Arbori de intervale (AINT)

Arborii de intervale sunt o structură de date (o modalitate de a organiza informații cu scopul de a realiza mai rapid un anumit tip de calcule).

I) Problema inițială

Pornim de la următorul enunț:
Se dă un tablou unidimensional (vector) `V` cu `n` numere naturale, asupra căruia se pot face două tipuri de operații:

- interogare pentru minim pe o secvență de elemente;
- actualizarea unui element (schimbarea valorii sale);

Să considerăm că vectorul poate avea poate avea cel mult `100000` de elemente și că sunt cel mult `100000` de interogări. De asemenea, valorile sale sunt de tip `int`.
În această problemă actualizările și interogările nu sunt separate, ele pot apărea amestecat și la fiecare interogare răspundem considerând configurația vectorului din acel moment.

Exemplu:

```
11
1 6 4 3 9 2 7 8 0 5 4
3
1 2 4
2 3 2
1 2 4
```

Semnificația datelor de intrare de mai sus: vectorul are `11` elemente, acestea sunt inițial `1 6 4 3 9 2 7 8 0 5 4` (să considerăm indexarea de la `1`). Sunt `3` operații. Mai întâi o interogare (simbolizată prin prezența valorii `1` prima pe linia ce o descrie), care ne cere să aflăm minimul din intervalul de indici de la `2` la `4`. Acesta are valoarea `3`. A doua operație este o actualizare (linia începe cu `2`). Adică elementul de pe poziția `3` devine `2`. Efectul este că vectorul va fi: `1 6 2 3 9`. A treia operație este iarăși o interogare pe intervalul de indici de la `2` la `4`. Acum minimul este `2`. Așadar ieșirea acestui program ar trebui să fie `3 2`.

Rezolvarea în mod brut, adică simularea a ceea ce este descris în enunț, presupune actualizarea în mod direct (operația de tip `2 p x` se rezolvă printr-o singură atribuire, `v[p] = x;`). În schimb, operația de tip interogare de minim pe intervalul de indici `[a, b]`, simbolizată prin `1 a b` presupune realizarea unei parcurgeri de la `a` la `b`, iar acesta, pe cazul cel mai defavorabil tinde să aibă spre `n` pași. Această metodă face actualizarea în timp constant iar interogarea în timp liniar. Așadar, dacă numărul de interogări este mare, iar acestea sunt mereu pe intervale mari, timpul de calcul ajunge să fie de ordin `n*m`.

Dacă interogările ar fi toate la final, fără ca printre ele să mai apară actualizări, se poate folosi structura de date numită *RMQ (Range Minimum Query)* pentru a face precalculări de minime pe anumite intervale și apoi se folosesc aceste minime pentru a răspunde în timp constant la o anumită interogare. Precalcularea elementelor structurii necesită timp și memorie de ordin $n \log_2 n$. Din păcate actualizarea sa la modificarea valorii unui element nu se mai face în timp logaritmic și astfel folosirea pentru cazul în care avem actualizări și interogări amestecate devine neeficientă.

Să revenim la problema enunțată inițial. Anunțăm de pe acum ce vom obține la final, urmând să arătăm apoi pașii :

- vom calcula (și stoca) minime din anumite intervale ale vectorului V (intervale stabilite clar de la început);
- intervalele pentru care avem minimele calculate vor fi suficiente ca pe baza lor să determinăm minimul din oricare alt interval. Mai mult, pentru un interval oarecare dat, numărul de intervale necesare, dintre cele cu rezultatul stocat, este de ordin $\log_2 n$;
- la actualizarea unui element din vectorul V numărul de intervale care trebuie actualizate este de ordin $\log_2 n$;

Astfel, vom obține timp de ordin logaritmic pe fiecare dintre dele două operații.

Mai întâi să vedem cum alegem setul de intervale fixat, pentru care calculăm minimele.
În primul rând observăm că numărul total de intervale este de ordin n^2 (ne imaginăm că împerechem fiecare număr de la 1 la n, reprezentând capătul stâng al unui interval cu fiecare număr mai mare decât el, reprezentând capătul drept).

Pentru datele noastre, este nepractic să calculăm informații pentru toate aceste intervale (atât ca timp dar și ca memorie avem n^2, iar pentru $n = 100000$ asta înseamnă foarte mult).

Pentru a găsi modalitatea de obținere a setului de intervale care ne interesează pornim de la următoarea funcție recursivă:

```
void rec(int st, int dr) {
    if (st == dr) {
        /// operație directă pe intervalul format
        /// cu un singur element (v[st])
    } else {
        /// intervalul de indici de la st la dr este
        /// împărțit în două intervale, folosind
        /// un indice de la mijloc, iar apoi facem
        /// autoapel în cele două intervale

        int mid = (st+dr)/2;
        rec(st, mid);
        rec(mid+1, dr);
    }
}
```

Apel inițial `rec(1, n);`

Pentru codul de mai sus avem:

- `n` = numarul de elemente ale vectorului dat (indexat, cum spuneam, de la `1`);
- parametrii funcției recursive au semnificația de indici din vectorul dat;

De exemplu, pentru `n = 11`, apelul de mai sus generează următoatele perechi de indici `(st, dr)`:

```
(1, 11), (1, 6), (1, 3), (1, 2), (1, 1), (2, 2), (3, 3), (4, 6), (4, 5), (4, 4), (5, 5), (6, 6), (7, 11), (7, 9), (7, 8), (7, 7), (8, 8), (9, 9), (10, 11), (10, 10), (11, 11)
```

Citindu-le pur și simplu ne este greu să găsim repede o semnificație a acestor numere, dar să vedem mai departe câteva moduri de a le imagina:

<table>
<tr><td colspan="16">1, 11</td></tr>
<tr><td colspan="8">1, 6</td><td colspan="8">7, 11</td></tr>
<tr><td colspan="4">1, 3</td><td colspan="4">4, 6</td><td colspan="4">7, 9</td><td colspan="4">10, 11</td></tr>
<tr><td colspan="3">1, 2</td><td>3,3</td><td colspan="3">4, 5</td><td>6, 6</td><td colspan="3">7, 8</td><td>9, 9</td><td>10,10</td><td>11,11</td></tr>
<tr><td>1, 1</td><td>2, 2</td><td></td><td></td><td>4, 4</td><td>5, 5</td><td></td><td></td><td>7, 7</td><td>8, 8</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Pentru `n=11` acestea vor fi intervalele pentru care vom ține în orice moment minimele (adică, după fiecare operație vom avea stocată valoare minimă din fiecare dintre aceste intervale). Va trebui să determinăm câte astfel de intervale sunt, cum actualizăm aceste intervale în timp logaritmic și cum putem determina minimul din oricare alt interval folosind doar dintre acestea, în număr de operații de ordin logaritmic. Bineînțeles, exemplul nostru este pentru `n=11`, dar descompunerea este valabilă în general, folosind template-ul de funcție recursivă de mai sus, împreună cu apelul inițial `rec(1, n)`.

O primă observație este că numărul de linii ale tabelului de mai sus este de ordin `log2n`. Justificare: Dacă intervalele de pe un rând au lungimea `L`, cele de pe rândul următor au lungimea `L/2` sau `L/2+1` (am semnificar prin `/` câtul împărțirii întregi). Dar numărul de înjumătățiri după care se ajunge la întervale de lungime `1` (celule de jos, care nu mai pot fi împărțite) este de ordin logaritmic.

Să vedem cum păstrăm informații despre aceste intervale. Pentru aceasta, vom mai introduce un parametru la funcția `rec`. Îl notăm cu `nod` și alegem să îl scriem primul (poziția lui între

parametrii funcției recursive nu este importantă, însă încercăm să păstrăm o ordine întâlnită în multe materiale despre arborii de intervale). Avem așadar:

```
void rec(int nod, int st, int dr) {
    if (st == dr) {
        /// operație directă pe intervalul format
        /// cu un singur element (v[st])
    } else {
        /// intervalul de indici de la st la dr este împărțit
        /// în două intervale, folosind un indice de la mijloc,
        /// iar apoi facem autoapel în acestea

        int mid = (st+dr)/2;
        rec(2*nod, st, mid);
        rec(2*nod+1, mid+1, dr);
    }
}
```

Apel inițial: `rec(1, 1, n)`

Urmăriți modul în în care realizăm autoapelurile.

