n] = 1 << (s[i] - 'a');
    }

    build();
  }

  void build() {
    for (int i = n - 1; i; i--) {
      v[i] = v[2 * i] | v[2 * i + 1];
    }
  }

  void update(int pos, char val) {
    pos += n;
    v[pos] = 1 << (val - 'a');

    for (pos /= 2; pos; pos /= 2) {
      v[pos] = v[2 * pos] | v[2 * pos + 1];
    }
  }

  int popcount_query(int l, int r) {
    int mask = 0;

    l += n;
    r += n;
```

```
    while (l <= r)  {
      if (l & 1) {
        mask |= v[l++];
      }
      l >>= 1;

      if (!(r & 1)) {
        mask |= v[r--];
      }
      r >>= 1;
    }

    return __builtin_popcount(mask);
  }
};

segment_tree st;

void read_string() {
  char s[MAX_N + 1];
  scanf("%s", s);
  st.init(s);
}

void process_ops() {
  int num_ops, type;
  scanf("%d", &num_ops);

  while (num_ops--) {
    scanf("%d", &type);

    if (type == T_UPDATE) {
      int pos;
      char val;
      scanf("%d %c", &pos, &val);
      st.update(pos - 1, val);
    } else {
      int l, r;
      scanf("%d %d", &l, &r);
      int num_distinct = st.popcount_query(l - 1, r - 1);
      printf("%d\n", num_distinct);
    }
  }
}

int main() {
  read_string();
  process_ops();

  return 0;
```

```
}
```

A.1.3 Problema K-query (SPOJ)

înapoi

Sursă cu arbori de intervale (versiune online).

```
// Complexitate: O(n log n)
#include <algorithm>
#include <stdio.h>

const int MAX_N = 32'768;
const int MAX_Q = 200'000;

int next_power_of_2(int n) {
  while (n & (n - 1)) {
    n += (n & -n);
  }

  return n;
}

struct segment_tree {
  short v[2 * MAX_N];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  void set(int pos) {
    pos += n;
    while (pos) {
      v[pos]++;
      pos /= 2;
    }
  }

  short range_count(int l, int r) {
    int sum = 0;

    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        sum += v[l++];
      }
```

```
      l >>= 1;

      if (!(r & 1)) {
        sum += v[r--];
      }
      r >>= 1;
    }

    return sum;
  }
};

struct elem {
  int val;
  short pos;
};

struct query {
  short left, right;
  int val;
  int orig_pos;
};

elem v[MAX_N];
query q[MAX_Q];
// Mai curat ar fi să punem răspunsurile în struct. Dar atunci ar trebui să
// sortăm interogările încă o dată la final.
short answer[MAX_Q];
segment_tree st;
int n, num_queries;

void read_data() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &v[i].val);
    v[i].pos = i + 1;
  }

  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%hd %hd %d", &q[i].left, &q[i].right, &q[i].val);
    q[i].orig_pos = i;
  }
}

void sort_array_by_val() {
  std::sort(v, v + n, [](elem a, elem b) {
    return a.val > b.val;
  });
}
```

```
void sort_queries_by_val() {
  std::sort(q, q + num_queries, [](query& a, query& b) {
    return a.val > b.val;
  });
}

void process_queries() {
  st.init(n);

  int j = 0;
  for (int i = 0; i < num_queries; i++) {
    while ((j < n) && (v[j].val > q[i].val)) {
      st.set(v[j++].pos);
    }
    answer[q[i].orig_pos] = st.range_count(q[i].left, q[i].right);
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", answer[i]);
  }
}

int main() {
  read_data();
  sort_array_by_val();
  sort_queries_by_val();
  process_queries();
  write_answers();

  return 0;
}
```

Sursă cu arbori indexați binar (versiune online).

```
// Complexitate: O(n log n)
#include <algorithm>
#include <stdio.h>

const int MAX_N = 30'000;
const int MAX_Q = 200'000;

struct fenwick_tree {
  short v[MAX_N + 1];
  int n;

  void init(int n) {
```

```
    this->n = n;
  }

  void set(int pos) {
    do {
      v[pos]++;
      pos += pos & -pos;
    } while (pos <= n);
  }

  short prefix_count(int pos) {
    int sum = 0;
    while (pos) {
      sum += v[pos];
      pos &= pos - 1;
    }
    return sum;
  }

  short range_count(int l, int r) {
    return prefix_count(r) - prefix_count(l - 1);
  }
};

struct elem {
  int val;
  short pos;
};

struct query {
  short left, right;
  int val;
  int orig_pos;
};

elem v[MAX_N];
query q[MAX_Q];
// Mai curat ar fi să punem răspunsurile în struct. Dar atunci ar trebui să
// sortăm interogările încă o dată la final.
short answer[MAX_Q];
fenwick_tree fen;
int n, num_queries;

void read_data() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &v[i].val);
    v[i].pos = i + 1;
  }
```

```
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%hd %hd %d", &q[i].left, &q[i].right, &q[i].val);
    q[i].orig_pos = i;
  }
}

void sort_array_by_val() {
  std::sort(v, v + n, [](elem a, elem b) {
    return a.val > b.val;
  });
}

void sort_queries_by_val() {
  std::sort(q, q + num_queries, [](query& a, query& b) {
    return a.val > b.val;
  });
}

void process_queries() {
  fen.init(n);

  int j = 0;
  for (int i = 0; i < num_queries; i++) {
    while ((j < n) && (v[j].val > q[i].val)) {
      fen.set(v[j++].pos);
    }
    answer[q[i].orig_pos] = fen.range_count(q[i].left, q[i].right);
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", answer[i]);
  }
}

int main() {
  read_data();
  sort_array_by_val();
  sort_queries_by_val();
  process_queries();
  write_answers();

  return 0;
}
```

A.1.4 Problema Sereja and Brackets (Codeforces)

‹ înapoi • versiune online

```
// Complexitate: O(n + q log n).
#include <stdio.h>
#include <string.h>

const int MAX_N = 1 << 20;

int next_power_of_2(int n) {
  while (n & (n - 1)) {
    n += (n & -n);
  }

  return n;
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

struct node {
  int matched, open, closed;

  node combine(node& other) {
    int new_matches = min(open, other.closed);
    return {
      .matched = matched + other.matched + 2 * new_matches,
      .open = open + other.open - new_matches,
      .closed = closed + other.closed - new_matches,
    };
  }
};

struct segment_tree {
  node v[2 * MAX_N];
  int n;

  void init(char* s) {
    int len = strlen(s);
    this->n = next_power_of_2(len);

    for (int i = 0; i < len; i++) {
      v[i + this->n] = {
        .matched = 0,
        .open = (s[i] == '('),
        .closed = (s[i] == ')'),
      };
    }
```

```
    build();
  }

  void build() {
    for (int i = n - 1; i; i--) {
      v[i] = v[2 * i].combine(v[2 * i + 1]);
    }
  }

  int query(int l, int r) {
    node left = { 0, 0, 0 };
    node right = { 0, 0, 0 };

    l += n;
    r += n;

    while (l <= r)  {
      if (l & 1) {
        left = left.combine(v[l++]);
      }
      l >>= 1;

      if (!(r & 1)) {
        right = v[r--].combine(right);
      }
      r >>= 1;
    }

    node answer = left.combine(right);
    return answer.matched;
  }
};

segment_tree st;

void read_string() {
  char s[MAX_N + 1];
  scanf("%s", s);
  st.init(s);
}

void process_ops() {
  int num_ops;
  scanf("%d", &num_ops);

  while (num_ops--) {
    int l, r;
    scanf("%d %d", &l, &r);
    printf("%d\n", st.query(l - 1, r - 1));
```

```
  }
}

int main() {
  read_string();
  process_ops();

  return 0;
}
```

A.1.5 Problema Copying Data (Codeforces)

‹ înapoi • versiune online

```
// Complexitate: O(q log n)
#include <stdio.h>

const int MAX_N = 1 << 17;
const int T_COPY = 1;

struct change {
  int time;
  int shift;
};

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

struct segment_tree {
  change v[2 * MAX_N];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  change query(int pos) {
    change latest = { 0, 0 };
    for (pos += n; pos; pos >>= 1) {
      if (v[pos].time > latest.time) {
        latest = v[pos];
      }
    }
    return latest;
  }

  void update(int l, int r, change c) {
    l += n;
```

```
    r += n;

    while (l <= r)  {
      if (l & 1) {
        v[l++] = c;
      }
      l >>= 1;

      if (!(r & 1)) {
        v[r--] = c;
      }
      r >>= 1;
    }
  }
};

segment_tree st;
int a[MAX_N], b[MAX_N];
int n, num_ops;

void read_arrays() {
  scanf("%d %d", &n, &num_ops);
  for (int i = 0; i < n; i++) {
    scanf("%d", &a[i]);
  }
  for (int i = 0; i < n; i++) {
    scanf("%d", &b[i]);
  }
}

void process_ops() {
  st.init(n);
  int type, x, y, k;
  for (int op = 1; op <= num_ops; op++) {
    scanf("%d", &type);
    if (type == T_COPY) {
      scanf("%d %d %d", &x, &y, &k);
      x--;
      y--;
      change c = { .time = op, .shift = x - y };
      st.update(y, y + k - 1, c);
    } else {
      scanf("%d", &x);
      x--;
      change latest = st.query(x);
      int val = latest.time ? a[x + latest.shift] : b[x];
      printf("%d\n", val);
    }
  }
}
```

```
int main() {
  read_arrays();
  process_ops();

  return 0;
}
```

A.1.6 Problema PHF (FMI No Stress 2013)

‹ înapoi • versiune online

```
#include <stdio.h>

typedef unsigned char byte;

const int MAX_N = 1'000'000;
const int MAX_SEGTREE_NODES = 1 << 21;

const int IDENTITY = 3;
const byte CROSS_TABLE[4][3] = {
  { 0, 1, 0 }, // P
  { 1, 1, 2 }, // H
  { 0, 2, 2 }, // F
  { 0, 1, 2 }, // identitate
};

char FROM_CHAR[27] = ".....2.1.......0..........";
char TO_CHAR[4] = "PHF";

byte from_char(char c) {
  return FROM_CHAR[c - 'A'] - '0';
}

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

struct segment_tree_node {
  byte f[3];

  void make_leaf(byte c) {
    for (int i = 0; i < 3; i++) {
      f[i] = CROSS_TABLE[c][i];
    }
  }

  segment_tree_node compose(segment_tree_node other) {
    return {
```

```
      other.f[f[0]],
      other.f[f[1]],
      other.f[f[2]],
    };
  }
};

struct segment_tree {
  segment_tree_node v[MAX_SEGTREE_NODES];
  int n;

  void init(char* s, int _n) {
    n = next_power_of_2(_n);
    for (int i = 0; i < _n; i++) {
      v[n + i].make_leaf(s[i]);
    }

    for (int i = _n; i < n; i++) {
      v[n + i].make_leaf(IDENTITY);
    }

    for (int i = n - 1; i; i--) {
      v[i] = v[2 * i].compose(v[2 * i + 1]);
    }
  }

  void update(int pos, byte b) {
    pos += n;
    v[pos].make_leaf(b);
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = v[2 * pos].compose(v[2 * pos + 1]);
    }
  }

  char get_value(byte input) {
    return TO_CHAR[v[1].f[input]];
  }
};

char s[MAX_N + 1];
segment_tree st;
int n, num_queries;

void read_string() {
  scanf("%d %d ", &n, &num_queries);
  for (int i = 0; i < n; i++) {
    s[i] = from_char(getchar());
  }
}
```

```
void process_updates() {
  int pos;

  while (num_queries--) {
    scanf("%d ", &pos);
    pos--;
    s[pos] = from_char(getchar());
    if (pos) {
      st.update(pos - 1, s[pos]);
    }
    putchar(st.get_value(s[0]));
  }
}

int main() {
  read_string();
  st.init(s + 1, n - 1);
  putchar(st.get_value(s[0]));
  process_updates();
  putchar('\n');

  return 0;
}
```

A.1.7 Problema Points (Codeforces)

‹ înapoi • versiune online

```
#include <algorithm>
#include <set>
#include <stdio.h>

const int MAX_N = 1 << 18;
const int INFINITY = 2'000'000'000;
const int T_ADD = 1;
const int T_REMOVE = 2;
const int T_FIND = 3;

struct query {
  char type;
  int x, y;
};

int next_power_of_2(int n) {
  while (n & (n - 1)) {
    n += (n & -n);
  }

  return n;
```

```
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct max_segment_tree {
  int v[2 * MAX_N];
  int n;

  void init(int n) {
    n++; // santinelă infinită la final
    n = next_power_of_2(n);
    this->n = n;

    for (int i = 1; i < 2 * n; i++) {
      v[i] = -1; // fără puncte
    }

    set(n - 1, INFINITY);
  }

  void set(int pos, int val) {
    pos += n;
    v[pos] = val;
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = max(v[2 * pos], v[2 * pos + 1]);
    }
  }

  // Returnează prima poziție după pos unde valoarea depășește val.
  int find_first_after(int pos, int val) {
    pos += n;
    pos++;

    // Urcă pînă cînd găsim un maxim mai mare decît val.
    while (v[pos] <= val) {
      if (pos & 1) {
        pos++;
      } else {
        pos >>= 1;
      }
    }

