Arbori de Intervale

Dicu Adrian-Emanuel

11 Decembrie 2021

Contents

1 La ce ne sunt folositori? 2

2 Ce sunt mai exact arborii de intervale? 2

3 Operatiile esentiale: problema Arbint 2

4 Problema de incalzire: Datorii 4

5 Tehnica de 'lazy update': problema Biscuiti 4

6 Subsecventa de suma maxima: problema Sequencequery 6

7 Aplicatie: problema Hotel 7

8 Arbori de intervale 2D: problema Zoo 9

9 Algoritmi de baleiere (line sweeping): problema Demolish 10

10 Aplicatii 11

11 Bibliografie 14

Acesta este un material introductiv in arborii de intervale si aplicatiile lor in problemele de olimpiada.

Daca voi mai tine inca un curs de arbori de intervale, voi prezenta aplicatii si tehnici avansate, precum arbori de intervale pe diferente (progresii), arbori de intervale persistenti, conectivitate dinamica cu arbori de intervale, segment tree beats, etc.

1 La ce ne sunt folositori?

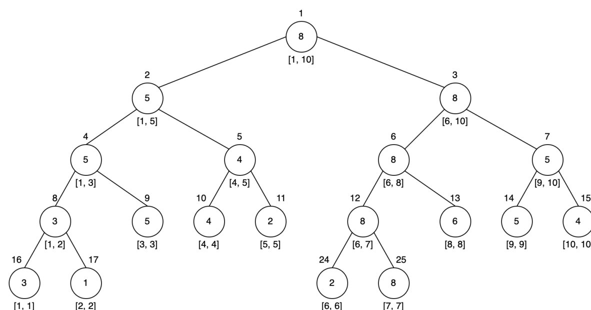
Foarte multe probleme din informatica de olimpiada si nu numai se reduc la calcule pe subsecvente ale unui sir (de exemplu, vrem suma / maximul / minimul pe o subsecventa). Cand sirul este constant si avem de raspuns doar la intrebari pe subsecvente, in general exista modalitati alternative de rezolvare (pentru suma pe subsecventa folosim sume partiale, pentru maximul pe subsecventa folosim RMQ, etc). Dar cand sirul se modifica 'in timp real' atunci avem nevoie de o structura de date mai complicata. Aici intervin arborii de intervale.

2 Ce sunt mai exact arborii de intervale?

Vrem sa rezolvam urmatoarea problema clasica de arbori de intervale de pe infoarena. Un *arbore de intervale* este un arbore binar in care fiecare nod contine informatii asociate unui anumit interval (subsecventa) din sirul original. Asociem asadar fiecarui nod o informatie suplimentara care ne va ajuta sa obtinem rezultatul dorit (in cazul de fata maximul) pe acest interval. Putem defini recursiv un arbore de intervale astfel:

- radacina arborelui va fi nodul 1 si va avea asociat intervalul [1, n]
- daca un nod k are asociat un interval [left, right] atunci cei 2 fii vor avea indicii $2k$ si $2k+1$ iar intervalele asociate lor vor fi [left, middle] respectiv $[middle + 1, right]$, unde $middle = [\frac{left+right}{2}]$

Pentru sirul $3, 1, 5, 4, 2, 2, 8, 6, 5, 4$, arborele de intervale va fi:



Observatii esentiale:

- un arbore de intervale va avea n frunze (cate una pentru fiecare interval de lungime 1) si deci $n-1$ noduri interne, rezultand astfel ca va avea in total $2n-1$ noduri. Insa, asa cum se poate vedea, ultimul nivel al arborelui poate fi incomplet, asa ca vom avea nevoie de un vector de lungime $4n$ (dublul fata de $2n$) pentru stocarea arborelui in memorie
- arborele este echilibrat (are adancimea mica, mai exact are adancimea egala cu $[\log(n)] + 1$)
- **FOARTE IMPORTANT** orice interval [left, right] din sirul initial poate fi compus din concatenarea a cel mult $2[log(n)]$ intervale ale arborelui. De exemplu, intervalul [2, 9] poate fi compus astfel: [2, 2] + [3, 3] + [4, 5] + [6, 8] + [9, 9].