Mai departe să vedem cum arată celulele din tabel, cu noul parametru.

<table>
<tr><td colspan="16">1
1,11</td></tr>
<tr><td colspan="8">2
1,6</td><td colspan="8">3
7,11</td></tr>
<tr><td colspan="4">4
1,3</td><td colspan="4">5
4,6</td><td colspan="4">6
7,9</td><td colspan="4">7
10,11</td></tr>
<tr><td colspan="3">8
1,2</td><td>9
3,3</td><td colspan="3">10
4,5</td><td>11
6,6</td><td colspan="3">12
7,8</td><td>13
9,9</td><td>14
10,10</td><td>15
11,11</td></tr>
<tr><td>16
1,1</td><td>17
2,2</td><td></td><td></td><td>20
4,4</td><td>21
5,5</td><td></td><td></td><td>24
7,7</td><td>25
8,8</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Valorile lui nod sunt figurate deasupra, boldat. Așadar, în fiecare celulă avem un triplet (`nod`, `st`, `dr`) unde *`[st, dr]` reprezintă un interval de indici din vectoriul `V`*.
Dar ce semnificație poate avea `nod`?
Citind de sus în jos și de la stânga la dreapta, valorile sale sunt consecutive începând cu `1`, excepție făcând câteva aflate mai la final, care lipsesc.
Acum să estimăm numărul de intervale (celule) care apar în tabel. Pentru asta, să considerăm că `n` este o putere de `2`. Pentru simplitate luăm `n=8`. Tabelul nostru va fi:

<table>
<tr><td colspan="8">1
1,8</td></tr>
<tr><td colspan="4">2
1,4</td><td colspan="4">3
5,8</td></tr>
<tr><td colspan="2">4
1,2</td><td colspan="2">5
3,4</td><td colspan="2">6
5,6</td><td colspan="2">7
7,8</td></tr>
<tr><td>8
1,1</td><td>9
2,2</td><td>10
3,3</td><td>11
4,4</td><td>12
5,5</td><td>13
6,6</td><td>14
7,7</td><td>15
8,8</td></tr>
</table>

Acum se observă ușor:

- valorile nod sunt toate numerele de la `1` la `2*n-1` (deci tot acesta este numarul de intervale care ne intereseaza);
- înălțimea este `1+log2n`;
- dacă `n` nu ar fi putere de `2`, ne dăm seama ușor că tabelul nu este mai înalt decât pentru cazul că am majora pe `n` la următoarea putere de `2`, iar numărul său de elemente este tot `2*n-1;`

În general, la tehnica noastră numărul de intervale care contează este `2*n-1`.
Valoarea `nod` o vom folosi ca indice în alt vector, pe care îl vom nota cu `A` și care este de fapt structura noastră de date.

Astfel, pentru un triplet (`nod`, `st`, `dr`) generat de funcția de mai sus, avem:
`A[nod]` = minimul elementelor din `V` aflate pe indici între `st` și `dr` incusiv.
Putem deci spune că la un triplet (`nod`, `st`, `dr`), `nod` este indice din `A` iar `st` si `dr` sunt indici din `V`.
Din cele analizate mai sus deducem că sunt importante `2*n-1` elemente din `A`, însă observăm că ele nu sunt plasate chiar pe toți indicii consecutiv de la `1` la `2*n-1` indiferent de valoarea lui `n`.
Gândindu-ne că la trecerea unui `n` peste o putere de `2` se merge până la a se dubla numărul de elemente (la următoarea putere de `2` fiecare frunză se expandează în alte două) deducem că `4*n` elemente pentru vectorul `A` este o limitare superioară suficientă. Nu face obiectul acestui articol, dar precizăm că există formule de calcul pentru indicii din `A`, altele decât cea folosită de noi (`mid = (st+dr)/2`) care permit reducerea semnificativă a memoriei.

Date fiind cele de mai sus, să trecem către implementare.
Primul pas ar fi să calculăm valorile inițiale din `A`. Adică minimele, conform elementelor date inițial în `V`, înainte de updateuri. Pentru asta, vom folosi următoarea funcție recursivă (o variantă a funcției `rec` pe care o vom numi `build`).

```
void build(int nod, int st, int dr) {
    if (st == dr) {
```

```
        A[nod] = V[st];
    } else {
        int mid = (st+dr)/2;
        build(2*nod, st, mid);
        build(2*nod+1, mid+1, dr);
        A[nod] = min(A[2*nod], A[2*nod+1]);
    }
}
```

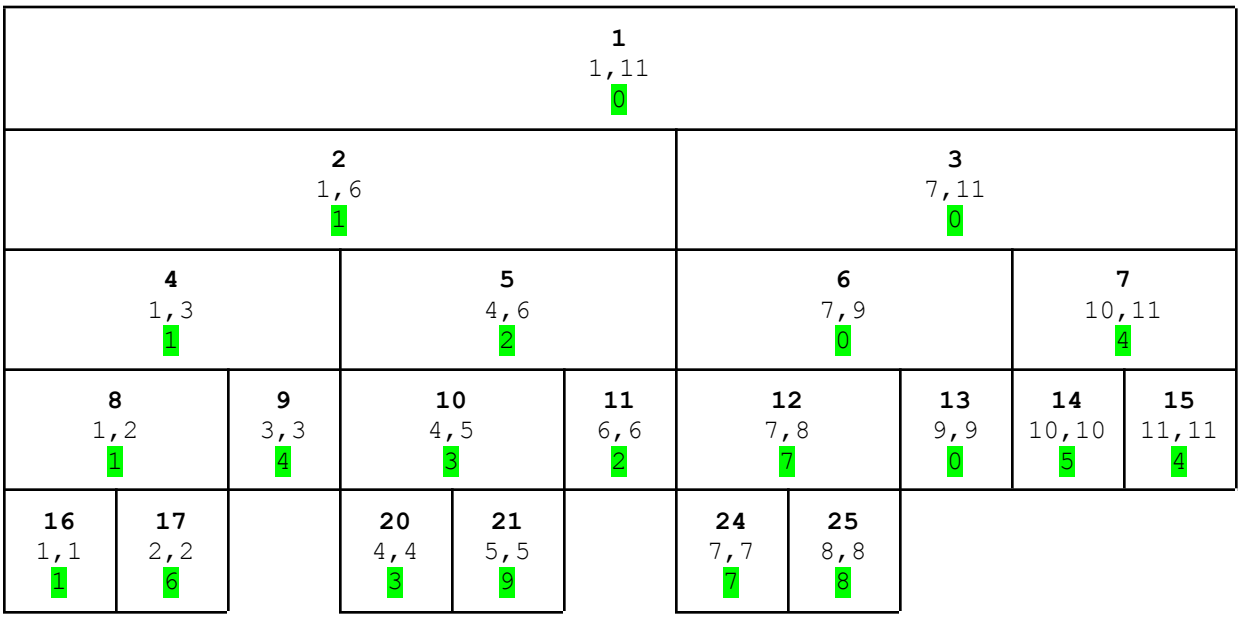
Apel: `build(1, n);`

Este un principiu ***divide et impera*** pe care îl putem interpreta așa:

- Dacă intervalul are lungimea `1`, valoarea din el este chiar cea din `V` de pe acea poziție.
- Altfel, cei doi fii sunt de fapt cele două jumătăți ale sale și valoarea din interval este minimul valorilor din cele două jumătăți.

Iată cum arată vectorul `A` dar și tabelul de mai sus în care acum completăm cu valorile calculate în noduri de apelul `build(1, 1, n)`.

