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.

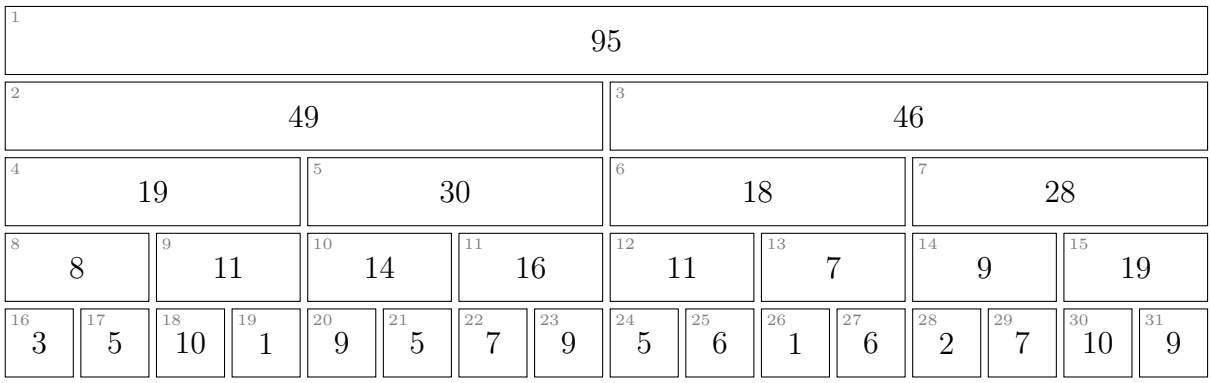

Figura 1.1: Un arbore de intervale cu 16 frunze și 15 noduri interne. Valorile din fiecare celulă reprezintă suma din frunzele subîntinse de acea celulă. Cu cifre mici este notat indicele fiecărei celule.

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.

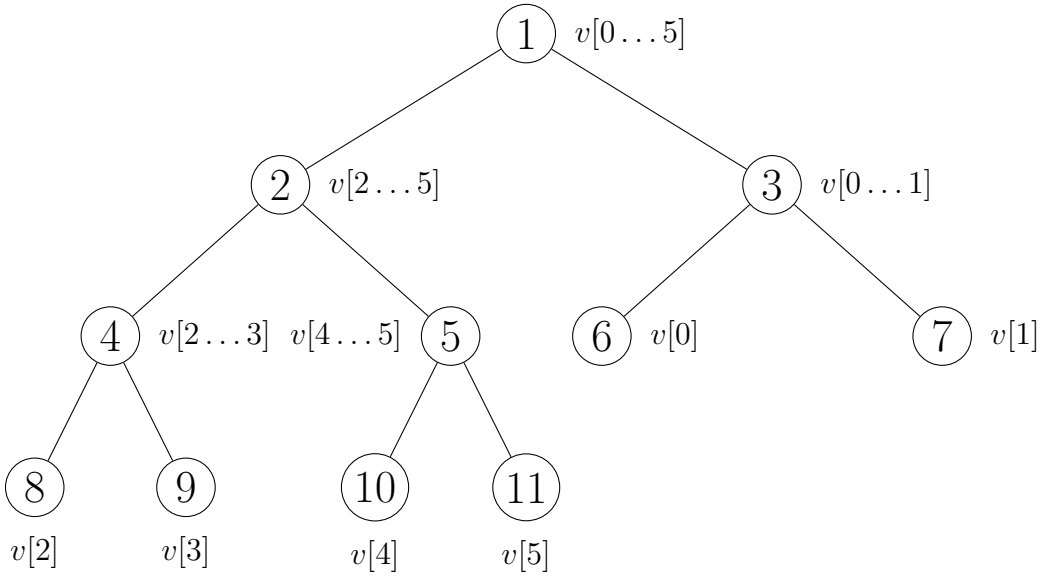

Figura 1.2: Reprezentarea arborilor de intervale cu exact $2n-1$ noduri.

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 modificare 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.