    // Coboară spre valoarea dorită, mergînd la stînga oricînd putem.
    while (pos < n) {
      pos = (v[2 * pos] > val) ? (2 * pos) : (2 * pos + 1);
    }

    return pos - n;
```

```
  }

};

query q[MAX_N];
int pos[MAX_N]; // pentru normalizare
int orig_x[MAX_N];
max_segment_tree segtree;
std::set<int> whys[MAX_N];
int n;

void read_queries() {
  char s[10];
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%s %d %d", s, &q[i].x, &q[i].y);
    switch (s[0]) {
      case 'a': q[i].type = T_ADD; break;
      case 'r': q[i].type = T_REMOVE; break;
      default: q[i].type = T_FIND;
    }
  }
}

void normalize_x() {
  for (int i = 0; i < n; i++) {
    pos[i] = i;
  }

  std::sort(pos, pos + n, [](int a, int b) {
    return q[a].x < q[b].x;
  });

  int ptr = 0;
  orig_x[ptr] = q[pos[0]].x;
  for (int i = 0; i < n; i++) {
    if (q[pos[i]].x != orig_x[ptr]) {
      orig_x[++ptr] = q[pos[i]].x;
    }
    q[pos[i]].x = ptr;
  }
}

void add_query(int x, int y) {
  whys[x].insert(y);
  segtree.set(x, *whys[x].rbegin());
}

void remove_query(int x, int y) {
  whys[x].erase(y);
```

```
  if (whys[x].empty()) {
    segtree.set(x, -1);
  } else {
    segtree.set(x, *whys[x].rbegin());
  }
}

void find_query(int x, int y) {
  int first = segtree.find_first_after(x, y);
  if (first < n) {
    int first_above = *whys[first].upper_bound(y);
    printf("%d %d\n", orig_x[first], first_above);
  } else {
    printf("-1\n");
  }
}

void process_queries() {
  segtree.init(n);
  for (int i = 0; i < n; i++) {
    switch (q[i].type) {
      case T_ADD: add_query(q[i].x, q[i].y); break;
      case T_REMOVE: remove_query(q[i].x, q[i].y); break;
      default: find_query(q[i].x, q[i].y);
    }
  }
}

int main() {
  read_queries();
  normalize_x();
  process_queries();

  return 0;
}
```

A.1.8 Problema Medwalk (Lot 2025)

‹ înapoi • versiune online

```
// Complexitate: O((N + Q) log N log MAX_VAL).
//
// v2: Nu actualiza aint-ul de minime dacă valoarea nu s-a schimbat.
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <stdio.h>

const int MAX_N = 100'000;
const int MAX_VALUE = 300'000;
```

```
const int MAX_POS_SEGTREE_NODES = 1 << 18;
const int MAX_VAL_SEGTREE_NODES = 1 << 20;
const int INF = 1'000'000;
const int OP_UPDATE = 1;

typedef __gnu_pbds::tree<
  int,
  __gnu_pbds::null_type,
  std::less<int>,
  __gnu_pbds::rb_tree_tag,
  __gnu_pbds::tree_order_statistics_node_update
  > set;

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct pair {
  int v[2];

  void set(int pos, int val) {
    v[pos] = val;
  }

  int get_min() {
    return min(v[0], v[1]);
  }

  int get_max() {
    return max(v[0], v[1]);
  }
};

pair mat[MAX_N];
int n, num_queries;

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

// Admite actualizări punctuale și interogări de minim pe interval.
struct min_segment_tree {
  int v[MAX_POS_SEGTREE_NODES];
  int n;

  void init(int size) {
```

```
    n = next_power_of_2(size);
  }

  void update(int pos, int val) {
    pos += n;
    v[pos] = val;
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = min(v[2 * pos], v[2 * pos + 1]);
    }
  }

  int range_min(int l, int r) {
    l += n;
    r += n;
    int result = INF;

    while (l <= r)  {
      if (l & 1) {
        result = min(result, v[l++]);
      }
      l >>= 1;

      if (!(r & 1)) {
        result = min(result, v[r--]);
      }
      r >>= 1;
    }

    return result;
  }
};

// Kudos https://stackoverflow.com/a/22075025/6022817
//
// Arbore de intervale pe valori. Fiecare nod care subîntinde valorile [x, y)
// reține un set cu pozițiile pe care apar acele valori.
struct val_segment_tree {
  set s[MAX_VAL_SEGTREE_NODES];
  int n;

  void init(int size) {
    n = next_power_of_2(size);
  }

  void insert(int pos, int val) {
    for (val += n; val; val /= 2) {
      s[val].insert(pos);
    }
  }
```

```
  void erase(int pos, int val) {
    for (val += n; val; val /= 2) {
      s[val].erase(pos);
    }
  }

  // Cîte poziţii din [l, r] sînt ocupate de valori subîntinse de acest nod?
  int count_positions_between(int node, int l, int r) {
    return s[node].order_of_key(r + 1) - s[node].order_of_key(l);
  }

  // Contract: k este 0-based, iar [l, r] este interval închis.
  int kth_element(int k, int l, int r) {
    int node = 1;

    while (node < n) {
      int on_left = count_positions_between(2 * node, l, r);
      if (on_left > k) {
        node = 2 * node;
      } else {
        k -= on_left;
        node = 2 * node + 1;
      }
    }

    return node - n;
  }
};

void read_data() {
  scanf("%d %d", &n, &num_queries);
  for (int r = 0; r < 2; r++) {
    for (int i = 0; i < n; i++) {
      int x;
      scanf("%d", &x);
      mat[i].set(r, x);
    }
  }
}

min_segment_tree maxima;
val_segment_tree occur;

void build_segment_trees() {
  maxima.init(n);
  for (int i = 0; i < n; i++) {
    maxima.update(i, mat[i].get_max());
  }

  occur.init(MAX_VALUE + 1);
```

```
  for (int i = 0; i < n; i++) {
    occur.insert(i, mat[i].get_min());
  }
}

void update(int row, int col, int val) {
  int old_min = mat[col].get_min();
  mat[col].set(row, val);
  int new_min = mat[col].get_min();

  maxima.update(col, mat[col].get_max());

  if (new_min != old_min) {
    occur.erase(col, old_min);
    occur.insert(col, new_min);
  }
}

int query(int left, int right) {
  int len = right - left + 2;
  int median_pos = (len - 1) / 2;
  int median = occur.kth_element(median_pos, left, right);
  int best_max = maxima.range_min(left, right);

  if (best_max >= median) {
    return median;
  } else {
    // best_max trebuie inserat înainte de median_pos. Răspunsul poate fi la
    // poziția median_pos - 1 sau poate fi chiar best_max.
    int prev = occur.kth_element(median_pos - 1, left, right);
    return max(prev, best_max);
  }
}

void process_queries() {
  while (num_queries--) {
    int type;
    scanf("%d", &type);
    if (type == OP_UPDATE) {
      int r, c, x;
      scanf("%d %d %d", &r, &c, &x);
      r--;
      c--;
      update(r, c, x);
    } else {
      int left, right;
      scanf("%d %d", &left, &right);
      left--;
      right--;
      printf("%d\n", query(left, right));
```

```
    }
  }
}

int main() {
  read_data();
  build_segment_trees();
  process_queries();

  return 0;
}
```

A.2 Arbori de intervale cu propagare *lazy*

A.2.1 Problema Polynomial Queries (CSES)

înapoi

Sursă cu AINT iterativ (versiune online).

```
// Complexitate: O(q log n)
//
// Metodă: menține un arbore de intervale care admite sume pe interval și
// adăugarea unei progresii pe interval.
#include <stdio.h>

const int MAX_SEGTREE_NODES = 1 << 19;
const int T_UPDATE = 1;

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

long long progression_sum(long long first, long long step, int len) {
  return first * len + step * (len - 1) * len / 2;
}

// Invarianți:
//
// 1. Valorile reale ale nodurilor subîntines sînt valorile lor v respective
//    plus o progresie aritmetică definită prin first și step.
// 2. Valoarea v a unui nod nu include progresia nodului.
struct segment_tree_node {
  long long s;
  long long first, step;

  long long get_value(int size) {
    return s + progression_sum(first, step, size);
```

```
  }
};


struct segment_tree {
  segment_tree_node v[MAX_SEGTREE_NODES];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  void set(int pos, int val) {
    v[pos + n].s = val;
  }


  void build() {
    for (int i = n - 1; i; i--) {
      v[i].s = v[2 * i].s + v[2 * i + 1].s;
    }
  }

  void push(int t, int size) {
    int l = 2 * t, r = 2 * t + 1;

    v[l].first += v[t].first;
    v[l].step += v[t].step;

    long long first_right = v[t].first + v[t].step * size / 2;
    v[r].first += first_right;
    v[r].step += v[t].step;

    v[t].s += progression_sum(v[t].first, v[t].step, size);
    v[t].first = 0;
    v[t].step = 0;
  }

  void push_path(int node, int size) {
    if (node) {
      push_path(node / 2, size * 2);
      push(node, size);
    }
  }

  void pull(int t, int size) {
    int l = 2 * t, r = 2 * t + 1;
    v[t].s = v[l].get_value(size / 2) + v[r].get_value(size / 2);
  }
```

```
void pull_path(int node) {
  for (int x = node / 2, size = 2; x; x /= 2, size <<= 1) {
    pull(x, size);
  }
}

void add_progression(int l, int r) {
  l += n;
  r += n;
  int orig_l = l, orig_r = r, size = 1;
  push_path(l / 2, 2);
  push_path(r / 2, 2);

  while (l <= r)  {
    // Prima frunză subîntinsă de nodul x este x * size.
    if (l & 1) {
      v[l].first += l * size - orig_l + 1;
      v[l].step++;
      l++;
    }
    l >>= 1;

    if (!(r & 1)) {
      v[r].first += r * size - orig_l + 1;
      v[r].step++;
      r--;
    }
    r >>= 1;
    size <<= 1;
  }

  pull_path(orig_l);
  pull_path(orig_r);
}

long long range_sum(int l, int r) {
  long long sum = 0;
  int size = 1;

  l += n;
  r += n;
  push_path(l / 2, 2);
  push_path(r / 2, 2);

  while (l <= r)  {
    if (l & 1) {
      sum += v[l++].get_value(size);
    }
    l >>= 1;
```

```
      if (!(r & 1)) {
        sum += v[r--].get_value(size);
      }
      r >>= 1;
      size <<= 1;
    }

    return sum;
  }

};

segment_tree st;
int n, num_ops;

void read_array() {
  scanf("%d %d", &n, &num_ops);
  st.init(n);
  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    st.set(i, x);
  }
  st.build();
}

void process_ops() {
  while (num_ops--) {
    int type, l, r;
    scanf("%d %d %d", &type, &l, &r);
    l--; r--;
    if (type == T_UPDATE) {
      st.add_progression(l, r);
    } else {
      printf("%lld\n", st.range_sum(l, r));
    }
  }
}

int main() {
  read_array();
  process_ops();

  return 0;
}
```

Sursă cu AINT recursiv (versiune online).

```
// Complexitate: O(q log n)
```

```
//
// Metodă: menține un arbore de intervale care admite sume pe interval și
// adăugarea unei progresii pe interval.
#include <stdio.h>

const int MAX_N = 1 << 18;
const int T_UPDATE = 1;

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

long long progression_sum(long long first, long long step, int len) {
  return first * len + step * (len - 1) * len / 2;
}

// Invarianți:
//
// 1. Valorile reale ale nodurilor subîntines sînt valorile lor v respective
//    plus o progresie aritmetică definită prin first și step.
// 2. v[k] include first[k] și step[k], dar nu și valorile de la strămoși.
// 3. Toate intervalele sînt [închis, deschis).
struct segment_tree {
  long long v[2 * MAX_N];
  long long first[2 * MAX_N];
  long long step[2 * MAX_N];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  void set(int pos, int val) {
    v[pos + n] = val;
  }

  void build() {
    for (int i = n - 1; i; i--) {
      v[i] = v[2 * i] + v[2 * i + 1];
    }
  }
```

```
void push(int t, int len) {
  first[2 * t] += first[t];
  step[2 * t] += step[t];
  v[2 * t] += progression_sum(first[t], step[t], len / 2);

  int right_first = first[t] + step[t] * len / 2;
  first[2 * t + 1] += right_first;
  step[2 * t + 1] += step[t];
  v[2 * t + 1] += progression_sum(right_first, step[t], len / 2);

  first[t] = 0;
  step[t] = 0;
}

void pull(int t) {
  v[t] = v[2 * t] + v[2 * t + 1];
}

// pos1 = poziția unde scriem 1
void add_progression_helper(int t, int pl, int pr, int l, int r, int pos1) {
  if (l >= r) {
    return;
  } else if ((l == pl) && (r == pr)) {
    int value_at_l = l - pos1 + 1;
    first[t] += value_at_l;
    step[t]++;
    v[t] += progression_sum(value_at_l, 1, r - l);
  } else {
    push(t, pr - pl);
    int mid = (pl + pr) >> 1;
    add_progression_helper(2 * t, pl, mid, l, min(r, mid), pos1);
    add_progression_helper(2 * t + 1, mid, pr, ::max(l, mid), r, pos1);
    pull(t);
  }
}

void add_progression(int left, int right) {
  add_progression_helper(1, 0, n, left, right, left);
}

long long range_sum_helper(int t, int pl, int pr, int l, int r) {
  if (l >= r) {
    return 0;
  } else if ((l == pl) && (r == pr)) {
    return v[t];
  } else {
    push(t, pr - pl);
    int mid = (pl + pr) >> 1;
    return range_sum_helper(2 * t, pl, mid, l, min(r, mid)) +
      range_sum_helper(2 * t + 1, mid, pr, ::max(l, mid), r);
```

```
    }
  }

  long long range_sum(int left, int right) {
    return range_sum_helper(1, 0, n, left, right);
  }
};

segment_tree s;
int n, num_ops;

void read_array() {
  scanf("%d %d", &n, &num_ops);
  s.init(n);
  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    s.set(i, x);
  }
  s.build();
}

void process_ops() {
  while (num_ops--) {
    int type, l, r;
    scanf("%d %d %d", &type, &l, &r);
    if (type == T_UPDATE) {
      s.add_progression(l - 1, r);
    } else {
      printf("%lld\n", s.range_sum(l - 1, r));
    }
  }
}

int main() {
  read_array();
  process_ops();

  return 0;
}
```

A.2.2 Problema Nezzar and Binary String (Codeforces)

‹ înapoi • versiune online

```
// Complexitate: O(q log n).
#include <stdio.h>

const int MAX_N = 256 * 1024;
```

```
const int MAX_OPS = 200'000;
const int ST_CLEAN = 2;