Indici:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Valori din V:	1	6	4	3	9	2	7	8	0	5	4														
Valori din A:	0	1	0	1	2	0	4	1	4	3	2	7	0	5	4	1	6	X	X	3	9	X	X	7	8



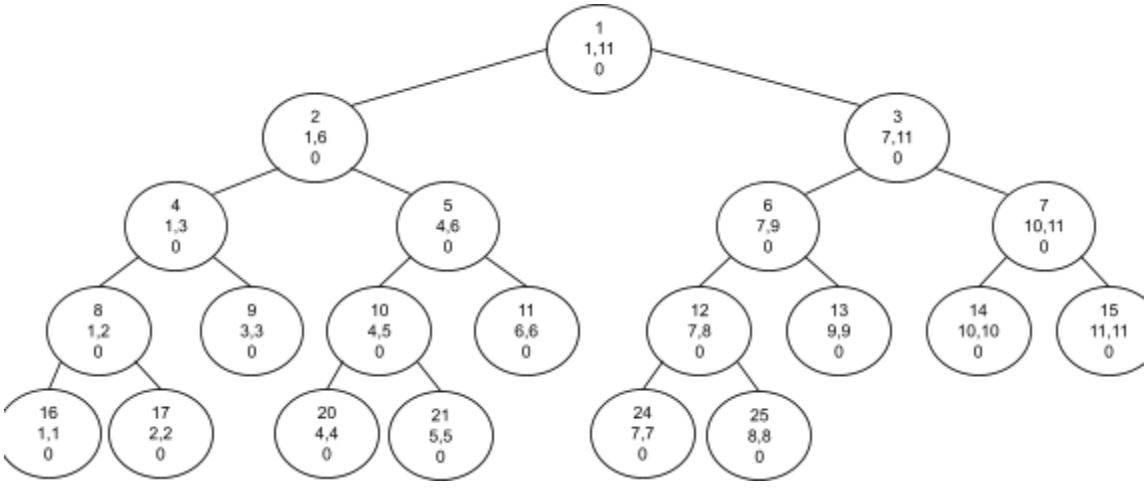
De remarcat că timpul de calcul pentru rularea acestui cod, `build(1, n)` este de ordin `n` (sunt `2*n-1` valori care se calculează). Asta ne poate duce puțin în eroare gândindu-ne că noi urmărim să obținem cod care rulează în timp logaritmic. Nu este nicio problemă întrucât apelul acesta se face o singură dată, nu la fiecare operație, iar el are aceeași complexitate în timp ca și citirea datelor.

Pornind de la observația că nodurile frunză (cele care corespund unor intervale de lungime `1`) se parcurg de la stânga la dreapta (acest lucru se întâmplă pentru că noi facem mai întâi autoapel în subintervalul stâng), și că valorile nodurilor se completează de jos în sus (deoarece instrucțiunea `A[nod] = min(A[2*nod], A[2*nod+1]);` se află după autoapeluri), putem face citirea datelor de intrare chiar în funcție, pe ramura fără autoapel, adică:

```
void build(int nod, int st, int dr) {
    if (st == dr) {
        fin>>A[nod];
    } else {
        int mid = (st+dr)/2;
        build(2*nod, st, mid);
        build(2*nod+1, mid+1, dr);
        A[nod] = min(A[2*nod], A[2*nod+1]);
    }
}
```

Am prezentat această variantă penrtu a ajuta la înțelegerea mai bună a modului de funcționare, dar nu o recomandăm, considerăm că este mai important să facem lucrurile separat, și pentru a le avea mai clare.

Pentru că tot am vorbit de frunze, noduri, dar și ținând cont și de la denumirea structurii (arbore de intervale), ne putem imagina și o reprezentare a datelor printr-un arbore binar (în care un nod este un interval iar cei doi fii ai săi sunt cele două jumătăți care îl compun). Astfel, fiecare celulă de tabel este un nod iar cei doi fii sunt cele două celule de sub ea, corespunzând celor două jumătăți în care se descompune intervalul pe care ea îl reprezintă.



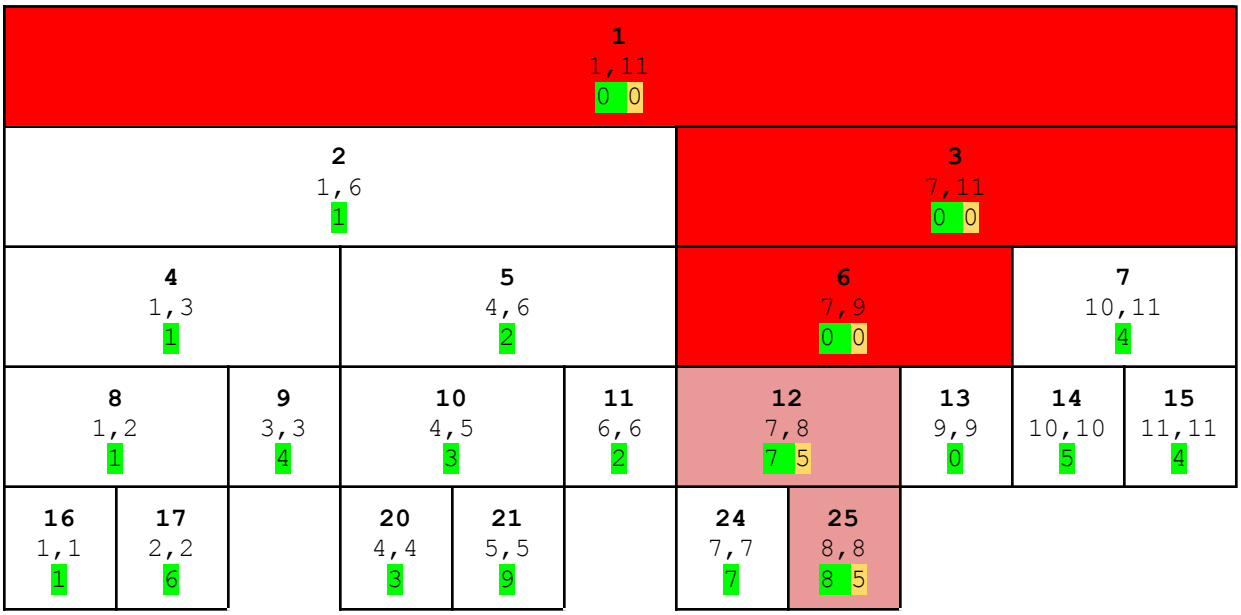
Operația de actualizare a unui element.

Noi am codificat-o prin `2 p x`, cu semnificația: elementul de pe poziția `p` din vectorul dat devine `x`. Să vedem ce modificări implică această operație în vectorul `A`.

- se va modifica valoarea frunzei corespunzătoare intervalului de lungime `1 [p, p];`

- pentru tot drumul de la această frunză până la rădăcină se vor calcula valorile din celula curentă în funcție de cele din cele două celule fiu (cele două jumătăți ale intervalului reprezentat de celula curentă) - care la revenire sunt deja calculate;

Figurăm cele explicate aici considerând exemplul inițial și actualizarea `V[8] = 5`; (adică valoarea de pe poziția `8`, în loc de `8`, va deveni `5`).



Celulele desenate cu background roșu sunt cele în care se recalculează valoarea (pentru unele ea se poate modifica iar pentru altele nu).