3 Operatiile esentiale: problema Arbint

3.1 build $O(n)$

Vom folosi o functie recursiva care construieste arborele de jos in sus (de la frunze spre radacina). Cand vrem sa calculam valoarea din nod (maximul pe intervalul asociat nodului curent), vom fi calculat deja valoarea din cei 2 fii si observam ca valoarea nodului este egala cu maximul celor 2 fii.

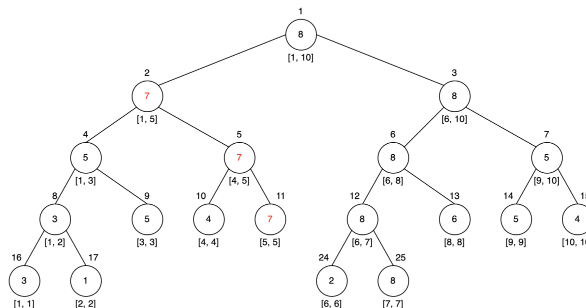
```
void build(int node, int left, int right)
{
  if (left == right) {
    segment_tree[node] = array[left];
  } else {
    int middle = (left + right) / 2;

    build(node * 2, left, middle);
    build(node * 2 + 1, middle + 1, right);

    segment_tree[node] = std::max(segment_tree[node * 2],
                                  segment_tree[node * 2 + 1]);
  }
}
```

3.2 update $O(log\ n)$

Vom folosi tot o functie recursiva care pleaca din radacina si se duce pana in frunza. Cand ne aflam in frunza, update-ul reprezinta o simpla inlocuire a valorii vechi cu cea noua. La intoarcere din recursivitate, trebuie sa reupdatam informatia din nod folosindu-ne de informatiile din cei 2 fii.

```
void update(int node, int left, int right, int position, int new_value)
{
  if (left == right) {
    segment_tree[node] = new_value;
  } else {
    int middle = (left + right) / 2;

    if (position <= middle)
      update(node * 2, left, middle, position, new_value);
    else
      update(node * 2 + 1, middle + 1, right, position, new_value);

    segment_tree[node] = std::max(segment_tree[node * 2],
                                  segment_tree[node * 2 + 1]);
  }
}
```

Mai jos se poate observa cum se modifica structura arborelui de intervale dupa ce modificam in sirul de mai sus $(3, 1, 5, 4, 2, 2, 8, 6, 5, 4)$ valoarea de pe pozitia 5 din 2 in 7.



3.3 query $O(log\ n)$

Ne folosim de observatia 3) si spargem intervalul de query in maxim $2[log(n)]$ segmente ale arborelui de intervale. Observam ca raspunsul pentru intervalul nostru este egal cu maximul tuturor valorilor asociate segmentelor arborelui.

```
int query(int node, int left, int right, int query_left, int query_right)
{
  if (query_left <= left and right <= query_right) {
    return segment_tree[node];
```

```
  } else {
    int middle = (left + right) / 2;

    if (query_right <= middle)
      return query(node * 2, left, middle, query_left, query_right);
    if (middle + 1 <= query_left)
      return query(node * 2 + 1, middle + 1, right, query_left, query_right);

    return std::max(query(node * 2, left, middle, query_left, query_right),
                    query(node * 2 + 1, middle + 1, right, query_left, query_right));
  }
}
```

Codul sursa aici

4 Problema de incalzire: Datorii

Aceasta problema merge rezolvata si cu AIB-uri. In scop didactic pentru incalzire, o vom rezolva si cu arbori de intervale. Vom tine intr-un nod din arborele de intervale suma pe acel interval. Rezolvarea e aproape identica cu cea de mai sus. Singura diferenta e ca acum la operatia de query, raspunsul pe un interval de query va fi egal cu suma intervalelor din arbore in care acesta se descompune.