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

// Arbore de intervale cu valori 0/1, atribuire pe interval și sumă pe
// interval.
//
// Contract: state[x] poate fi:
// * 0/1 pentru a indica o valoare de atribuit pe toți descendenții lui x, sau
// * ST_CLEAN pentru a arăta că am apelat deja push.
//
// sum[node] ține cont și de state[node].
struct segment_tree {
  int sum[2 * MAX_N];
  int state[2 * MAX_N];
  int n;

  void init(char* s, int len) {
    n = next_power_of_2(len);
    for (int i = 0; i < len; i++) {
      sum[n + i] = s[i] - '0';
    }
    for (int i = len; i < n; i++) {
      sum[n + i] = 0;
    }
    for (int i = 1; i < 2 * n; i++) {
      state[i] = ST_CLEAN;
    }

    build();
  }

  void build () {
    for (int i = n - 1; i; i--) {
      sum[i] = sum[2 * i] + sum[2 * i + 1];
    }
  }

  void push(int x) {
    if (state[x] != ST_CLEAN) {
      state[2 * x] = state[2 * x + 1] = state[x];
      sum[2 * x] = sum[2 * x + 1] = sum[x] / 2;
      state[x] = ST_CLEAN;
    }
  }

  void push_all() {
```

```
  for (int i = 1; i < n; i++) {
    push(i);
  }
}

void push_path(int x) {
  int bits = __builtin_popcount(n - 1);
  for (int b = bits; b; b--) {
    push(x >> b);
  }
}

void pull_path(int x) {
  for (x /= 2; x; x /= 2) {
    if (state[x] == ST_CLEAN) {
      sum[x] = sum[2 * x] + sum[2 * x + 1];
    }
  }
}

void range_set(int l, int r, int val) {
  l += n;
  r += n;
  int orig_l = l, orig_r = r;
  int size = 1;

  push_path(l);
  push_path(r);

  while (l <= r)  {
    if (l & 1) {
      sum[l] = size * val;
      state[l++] = val;
    }
    l >>= 1;

    if (!(r & 1)) {
      sum[r] = size * val;
      state[r--] = val;
    }
    r >>= 1;
    size <<= 1;
  }

  pull_path(orig_l);
  pull_path(orig_r);
}

int range_sum(int l, int r) {
  int result = 0;
```

```
    l += n;
    r += n;
    push_path(l);
    push_path(r);

    while (l <= r)  {
      if (l & 1) {
        result += sum[l++];
      }
      l >>= 1;

      if (!(r & 1)) {
        result += sum[r--];
      }
      r >>= 1;
    }

    return result;
  }

  bool equals(char* s, int len) {
    push_all();

    int i = 0;
    while ((i < len) && (s[i] - '0' == sum[n + i])) {
      i++;
    }

    return (i == len);
  }
};

struct operation {
  int l, r;
};

segment_tree st;
char start[MAX_N + 1], finish[MAX_N + 1];
operation op[MAX_OPS];
int len, num_ops;

void read_data() {
  scanf("%d %d %s %s", &len, &num_ops, start, finish);
  for (int i = 0; i < num_ops; i++) {
    scanf("%d %d\n", &op[i].l, &op[i].r);
    op[i].l--;
    op[i].r--;
  }
}
```

```
bool process_op(operation op) {
  int num_ones = st.range_sum(op.l, op.r);
  int num_zeroes = (op.r - op.l + 1) - num_ones;
  if (num_ones == num_zeroes) {
    return false;
  } else {
    int majority = (num_ones > num_zeroes) ? 1 : 0;
    st.range_set(op.l, op.r, majority);
    return true;
  }
}

void process_test() {
  read_data();
  st.init(finish, len);
  int i = num_ops - 1;
  while ((i >= 0) && process_op(op[i])) {
    i--;
  }

  bool success = (i < 0) && st.equals(start, len);
  printf("%s\n", success ? "YES" : "NO");
}

int main() {
  int num_tests;
  scanf("%d", &num_tests);
  while (num_tests--) {
    process_test();
  }

  return 0;
}
```

A.2.3 Problema Simple (infO(1)Cup 2019)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_N = 200'000;
const int MAX_SEGTREE_NODES = 1 << 19;
const long long INF = 1LL << 60;
const int OP_ADD = 0;

long long min(long long x, long long y) {
  return (x < y) ? x :  y;
}
```

```
long long max(long long x, long long y) {
  return (x > y) ? x :  y;
}

// Un arbore de intervale cu propagare lazy. Fiecare nod reține
// * minimul par,  maximul par, minimul impar și maximul impar;
// * o cantitate delta de adăugat pe tot subarborele.
//
// Contract: Nodul curent și-a aplicat deja delta sie însuși.
struct node {
  long long e, E, o, O;
  long long delta;

  void empty() {
    // Astfel operațiile de minim/maxim funcționează fără cazuri particulare.
    // 2x pentru că ne lăsăm loc să creștem/scădem.
    e = o = 2 * INF;
    E = O = -2 * INF;
    delta = 0;
  }

  void set(int val) {
    empty();
    if (val % 2) {
      o = O = val;
    } else {
      e = E = val;
    }
  }

  void swap() {
    long long tmp = e; e = o; o = tmp;
    tmp = E; E = O; O = tmp;
  }

  void push(node& a, node& b) {
    a.add(delta);
    b.add(delta);
    delta = 0;
  }

  void pull(node a, node b) {
    // Dacă nodul este *dirty*, atunci el știe mai bine situația curentă. Fiii
    // săi au informație perimată.
    if (!delta) {
      e = min(a.e, b.e);
      E = max(a.E, b.E);
      o = min(a.o, b.o);
      O = max(a.O, b.O);
```

```
    }
  }

  void add(long long val) {
    delta += val;
    if (val % 2) {
      // Exemplu: [9,15] și [6,30] +5 ⇒ [11,35] și [14,20].
      swap();
    }
    e += val;
    E += val;
    o += val;
    O += val;
  }
};

struct segment_tree {
  node v[MAX_SEGTREE_NODES];
  int n, bits;

  void init(int _n) {
    bits = 32 - __builtin_clz(_n - 1);
    n = 1 << bits;

    for (int i = n + _n; i < 2 * n; i++) {
      // Necesar deoarece altfel nodurile de la _n la n rămîn pe zero.
      v[i].empty();
    }
  }

  void raw_set(int pos, int val) {
    v[pos + n].set(val);
  }

  void build() {
    for (int i = n - 1; i; i--) {
      v[i].pull(v[2 * i], v[2 * i + 1]);
    }
  }

  void push_path(int pos) {
    for (int b = bits - 1; b; b--) {
      int t = pos >> b;
      v[t].push(v[2 * t], v[2 * t + 1]);
    }
  }

  void pull_path(int pos) {
    for (pos /= 2; pos; pos /= 2) {
      v[pos].pull(v[2 * pos], v[2 * pos + 1]);
```

```
    }
  }

  void range_add(int l, int r, int val) {
    l += n;
    r += n;
    int orig_l = l, orig_r = r;
    push_path(l);
    push_path(r);

    while (l <= r)  {
      if (l & 1) {
        v[l++].add(val);
      }
      l >>= 1;

      if (!(r & 1)) {
        v[r--].add(val);
      }
      r >>= 1;
    }

    pull_path(orig_l);
    pull_path(orig_r);
  }

  node range_query(int l, int r) {
    node accumulator;
    accumulator.empty();

    l += n;
    r += n;
    push_path(l);
    push_path(r);

    while (l <= r)  {
      if (l & 1) {
        accumulator.pull(accumulator, v[l++]);
      }
      l >>= 1;

      if (!(r & 1)) {
        accumulator.pull(accumulator, v[r--]);
      }
      r >>= 1;
    }

    return accumulator;
  }
};
```

```
segment_tree st;
int n;

void read_array_into_segtree() {
  scanf("%d", &n);
  st.init(n);

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    st.raw_set(i, x);
  }

  st.build();
}

void process_ops() {
  int num_ops, type, l, r, val;

  scanf("%d", &num_ops);
  while (num_ops--) {
    scanf("%d %d %d", &type, &l, &r);
    l--;
    r--;
    if (type == OP_ADD) {
      scanf("%d", &val);
      st.range_add(l, r, val);
    } else {
      node nd = st.range_query(l, r);
      long long e = (nd.e > INF) ? -1 : nd.e;
      long long O = (nd.O < -INF) ? -1 : nd.O;
      printf("%lld %lld\n", e, O);
    }
  }
}

int main() {
  read_array_into_segtree();
  process_ops();
  return 0;
}
```

A.2.4 Problema Balama (Baraj ONI 2024)

‹ înapoi

Sursă cu AINT iterativ (versiune online).

```
#include <algorithm>
#include <stdio.h>

const int MAX_N = 200'000;
const int MAX_SEGTREE_NODES = 1 << 19;

struct hinge {
  int r, pos;
};

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

int next_power_of_2(int n) {
  return 1 << (32 - __builtin_clz(n - 1));
}

struct segment_tree_node {
  int m, delta;
};

struct segment_tree {
  segment_tree_node v[MAX_SEGTREE_NODES];
  int n, bits;

  void init(int _n) {
    n = next_power_of_2(_n);
    bits = 31 - __builtin_clz(n);
  }

  void push_path(int node) {
    for (int b = bits; b; b--) {
      int t = node >> b;
      v[2 * t].delta += v[t].delta;
      v[2 * t].m += v[t].delta;
      v[2 * t + 1].delta += v[t].delta;
      v[2 * t + 1].m += v[t].delta;
      v[t].delta = 0;
    }
  }

  void pull_path(int node) {
    for (int t = node / 2; t; t /= 2) {
      v[t].m = v[t].delta + max(v[2 * t].m, v[2 * t + 1].m);
```

```
    }
  }

  int range_max_and_inc(int l, int r) {
    l += n;
    r += n;
    push_path(l);
    push_path(r);

    int result = 0;
    int orig_l = l, orig_r = r;

    while (l <= r)  {
      if (l & 1) {
        result = max(result, v[l].m);
        v[l].m++;
        v[l].delta++;
        l++;
      }
      l >>= 1;

      if (!(r & 1)) {
        result = max(result, v[r].m);
        v[r].m++;
        v[r].delta++;
        r--;
      }
      r >>= 1;
    }

    pull_path(orig_l);
    pull_path(orig_r);

    return result;
  }
};

hinge h[MAX_N];
int sol[MAX_N];
segment_tree st;
int n, k;

void read_data() {
  FILE* f = fopen("balama.in", "r");
  fscanf(f, "%d %d", &n, &k);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d", &h[i].r);
    h[i].pos = i;
  }
  fclose(f);
```

```
}

void sort_by_r() {
  std::sort(h, h + n, [](hinge a, hinge b) {
    return a.r > b.r;
  });
}

void scan_hinges() {
  int next_to_fill = 0;
  int i = 0;

  st.init(n);

  while (next_to_fill < k) {
    int left = max(h[i].pos - k + 1, 0);
    int right = min(h[i].pos, n - k);
    int m = st.range_max_and_inc(left, right);
    if (m == next_to_fill) {
      sol[next_to_fill++] = h[i].r;
    }
    i++;
  }
}

void write_answers() {
  FILE* f = fopen("balama.out", "w");
  for (int i = k - 1; i >= 0; i--) {
    fprintf(f, "%d ", sol[i]);
  }
  fprintf(f, "\n");
  fclose(f);
}

int main() {
  read_data();
  sort_by_r();
  scan_hinges();
  write_answers();

  return 0;
}
```

Sursă cu AINT recursiv (versiune online).

```
#include <algorithm>
#include <stdio.h>

const int MAX_N = 1 << 18;
```

```
struct hinge {
  int r, pos;
};

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

int next_power_of_2(int n) {
  while (n & (n - 1)) {
    n += (n & -n);
  }

  return n;
}

// Arbore de segmente cu operațiile:
//
// 1. update: inc(left, right) -- incrementează pozițiile [left, right]
// 2. query: max(left, right) -- returnează maximul din [left, right]
//
// Invariant:
//
// * lazy_sum este valoarea de adăugat pe fiecare nod din subarbore
// * m este maximul real din subarbore, inclusiv lazy_sum
struct segment_tree {
  int m[2 * MAX_N];
  int lazy_sum[2 * MAX_N];
  int n;

  void init(int n) {
    this->n = next_power_of_2(n);
  }

  void push(int t) {
    lazy_sum[2 * t] += lazy_sum[t];
    m[2 * t] += lazy_sum[t];
    lazy_sum[2 * t + 1] += lazy_sum[t];
    m[2 * t + 1] += lazy_sum[t];
    lazy_sum[t] = 0;
  }

  void pull(int t) {
    m[t] = ::max(m[2 * t], m[2 * t + 1]);
  }
```

```
  void inc_helper(int t, int pl, int pr, int l, int r) {
    if (l >= r) {
      return;
    } else if ((l == pl) && (r == pr)) {
      lazy_sum[t]++;
      m[t]++;
    } else {
      push(t);
      int mid = (pl + pr) >> 1;
      inc_helper(2 * t, pl, mid, l, min(r, mid));
      inc_helper(2 * t + 1, mid, pr, ::max(l, mid), r);
      pull(t);
    }
  }

  void inc(int left, int right) {
    inc_helper(1, 0, n, left, right + 1);
  }

  int max_helper(int t, int pl, int pr, int l, int r) {
    if (l >= r) {
      return 0;
    } else if ((l == pl) && (r == pr)) {
      return m[t];
    } else {
      push(t);
      int mid = (pl + pr) >> 1;
      return ::max(max_helper(2 * t, pl, mid, l, min(r, mid)),
                   max_helper(2 * t + 1, mid, pr, ::max(l, mid), r));
    }
  }

  int max(int left, int right) {
    return max_helper(1, 0, n, left, right + 1);
  }
};

hinge h[MAX_N];
int sol[MAX_N];
segment_tree st;
int n, k;

void read_data() {
  FILE* f = fopen("balama.in", "r");
  fscanf(f, "%d %d", &n, &k);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d", &h[i].r);
    h[i].pos = i;
  }
```

```
  fclose(f);
}

void sort_by_r() {
  std::sort(h, h + n, [](hinge a, hinge b) {
    return a.r > b.r;
  });
}

void scan_hinges() {
  int next_to_fill = 0;
  int i = 0;

  st.init(n);

  while (next_to_fill < k) {
    int left = max(h[i].pos - k + 1, 0);
    int right = min(h[i].pos, n - k);
    int m = st.max(left, right);
    st.inc(left, right);
    if (m == next_to_fill) {
      sol[next_to_fill++] = h[i].r;
    }
    i++;
  }
}

void write_answers() {
  FILE* f = fopen("balama.out", "w");
  for (int i = k - 1; i >= 0; i--) {
    fprintf(f, "%d ", sol[i]);
  }
  fprintf(f, "\n");
  fclose(f);
}

int main() {
  read_data();
  sort_by_r();
  scan_hinges();
  write_answers();

  return 0;
}
```

A.3 Arbori indexați binar

A.3.1 Problema The Permutation Game Again (SPOJ)

înapoi • versiune online

```
#include <stdio.h>

const int MAX_N = 200'000;
const int MOD = 1'000'000'007;

struct fenwick_tree {
  int v[MAX_N + 1];
  int n;

  void init(int n) {
    this->n = n;
    for (int i = 1; i <= n; i++) {
      v[i] = 0;
    }
  }

  void check(int pos) {
    do {
      v[pos]++;
      pos += pos & -pos;
    } while (pos <= n);
  }

  int prefix_sum(int pos) {
    int s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }
};

fenwick_tree fen;

void solve_test() {
  int n, x;
  scanf("%d", &n);
  fen.init(n);
  long long rank = 0;

  for (int place = n; place; place--) {
    scanf("%d", &x);
```

```
    int not_seen_before = x - 1 - fen.prefix_sum(x);
    rank = (rank * place + not_seen_before) % MOD;
    fen.check(x);
  }

  rank++; // Noi calculăm rangul începînd cu 0.

  printf("%lld\n", rank);
}

int main() {
  int num_tests;
  scanf("%d", &num_tests);
  while (num_tests--) {
    solve_test();
  }

  return 0;
}
```