Pentru a realiza acestea, facem o altă funcție recursivă (o vom numi `update`), în care avem ca parametri, pe de o parte pe cei `3` care identifică tripletele (`nod`, `st`, `dr`) și încă doi specifici operației: `p` și `x`.
Reamintim că este esențial să generăm la toate funcțiile care operează pe arbore, aceleși triplete.

```
void update(int nod, int st, int dr, int p, int x) {
    if (st == dr) {
        A[nod] = x;
    } else {
        int mid = (st+dr)/2;
        if (p <= mid)
            update(2*nod, st, mid, p, x);
        if (p > mid)
            update(2*nod+1, mid+1, dr, p, x);
        A[nod] = min(A[2*nod], A[2*nod+1]);
    }
```

```
}
```

Apel: `update(1, 1, n, p, x)`

Câteva observații legate de codul funcției de mai sus:

- din nodul curent avansăm exact în unul dintre cei doi fii (cel care va conține poziția `p`);
- întrucât înălțimea arborelui este de ordin logaritmic iar noi coborâm un nivel la fiecare autoapel, ajungem la complexitatea dorită;
- cele două `if`-uri puteau fi scrise mai compact printr-un `if-else`, însă am preferat această variantă pentru a introduce un standard ce va fi folosit și la alte operații pe arbore; odată stăpânit modul în care funcționează structura, este la alegerea celui care codează să scrie cum i se pare mai potrivit;
- `p` și `x` se transmit cu aceeași valoare către autoapeluri dar am ales să îi punem parametri pentru claritate; dacă dorim, îi putem folosi ca variabile globale;

Operația de interogare

O codificăm așadar prin `1 a b` cu semnificația dată (să aflăm minimul dintre valorile: `V[a]`, `V[a+1]`, ..., `V[b]`).

De această dată vom scrie întâi codul care realizează interogarea și vom veni cu explicații asupra lui.

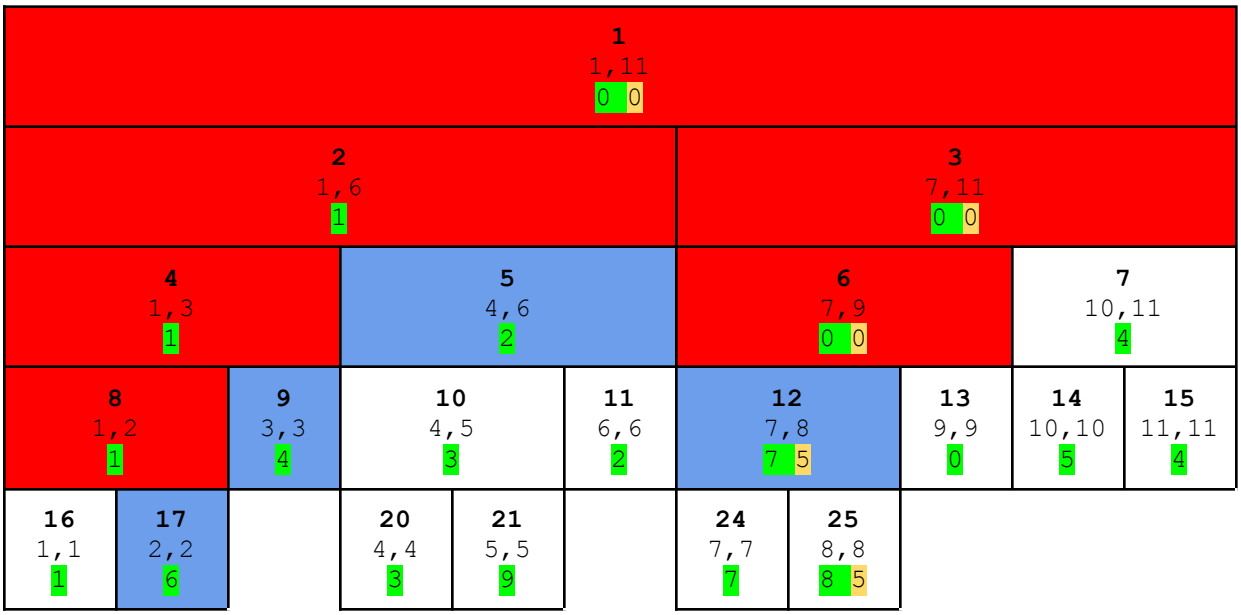
```
void query(int nod, int st, int dr, int a, int b) {
    if (a<=st && dr<=b) {
        sol = min(sol, A[nod]);
    } else {
        int mid = (st+dr)/2;
        if (a <= mid)
            query(2*nod, st, mid, a, b);
        if (b >= mid+1)
            query(2*nod+1, mid+1, dr, a, b);
    }
}
```

```
sol = -INF;
query(1, 1, n, a, b);
fout<<sol;
```

O explicație sumară a acestui cod ar fi următoarea: dacă pentru intervalul curent `[st, dr]`, intervalul de interogare `[a, b]` are elemente și într-o jumătate și în alta, facem autoapel în amândouă jumătățile, în caz contrar face, apel în jumătatea în care el conține elemente. Când ajungem la un interval `[st, dr]` complet inclus în `[a, b]`, comparăm cu minimul global (păstrat de noi în variabila `sol`) valoarea `A[nod]` (corespunzătoatr lui `[st, dr]`).

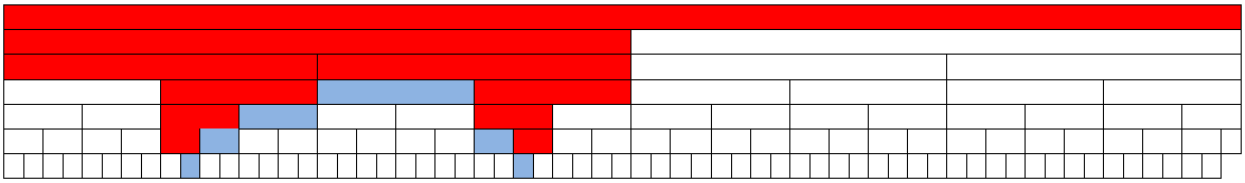
Această explicație sumară nu justifică nici eficiența (de exemplu, ne punem întrebarea, “dacă facem apel în ambii fii, nu putem ajunge la timp de ordin `n`, precum la `build` ?”) și nici nu ajută la înțelegerea temeinică a modului în care se face descompunerea (iar această bună înțelegere este esențială și pentru priceperea modului de funcționare a structurii atunci când avem actualizare pe interval de indici - să nu uităm că aici noi am făcut explicație doar pentru `update` pe un singur element).

Așadar, să ne concentrăm pe modul în care funcționează funcția `query`, scrisă mai sus.



Pe figura de mai sus sunt evidențiate (cu alt background decât alb) nodurile prin care se trece la un apel pentru intervalul `[a, b] = [2, 8]`.
Nodurile figurate cu roșu au intersecție nevidă cu intervalul `[a, b]` însă nu sunt complet incluse în el, așa că pentru ele nu se răspunde cu informația `A[nod]` ci se autoapelează în fii. Din unele dintre ele se fac două autoapeluri iar din altele doar unul, depinde de care dintre jumătăți au intersecție nevidă cu `[a, b]`.
De exemplu, când pornim din nodul de sus, cu apelul inițial, adică `[st, dr] = [1, 11]`, se fac autoapeluri în ambii fii, iar când ajungem în `[7, 11]`, se mai face autoapel doar în fiul stâng, `[7, 9]`, pentru că doar el are intersecție nevidă cu intervalul `[a, b]`.
Cu albastru am figurat intervalele complet incluse în `[a, b]`. Deci, când se ajunge cu autoapel în ele se returnează valoarea din `A` corespunzătoate fără să se mai facă autoapeluri. Se observă pe figură că, dacă pe acestea albastre le concatenăm de la stânga la dreapta, obținem chiar intervalul `[a, b]`.
Acum ne punem problema: cât de multe pot fi intervalele prin care trece un autoapel (adică în total cele roșii și cele albastre) ? De numărul lor depinde timpul de executare, iar noi ne dorim ca el să fie de ordin logaritmic.