Codul sursa aici

5 Tehnica de 'lazy update': problema Biscuiti

Pana acum am intalnit numai aplicatii in care update-ul se executa pe o pozitie (pe un singur element). Ce facem in schimb daca vrem sa modificam un interval de elemente? Clar putem sa updatam fiecare element individual, dar este foarte ineficient. Aici intervine tehnica de *lazy update*.

Cum functioneaza aceasta tehnica? Vom mai tine inca o valoare auxiliara in structura nodului din arbore in care specificam o actiune pe care vrem sa o executam asupra tuturor elementelor din intervalul asociat nodului curent. Partea desteapta a tehnicii este ca actiunea nu trebuie executata imediat pentru toate pozitiile corespunzatoare acestui interval, ci reprezinta mai degraba o 'promisiune' ca actiunea va fi executata in viitor. Cand ajungem intr-un nod care trebuie sa execute o actiune pentru toate pozitiile din intervalul sau si vrem sa ne ducem in fii lui, mai intai 'instruim' cei 2 fii sa execute ei pentru nodul nostru actiunea pe care o are el de facut. Astfel, nodul 'paseaza responsabilitatea' celor 2 fii si deci el nu mai trebuie sa execute nicio actiune. In schimb, acum cei 2 fii vor avea de executat actiunea, dar si ei la randul lor nu o executa imediat, ci 'fac si ei o promisiune' ca o vor executa, dar doar cand este absolut necesar, similar cu nodul curent.

In problema noastra, avem nevoie de urmatoarele operatii:

- gasirea pozitiei scandurii cu inaltimea minima din tot sirul
- toate scandurile dintr-un interval [left, right] cresc in inaltime cu k unitati

Vom tine asadar intr-un nod al arborelui de intervale inaltimea scandurii minime din intervalul curent, pozitia pe care se afla scandura cu inaltimea minima din intervalul curent, si in plus vom mai tine o valoare 'lazy' care ne indica faptul ca toate scandurile din intervalul nodului curent trebuie crescute cu 'lazy' unitati (actiunea in cazul de fata este deci: cresterea tuturor valorilor de pe pozitiile din intervalul curent cu o anumita valoare). Sa presupunem ca ne aflam intr-un nod 'nod' care trebuie sa creasca toate scandurile din intervalul sau cu 'lazy-nod'. Atunci, inainte de a ne duce in fiii nodului, vom 'instrui' cei 2 fii sa creasca ei scandurile pentru nodul curent (echivalent cu a spune: valorile 'lazy' asociate celor 2 fii vor creste cu valoarea 'lazy' a nodului curent). De asemenea, din moment ce stim ca toate scandurile din intervalul curent cresc cu aceeasi valoare implicit si lungimea scandurii minime din acest interval va creste tot cu aceeasi valoare 'lazy'. De abia dupa ce am pasat responsabilitatea de la nod la fii sai, ne putem duce in fii si sa continuam update-ul / query-ul.

5.1 build $O(n)$

Pentru inceput introducem functia *update_node*(*int node*); care updateaza inaltimea scandurii minime, precum si pozitia ei, stiind ca informatia din cei 2 fii este deja calculata:

```
void update_node(int node)
{
```

```
  // updatam scandura minima
  segment_tree[node].minim = std::min(segment_tree[node * 2].minim,
                                      segment_tree[node * 2 + 1].minim);

  // verificam daca scandura minima vine din fiul stang sau din cel drept
  if (segment_tree[node].minim == segment_tree[node * 2].minim)
    segment_tree[node].position = segment_tree[node * 2].position;
  else
    segment_tree[node].position = segment_tree[node * 2 + 1].position;
}
```

Folosindu-ne de aceasta functie, putem scrie update-ul foarte usor astfel:

```
void build(int node, int left, int right)
{
  if (left == right) { // interval de lungime 1
    segment_tree[node].minim = height[left];
    segment_tree[node].position = left;
  } else {
    int middle = (left + right) / 2;

    build(node * 2, left, middle);
    build(node * 2 + 1, middle + 1, right);

    update_node(node); // updatam nodul curent
  }
}
```

5.2 query $O(1)$

Nu avem query. Cand interogam arborele de intervale ne uitam direct in radacina deoarece ne trebuie scandura minima pe intreg sirul (interval corespunzator radacinii).