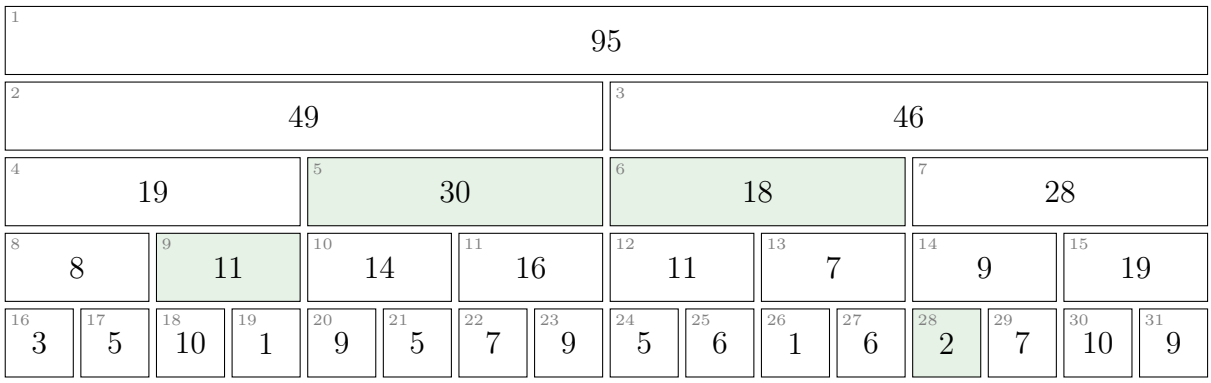

Figura 1.3: Suma intervalului [18, 28] este egală cu suma valorilor nodurilor 9, 5, 6 și 28.

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 modificări 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 modificare 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.

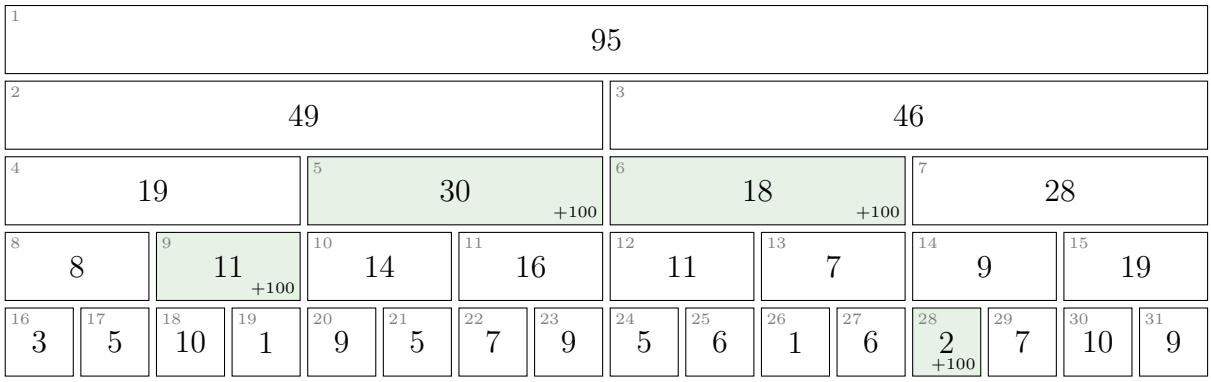

Figura 2.1: Pentru a adăuga 100 pe intervalul [18, 28], notăm valoarea *lazy* 100 pe nodurile 9, 5, 6 și 28.

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):

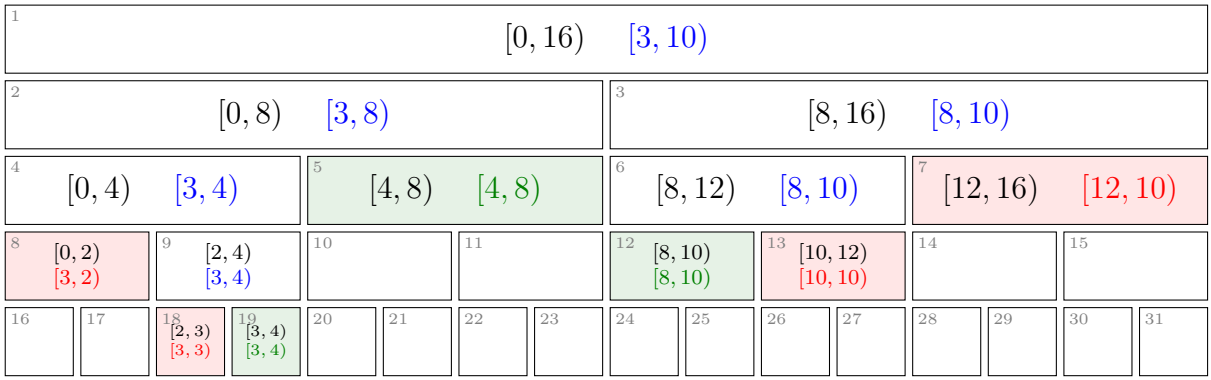

Figura 2.2: Arborele de apeluri în funcția de actualizare recursivă pe intervalul $[3, 10)$.

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 Probleme

2.4.1 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.

2.4.2 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).

2.4.3 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.

2.4.4 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 Modificări 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.