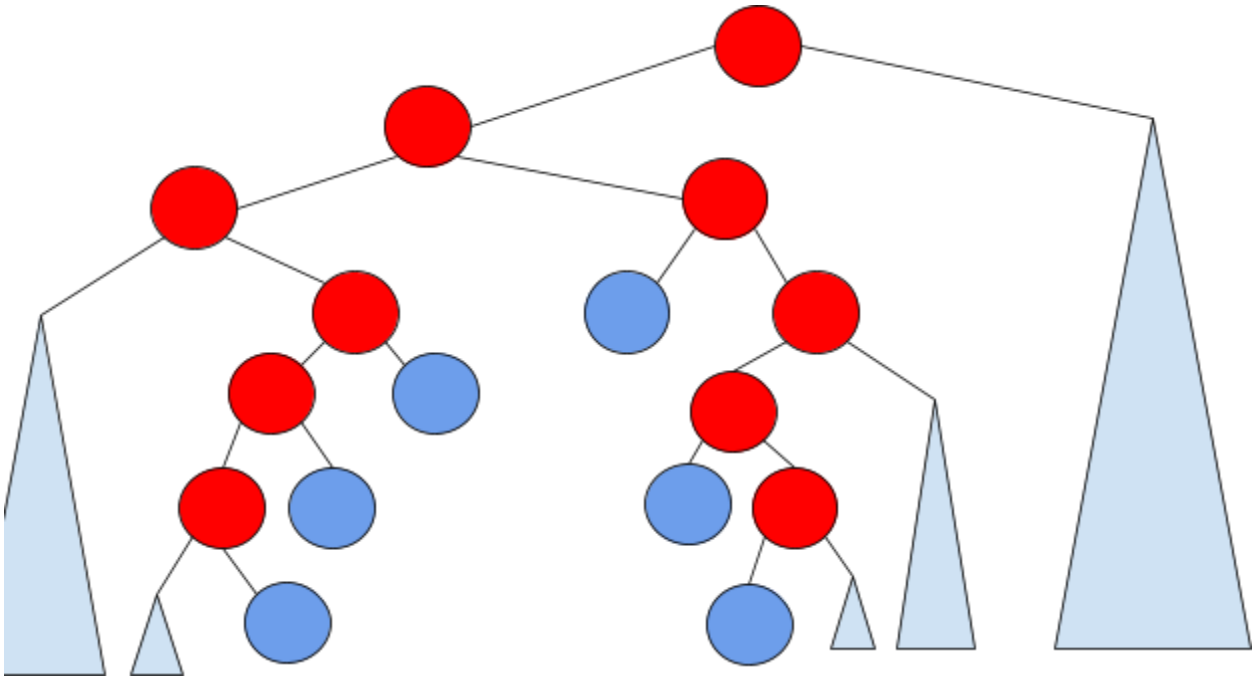
Din această figură pare că numărul de intervale implicate este mare raportat la numărul total de intervale. Dar asta pentru că avem un n mic iar intervalele aflate mai sus sunt desenate “mai late”. Dar să vedem o figură mai în ansamblu.



Este esențial de observat că sub intervalele colorate cu albastru nu mai avem culori. Semnificația este că nu se mai ajunge cu autoapelul în intervale de sub acestea. **Practic, pentru fiecare valoare din intervalul de query se ajunge doar până la cel mai înalt nod din AINT care o conține și care este complet inclus în intervalul de query `[a, b]`.** Reținem în mod special această observație, esențială în înțelegerea situației cu update pe interval, despre care am mai amintit și pe care o vom explica ulterior (tehnica numită “**lazy update**”).

Observăm că este o singură linie (nivel) pe care se află două dreptunghiuri roșii. Celelalte linii pe care sunt două dreptunghiuri colorate au proprietatea că unul dintre ele esle colorat cu albastru, adică cel dinspre mijloc. Ce se întâmplă de fapt ? Odată făcută ramificarea pe un nivel, pe nivelele de mai jos în care se mai fac două autoapeluri, se întâmplă că unul dintre cele două (cel dinspre mijloc, adică din dreapta daca am mers la ramificare pe ramura din stânga, și cel din stânga, dacă am mers la ramificare pe ramura din dreapta) reprezintă un interval al `AINT`-ului complet inclus în `[a, b]`, deci din el se va răspunde direct cu valoarea (`A[nod]`) fără a se mai face autoapeluri.

Dacă am figura mai contractat nodurile, am observa și mai bine ce se întâmplă de fapt.



Nu e obligatoriu ca ramificarea să se facă de pe primul nivel, dar de acolo unde se face, se va merge în jos pe două ramuri, iar pe fiecare dintre acestea va fi cel mult un nod în care se mai fac două autoapeluri și în plus, în fiecare astfel de caz, unul dintre autoapeluri returnează direct valoarea.
Astfel, de pe fiecare nivel se pot vizita maxim 4 noduri. Pe de altă parte, numărul de niveluri este maxim $1 + \log_2 n$. Obținem timp de calcul logaritmic.
Prezentăm mai jos o sursă care rezolvă problema enunțată la început.
Un enunț complet al acesteia se află aici:
https://www.pbinfo.ro/probleme/2090/actualizare-element-minim-interval

```
#include <fstream>
#include <climits>
#define DIM 100010
#define INF INT_MAX

using namespace std;

ifstream fin ("aemi.in");
ofstream fout("aemi.out");

int A[4*DIM];
int n, m, a, b, t, i, minim;

void build(int nod, int st, int dr) {
    if (st == dr) {
        fin>>A[nod];
```

```
    } else {
        int mid = (st + dr)/2;
        build(2*nod, st, mid);
        build(2*nod+1, mid+1, dr);
        A[nod] = min(A[2*nod], A[2*nod+1]);
    }
}

void query(int nod, int st, int dr, int a, int b) {
    if (a <= st && dr <= b) {
        minim = min(minim, A[nod]);
    } else {
        int mid = (st + dr)/2;
        if (a <= mid)
            query(2*nod, st, mid, a, b);
        if (b > mid)
            query(2*nod+1, mid+1, dr, a, b);
    }
}

void update(int nod, int st, int dr, int a, int b) {
    if (st == dr) {
        A[nod] = b;
    } else {
        int mid = (st + dr)/2;
        if (a <= mid)
            update(2*nod, st, mid, a, b);
        if (a > mid)
            update(2*nod+1, mid+1, dr, a, b);
        A[nod] = min(A[2*nod], A[2*nod+1]);
    }
}

int main () {

    fin>>n;
    build(1, 1, n);
    fin>>m;
    for (int i=1;i<=m;i++) {
        fin>>t>>a>>b;
        if (t == 1) {
            minim = INF;
            query(1, 1, n, a, b);
            fout<<minim<<"\n";
        } else {
            update(1, 1, n, a, b);
        }
    }
    return 0;
}
```

II) Folosirea arborilor de intervale atunci când avem update pe un interval cu mai mult de un element (tehnica “lazy update”).

La problema anterioară, chiar dacă interogarea se poate face pe un interval, operația de actualizare a fost dată pentru un singur element.
Acum rezolvăm problema în care operația de actualizare este de forma: `2 a b x`, cu semnificația: “toate elementele aflate în `V` pe poziții de la `a` la `b` inclusiv primesc valoarea `x`”.

Pornind de la ce avem deja, am putea folosi funcția update (scrisă mai sus) pentru fiecare element din intervalul `[a, b]`.

```
for (i=a;i<=b;i++)
    update(1, 1, n, i, x);
```

Avem astfel $n*\log_2 n$ pe update, dar noi am promis $\log_2 n$.