5.3 update $O(log\ n)$

Mai intai introducem functia $update_lazy(int\ node, int\ left, int\ right)$; care primeste un nod, precum si intervalul lui asociat si executa actiunea pe care nodul curent a amanat-o pana acum si eventual o paseaza fiilor sai (daca nu este nod frunza)

```
void update_lazy(int node, int left, int right)
{
  if (segment_tree[node].lazy) { // daca nodul curent trebuie sa creasca
                                 // scandurile din intervalul curent
    segment_tree[node].minim += segment_tree[node].lazy; // minimul creste si el

    if (left != right) { // vedem daca nodul are fii carora
                         // sa le paseze responsabilitatea
      segment_tree[node * 2].lazy += segment_tree[node].lazy;
      segment_tree[node * 2 + 1].lazy += segment_tree[node].lazy;
    }

    segment_tree[node].lazy = 0; // nodul curent si-a terminat treaba
  }
}
```

Avand functia de updatare a lazy-ului, putem sa implementam update-ul astfel.

```
void update(int node, int left, int right,
            int query_left, int query_right, long long increase)
{
  // primul pas este sa vedem daca nodul are de executat vreo
  // actiune pe care a amanat-o (ne uitam la lazy)
```

```
  update_lazy(node, left, right);

  // verificam daca intervalul curent e complet inclus in intervalul de query
  if (query_left <= left and right <= query_right) {
    // in caz afirmativ, toate scandurile cresc cu
    // aceeasi valoare, asa ca ne folosim de lazy
    segment_tree[node].lazy += increase;
    update_lazy(node, left, right);
  } else {
    int middle = (left + right) / 2;

    if (query_left <= middle)
      update(node * 2, left, middle, query_left, query_right, increase);
    if (middle + 1 <= query_right)
      update(node * 2 + 1, middle + 1, right, query_left, query_right, increase);

    // indiferent de care fiu este updatat, noi trebuie sa recalculam
    // valoarea din nodul curent si pentru asta avem nevoie de inaltimile
    // curente ale celor 2 fii, asa ca ii 'fortam' sa isi execute actiunile
    // (updatam fortat lazy-urile lor)
    update_lazy(node * 2, left, middle);
    update_lazy(node * 2 + 1, middle + 1, right);

    update_node(node); // updatam nodul curent
  }
}
```

Codul sursa aici

6 Subsecventa de suma maxima: problema Sequencequery

Problema aceasta nu are update-uri, dar tehnica folosita la query e suficient de interesanta si des intalnita la olimpiada incat sa merite sa fie prezentata aici.

Dupa cum am observat pana acum, ideea generala pe care o folosim atunci cand abordam o problema cu arbori de intervale este descompunerea unei subsecvente in mai multe subsecvente care 'se pot combina usor'.

Ce intelegem prin 'se pot combina usor'? De exemplu, la problema clasica a arborilor de intervale, ca sa obtinem maximul pe un interval mare de query, il descompunem in intervale mai mici din care luam maximul fiecaruia si apoi le 'reunim' facand maximul tuturor intervalelor. Similar la suma, 'reunim' intervalele adunand sumele lor corespunzatoare.

Cum putem sa aplicam ideea de 'concatenare/reuniune' a 2 intervale in problema curenta? Sa presupunem ca avem pentru un nod informatia celor 2 fii de-ai sai si vrem sa gasim rapid informatia pentru nodul curent (cu alte cuvinte avem intervalele $[left, middle]$ si $[middle + 1, right]$ si vrem sa gasim informatia pentru intervalul $[left, right]$). Daca stocam in nod doar subsecventa de suma maxima din intervalul fiilor din pacate nu putem sa reunim cei 2 fii ca sa obtinem subsecventa de suma maxima a nodului curent.