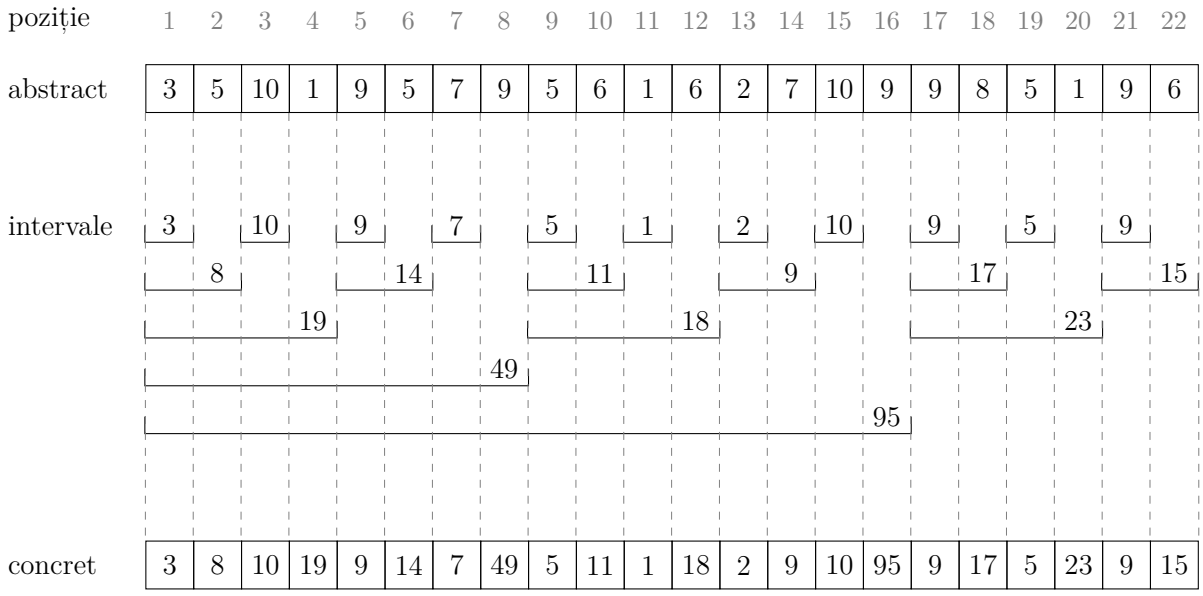

Figura 3.1: Un arbore indexat binar cu 22 de poziții. Vectorul de sus este cel abstract, iar vectorul de jos este cel stocat concret în memorie. Fiecare interval arată pozițiile a căror sumă o notăm în capătul din dreapta al intervalului.

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 modificarea 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 același 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 modificare, 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.

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).

```
// Complexity: 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];
// This belongs in the struct, but that would require sorting the queries once
// more at the end.
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).

```
// Complexity: 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];
// This belongs in the struct, but that would require sorting the queries once
// more at the end.
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

```
// Complexity: 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

```
// Complexity: 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++; // infinite sentinel at the end
    n = next_power_of_2(n);
    this->n = n;

    for (int i = 1; i < 2 * n; i++) {
      v[i] = -1; // no points
    }

    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]);
    }
  }

  // Returns the first position after pos where the value is greater than val.
  int find_first_after(int pos, int val) {
    pos += n;
    pos++;

    // Climb until we encounter a maximum greater than val.
    while (v[pos] <= val) {
      if (pos & 1) {
        pos++;
      } else {
        pos >>= 1;
      }
    }

    // Descend and zoom in on the desired value, keeping to the left when we
    // can.
    while (pos < n) {
      pos = (v[2 * pos] > val) ? (2 * pos) : (2 * pos + 1);
    }
```

```
    return pos - n;
  }

};

query q[MAX_N];
int pos[MAX_N]; // used for normalization
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 poziții 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).

```
// Complexity: O(q log n)
//
// Method: maintain a segment tree that supports range sums and adding an
// arithmetic progression to a range.
#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;
}

// Invariants:
//
// 1. The real values of spanned nodes are their respective v values plus an
//    arithmetic progression defined by first and step.
// 2. A node's v does not include its own progression.
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)  {
    // The first leaf spanned by node x is 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).

```
// Complexity: O(q log n)
//
// Method: maintain a segment tree that supports range sums and adding
// an arithmetic progression to a range.
#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;
}

// Invariants:
//
// 1. first[k] and step[k] mean that the real values of spanned nodes are the
//    values in v[] plus an arithmetic progression with the given parameters.
// 2. v[k] includes first[k] and step[k], but not values from ancestors.
// 3. All intervals are [closed, open).
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 = position where the 1 is written
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

```
// Complexity: 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));
}

// Segment tree with 0/1 values, range set and range sum operations.
//
// Contract: state[x] can be:
// * 0/1 to indicate a value to be set on all of x's descendants, or
// * ST_CLEAN to indicate that the value has been pushed.
//
// sum[node] takes into account 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).

```
// v3: Unifică range_max() și inc(), care sînt apelate consecutiv.
#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;
}
```