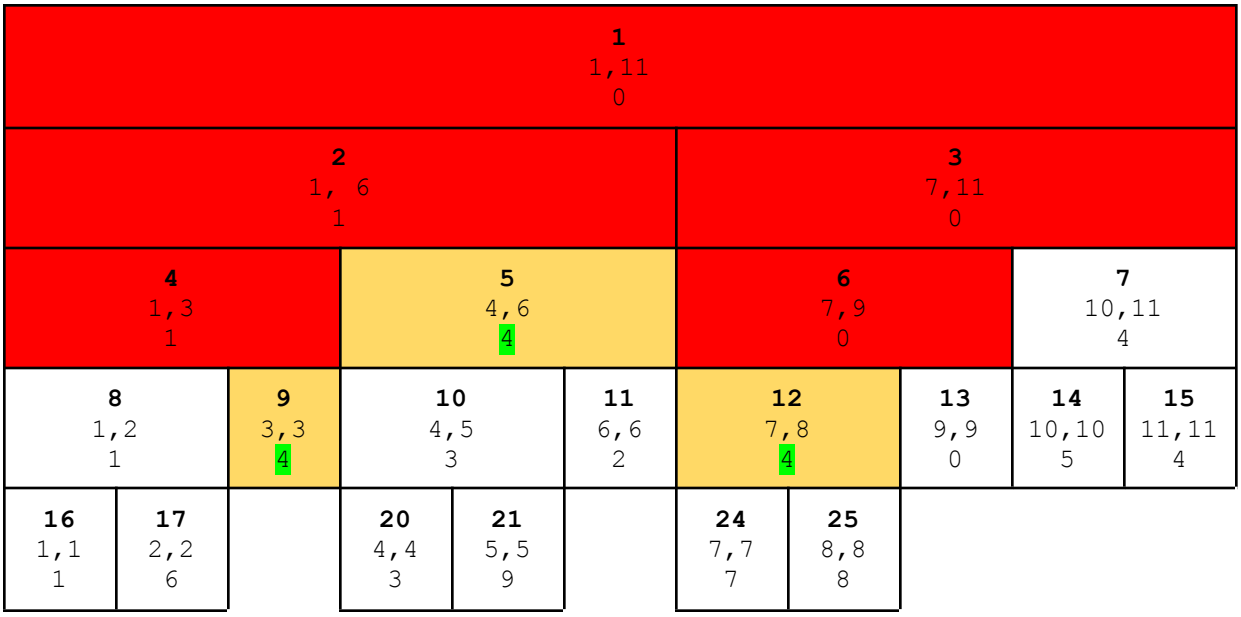
Altă variantă ar fi ca la funcția update să facem cam ca la `query` și să mergem până la frunze.

```
void update(int nod, int st, int dr, int a, int b, int x) {
    if (st == dr) {
        A[nod] = x;
    } else {
        int mid = (st+dr)/2;
        if (a <= mid)
            update(2*nod, st, mid, a, b, x);
        if (b >= mid+1)
            update(2*nod+1, mid+1, dr, a, b, x);
        A[nod] = min(A[2*nod], A[2*nod+1]);
    }
}
```

Practic avansăm peste tot până ajungem la frunze, având grijă să reconstituim nodurile interioare, adică după autoapeluri.
Din păcate, ajungem să actualizăm fiecare frunză individual, iar numărul acestora este de ordin `n`.

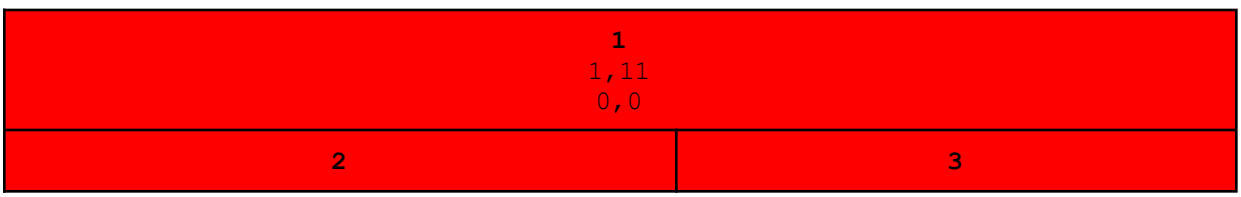
Soluția optimă se obține pornind de la cea de mai sus plus câteva observații pentru a nu mai ajunge la frunze. *Acest lucru se obține oprindu-ne mereu cu actualizarea la intervalele complet incluse în `[a, b]`, adică la cele figurate cu albastru pe desenele anterioare, la explicațiile despre operația de interogare.*
Imediat apare însă întrebarea: dar ce se întâmplă dacă după un astfel de update apare un `query` (sau un alt `update`) strict în interiorul unui astfel de interval pentru care nu am mers și în fii?

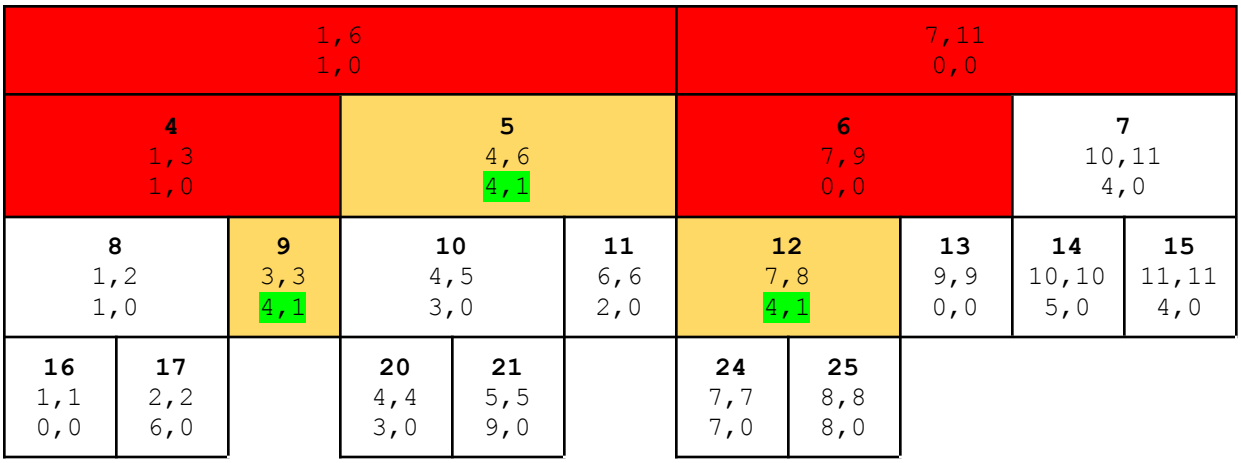
Mai precis, dacă pe exemplul inițial facem o operație de tipul `2 3 8 4` (adică toate valorile din intervalul de indici `3 … 8` devin `4`), arborele nostru ar arăta:



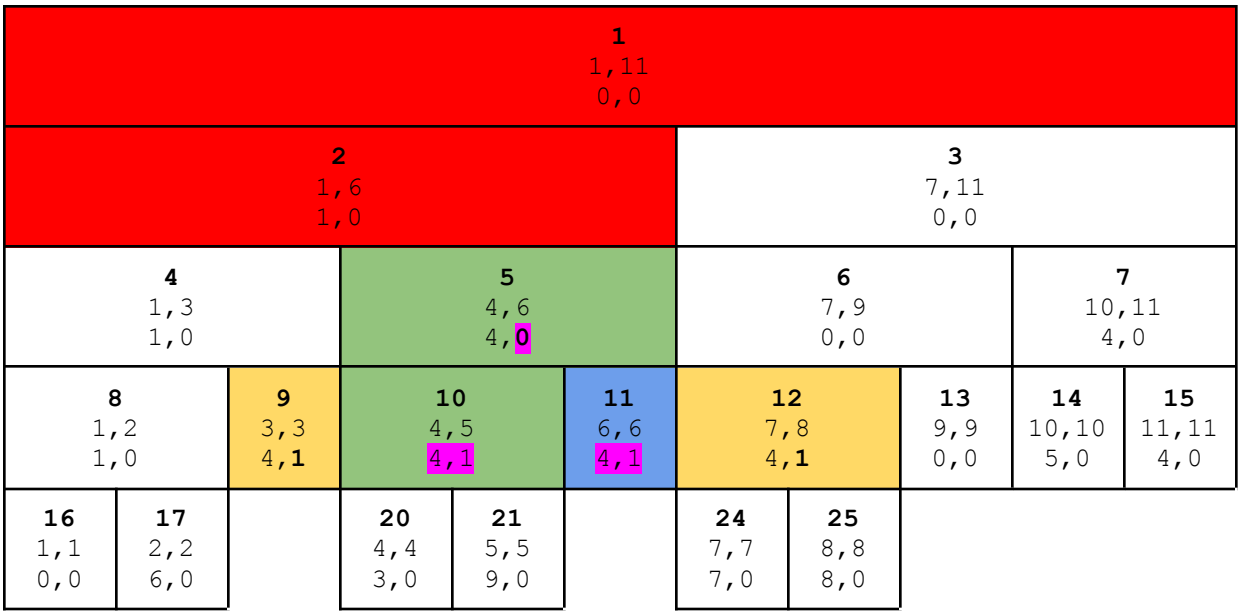
Adică valorile setate pentru intervalele în care se descompune `[a,b]` sunt actualizate la `4`, însă cele de sub ele rămân cu valorile anterioare. Acest lucru ar face ca un query pentru poziția `5`, spre exemplu, să dea valoare greșită, `9` (acest `query` ar ajunge chiar în acea frunză pentru care valoarea acum nu mai este corectă).
Pentru a rezolva problema, vom reține suplimentar în fiecare nod al arborelui de intervale un flag (valoare care poate fi `0` sau `1`) iar când acesta este marcat la `1` semnifică faptul că s-a făcut în intervalul reprezentat de acel nod o actualizare care nu a mai fost trimisă fiilor.
În momentul în care se trece în jos printr-un nod, vom verifica valoarea flagului și în cazul în care el este `1`, înainte să facem autoapel în unul sau în ambii fii ai nodului curent, vom transmite valoarea din nod **în ambii săi fii** și vom seta la `1` flagul din aceștia, făcând totodată `0` flagul din nodul prin care am trecut.
Aceste operații se execută în timp constant, deci nu afectează complexitatea, iar prezența flagului anunță că valorile din nodurile de sub un nod cu flagul `1` nu sunt neapărat reale, dar ele pot fi actualizate când este nevoie de ele, întrucât orice apel pornește de sus în jos și poate afla valoarea flag-ului pentru nodurile prin care trece.