Solutia problemei este sa stocam si alte valori aditionale care sa ne ajute sa gasim subsecventa de suma maxima a nodului rapid. Vom stoca 4 valori: subsecventa de suma maxima, prefixul de suma maxima, sufixul de suma maxima, respectiv suma intregului interval.

```
struct tree_node {
  long long sum; // suma subsecventei
  long long pref_scmax; // prefixul de suma maxim
  long long suff_scmax; // sufixul de suma maxima
  long long scmax; // subsecventa de suma maxima
};
```

Cele 4 valori auxiliare corespunzatoare nodului se updateaza din cei 2 fii astfel:

```
tree_node update_node(tree_node left_node, tree_node right_node)
{
  tree_node current_node;
```

```
  current_node.sum =
      left_node.sum + right_node.sum;

  current_node.pref_scmax =
      std::max(left_node.pref_scmax,
               left_node.sum + right_node.pref_scmax);

  current_node.suff_scmax =
      std::max(right_node.suff_scmax,
               right_node.sum + left_node.suff_scmax);

  current_node.scmax =
      std::max(left_node.suff_scmax + right_node.pref_scmax,
               std::max(left_node.scmax, right_node.scmax));

  return current_node;
}
```

Explicatie:

- suma intervalului $[left, right]$ este egala cu suma intervalelor fiilor
- pentru prefixul de suma maxima, avem 2 cazuri: fie 'nu trece' de middle, caz in care prefixul de suma maxima este complet inclus in intervalul fiului stang (deci deja calculat), fie 'trece' de middle, caz in care prefixul este egal cu intreg intervalul fiului stang plus un prefix din intervalul fiului drept
- similar pentru sufixul de suma maxima
- pentru subsecventa de suma maxima, avem iar 2 cazuri: fie 'nu intersecteaza' marginea dintre cei 2 fii, caz in care subsecventa de suma maxima a nodului curent este complet inclusa in unul din cei 2 fii (si implicit o alegem pe cea mai buna), fie 'trece din fiul stang in cel drept', caz in care subsecventa de suma maxima va fi egala cu un sufix al fiului din stanga plus un prefix al fiului din dreapta

Codul sursa aici

7 Aplicatie: problema Hotel

Aceasta este o combinatie a problemei 'biscuiti' (lazy update) si a problemei 'sequencequery' (subsecventa de lungime maxima plina cu 0, care se rezolva similar cu subsecventa de suma maxima). Initial avem un sir de lungime n plin cu 0 (toate camerele sunt libere). Avem nevoie de urmatoarele 2 operatii

- $set(left,\ right,\ state)$; unde state poate fi 0 (se elibereaza camerele din intervalul $[left, right]$) sau 1 (se ocupa camerele din intervalul $[left, right]$)
- $query$; trebuie sa gasim cel mai lung interval de camere libere (cea mai lunga subsecventa plina cu 0)

Pentru a realiza cele 2 operatii, vom tine in structura nodului asemanator cu problema sequencequery urmatoarele date: cea mai lunga subsecventa plina cu 0, cel mai lung prefix plin cu 0, cel mai lung sufix plin cu 0, lungimea subsecventei. Dar cum avem update pe interval, trebuie sa mai tinem si o valoare lazy care poate avea 3 valori:

- 0 - nu trebuie sa facem nimic
- 1 - toate camerele din intervalul curent trebuie ocupate
- 2 - toate camerele din intervalul curent trebuie eliberate

Structura nodului este asadar:

```
struct tree_node {
  int max;
  int left_max;
  int right_max;
  int lazy;
};
```