A.3.2 Problema Multiset (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_N = 1'000'000;

struct fenwick_tree {
  int v[MAX_N + 1];
  int n, max_p2;

  void init(int n) {
    this->n = n;
    max_p2 = 1 << (31 - __builtin_clz(n));
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  // Returnează poziția unde suma parțială atinge sau depășește sum.
  int bin_search(int sum) {
    int pos = 0;

    for (int interval = max_p2; interval; interval >>= 1) {
```

```
      if ((pos + interval <= n) && (v[pos + interval] < sum)) {
        sum -= v[pos + interval];
        pos += interval;
      }
    }

    return pos + 1;
  }

  void remove_kth_element(int k) {
    int pos = bin_search(k);
    add(pos, -1);
  }

  int get_smallest() {
    int pos = bin_search(1);
    return (pos <= n) ? pos : 0;
  }
};

fenwick_tree fen;
int n, num_ops;

void read_initial_multiset() {
  scanf("%d %d", &n, &num_ops);
  fen.init(n);
  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    fen.add(x, 1);
  }
}

void process_queries() {
  while (num_ops--) {
    int x;
    scanf("%d", &x);
    if (x > 0) {
      fen.add(x, 1);
    } else {
      fen.remove_kth_element(-x);
    }
  }
}

void write_answer(int answer) {
  printf("%d\n", answer);
}

int main() {
```

```
  read_initial_multiset();
  process_queries();
  int answer = fen.get_smallest();
  write_answer(answer);

  return 0;
}
```

A.3.3 Problema Hanoi Factory (Codeforces)

‹ înapoi • versiune online

```
#include <algorithm>
#include <stdio.h>

const int MAX_N = 100'000;

struct ring {
  int in, out, h;
};

long long max(long long x, long long y) {
  return (x > y) ? x : y;
}

struct max_fenwick_tree {
  long long* v;
  int n;

  void init(long long* v, int n) {
    this->v = v;
    this->n = n;
    for (int i = 0; i <= n; i++) {
      v[i] = 0;
    }
  }

  void improve(int pos, long long val) {
    do {
      v[pos] = max(v[pos], val);
      pos += pos & -pos;
    } while (pos <= n);
  }

  long long prefix_max(int pos) {
    long long m = 0;
    while (pos) {
      m = max(m, v[pos]);
      pos &= pos - 1;
```

```
    }
    return m;
  }
};

ring r[MAX_N];
// folosit și pentru arborele fenwick, și pentru normalizarea mărimilor
long long v[2 * MAX_N + 1];
max_fenwick_tree fen;
int n;

void read_rings() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d %d %d", &r[i].in, &r[i].out, &r[i].h);
  }
}

void sort_rings() {
  std::sort(r, r + n, [](ring& a, ring& b) {
    return (a.out > b.out) || ((a.out == b.out) && (a.in > b.in));
  });
}

int bin_search(int val) {
  int l = 0, r = 2 * n;
  while (v[l] != val) { // valoarea există garantat în v
    int m = (l + r) / 2;
    if (v[m] > val) {
      r = m;
    } else {
      l = m;
    }
  }
  return l;
}

void normalize_diameters() {
  for (int i = 0; i < n; i++) {
    v[2 * i] = r[i].in;
    v[2 * i + 1] = r[i].out;
  }
  std::sort(v, v + 2 * n);

  for (int i = 0; i < n; i++) {
    r[i].in = 1 + bin_search(r[i].in);
    r[i].out = 1 + bin_search(r[i].out);
  }
}
```

```
long long solve_recurrence() {
  fen.init(v, 2 * n);

  long long max_height = 0;
  for (int i = 0; i < n; i++) {
    long long best = r[i].h + fen.prefix_max(r[i].out - 1);
    fen.improve(r[i].in, best);
    max_height = max(max_height, best);
  }

  return max_height;
}

void write_answer(long long answer) {
  printf("%lld\n", answer);
}

int main() {
  read_rings();
  sort_rings();
  normalize_diameters();
  long long answer = solve_recurrence();
  write_answer(answer);

  return 0;
}
```

A.3.4 Problema Subsequences (Codeforces)

‹ înapoi • versiune online

```
// Folosim un singur vector pentru cnt, înlocuind cîte un element pe rînd.
#include <stdio.h>

const int MAX_N = 100'000;

struct fenwick_tree {
  long long v[MAX_N + 1];
  int n;

  void init(int n) {
    this->n = n;
    for (int i = 1; i <= n; i++) {
      v[i] = 0;
    }
  }

  void add(int pos, long long val) {
    do {
```

```
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  long long prefix_sum(int pos) {
    long long s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }
};

fenwick_tree fen;
int a[MAX_N];
long long cnt[MAX_N];
int n, k;

void read_data() {
  scanf("%d %d", &n, &k);
  for (int i = 0; i < n; i++) {
    scanf("%d", &a[i]);
  }
}

void iterate_lengths() {
  for (int i = 0; i < n; i++) {
    cnt[i] = 1;
  }

  while (k--) {
    fen.init(n);

    for (int i = 0; i < n; i++) {
      long long old_cnt = cnt[i];
      cnt[i] = fen.prefix_sum(a[i] - 1);
      fen.add(a[i], old_cnt);
    }
  }
}

long long array_sum(long long* v, int n) {
  long long sum = 0;
  for (int i = 0; i < n; i++) {
    sum += v[i];
  }
  return sum;
}
```

```
void write_answer(long long answer) {
  printf("%lld\n", answer);
}

int main() {
  read_data();
  iterate_lengths();
  long long total = array_sum(cnt, n);
  write_answer(total);

  return 0;
}
```

A.3.5 Problema D-query (SPOJ)

‹ înapoi • versiune online

```
#include <algorithm>
#include <stdio.h>

const int MAX_N = 30000;
const int MAX_QUERIES = 200000;
const int MAX_VAL = 1000000;

struct query {
  short l, r;
  int orig_index;
  short answer;
};

struct fenwick_tree {
  short v[MAX_N + 1];
  int n;

  void init(int n) {
    this->n = n;
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  short prefix_sum(int pos) {
    int sum = 0;
    while (pos) {
```

```
      sum += v[pos];
      pos &= pos - 1;
    }
    return sum;
  }

  short range_sum(int l, int r) {
    return prefix_sum(r) - prefix_sum(l - 1);
  }
};

int a[MAX_N + 1];
query q[MAX_QUERIES];
short prev_occur[MAX_VAL + 1];
fenwick_tree fen;
int n, num_queries;

void read_data() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d", &a[i]);
  }
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%hd %hd", &q[i].l, &q[i].r);
    q[i].orig_index = i;
  }
}

void sort_queries_by_right_end() {
  std::sort(q, q + num_queries, [](query& a, query& b) {
    return a.r < b.r;
  });
}

void update_last_occurrence(int pos) {
  if (prev_occur[a[pos]]) {
    fen.add(prev_occur[a[pos]], -1);
  }
  prev_occur[a[pos]] = pos;
  fen.add(pos, +1);
}

int answer_queries_ending_at(int right, int q_index) {
  while ((q_index < num_queries) && (q[q_index].r == right)) {
    q[q_index].answer = fen.range_sum(q[q_index].l, q[q_index].r);
    q_index++;
  }

  return q_index;
```

```
}

void scan_array() {
  fen.init(n);
  int q_index = 0;

  for (int i = 1; i <= n; i++) {
    update_last_occurrence(i);
    q_index = answer_queries_ending_at(i, q_index);
  }
}

void sort_queries_by_orig_index() {
  std::sort(q, q + num_queries, [](query& a, query& b) {
    return a.orig_index < b.orig_index;
  });
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", q[i].answer);
  }
}

int main() {
  read_data();
  sort_queries_by_right_end();
  scan_array();
  sort_queries_by_orig_index();
  write_answers();

  return 0;
}
```

A.3.6 Problema Magic Board (CodeChef)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_SIZE = 500'000;
const int MAX_TIME = 500'000;
const int MAX_WORD_LENGTH = 8;
enum op_type { OP_ROW_SET, OP_COL_SET, OP_ROW_QUERY, OP_COL_QUERY };

struct operation {
  op_type type;
  int index, value;
};
```

```
struct fenwick_tree {
  int v[MAX_TIME + 1];
  int n;

  void init(int n) {
    this->n = n;
  }

  void add(int pos, int val) {
    do {
      v[pos] += val;
      pos += pos & -pos;
    } while (pos <= n);
  }

  void set(int pos) {
    add(pos, +1);
  }

  void unset(int pos) {
    add(pos, -1);
  }

  int count(int pos) {
    int s = 0;
    while (pos) {
      s += v[pos];
      pos &= pos - 1;
    }
    return s;
  }
};

struct line_info {
  int time[MAX_SIZE + 1];
  bool value[MAX_SIZE + 1];
  fenwick_tree change[2];
  int size, num_ops;

  void init(int size, int num_ops) {
    this->size = size;
    this->num_ops = num_ops;
    change[0].init(num_ops);
    change[1].init(num_ops);
  }

  void update(int index, int now, int new_value) {
    int old_value = value[index];
    int old_time = time[index];
```

```
    if (old_time) {
      change[old_value].unset(old_time);
    }

    change[new_value].set(now);
    time[index] = now;
    value[index] = new_value;
  }

  int count_zeroes(int index, int now, line_info& other) {
    int last_reset = time[index];
    if (!last_reset) {
      last_reset = num_ops;
    }
    int last_value = value[index];
    int changes = other.change[1 - last_value].count(last_reset);

    return (last_value == 0)
      ? (size - changes)
      : changes;
  }

  void query(int index, int now, line_info& other) {
    int num_zeroes = count_zeroes(index, now, other);
    printf("%d\n", num_zeroes);
  }

};

line_info rows, cols;

operation read_op() {
  char word[MAX_WORD_LENGTH + 1];
  operation op;

  scanf("%s", word);
  if (word[3] == 'Q') {
    op.type = (word[0] == 'R') ? OP_ROW_QUERY : OP_COL_QUERY;
    scanf("%d", &op.index);
  } else {
    op.type = (word[0] == 'R') ? OP_ROW_SET : OP_COL_SET;
    scanf("%d %d", &op.index, &op.value);
  }

  return op;
}

void process_op(operation op, int time) {
  if (op.type == OP_ROW_SET) {
```

```
    rows.update(op.index, time, op.value);
  } else if (op.type == OP_COL_SET) {
    cols.update(op.index, time, op.value);
  } else if (op.type == OP_ROW_QUERY) {
    rows.query(op.index, time, cols);
  } else { // OP_COL_QUERY
    cols.query(op.index, time, rows);
  }
}

void process_ops() {
  int size, num_ops;
  scanf("%d %d", &size, &num_ops);
  rows.init(size, num_ops);
  cols.init(size, num_ops);

  // Inversăm direcția timpului astfel încît operațiile din AIB-uri să fie pe
  // prefix, nu pe sufix.
  for (int time = num_ops; time; time--) {
    operation op = read_op();
    process_op(op, time);
  }
}

int main() {
  process_ops();

  return 0;
}
```

A.3.7 Problema Ball (Codeforces)

‹ înapoi • versiune online

```
#include <algorithm>
#include <stdio.h>

const int MAX_LADIES = 500'000;

struct lady {
  int x, y, z;
};

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct suffix_max_fenwick_tree {
  int v[MAX_LADIES + 1];
```

```
  int n;

  void init(int n) {
    this->n = n;
  }

  void update(int pos, int val) {
    pos = n + 1 - pos;
    do {
      v[pos] = max(v[pos], val);
      pos += pos & -pos;
    } while (pos <= n);
  }

  int suffix_max(int pos) {
    pos = n + 1 - pos;
    int result = 0;
    while (pos) {
      result = max(result, v[pos]);
      pos &= pos - 1;
    }
    return result;
  }
};

lady l[MAX_LADIES];
suffix_max_fenwick_tree fen;
int pos[MAX_LADIES]; // folosit pentru normalizare
int n;

void read_data() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    scanf("%d", &l[i].x);
  }
  for (int i = 0; i < n; i++) {
    scanf("%d", &l[i].y);
  }
  for (int i = 0; i < n; i++) {
    scanf("%d", &l[i].z);
  }
}

void normalize_x() {
  for (int i = 0; i < n; i++) {
    pos[i] = i;
  }

  std::sort(pos, pos + n, [](int a, int b) {
    return l[a].x < l[b].x;
```

```
  });

  int old_x = l[pos[0]].x, new_x = 1;
  for (int i = 0; i < n; i++) {
    if (l[pos[i]].x != old_x) {
      old_x = l[pos[i]].x;
      new_x++;
    }
    l[pos[i]].x = new_x;
  }
}

void sort_ladies_by_z() {
  std::sort(l, l + n, [](lady& a, lady& b) {
    return a.z > b.z;
  });
}

int process_equal_z_batch(int start, int end) {
  int result = 0;

  for (int i = start; i < end; i++) {
    int prev_max_y = fen.suffix_max(l[i].x + 1);
    result += (prev_max_y > l[i].y);
  }
  for (int i = start; i < end; i++) {
    fen.update(l[i].x, l[i].y);
  }

  return result;
}

int count_self_murderers() {
  fen.init(n);

  int result = 0;

  // Procesează calupuri de valori z egale. Aceste doamne nu se domină una pe
  // alta.
  int i = 0;
  while (i < n) {
    int j = i;
    while ((j < n) && (l[j].z == l[i].z)) {
      j++;
    }
    result += process_equal_z_batch(i, j);
    i = j;
  }

  return result;
```

```
}

void write_answer(int answer) {
  printf("%d\n", answer);
}

int main() {
  read_data();
  normalize_x();
  sort_ladies_by_z();
  int answer = count_self_murderers();
  write_answer(answer);

  return 0;
}
```

A.3.8 Problema Medwalk revizitată (Lot 2025)