Arborele nostru, după update-ul de mai sus, pentru care figurăm în fiecare nod și minimul și flagul, arată astfel:





Să presupunem că avem mai departe un `query` pentru valoarea din nodul `6` (adică de forma `1 6 6`). Un apel al funcției `query` are nevoie de nodul `11` din `A`, care nu are valoarea corectă însă pentru a ajunge cu autoapelul la el se va trece prin tatăl său, nodul de pe poziția `5`, care are flagul setat. Chiar dacă este necesar să se avanseze doar spre fiul său drept, la trecerea prin nodul `5` se va seta valoarea și flagul către ambii fii ai săi și se va deseta valoarea sa. Astfel, avem:



Pe scurt, artificiul nostru se realizează în următorii pași:

- se operează înainte de autoapeluri;
- **atât la `query` cât și la `update`**, dacă se întâlnește un interval cu flagul `1` pentru care sunt necesare autoapeluri:
  - se setează la `0` flagul;
  - se trimite la fii valoarea sa;
  - se setează la `1` flagul fiilor;

- la update, dacă se întâlnește un interval `[st, dr]` complet inclus în `[a, b]`
    - se actualizează valorea nodului corespunzător
    - se setează flagul la `1`
    - nu se mai face autoapel în fii.

Prezentăm mai jos rezolvarea problemei (actualizare pe interval și interogare de minim pe interval).
Enunțul complet se găsește aici:
https://www.pbinfo.ro/probleme/2091/actualizare-interval-minim-interval

```
#include <fstream>
#include <climits>
#define DIM 100010
#define INF INT_MAX

using namespace std;

ifstream fin ("aimi.in");
ofstream fout("aimi.out");

struct nod {
    int value;
    int flag;
};


nod A[4*DIM];
int n, m, a, b, t, i, minim, x;

void build(int nod, int st, int dr) {
    if (st == dr) {
        fin>>A[nod].value;
        A[nod].flag = 0;
    } else {
        int mid = (st + dr)/2;
        build(2*nod, st, mid);
        build(2*nod+1, mid+1, dr);
        A[nod].value = min(A[2*nod].value, A[2*nod+1].value);
        A[nod].flag = 0;
    }
}

void query(int nod, int st, int dr, int a, int b) {
    if (a <= st && dr <= b) {
        minim = min(minim, A[nod].value);
    } else {
        if (A[nod].flag == 1) {
            A[2*nod].value = A[nod].value;
```

```
            A[2*nod].flag = 1;
            A[2*nod+1].value = A[nod].value;
            A[2*nod+1].flag = 1;
            A[nod].flag = 0;
        }

        int mid = (st + dr)/2;
        if (a <= mid)
            query(2*nod, st, mid, a, b);
        if (b > mid)
            query(2*nod+1, mid+1, dr, a, b);

        A[nod].value = min(A[2*nod].value, A[2*nod+1].value);
    }
}

void update(int nod, int st, int dr, int a, int b, int x) {
    if (a <= st && dr <= b) {
        A[nod].value = x;
        A[nod].flag = 1;
    } else {

        if (A[nod].flag == 1) {
            A[2*nod].value = A[nod].value;
            A[2*nod].flag = 1;
            A[2*nod+1].value = A[nod].value;
            A[2*nod+1].flag = 1;
            A[nod].flag = 0;
        }

        int mid = (st + dr)/2;
        if (a <= mid)
            update(2*nod, st, mid, a, b, x);
        if (b > mid)
            update(2*nod+1, mid+1, dr, a, b, x);
        A[nod].value = min(A[2*nod].value, A[2*nod+1].value);
    }
}

int main () {

    fin>>n;
    build(1, 1, n);
    fin>>m;
    for (int i=1;i<=m;i++) {
        fin>>t>>a>>b;
        if (t == 1) {
            minim = INF;
            query(1, 1, n, a, b);
            fout<<minim<<"\n";
        } else {
```

```
            fin>>x;
            update(1, 1, n, a, b, x);
        }
    }
    return 0;
}
```

Întrucât se fac modificări în noduri și în cazul unui query (atunci când se trece prin cele cu flagul `1`), acestea pot afecta și minimele din nodurile de deasupra lor, așadar, și la revenirea din autoapeluri trebuie pusă instrucțiunea de actualizare a valorii din nodul curent în funcție de valoarea din fii:
`A[nod].value = min(A[2*nod].value, A[2*nod+1].value);`

Semnificația flag-ului a fost descrisă conform cu specificul problemei de mai sus. În funcție de caz, putem da acestuia și alte roluri.

III Folosirea arborilor de intervale pentru operații mai complexe

Problema: Avem un șir de numere întregi asupra căruia putem face două tipuri de operați:

- aflarea minimului urmată de ștergerea sa (dacă minimul apare de mai multe ori se șterge doar o apariție, cea de la indicele mai mic); lungimea șirului se actualizează (scade cu `1`);
- schimbarea valorii unui element dat;

Enunțul complet se află aici:
https://www.pbinfo.ro/probleme/2093/actualizare-element-stergere-minim

Dacă ne spune cineva soluția este folosind arbori de intervale, ne putem gândi că apare o problemă prin faptul că se modifică lungimea vectorului de la o operație la alta (la ștergere). Noi stim că trebuie să formăm același set de triplete (`nod`, `st`, `dr`) la fiecare funcție recursivă. Dar modificându-se `n`, lungimea vectorului, lucrurile se dereglează.

Pentru a evita acest lucru apelăm la următorul truc: nu ștergem efectiv elementele ci doar le marcăm drept șterse.
Procedând astfel, alt lucru de care trebuie ținut cont este că la operațiile de actualizare se dă poziția din momentul curent, ori în urma ștergerilor anterioare ea nu mai corespunde neapărat cu cea inițială, deci trebuie aflată.
De asemenea, în afară de minim ne mai interesează și cea mai din stânga poziție pe care el se află.
Pentru a îndeplini toate cele de mai sus, vom ține, pentru fiecare nod din `AINT` (adică pentru fiecare interval) trei valori:

```
struct nod {
    int minim;
```

```
    int poz;
    int cnt;
};
```

- `minim` reprezintă valoarea minimă din intervalul corespunzător nodului;
- `poz` reprezintă cea mai din stânga poziție pe care se află minimul din interval; această valoare este relativă la dimensiunea inițială a vectorului;
- `cnt` reprezintă numărul de elemente nesterse încă în intervalul corespunzător lui `nod` (deci, dacă acest interval este `[st, dr]`, `dr-st+1 - A[nod].cnt` reprezintă numărul de elemete șterse).