Functia care combina intervalele fiilor in intervalul nodului arata astfel:

```
void update_node(int node, int left, int right)
{
  int middle = (left + right) / 2,
      left_son = node * 2,
      right_son = node * 2 + 1;

  segment_tree[node].left_max =
    (segment_tree[left_son].left_max == middle - left + 1) ?
        (middle - left + 1) + segment_tree[right_son].left_max :
        segment_tree[left_son].left_max;

  segment_tree[node].right_max =
    (segment_tree[right_son].right_max == right - middle) ?
        (right - middle) + segment_tree[left_son].right_max :
        segment_tree[right_son].right_max;

  segment_tree[node].max =
    std::max(segment_tree[left_son].right_max + segment_tree[right_son].left_max,
             std::max(segment_tree[left_son].max,
                      segment_tree[right_son].max));
}
```

Si functia care updateaza lazy-ul unui nod este:

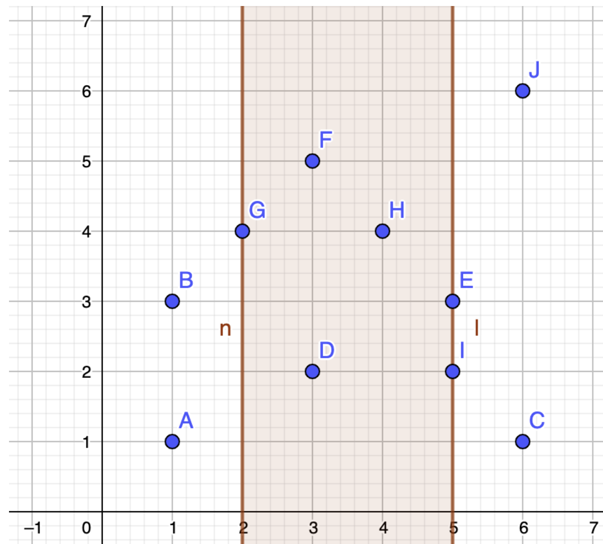
```
void update_lazy(int node, int left, int right)
{
  if (segment_tree[node].lazy == 1) {
    segment_tree[node].max = 0;
    segment_tree[node].left_max = 0;
    segment_tree[node].right_max = 0;

    if (left != right) {
      int left_son = node * 2,
          right_son = node * 2 + 1;

      segment_tree[left_son].lazy = 1;
      segment_tree[right_son].lazy = 1;
    }

    segment_tree[node].lazy = 0;
  } else
  if (segment_tree[node].lazy == 2) {
    segment_tree[node].max = right - left + 1;
    segment_tree[node].left_max = right - left + 1;
    segment_tree[node].right_max = right - left + 1;

    if (left != right) {
      int left_son = node * 2,
          right_son = node * 2 + 1;

      segment_tree[left_son].lazy = 2;
      segment_tree[right_son].lazy = 2;
    }

    segment_tree[node].lazy = 0;
  }
}
```

Codul sursa aici

8 Arbori de intervale 2D: problema Zoo

Observam ca arborii de intervale se pot folosi in multe probleme variate, datorita flexibilitatii lor. Putem in structura nodului din arborele de intervale sa tinem si vectori sau alte structuri liniare, astfel avand un arbore de intervale pe 2 dimensiuni (2D). Acestia au aplicatii in special in problemele de 'range searching' sau probleme de geometrie 2D in care trebuie facute query-uri pe dreptunghiuri (in problema zoo trebuie sa numaram cate puncte sunt intr-un dreptunghi).

Pentru inceput, se va face o normalizare a coordonatelor a.i toate punctele si colturile dreptunghiurilor sa fie in intervalul $[1, 16000]$. Apoi putem sa facem un arbore de intervale peste dimensiunea Ox astfel: fiecare nod din arborele de intervale are asociat o 'fasie verticala infinita' $[left, right]$. De exemplu fasia asociata intervalului $[2, 5]$ din exemplul de mai jos arata cam asa:



Pentru fiecare fasie, vom tine toate punctele din interiorul acestei fasii in ordine sortata pe Oy (pentru exemplul de mai sus, in fasia $[2, 5]$ punctele vor fi D, I, E, G, H, F)