‹ înapoi • versiune online

```
#include <algorithm>
#include <stdio.h>
#include <vector>

const int MAX_N = 100'000;
const int MAX_QUERIES = 100'000;
const int MAX_VALUE = 300'000;
const int MAX_SEGTREE_NODES = 1 << 18;
const int INF = 1'000'000;
const int OP_UPDATE = 1;

int min(int x, int y) {
  return (x < y) ? x : y;
}

int max(int x, int y) {
  return (x > y) ? x : y;
}

struct matrix {
  int v[2][MAX_N + 1];

  void set(int row, int col, int val) {
    v[row][col] = val;
  }

  int get_min(int col) {
    return min(v[0][col], v[1][col]);
  }
```

```
  int get_max(int col) {
    return max(v[0][col], v[1][col]);
  }
};

struct query {
  int type;
  // Syntactic sugar ca să putem folosi <row, col, val> sau <left, right>, nu
  // <x, y, z>.
  union { int row; int left; };
  union { int col; int right; };
  int val;
};

matrix mat;
query q[MAX_QUERIES];
int n, num_queries;

int next_power_of_2(int x) {
  return 1 << (32 - __builtin_clz(x - 1));
}

// Admite actualizări punctuale și interogări de minim pe interval.
struct min_segment_tree {
  int v[MAX_SEGTREE_NODES];
  int n;

  void init(int size) {
    n = next_power_of_2(size);
  }

  void update(int pos, int val) {
    pos += n;
    v[pos] = val;
    for (pos /= 2; pos; pos /= 2) {
      v[pos] = min(v[2 * pos], v[2 * pos + 1]);
    }
  }

  int range_min(int l, int r) {
    l += n;
    r += n;
    int result = INF;

    while (l <= r)  {
      if (l & 1) {
        result = min(result, v[l++]);
      }
      l >>= 1;
```

```
      if (!(r & 1)) {
        result = min(result, v[r--]);
      }
      r >>= 1;
    }

    return result;
  }
};

// Kudos https://usaco.guide/plat/2DRQ?lang=cpp#offline-2d-bit și
// https://kilonova.ro/submissions/752782
//
// Arbore Fenwick 2D offline.
struct fenwick_2d {
  // Valorile distincte pe fiecare coloană.
  std::vector<int> col_rows[MAX_VALUE + 1];

  // Arborele 1D pe fiecare coloană, peste valorile existente.
  std::vector<int> col_fen[MAX_VALUE + 1];

  int n, max_p2;

  void init(int n) {
    this->n = n;
    max_p2 = 1 << (31 - __builtin_clz(n));
    for (int i = 0; i <= n; i++) {
      col_rows[i].push_back(0);
    }
  }

  // Notează faptul că rîndul row va fi folosit cel puțin o dată pe coloana
  // col.
  void reserve(int row, int col) {
    do {
      col_rows[col].push_back(row);
      col += col & -col;
    } while (col <= n);
  }

  void build() {
    for (int col = 1; col <= n; col++) {
      std::vector<int>& v = col_rows[col]; // syntactic sugar
      std::sort(v.begin(), v.end());
      v.erase(std::unique(v.begin(), v.end()), v.end());
      col_fen[col].resize(v.size() + 1);
    }
  }
```

```
int leftmost_gte(int row, int col) {
  std::vector<int>& v = col_rows[col];
              return std::lower_bound(v.begin(), v.end(), row) - v.begin();
}

// Adaugă delta (bifează / debifează) pentru rîndul row și toți succesorii
// săi în fiecare AIB 1D, atît pe coloana col cît și pe toate coloanele
// succesoare în AIB-ul 2D.
void add(int row, int col, int delta) {
  do {
    int pos = leftmost_gte(row, col);
    do {
      col_fen[col][pos] += delta;
      pos += pos & -pos;
    } while (pos <= (int)col_fen[col].size());
    col += col & -col;
  } while (col <= n);
}

int prefix_sum(int pos, int col) {
  int s = 0;
  while (pos) {
    s += col_fen[col][pos];
    pos &= pos - 1;
  }
  return s;
}

int count_less_than(int val, int col) {
  // Reminder: AIB-ul 1D este normalizat. Află pe ce rînd se află val.
  int pos = leftmost_gte(val, col);
  return prefix_sum(pos - 1, col);
}

int count_in_range(int col, int l, int r) {
  return count_less_than(r + 1, col) - count_less_than(l, col);
}

// Contract: k este 0-based, iar [l, r] este un interval închis de valori
// (rînduri).
int kth_element(int k, int l, int r) {
  int col = 0;

  for (int interval = max_p2; interval; interval >>= 1) {
    if (col + interval <= n) {
      int cnt = count_in_range(col + interval, l, r);
      if (cnt <= k) {
        k -= cnt;
        col += interval;
      }
```

```
      }
    }

    return col + 1;
  }
};

min_segment_tree maxima;
fenwick_2d minima;

void read_data() {
  scanf("%d %d", &n, &num_queries);
  for (int r = 0; r < 2; r++) {
    for (int c = 1; c <= n; c++) {
      int x;
      scanf("%d", &x);
      mat.set(r, c, x);
    }
  }

  for (int i = 0; i < num_queries; i++) {
    query& z = q[i];
    scanf("%d", &z.type);
    if (z.type == OP_UPDATE) {
      scanf("%d %d %d", &z.row, &z.col, &z.val);
      z.row--;
    } else {
      scanf("%d %d", &z.left, &z.right);
    }
  }
}

void build_maxima_segtree() {
  maxima.init(n + 1);
  for (int i = 1; i <= n; i++) {
    maxima.update(i, mat.get_max(i));
  }
}

void simulate() {
  // Rezervă minimele inițiale.
  for (int c = 1; c <= n; c++) {
    minima.reserve(c, mat.get_min(c));
  }

  // Rezervă minimele care iau naștere prin actualizări.
  matrix mat_copy = mat;
  for (int i = 0; i < num_queries; i++) {
    if (q[i].type == OP_UPDATE) {
      mat_copy.set(q[i].row, q[i].col, q[i].val);
```

```
      minima.reserve(q[i].col, mat_copy.get_min(q[i].col));
    }
  }
}

void build_fenwick() {
  minima.init(MAX_VALUE);
  simulate();
  minima.build();

  for (int c = 1; c <= n; c++) {
    minima.add(c, mat.get_min(c), +1);
  }
}

void update(int row, int col, int val) {
  int old_min = mat.get_min(col);
  mat.set(row, col, val);
  int new_min = mat.get_min(col);

  maxima.update(col, mat.get_max(col));

  if (new_min != old_min) {
    minima.add(col, old_min, -1);
    minima.add(col, new_min, +1);
  }
}

int query(int left, int right) {
  int len = right - left + 2;
  int median_pos = (len - 1) / 2;
  int median = minima.kth_element(median_pos, left, right);
  int best_max = maxima.range_min(left, right);

  if (best_max >= median) {
    return median;
  } else {
    // best_max trebuie inserat înainte de median_pos. Răspunsul poate fi la
    // poziția median_pos - 1 sau poate fi chiar best_max.
    int prev = minima.kth_element(median_pos - 1, left, right);
    return max(prev, best_max);
  }
}

void process_queries() {
  for (int i = 0; i < num_queries; i++) {
    if (q[i].type == OP_UPDATE) {
      update(q[i].row, q[i].col, q[i].val);
    } else {
      printf("%d\n", query(q[i].left, q[i].right));
```

```
    }
  }
}

int main() {
  read_data();
  build_maxima_segtree();
  build_fenwick();
  process_queries();

  return 0;
}
```

A.4 Descompunere în radical

A.4.1 Problema Mexitate (ONI 2018 clasa a 9-a)

‹ înapoi

Sursă naivă (versiune online).

```
#include <stdio.h>

const int MAX_ELEMS = 400'000;
const int MOD = 1'000'000'007;

// Costul calculării funcției mex() este mare. Versiunea 2 va folosi o
// structură mai eficientă pentru frequency_tracker.
struct frequency_tracker {
  int f[MAX_ELEMS + 2]; // rows * cols + 1 în cel mai rău caz

  void add(int x) {
    f[x]++;
  }

  void remove(int x) {
    f[x]--;
  }

  int mex() {
    int x = 1;
    while (f[x]) {
      x++;
    }
    return x;
  }
};
```

```
int mat[MAX_ELEMS];
frequency_tracker ft;
int rows, cols, k, l;

void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

void read_right_matrix(FILE* f) {
  for (int r = 0; r < rows; r++) {
    for (int c = 0; c < cols; c++) {
      fscanf(f, "%d", &mat[r * cols + c]);
    }
  }
}

void read_transposed_matrix(FILE* f) {
  for (int r = 0; r < rows; r++) {
    for (int c = 0; c < cols; c++) {
      fscanf(f, "%d", &mat[c * rows + r]);
    }
  }
}

void read_data() {
  FILE* f = fopen("mexitate.in", "r");
  fscanf(f, "%d %d %d %d", &rows, &cols, &k, &l);
  if (k <= l) {
    read_right_matrix(f);
  } else {
    read_transposed_matrix(f);
    swap(rows, cols);
    swap(k, l);
  }
  fclose(f);
}

int get(int r, int c) {
  return mat[r * cols + c];
}

int north_west_corner() {
  for (int r = 0; r < k; r++) {
    for (int c = 0; c < l; c++) {
      ft.add(get(r, c));
    }
  }
  return ft.mex();
```

```
}

void move_right(int r, int c) {
  for (int i = r; i < r + k; i++) {
    ft.remove(get(i, c));
    ft.add(get(i, c + l));
  }
}

void move_left(int r, int c) {
  for (int i = r; i < r + k; i++) {
    ft.remove(get(i, c + l - 1));
    ft.add(get(i, c - 1));
  }
}

void move_down(int r, int c) {
  for (int j = c; j < c + l; j++) {
    ft.remove(get(r, j));
    ft.add(get(r + k, j));
  }
}


int left_right_snake() {
  north_west_corner();
  long long result = ft.mex();
  int r = 0, c = 0;
  int final_r = rows - k;
  int final_c = (final_r % 2) ? 0 : (cols - l);

  while ((r != final_r) || (c != final_c)) {
    if ((r % 2 == 0) && (c < cols - l)) {
      move_right(r, c++);
    } else if ((r % 2 == 1) && (c > 0)) {
      move_left(r, c--);
    } else {
      move_down(r++, c);
    }
    result = result * ft.mex() % MOD;
  }

  return result;
}

void write_answer(int answer) {
  FILE* f = fopen("mexitate.out", "w");
  fprintf(f, "%d\n", answer);
  fclose(f);
}
```

```
int main() {
  read_data();
  int answer = left_right_snake();
  write_answer(answer);

  return 0;
}
```

Sursă eficientă (versiune online).

```
#include <stdio.h>

const int MAX_ELEMS = 400'000;
const int BLOCK_SIZE = 400;
const int MAX_BLOCKS = (MAX_ELEMS - 1) / BLOCK_SIZE + 1;
const int MOD = 1'000'000'007;

struct frequency_tracker {
  int f[MAX_ELEMS + 2]; // rows * cols + 1 în cel mai rău caz
  int block_nonzero[MAX_BLOCKS];

  void init() {
    add(0); // Ca să nu-l returnăm niciodată.
  }

  void add(int x) {
    if (++f[x] == 1) {
      block_nonzero[x / BLOCK_SIZE]++;
    }
  }

  void remove(int x) {
    if (--f[x] == 0) {
      block_nonzero[x / BLOCK_SIZE]--;
    }
  }

  int mex() {
    int b = 0;
    while (block_nonzero[b] == BLOCK_SIZE) {
      b++;
    }
    int x = b * BLOCK_SIZE;
    while (f[x]) {
      x++;
    }
    return x;
  }
```

```
};

int mat[MAX_ELEMS];
frequency_tracker ft;
int rows, cols, k, l;

void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

void read_right_matrix(FILE* f) {
  for (int r = 0; r < rows; r++) {
    for (int c = 0; c < cols; c++) {
      fscanf(f, "%d", &mat[r * cols + c]);
    }
  }
}

void read_transposed_matrix(FILE* f) {
  for (int r = 0; r < rows; r++) {
    for (int c = 0; c < cols; c++) {
      fscanf(f, "%d", &mat[c * rows + r]);
    }
  }
}

void read_data() {
  FILE* f = fopen("mexitate.in", "r");
  fscanf(f, "%d %d %d %d", &rows, &cols, &k, &l);
  if (k <= l) {
    read_right_matrix(f);
  } else {
    read_transposed_matrix(f);
    swap(rows, cols);
    swap(k, l);
  }
  fclose(f);
}

int get(int r, int c) {
  return mat[r * cols + c];
}

int north_west_corner() {
  for (int r = 0; r < k; r++) {
    for (int c = 0; c < l; c++) {
      ft.add(get(r, c));
    }
```

```
  }
  return ft.mex();
}

void move_right(int r, int c) {
  for (int i = r; i < r + k; i++) {
    ft.remove(get(i, c));
    ft.add(get(i, c + l));
  }
}

void move_left(int r, int c) {
  for (int i = r; i < r + k; i++) {
    ft.remove(get(i, c + l - 1));
    ft.add(get(i, c - 1));
  }
}

void move_down(int r, int c) {
  for (int j = c; j < c + l; j++) {
    ft.remove(get(r, j));
    ft.add(get(r + k, j));
  }
}


int left_right_snake() {
  ft.init();
  north_west_corner();
  long long result = ft.mex();
  int r = 0, c = 0;
  int final_r = rows - k;
  int final_c = (final_r % 2) ? 0 : (cols - l);

  while ((r != final_r) || (c != final_c)) {
    if ((r % 2 == 0) && (c < cols - l)) {
      move_right(r, c++);
    } else if ((r % 2 == 1) && (c > 0)) {
      move_left(r, c--);
    } else {
      move_down(r++, c);
    }
    result = result * ft.mex() % MOD;
  }

  return result;
}

void write_answer(int answer) {
  FILE* f = fopen("mexitate.out", "w");
```

```
  fprintf(f, "%d\n", answer);
  fclose(f);
}

int main() {
  read_data();
  int answer = left_right_snake();
  write_answer(answer);

  return 0;
}
```

A.4.2 Problema Give Away (SPOJ)