Construim mai întâi arborele de intervale corespunzător vectorului dat:

```
void build(int nod, int st, int dr) {
    if (st == dr) {
/*
  pentru intervalele de lungime 1:
   -  minimul este chiar valoarea de la intrare;
   -  numărul de valori nesterse este 1 (inițial nu e nimic șters);
   -  poziția minimului este chiar cea care se dă la intrare;
*/
        fin>>A[nod].minim;
        A[nod].poz = st;
        A[nod].cnt = 1;
    } else {
        int mid = (st + dr)/2;
        build(2*nod, st, mid);
        build(2*nod+1, mid+1, dr);

/*
    Dacă minimul provine din jumătatea stângă ținem în poz, pentru
    nodul curent, valoarea lui poz din fiul stâng (chiar dacă am
    avea aceeași valoare minimă și în fiul drept)
*/
        A[nod].minim = A[2*nod].minim;
        A[nod].poz = A[2*nod].poz;
        if (A[2*nod+1].minim < A[nod].minim) {
            A[nod].minim = A[2*nod+1].minim;
            A[nod].poz = A[2*nod+1].poz;
        }
/*
   -  inițial toate valorile sunt nesterse;
   -  altfel, puteam scrie: A[nod].cnt = dr-st+1;
*/
        A[nod].cnt = A[2*nod].cnt + A[2*nod+1].cnt;

    }
}
```

Pentru operația de interogare lucrurile sunt relativ simple: În rădăcină avem și minimul și poziția sa. Astfel, putem afișa direct valoarea `A[1].minim`.
Acum trebuie să ștergem elementul de pe poziția `A[1].poz`. Noi am stabilit că vom face o *ștergere logică*, adică doar vom marca drept șters elementul de pe această poziție. Această marcare provoacă modificarea unor informații din `AINT` (scade cu `1` valoarea `cnt` din nodurile care conțin poziția de pe care ștergem, dar se poate modifica și minimul sau poziția sa în aceste noduri).
Practic această ștergere este o actualizare mai specială a valorii unui element (vom pune în el o valoare foarte mare, dar față de actualizarea de tipul `2`, vom mai și seta la `0` valoatea `cnt` din nodul frunză corespunzător nodului marcat ca șters).
Vom folosi deci același cod atât pentru ștergerea unui element cât și pentru actualizarea valorii unui element.

```
void update(int nod, int st, int dr, int poz, int x) {
    if (st == dr) {
        A[nod].minim = x;
        A[nod].poz = st;
        if (x == INF)
            A[nod].cnt = 0; /// pentru ștergere
        else
            A[nod].cnt = 1; /// pentru actualizare
    } else {
        int mid = (st + dr)/2;
        if (poz <= mid)
            update(2*nod, st, mid, poz, x);
        if (poz > mid)
            update(2*nod+1, mid+1, dr, poz, x);

        A[nod].minim = A[2*nod].minim;
        A[nod].poz = A[2*nod].poz;

        if (A[2*nod+1].minim < A[nod].minim) {
            A[nod].minim = A[2*nod+1].minim;
            A[nod].poz = A[2*nod+1].poz;
        }

        A[nod].cnt = A[2*nod].cnt + A[2*nod+1].cnt;
    }
}
```

Pentru ștergere facem apelul:

```
update(1, 1, n, A[1].poz, INF);
```

Pentru actualizare facem apelul:

```
fin>>p>>x;
update(1, 1, n, ?, x);
```

Acum mai rămâne de văzut cine este valoarea **?**.
Cum spuneam, la intrare ni se dă o poziție dinamică, adică ținând cont de ștergerile anterioare, pe când noi păstrăm elementele nesterse pe pozițiile lor inițiale. Trebuie deci să aflăm pe ce poziție se află la un moment dat al `p`-lea element nesters. Pentru asta ne vom folosi de câmpul cnt al nodurilor din `AINT`.

Vom mai face o altă funcție care operează pe `AINT`: `getPoz(int nod, int st, int dr, int p)`, cu semnificația: returnează al `p`-lea nod nesters din intervalul `[st, dr]`, cu informații păstrate în `A` pe poziția `nod`.

Codul este mai jos:

```
int getPoz(int nod, int st, int dr, int p) {
    if (st == dr) {
        return st;
    } else {
        int mid = (st + dr)/2;
        if (A[2*nod].cnt >= p)
            return getPoz(2*nod, st, mid, p);
        else
            return getPoz(2*nod+1, mid+1, dr, p-A[2*nod].cnt);
    }
}
```

Modul de funcționare este ca la o căutare binară:
Al `p`-lea nod nesters din intervalul `[st, dr]` este al `p`-lea nod nesters din intervalul `[st, mid]` dacă în partea stângă sunt cel puțin `p` noduri nesterse, respectiv al `x` - lea nod din intervalul `[mid+1, dr]` dacă în stânga sunt mai puțin de `p` noduri nesterse (unde `x` este egal cu *`p` minus numărul de noduri nesterse în fiul stâng al lui `nod`*). Practic la fiecare pas se coboară un nivel în arbore. Toate aceste funcții au timp de calcul de ordin logaritmic.

Programul complet este aici:

```
#include <fstream>
#define INF 2000000001
#define DIM 100010
using namespace std;

ifstream fin ("aesm.in");
ofstream fout("aesm.out");

struct nod {
    int minim;
    int poz;
    int cnt;
};
```

```
nod A[DIM * 4];
int n, m, x, t, p;

void build(int nod, int st, int dr) {
    if (st == dr) {
        fin>>A[nod].minim;
        A[nod].poz = st;
        A[nod].cnt = 1;
    } else {
        int mid = (st + dr)/2;
        build(2*nod, st, mid);
        build(2*nod+1, mid+1, dr);

        A[nod].minim = A[2*nod].minim;
        A[nod].poz = A[2*nod].poz;

        if (A[2*nod+1].minim < A[nod].minim) {
            A[nod].minim = A[2*nod+1].minim;
            A[nod].poz = A[2*nod+1].poz;
        }
        A[nod].cnt = A[2*nod].cnt + A[2*nod+1].cnt;
    }
}

void update(int nod, int st, int dr, int poz, int x) {
    if (st == dr) {
        A[nod].minim = x;
        A[nod].poz = st;
        if (x == INF)
            A[nod].cnt = 0;
        else
            A[nod].cnt = 1;
    } else {
        int mid = (st + dr)/2;
        if (poz <= mid)
            update(2*nod, st, mid, poz, x);
        if (poz > mid)
            update(2*nod+1, mid+1, dr, poz, x);

        A[nod].minim = A[2*nod].minim;
        A[nod].poz = A[2*nod].poz;

        if (A[2*nod+1].minim < A[nod].minim) {
            A[nod].minim = A[2*nod+1].minim;
            A[nod].poz = A[2*nod+1].poz;
        }

        A[nod].cnt = A[2*nod].cnt + A[2*nod+1].cnt;
    }
}
```

```
int getPoz(int nod, int st, int dr, int p) {
    if (st == dr) {
        return st;
    } else {
        int mid = (st + dr)/2;
        if (A[2*nod].cnt >= p)
            return getPoz(2*nod, st, mid, p);
        else
            return getPoz(2*nod+1, mid+1, dr, p-A[2*nod].cnt);
    }
}

int main () {
    fin>>n;

    build(1, 1, n);

    fin>>m;
    for (int i=1;i<=m;i++) {
        fin>>t;
        if (t == 1) {
            fout<<A[1].minim<<"\n";
            update(1, 1, n, A[1].poz, INF);
        } else {
            fin>>p>>x;
            update(1, 1, n, getPoz(1, 1, n, p), x);
        }
    }
    return 0;
}
```