înapoi

Sursă cu multiseturi PBDS (versiune online).

```
// Complexitate: O(Q sqrt N log N).
//
// Metodă: Descompunere în radical. Pe fiecare bloc reține vectorul naiv și un
// set de valori pentru căutări în timp logaritmic. Avem nevoie de multisets
// pentru că pot exista duplicate și avem nevoie de PBDS pentru funcția
// order_of_key.
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <stdio.h>

const int MAX_N = 500000;
const int BLOCK_SIZE = 10000;
const int MAX_BLOCKS = (MAX_N - 1) / BLOCK_SIZE + 1;
const int T_QUERY = 0;

// Multiseturile PBDS sînt obscene, dar par să meargă. Ștergerile necesită cod
// extra. Vezi https://stackoverflow.com/q/59731946/6022817
typedef __gnu_pbds::tree<
  int,
  __gnu_pbds::null_type,
  std::less_equal<int>,
  __gnu_pbds::rb_tree_tag,
  __gnu_pbds::tree_order_statistics_node_update
  > ordered_set;

// Toate intervalele sînt [închis, deschis).
struct block {
  int v[BLOCK_SIZE];
  ordered_set s;

  void set(int pos, int val) {
```

```
    if (v[pos]) {
      int rank = s.order_of_key(v[pos]);
      ordered_set::iterator it = s.find_by_order(rank);
      s.erase(it);
    }
    v[pos] = val;
    s.insert(v[pos]);
  }

  int partial_count(int l, int r, int val) {
    int cnt = 0;
    while (l < r) {
      cnt += (v[l++] >= val);
    }
    return cnt;
  }

  int prefix_count(int end, int val) {
    return partial_count(0, end, val);
  }

  int suffix_count(int start, int val) {
    return partial_count(start, BLOCK_SIZE, val);
  }

  int whole_count(int val) {
    return s.size() - s.order_of_key(val);
  }
};

block b[MAX_BLOCKS];
int n;

void read_array() {
  scanf("%d", &n);

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    b[i / BLOCK_SIZE].set(i % BLOCK_SIZE, x);
  }
}

int process_query(int l, int r, int val) {
  int bl = l / BLOCK_SIZE, offset_l = l % BLOCK_SIZE;
  int br = r / BLOCK_SIZE, offset_r = r % BLOCK_SIZE;
  if (bl == br) {
    return b[bl].partial_count(offset_l, offset_r, val);
  } else {
    int cnt = b[bl].suffix_count(offset_l, val)
```

```
      + b[br].prefix_count(offset_r, val);
    for (int i = bl + 1; i < br; i++) {
      cnt += b[i].whole_count(val);
    }
    return cnt;
  }
}

void process_update(int pos, int val) {
  b[pos / BLOCK_SIZE].set(pos % BLOCK_SIZE, val);
}

void process_ops() {
  int num_ops, type, pos1, pos2, val;
  scanf("%d", &num_ops);

  while (num_ops--) {
    scanf("%d", &type);
    if (type == T_QUERY) {
      scanf("%d %d %d", &pos1, &pos2, &val);
      int count = process_query(pos1 - 1, pos2, val);
      printf("%d\n", count);
    } else {
      scanf("%d %d", &pos1, &val);
      process_update(pos1 - 1, val);
    }
  }
}

int main() {
  read_array();
  process_ops();

  return 0;
}
```

Sursă elementară (versiune online).

```
// Complexitate: O(Q sqrt N log N).
//
// Metodă: Descompunere în radical. Pe fiecare bloc reține vectorul naiv și un
// vector sortat pentru căutări în timp logaritmic.
#include <algorithm>
#include <stdio.h>

const int MAX_N = 500'000;
const int BLOCK_SIZE = 3'000;
const int MAX_BLOCKS = (MAX_N - 1) / BLOCK_SIZE + 1;
const int T_QUERY = 0;
```

```
// Toate intervalele sînt [închis, deschis).
struct block {
  int v[BLOCK_SIZE];
  int s[BLOCK_SIZE];
  int size;

  void push(int val) {
    v[size] = s[size] = val;
    size++;
  }

  void sort() {
    std::sort(s, s + size);
  }

  // Returnează cea mai din stînga poziție a unui element >= val.
  int bin_search(int val) {
    if (val < s[0]) {
      return 0;
    } else if (val > s[size - 1]) {
      return size;
    }
    int l = -1, r = size - 1; // (l, r]

    while (r - l > 1) {
      int mid = (l + r) >> 1;
      if (s[mid] < val) {
        l = mid;
      } else {
        r = mid;
      }
    }

    return r;
  }

  void migrate_left(int pos) {
    int save = s[pos];
    while (pos && (s[pos - 1] > save)) {
      s[pos] = s[pos - 1];
      pos--;
    }
    s[pos] = save;
  }

  void migrate_right(int pos) {
    int save = s[pos];
    while ((pos < size - 1) && (s[pos + 1] < save)) {
      s[pos] = s[pos + 1];
```

```
      pos++;
    }
    s[pos] = save;
  }

  void set(int pos, int val) {
    int sorted_pos = bin_search(v[pos]);
    s[sorted_pos] = val;
    migrate_left(sorted_pos);
    migrate_right(sorted_pos);
    v[pos] = val;
  }

  int partial_count(int l, int r, int val) {
    int cnt = 0;
    while (l < r) {
      cnt += (v[l++] >= val);
    }
    return cnt;
  }

  int prefix_count(int end, int val) {
    return partial_count(0, end, val);
  }

  int suffix_count(int start, int val) {
    return partial_count(start, BLOCK_SIZE, val);
  }

  int whole_count(int val) {
    return size - bin_search(val);
  }
};

block b[MAX_BLOCKS];
int n;

void read_array() {
  scanf("%d", &n);

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    b[i / BLOCK_SIZE].push(x);
  }

  int end_block = (n - 1) / BLOCK_SIZE + 1;
  for (int i = 0; i < end_block; i++) {
    b[i].sort();
  }
```

```
}

int process_query(int l, int r, int val) {
  int bl = l / BLOCK_SIZE, offset_l = l % BLOCK_SIZE;
  int br = r / BLOCK_SIZE, offset_r = r % BLOCK_SIZE;
  if (bl == br) {
    return b[bl].partial_count(offset_l, offset_r, val);
  } else {
    int cnt = b[bl].suffix_count(offset_l, val)
      + b[br].prefix_count(offset_r, val);
    for (int i = bl + 1; i < br; i++) {
      cnt += b[i].whole_count(val);
    }
    return cnt;
  }
}

void process_update(int pos, int val) {
  b[pos / BLOCK_SIZE].set(pos % BLOCK_SIZE, val);
}

void process_ops() {
  int num_ops, type, pos1, pos2, val;
  scanf("%d", &num_ops);

  while (num_ops--) {
    scanf("%d", &type);
    if (type == T_QUERY) {
      scanf("%d %d %d", &pos1, &pos2, &val);
      int count = process_query(pos1 - 1, pos2, val);
      printf("%d\n", count);
    } else {
      scanf("%d %d", &pos1, &val);
      process_update(pos1 - 1, val);
    }
  }
}

int main() {
  read_array();
  process_ops();

  return 0;
}
```

A.4.3 Problema Holes (Codeforces)

‹ înapoi • versiune online

```
#include <stdio.h>

const int MAX_N = 100'000;
// Preferăm blocuri puțin mai mari deoarece actualizările au *cache locality*,
// pe cînd interogările sar mai mult.
const int BUCKET_SIZE = 1'000;
const int OP_UPDATE = 0;

struct hole {
  int power;
  int last;  // ultima destinație din același bloc
  int jumps; // numărul de salturi pînă la last
};

hole h[MAX_N + 1];
int n, num_queries;

void read_data() {
  scanf("%d %d", &n, &num_queries);
  for (int i = 0; i < n; i++) {
    scanf("%d", &h[i].power);
  }
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

// Actualizează găurile de la începutul blocului pînă la pos invlusiv.
void update_bucket_of(int pos) {
  int start = pos / BUCKET_SIZE * BUCKET_SIZE;
  int end = min(start + BUCKET_SIZE, n);
  for (int i = pos; i >= start; i--) {
    int dest = i + h[i].power;
    if (dest < end) {
      h[i].last = h[dest].last;
      h[i].jumps = h[dest].jumps + 1;
    } else {
      h[i].last = i;
      h[i].jumps = 0;
    }
  }
}

void init_buckets() {
  for (int start = 0; start < n; start += BUCKET_SIZE) {
    int end = min(start + BUCKET_SIZE - 1, n - 1);
    update_bucket_of(end);
  }
```

```
}

void query(int pos, int* last, int* count) {
  *count = 0;

  do {
    *count += h[pos].jumps + 1;
    *last = h[pos].last;
    pos = *last + h[*last].power;
  } while (pos < n);
}

void process_queries() {
  while (num_queries--) {
    int type, a;
    scanf("%d %d", &type, &a);
    a--;

    if (type == OP_UPDATE) {
      int power;
      scanf("%d", &power);
      h[a].power = power;
      update_bucket_of(a);
    } else {
      int last, num_jumps;
      query(a, &last, &num_jumps);
      printf("%d %d\n", 1 + last, num_jumps);
    }
  }
}

int main() {
  read_data();
  init_buckets();
  process_queries();

  return 0;
}
```

A.4.4 Problema Piezișă (Baraj ONI 2022)

înapoi

Sursă forță brută (versiune online).

```
#include <algorithm>
#include <stdio.h>

#define MAX_N 500000
```

```
#define INFINITY (MAX_N + 1)
#define NONE -1

int v[MAX_N + 1]; // partial xor's
int pos[MAX_N + 1];
int first[MAX_N + 2];
int n, distinct;

int max(int x, int y) {
  return (x > y) ? x : y;
}

// Returnează cea mai din dreapta apariție a lui val pe poziția p sau înainte,
// sau NONE dacă val nu apare pe poziția p sau înainte.
int rightmost(int val, int p) {
  int l = first[val], r = first[val + 1]; // [l, r)
  while (r - l > 1) {
    int m = (l + r) >> 1;
    if (pos[m] > p) {
      r = m;
    } else {
      l = m;
    }
  }

  // Caz special: p < orice poziție unde apare val. Căutarea binară normală ar
  // returna pos[l].
  return (pos[l] <= p) ? pos[l] : NONE;
}

int main() {
  FILE* fin = fopen("piezisa.in", "r");
  FILE* fout = fopen("piezisa.out", "w");
  fscanf(fin, "%d", &n);

  // Citește datele și calculează xor-uri parțiale.
  v[0] = 0;
  for (int i = 1; i <= n; i++) {
    int x;
    fscanf(fin, "%d", &x);
    v[i] = v[i - 1] ^ x;
  }

  // Sortează pozițiile după valoarea xor parțială, apoi după poziție.
  for (int i = 0; i <= n; i++) {
    pos[i] = i;
  }
  std::sort(pos, pos + n + 1, [](int a, int b) {
    return (v[a] < v[b]) || ((v[a] == v[b]) && (a < b));
  });
```

```
  // Renumerotează valorile începînd cu n; colectează pozițiile.
  int from = -1;
  distinct = 0;
  for (int i = 0; i <= n; i++) {
    if (v[pos[i]] != from) {
      from = v[pos[i]];
      first[distinct++] = i;
    }
    v[pos[i]] = distinct - 1;
  }
  first[distinct] = n + 1;
  // Acum pos[first[i]...first[i+1]] conține o listă ordonată cu pozițiile
  // aparițiilor valorii i.

  int num_queries;
  fscanf(fin, "%d", &num_queries);
  while (num_queries--) {
    int l, r;
    fscanf(fin, "%d %d", &l, &r);
    r++;

    int end = r, best = INFINITY;
    // cît timp avem loc să avansăm și sperăm să îmbunătățim soluția existentă
    while ((end <= n) && (end - l < best)) {
      int start = rightmost(v[end], l);
      if ((start != NONE) && (end - start < best)) {
        best = end - start;
      }
      end++;
    }

    fprintf(fout, "%d\n", (best == INFINITY) ? NONE : best);
  }

  fclose(fin);
  fclose(fout);

  return 0;
}
```

Sursă cu metoda 1 (versiune online).

```
#include <algorithm>
#include <math.h>
#include <stdio.h>

const int MAX_BUCKETS = 354;
const int MAX_BUCKET_SIZE = 1416;
```

```
const int MAX_N = MAX_BUCKETS * MAX_BUCKET_SIZE;
const int MAX_Q = 500000;
const int INF = MAX_N + 1;
const int NONE = -1;

struct query {
  int l, r, orig_index;
};

int v[MAX_N + 1]; // xor-uri parțiale
int pos[MAX_N + 1];
int ptr[MAX_N + 1];
int last[MAX_N + 1];
int best[MAX_BUCKETS];

query q[MAX_Q];
// Logic ar trebui stocat în query.answer, dar astfel evităm o sortare.
int answer[MAX_Q];

int n, num_queries;
int bs, nb;

void read_array() {
  scanf("%d", &n);

  v[0] = 0;
  for (int i = 1; i <= n; i++) {
    scanf("%d", &v[i]);
    v[i] ^= v[i - 1];
  }
}

void read_queries() {
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%d %d", &q[i].l, &q[i].r);
    q[i].r++;
    q[i].orig_index = i;
  }
}

void sort_queries() {
  std::sort(q, q + num_queries, [](query a, query b) {
    return (a.l < b.l);
  });
}

void sort_positions() {
  for (int i = 0; i <= n; i++) {
    pos[i] = i;
```

```
  }
  std::sort(pos, pos + n + 1, [](int a, int b) {
    return (v[a] < v[b]) || ((v[a] == v[b]) && (a > b));
  });
}

void preprocess_values() {
  nb = 0.5 * sqrt(n + 1); // determinat experimental
  bs = n / nb + 1;

  sort_positions();

  // renumerotează valorile de la 0; creează pointeri
  int prev = -1, distinct = 0;
  for (int i = 0; i <= n; i++) {
    if (v[pos[i]] == prev) {
      v[pos[i]] = distinct - 1;
      bool same_bucket = (pos[i] / bs == pos[i - 1] / bs);
      ptr[pos[i]] = same_bucket
        ? ptr[pos[i - 1]]
        : pos[i - 1];
    } else {
      // începi o serie nouă de la sfîrșitul vectorului
      prev = v[pos[i]];
      v[pos[i]] = distinct++;
      ptr[pos[i]] = NONE;
    }
  }
}

inline int min(int x, int y) {
  return (x < y) ? x : y;
}

int answer_query(int l, int r) {
  int result = INF;

  // procedează incremental pînă la blocul următor
  int b = r / bs + 1, boundary = min(b * bs, n + 1);
  while (r < boundary) {
    result = min(result, r - last[v[r]]);
    r++;
  }

  // procedează bloc cu bloc restul vectorului
  while (b < nb) {
    result = min(result, best[b++]);
  }

  return (result == INF) ? NONE : result;
```

```
}

void scan() {
  for (int i = 0; i <= n; i++) {
    last[i] = -INF;
  }
  for (int i = 0; i < nb; i++) {
    best[i] = INF;
  }

  int qi = 0;
  for (int l = 0; l <= n; l++) {
    // actualizează-l pe last
    last[v[l]] = l;

    // actualizează-l pe best
    for (int i = ptr[l]; i != NONE; i = ptr[i]) {
      int b = i / bs;
      best[b] = min(best[b], i - l);
    }

    // răspunde la interogările care încep la l
    while (qi < num_queries && q[qi].l == l) {
      answer[q[qi].orig_index] = answer_query(q[qi].l, q[qi].r);
      qi++;
    }
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", answer[i]);
  }
}

int main() {
  read_array();
  read_queries();
  sort_queries();
  preprocess_values();
  scan();
  write_answers();

  return 0;
}
```

Sursă cu metoda 2 (versiune online).

```
// Rescrisă după https://infoarena.ro/job_detail/3032904
```

```
//
// Complexitate: O((N + Q) sqrt(N)).
#include <algorithm>
#include <stdio.h>

const int MAX_N = 500'000;
const int BLOCK_SIZE = 700;
const int NUM_BLOCKS = MAX_N / BLOCK_SIZE + 1;
const int MAX_Q = 500'000;
const int INFINITY = 1'000'000;
const int NONE = -1;

struct query {
  int l, r;
  int orig_index;
  int block;
};

int v[MAX_N + 1];
query q[MAX_Q];
int left[MAX_N + 1], right[MAX_N + 1];
int* pos = left; // folosit inițial pentru normalizare;

// Logic ar trebui stocat în query.answer, dar astfel evităm o sortare.
int answer[MAX_Q];

int n, num_queries, max_value;

void read_array() {
  scanf("%d", &n);

  v[0] = 0;
  for (int i = 1; i <= n; i++) {
    scanf("%d", &v[i]);
    v[i] ^= v[i - 1];
  }
}

void read_queries() {
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%d %d", &q[i].l, &q[i].r);
    q[i].l++;
    q[i].r++; // inteval închis, indexat de la 1
    q[i].orig_index = i;
    q[i].block = q[i].l / BLOCK_SIZE;
  }
}

void normalize_array() {
```

```
  for (int i = 1; i <= n; i++) {
    pos[i] = i;
  }
  std::sort(pos + 1, pos + n + 1, [](int a, int b) {
    return v[a] < v[b];
  });

  int prev = NONE;
  max_value = NONE;
  for (int i = 0; i <= n; i++) {
    if (v[pos[i]] != prev) {
      max_value++;
    }
    prev = v[pos[i]];
    v[pos[i]] = max_value;
  }
}

void sort_queries_by_left_block() {
  std::sort(q, q + num_queries, [](query a, query b) {
    return (a.block < b.block) ||
      ((a.block == b.block) && (a.r > b.r));
  });
}

void init_left() {
  for (int i = 0; i <= max_value; i++) {
    left[i] = -INFINITY;
  }
}

void init_right() {
  for (int i = 0; i <= max_value; i++) {
    right[i] = +INFINITY;
  }
}

int min(int x, int y) {
  return (x < y) ? x : y;
}

void answer_query(query q, int closest_left_right) {
  int best = closest_left_right;
  for (int i = q.block * BLOCK_SIZE; i < q.l; i++) {
    best = min(best, right[v[i]] - i);
  }

  answer[q.orig_index] = (best > n) ? NONE : best;
}
```

```
int answer_queries_in_block(int block, int q_index) {
  init_right();
  int ptr = n + 1;
  int closest_left_right = INFINITY;
  while ((q_index < num_queries) && (q[q_index].block == block)) {
    while (ptr > q[q_index].r) {
      --ptr;
      right[v[ptr]] = ptr;
      int dist = ptr - left[v[ptr]];
      closest_left_right = min(closest_left_right, dist);
    }
    answer_query(q[q_index], closest_left_right);
    q_index++;
  }

  return q_index;
}

void update_left(int block) {
  int start = block * BLOCK_SIZE;
  int end = min(start + BLOCK_SIZE - 1, n);
  for (int i = start; i <= end; i++) {
    left[v[i]] = i;
  }
}

void scan_blocks() {
  init_left();
  int q_index = 0;

  int last_block = n / BLOCK_SIZE;
  for (int b = 0; b <= last_block; b++) {
    q_index = answer_queries_in_block(b, q_index);
    update_left(b);
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%d\n", answer[i]);
  }
}

int main() {
  read_array();
  read_queries();
  normalize_array();
  sort_queries_by_left_block();

  scan_blocks();
```

```
  write_answers();

  return 0;
}
```

A.4.5 Problema Serega and Fun (Codeforces)

înapoi

Sursă cu deque (versiune online).

```
#include <deque>
#include <stdio.h>
#include <unordered_map>

const int MAX_N = 100'000;
const int BUCKET_SIZE = 2'000;
const int MAX_BUCKETS = (MAX_N - 1) / BUCKET_SIZE + 1;
const int TYPE_ROTATE = 1;
const int TYPE_COUNT = 2;

struct freq {
  std::unordered_map<int, short> map;

  void add(int val) {
    map[val]++;
  }

  void remove(int val) {
    auto it = map.find(val);
    if (it->second == 1) {
      map.erase(it);
    } else {
      it->second--;
    }
  }

  int count(int val) {
    auto it = map.find(val);
    return (it == map.end()) ? 0 : it->second;
  }
};

struct bucket {
  freq f;
  std::deque<int> deq;

  void push(int val) {
```

```
  deq.push_back(val);
  f.add(val);
}

void set(int pos, int val) {
  f.remove(deq[pos]);
  deq[pos] = val;
  f.add(val);
}

int shift(int val) {
  deq.push_front(val);
  f.add(val);
  int last = deq.back();
  deq.pop_back();
  f.remove(last);
  return last;
}

void rotate(int l, int r) {
  int val = deq[r];
  deq.erase(deq.begin() + r);
  deq.insert(deq.begin() + l, val);
}

int remove(int pos) {
  int val = deq[pos];
  f.remove(val);
  deq.erase(deq.begin() + pos);
  return val;
}

int remove_last() {
  return remove(BUCKET_SIZE - 1);
}

void insert(int pos, int val) {
  deq.insert(deq.begin() + pos, val);
  f.add(val);
}

void insert_first(int val) {
  insert(0, val);
}

int partial_count(int l, int r, int k) {
  int result = 0;
  while (l <= r) {
    result += (deq[l++] == k);
  }
```

```
    return result;
  }

  int prefix_count(int pos, int k) {
    return partial_count(0, pos, k);
  }

  int suffix_count(int pos, int k) {
    return partial_count(pos, BUCKET_SIZE - 1, k);
  }

};

bucket buck[MAX_BUCKETS];
int n, num_ops;
int last_answer;

void read_data() {
  scanf("%d", &n);

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    buck[i / BUCKET_SIZE].push(x);
  }

  scanf("%d", &num_ops);
}

int bucket_count(int l, int r, int k) {
  int result = 0;
  while (l < r) {
    result += buck[l++].f.count(k);
  }
  return result;
}

void process_count_op(int l, int r, int k) {
  int bl = l / BUCKET_SIZE, offset_l = l % BUCKET_SIZE;
  int br = r / BUCKET_SIZE, offset_r = r % BUCKET_SIZE;
  if (bl == br) {
    last_answer = buck[bl].partial_count(offset_l, offset_r, k);
  } else {
    last_answer =
      buck[bl].suffix_count(offset_l, k) +
      bucket_count(bl + 1, br, k) +
      buck[br].prefix_count(offset_r, k);
  }

  printf("%d\n", last_answer);
```

```
}

void process_rotate_op(int l, int r) {
  int bl = l / BUCKET_SIZE, offset_l = l % BUCKET_SIZE;
  int br = r / BUCKET_SIZE, offset_r = r % BUCKET_SIZE;
  if (bl == br) {
    buck[bl].rotate(offset_l, offset_r);
  } else {
    int from_left = buck[bl].remove_last();
    for (int b = bl + 1; b < br; b++) {
      from_left = buck[b].shift(from_left);
    }
    int to_right = buck[br].remove(offset_r);
    buck[br].insert_first(from_left);
    buck[bl].insert(offset_l, to_right);
  }
}

int transform(int x) {
  return last_answer
    ? (x + last_answer - 1) % n + 1
    : x;
}

void process_ops() {
  while (num_ops--) {
    int type, l, r, k;
    scanf("%d %d %d", &type, &l, &r);
    l = transform(l) - 1;
    r = transform(r) - 1;
    if (l > r) {
      int tmp = l; l = r; r = tmp;
    }

    if (type == TYPE_COUNT) {
      scanf("%d", &k);
      k = transform(k);
      process_count_op(l, r, k);
    } else {
      process_rotate_op(l, r);
    }
  }
}

int main() {
  read_data();
  process_ops();

  return 0;
```

```
}
```

Sursă cu descompunere după poziții (versiune online).

```
#include <math.h>
#include <stdio.h>

const int MAX_BUCKETS = 160;
const int MAX_BUCKET_SIZE = 634;
const int MAX_N = MAX_BUCKETS * MAX_BUCKET_SIZE;
const int TYPE_ROTATE = 1;
const int TYPE_COUNT = 2;

typedef struct {
  short freq[MAX_N];
  int circ[MAX_BUCKET_SIZE];
  int start;
} bucket;

bucket buck[MAX_BUCKETS];
int modulo[2 * MAX_BUCKET_SIZE];
int bs; // bucket size
int n, numOps;
int lastAnswer;

void initBuckets() {
  bs = 2 * sqrt(n);

  for (int i = 0; i < 2 * bs; i++) {
    modulo[i] = i % bs;
  }
}

void bucketSetInitial(int pos, int val) {
  buck[pos / bs].circ[pos % bs] = val;
  buck[pos / bs].freq[val]++;
}

void readInputData() {
  scanf("%d", &n);
  initBuckets();

  for (int i = 0; i < n; i++) {
    int x;
    scanf("%d", &x);
    bucketSetInitial(i, x);
  }

  scanf("%d", &numOps);
```

```
}

int realPos(int b, int pos) {
  return modulo[pos + buck[b].start];
}

int next(int pos) {
  return modulo[pos + 1];
}

int prev(int pos) {
  return modulo[pos + bs - 1];
}

void bucketSet(int b, int pos, int val) {
  bucket& g = buck[b];
  int realP = realPos(b, pos);
  int oldVal = g.circ[realP];
  g.circ[realP] = val;
  g.freq[oldVal]--;
  g.freq[val]++;
}

int partialCount(int b, int l, int r, int k) {
  int realL = realPos(b, l);
  int realR = realPos(b, r);

  // Numără-l separat pe realR ca să îl putem folosi ca terminator de buclă.
  int result = (buck[b].circ[realR] == k);

  while (realL != realR) {
    result += (buck[b].circ[realL] == k);
    realL = next(realL);
  }
  return result;
}

int bucketCount(int l, int r, int k) {
  int result = 0;
  while (l < r) {
    result += buck[l++].freq[k];
  }
  return result;
}

void processCountOp(int l, int r, int k) {
  int bl = l / bs, br = r / bs;
  int lpos = l - bl * bs, rpos = r - br * bs;
  if (bl == br) {
    lastAnswer = partialCount(bl, lpos, rpos, k);
```

```
  } else {
    lastAnswer =
      partialCount(bl, lpos, bs - 1, k) +
      partialCount(br, 0, rpos, k) +
      bucketCount(bl + 1, br, k);
  }

  printf("%d\n", lastAnswer);
}

int partialRotate(int b, int l, int r) {
  int *v = buck[b].circ;
  int realL = realPos(b, l);
  int realR = realPos(b, r);
  int prevR = prev(realR);
  int save = v[realR];

  while (realR != realL) {
    v[realR] = v[prevR];
    realR = prevR;
    prevR = prev(realR);
  }

  v[realL] = save;
  return save;
}

int bucketRotate(int b) {
  bucket& g = buck[b];
  g.start = prev(g.start);
  return g.circ[g.start];
}

void processRotateOp(int l, int r) {
  int bl = l / bs, br = r / bs;
  int lpos = l - bl * bs, rpos = r - br * bs;
  if (bl == br) {
    partialRotate(bl, lpos, rpos);
  } else {
    int fromLeft = partialRotate(bl, lpos, bs - 1);
    for (int b = bl + 1; b < br; b++) {
      int toRight = bucketRotate(b);
      bucketSet(b, 0, fromLeft);
      fromLeft = toRight;
    }
    int toRight = partialRotate(br, 0, rpos);
    bucketSet(br, 0, fromLeft);
    bucketSet(bl, lpos, toRight);
  }
}
```

```
int transform(int x) {
  return lastAnswer
    ? (x + lastAnswer - 1) % n + 1
    : x;
}

void processOps() {
  while (numOps--) {
    int type, l, r, k;
    scanf("%d %d %d", &type, &l, &r);
    l = transform(l) - 1;
    r = transform(r) - 1;
    if (l > r) {
      int tmp = l; l = r; r = tmp;
    }

    if (type == TYPE_COUNT) {
      scanf("%d", &k);
      k = transform(k);
      processCountOp(l, r, k);
    } else {
      processRotateOp(l, r);
    }
  }
}

int main() {
  readInputData();
  processOps();

  return 0;
}
```

Sursă cu descompunere după operații (versiune online).

```
#include <math.h>
#include <stdio.h>

const int MAX_BUCKETS = 160;
const int MAX_BUCKET_SIZE = 634;
const int OVERFLOW_FACTOR = 2;
const int MAX_N = MAX_BUCKETS * MAX_BUCKET_SIZE;
const int TYPE_ROTATE = 1;
const int TYPE_COUNT = 2;

struct bucket {
  short freq[MAX_N];
  int data[OVERFLOW_FACTOR * MAX_BUCKET_SIZE];
```

```
int size;

void append(int x) {
  data[size++] = x;
  freq[x]++;
}

// Returnează numărul de elemente vărsate. Golește data, size și freq.
int empty(int* dest) {
  for (int i = 0; i < size; i++) {
    dest[i] = data[i];
    freq[data[i]]--;
  }
  int old_size = size;
  size = 0;

  return old_size;
}

int partial_count(int l, int r, int val) {
  int cnt = 0;
  while (l <= r) {
    cnt += (data[l++] == val);
  }

  return cnt;
}

int prefix_count(int pos, int val) {
  return partial_count(0, pos, val);
}

int suffix_count(int pos, int val) {
  return partial_count(pos, size - 1, val);
}

void insert(int pos, int val) {
  freq[val]++;
  size++;
  for (int i = size - 1; i > pos; i--) {
    data[i] = data[i - 1];
  }
  data[pos] = val;
}

int remove(int pos) {
  int result = data[pos];
  freq[result]--;
  for (int i = pos; i < size - 1; i++) {
    data[i] = data[i + 1];
```

```
    }
    size--;

    return result;
  }
};

bucket buck[MAX_BUCKETS];
int naive[MAX_N];
int bs, nb; // mărimea blocului, numărul de blocuri
int n;
int last_answer;

// Stochează informații despre blocul în care se află un indice real.
struct bucket_info {
  int b;      // numărul blocului
  int start;  // indicele primului element din bloc
  int offset; // offset-ul indicelui dat față de start

  void find_next_bucket(int index) {
    while (start + buck[b].size <= index) {
      start += buck[b++].size;
    }
    offset = index - start;
  }
};

int min(int x, int y) {
  return (x < y) ? x : y;
}

void distribute_buckets() {
  // Evită O(n) împărțiri deoarece vom rula acest cod de O(sqrt(n)) ori.
  for (int b = 0; b < nb; b++) {
    int start = b * bs;
    int end = min(start + bs, n);
    for (int i = start; i < end; i++) {
      buck[b].append(naive[i]);
    }
  }
}

void collect_buckets() {
  int ptr = 0;
  for (int i = 0; i < nb; i++) {
    ptr += buck[i].empty(&naive[ptr]);
  }
}

void read_data() {
```

```
  scanf("%d", &n);

  for (int i = 0; i < n; i++) {
    scanf("%d", &naive[i]);
  }
}

void init_buckets() {
  bs = 2 * sqrt(n);
  nb = (n - 1) / bs + 1;
  distribute_buckets();
}

int cross_bucket_count(int l, int r, int k) {
  int result = 0;
  while (l < r) {
    result += buck[l++].freq[k];
  }
  return result;
}

// [l, r] inclusiv
void process_count_op(int l, int r, int k) {
  bucket_info bl = { 0, 0, 0 };
  bl.find_next_bucket(l);
  bucket_info br = bl;
  br.find_next_bucket(r);

  if (bl.b == br.b) {
    last_answer = buck[bl.b].partial_count(bl.offset, br.offset, k);
  } else {
    last_answer =
      buck[bl.b].suffix_count(bl.offset, k) +
      buck[br.b].prefix_count(br.offset, k) +
      cross_bucket_count(bl.b + 1, br.b, k);
  }

  printf("%d\n", last_answer);
}

void process_rotate_op(int l, int r) {
  bucket_info bl = { 0, 0, 0 };
  bl.find_next_bucket(l);
  bucket_info br = bl;
  br.find_next_bucket(r);

  int x = buck[br.b].remove(br.offset);
  buck[bl.b].insert(bl.offset, x);

  if (buck[bl.b].size == OVERFLOW_FACTOR * bs) {
```

```
    collect_buckets();
    distribute_buckets();
  }
}

int transform(int x) {
  return last_answer
    ? (x + last_answer - 1) % n + 1
    : x;
}

void process_ops() {
  int num_ops;
  scanf("%d", &num_ops);

  while (num_ops--) {
    int type, l, r, k;
    scanf("%d %d %d", &type, &l, &r);
    l = transform(l) - 1;
    r = transform(r) - 1;
    if (l > r) {
      int tmp = l; l = r; r = tmp;
    }

    if (type == TYPE_COUNT) {
      scanf("%d", &k);
      k = transform(k);
      process_count_op(l, r, k);
    } else {
      process_rotate_op(l, r);
    }
  }
}

int main() {
  read_data();
  init_buckets();
  process_ops();

  return 0;
}
```

A.4.6 Problema Time to Raid Cowavans (Codeforces)

‹ înapoi • versiune online

```
// Complexitate: O((q + n) sqrt n).
#include <algorithm>
#include <stdio.h>
```

```
const int MAX_N = 300'000;
const int MAX_Q = 300'000;
const int PREPROCESS_LIMIT = 547;

struct query {
  int id, first, step;
};

int w[MAX_N + 1];
long long prep[MAX_N + 1];
query q[MAX_Q];
long long answer[MAX_Q];
int n, num_queries;

void read_data() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d", &w[i]);
  }
  scanf("%d", &num_queries);
  for (int i = 0; i < num_queries; i++) {
    scanf("%d %d", &q[i].first, &q[i].step);
    q[i].id = i;
  }
}

void sort_queries() {
  std::sort(q, q + num_queries, [](query a, query b) {
    return a.step < b.step;
  });
}

long long naive_prog_sum(int first, int step) {
  long long sum = 0;
  for (int i = first; i <= n; i += step) {
    sum += w[i];
  }
  return sum;
}

void preprocess(int step) {
  for (int i = n; (i > n - step) && (i >= 1); i--) {
    prep[i] = w[i];
  }
  for (int i = n - step; i >= 1; i--) {
    prep[i] = w[i] + prep[i + step];
  }
}
```

```
void process_queries() {
  sort_queries();
  for (int i = 0; i < num_queries; i++) {
    if (q[i].step > PREPROCESS_LIMIT) {
      answer[q[i].id] = naive_prog_sum(q[i].first, q[i].step);
    } else {
      if (!i || (q[i].step != q[i - 1].step)) {
        preprocess(q[i].step);
      }
      answer[q[i].id] = prep[q[i].first];
    }
  }
}

void write_answers() {
  for (int i = 0; i < num_queries; i++) {
    printf("%lld\n", answer[i]);
  }
}

int main() {
  read_data();
  process_queries();
  write_answers();

  return 0;
}
```

A.4.7 Problema Sandor (Baraj ONI 2025)

‹ înapoi • versiune online

```
// Complexitate O(N sqrt N), provenită din 3 for-uri imbricate a cîte
// O(sqrt N) iterații.
#include <stdio.h>

const int MAX_N = 400'000;
const int MAX_WEIGHT = 800'000;

struct run {
  int val, cnt;
};

run r[MAX_N];
// jump[w] este obiectul maxim care nu depășește greutatea w. Mai exact,
// jump[w] este cel mai mic i a.î r[i].val ≤ w
int jump[MAX_WEIGHT + 1];
long long sol[MAX_WEIGHT + 1];
int task, n, num_runs, capacity;
```

```
int min(int x, int y) {
  return (x < y) ? x : y;
}

void read_data() {
  FILE* f = fopen("sandor.in", "r");
  int x;

  fscanf(f, "%d %d %d", &task, &n, &capacity);
  for (int i = 0; i < n; i++) {
    fscanf(f, "%d", &x);
    if (num_runs && (x == r[num_runs - 1].val)) {
      r[num_runs - 1].cnt++;
    } else {
      r[num_runs++] = { x, 1 };
    }
  }

  fclose(f);
}

void compute_jumps() {
  int i = 0;
  for (int w = MAX_WEIGHT; w >= 0; w--) {
    while ((i < num_runs) && (r[i].val > w)) {
      i++;
    }
    jump[w] = i;
  }
}

// Rulează algoritmul lui Sandor pentru capacitatea dată. Returnează greutatea
// acumulată.
int do_the_sandor(int start, int capacity) {
  int c = capacity;
  int i = start;

  while (i < num_runs) {
    int taken = min(c / r[i].val, r[i].cnt);
    c -= taken * r[i].val;
    // Dacă din c = 100 am luat 2 * r[i].val = 20, fiindcă atîtea erau, nu are
    // sens să continui de la 80. Următorul număr poate fi doar 9 sau mai mic.
    i = jump[min(c, r[i].val - 1)];
  }

  return capacity - c;
}

// Presupunînd că am tăiat deja două obiecte înainte de start și am obținut
```

```
// greutatea weight_so_far, rulează Sandor simplu pe restul vectorului și
// adaugă rezultatul la soluție.
void cut_0(int start, int weight_so_far, long long multiplier) {
  int s = do_the_sandor(start, capacity - weight_so_far);
  sol[weight_so_far + s] += multiplier;
}

// Presupunînd că am tăiat deja un obiect înainte de start și am obținut
// greutatea weight_so_far, și că de la poziția start începînd mai am num_num
// obiecte, încearcă să elimini cîte un obiect în toate modurile posibile.
//
// Observăm că, dacă tăiem alt obiect decît cele pe care le-ar pune Sandor în
// rucsac, vom obține aceeași sumă ca și pe vectorul nemodificat.
void cut_1(int start, int weight_so_far, int num_num, long long multiplier) {
  int c = capacity - weight_so_far;
  int i = start;

  while (i < num_runs) {
    int taken = min(c / r[i].val, r[i].cnt);
    if (taken == r[i].cnt) {
      num_num -= taken;
      int all_but_one = (taken - 1) * r[i].val;
      cut_0(i + 1, weight_so_far + all_but_one, multiplier * taken);
    }
    weight_so_far += taken * r[i].val;
    c -= taken * r[i].val;
    i = jump[min(c, r[i].val - 1)];
  }

  // Toate numerele pe care nu le-am tăiat explicit merg pe soluția originală,
  // care duce la sumă weight_so_far.
  sol[weight_so_far] += multiplier * num_num;
}

// Pentru bucla exterioară este important să obținem tot complexitate
// O(sqrt N). De aceea, considerăm simultan fiecare grupă de obiecte
// consecutive NEincluse în rucsac.
void cut_2() {
  long long multiplier = 0;
  int weight_so_far = 0;
  int c = capacity;
  int num_right = n;

  for (int i = 0; i < num_runs; i++) {
    if (r[i].val > c) {
      multiplier += r[i].cnt;
    } else {
      int cnt = r[i].cnt;
      // Taie două obiecte dinainte de i. Rămîn 0 de tăiat.
      cut_0(i, weight_so_far, multiplier * (multiplier - 1) / 2);
```

```
      // Taie un obiect dinainte de i. Rămîne unul de tăiat.
      cut_1(i, weight_so_far, num_right, multiplier);

      // Taie două obiecte din grupa i. În rest, pune în rucsac ce se poate.
      if (r[i].cnt >= 2) {
        int taken = min(c / r[i].val, r[i].cnt - 2);
        cut_0(i + 1, weight_so_far + taken * r[i].val, cnt * (cnt - 1) / 2);
      }

      // Taie un obiect din grupa i. În rest, pune în rucsac ce se poate.
      int taken = min(c / r[i].val, r[i].cnt - 1);
      cut_1(i + 1, weight_so_far + taken * r[i].val, num_right - r[i].cnt, cnt);

      // Nu tăia nimic, bagă în rucsac.
      taken = min(c / r[i].val, r[i].cnt);
      c -= taken * r[i].val;
      weight_so_far += taken * r[i].val;

      multiplier = 0;
    }

    num_right -= r[i].cnt;
  }

  // Nu mai putem băga nimic în rucsac, dar din ultimele @multiplier obiecte
  // trebuie să tăiem două.
  sol[weight_so_far] += multiplier * (multiplier - 1) / 2;
}

void write_solution() {
  FILE* f = fopen("sandor.out", "w");
  for (int i = 0; i <= capacity; i++) {
    fprintf(f, "%lld ", sol[i]);
  }
  fprintf(f, "\n");
  fclose(f);
}

int main() {
  read_data();
  compute_jumps();

  if (task == 1) {
    cut_1(0, 0, n, 1);
  } else {
    cut_2();
  }

  write_solution();
```

```
  return 0;
}
```

A.4.8 Problema Puzzle-bila (Lot 2025)

‹ înapoi • versiune online

```
// Complexitate: O(n·m·sqrt(m) + n·m·log(m)).
#include <stdio.h>

const int MAX_COLS = 50'000;
const int MAX_LOG = 16;
const int MAX_DISTINCT_LENGTHS = 320;
const int INFTY = 2'000'000;
const int NONE = -1;

int min(int x, int y) {
  return (x < y) ? x : y;
}

int log2(int x) {
  return 31 - __builtin_clz(x);
}

struct sparse_table {
  int v[MAX_LOG][MAX_COLS];
  int n;

  void build(int* src, int n, bool with_shifts) {
    for (int i = 0; i < n; i++) {
      v[0][i] = with_shifts ? (src[i] - i) : src[i];
    }
    for (int p = 1; (1 << p) <= n; p++) {
      for (int i = 0; i <= n - (1 << p); i++) {
        v[p][i] = min(v[p - 1][i], v[p - 1][i + (1 << (p - 1))]);
      }
    }
  }

  int range_min(int l, int r) {
    l = (l < 0) ? 0 : l;
    if (l > r) {
      return INFTY;
    }
    int row = log2(r - l + 1);
    return  min(v[row][l], v[row][r - (1 << row) + 1]);
  }
};
```

```
sparse_table st, st_shift;
bool row[MAX_COLS];
int prev1[MAX_COLS];
int distinct_len[MAX_DISTINCT_LENGTHS], num_lengths;
int dp[MAX_COLS];
int end[MAX_COLS + 1]; // coloana pe care se termină ultima fereastră de lățime l
int num_rows, num_cols;

void read_size() {
  scanf("%d %d ", &num_rows, &num_cols);
}

void read_row() {
  for (int c = 0; c < num_cols; c++) {
    row[c] = getchar() - '0';
  }
  getchar();
}

void init_all() {
  dp[0] = 0;
  for (int c = 1; c < num_cols; c++) {
    dp[c] = INFTY;
  }

  for (int l = 0; l <= num_cols; l++) {
    end[l] = NONE;
  }
}

void compute_prev1() {
  prev1[0] = row[0] ? 0 : -1;
  for (int c = 1; c < num_cols; c++) {
    prev1[c] = row[c] ? c : prev1[c - 1];
  }
}

void add_length(int len, int end_col) {
  if (len && (end[len] == NONE)) {
    distinct_len[num_lengths++] = len;
  }
  end[len] = end_col;
}

void reset_distinct_lengths() {
  while (num_lengths) {
    end[distinct_len[--num_lengths]] = NONE;
  }
}
```

```
// Adu ferestre de zerouri din stînga, deja închise, aliniindu-le cu col.
void slide_windows_right(int col) {
  dp[col] = INFTY;
  for (int i = 0; i < num_lengths; i++) {
    int len = distinct_len[i];
    int cost = st.range_min(col - len + 1, col) + (col - end[len]);
    dp[col] = min(dp[col], cost);
  }
}

// Adu ferestre de zerouri din dreapta, deja închise, aliniindu-le cu col.
void slide_windows_left(int col) {
  for (int i = 0; i < num_lengths; i++) {
    int len = distinct_len[i];
    int cost = st_shift.range_min(col - len + 1, col) + end[len] - len + 1;
    dp[col] = min(dp[col], cost);
  }
}

void slide_current_window(int col, int r_len) {
  if (!row[col]) {
    int l = 1 + prev1[col];
    int r = col + r_len - 1;
    int len = r - l + 1;
    dp[col] = min(dp[col], st.range_min(l, col));
    int cost_r = st_shift.range_min(col - len + 1, l - 1) + l;
    dp[col] = min(dp[col], cost_r);
  }
}

void scan_left_to_right() {
  reset_distinct_lengths();

  int cur_len = 0;
  for (int c = 0; c < num_cols; c++) {
    if (row[c]) {
      add_length(cur_len, c - 1);
      cur_len = 0;
    } else {
      cur_len++;
    }
    slide_windows_right(c);
  }
}

void scan_right_to_left() {
  reset_distinct_lengths();

  int cur_len = 0;
```

```
  for (int c = num_cols - 1; c >= 0; c--) {
    if (row[c]) {
      add_length(cur_len, c + cur_len);
      cur_len = 0;
    } else {
      cur_len++;
    }
    slide_windows_left(c);
    slide_current_window(c, cur_len);
  }
}

void process_rows() {
  init_all();
  for (int r = 0; r < num_rows; r++) {
    read_row();
    compute_prev1();
    st.build(dp, num_cols, false);
    st_shift.build(dp, num_cols, true);
    scan_left_to_right();
    scan_right_to_left();
  }
}

void write_result() {
  int res = dp[num_cols - 1];
  printf("%d\n", (res == INFTY) ? NONE : res);
}

int main() {
  read_size();
  process_rows();
  write_result();

  return 0;
}
```