8.1 build $O(n\ log\ n)$ sau $O(n\ log^2 n)$

Operatia de build consta in adaugarea fiecarui punct in cele log intervale din care face parte, rezultand astfel o complexitate de $O(n\ log\ n)$ sau $O(n\ log^2 n)$ daca se resorteaza punctele din cadrul fiecarei fasii

```
void build(int node, int left, int right)
{
  if (left == right) {
    std::swap(segment_tree[node], list[left]);
    std::sort(std::begin(segment_tree[node]),
              std::end(segment_tree[node]));
  } else {
    int middle = (left + right) / 2;

    build(node * 2, left, middle);
    build(node * 2 + 1, middle + 1, right);

    std::merge(
        std::begin(segment_tree[node * 2]),
        std::end(segment_tree[node * 2]),
        std::begin(segment_tree[node * 2 + 1]),
        std::end(segment_tree[node * 2 + 1]),
        std::back_inserter(segment_tree[node]));
  }
}
```

8.2 query $O(log^2 n)$

Se sparge dreptunghiul de query in subfasii verticale corespunzatoare arborelui de intervale pe Ox. Pentru fiecare astfel de fasie se determina toate punctele care se afla si in interiorul intervalului de query pe Oy (marginile de

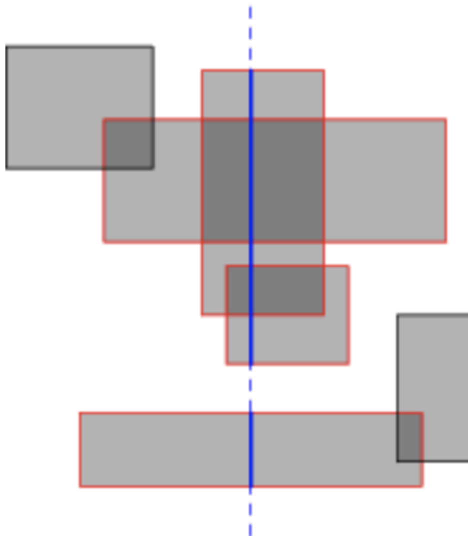
sus si de jos ale dreptunghiului) prin 2 cautari binare. Avem $log\ n$ fasii si pentru fiecare fasie se vor executa 2 cautari binare, rezultand astfel o complexitate totala de $log^2 n$:

```
int query(int node, int left, int right,
          int x_left, int x_right, int y_left, int y_right)
{
  if (x_left <= left and right <= x_right) {
    return std::upper_bound(
              std::begin(segment_tree[node]),
              std::end(segment_tree[node]), y_right) -
           std::lower_bound(
              std::begin(segment_tree[node]),
              std::end(segment_tree[node]), y_left);
  } else {
    int middle = (left + right) / 2;

    if (x_right <= middle)
      return query(node * 2, left, middle,
                   x_left, x_right, y_left, y_right);
    if (middle + 1 <= x_left)
      return query(node * 2 + 1, middle + 1, right,
                   x_left, x_right, y_left, y_right);

    int answer_left = query(node * 2, left, middle,
                            x_left, x_right, y_left, y_right),
        answer_right = query(node * 2 + 1, middle + 1, right,
                             x_left, x_right, y_left, y_right);

    return answer_left + answer_right;
  }
}
```

Codul sursa aici

9 Algoritmi de baleiere (line sweeping): problema Demolish

In primul rand in ce consta algoritmul de baleiera? Este o tehnica de 'scanare' in care se fixeaza o dreapta (de obicei verticala sau orizontala) care are rolul de scanner si inregistreaza evenimente precum aparitia unui obstacol, disparitia unui obstacol sau o interogare (un query).



Pe exemplul de mai sus, putem considera dreptunghiurile ca fiind obstacolele iar baleiera ca fiind dreapta verticala albastra care se misca de la stanga spre dreapta si tine evidenta tuturor dreptunghiurilor pe care le-a

intalnit pana acum. Dreptunghiurile cu marginile rosii sunt cele care intersecteaza baleiera (au fost detectate ca aparute dar inca nu au disparut).

De ce ne trebuie o astfel de tehnica atunci cand avem arbori de intervale 2D? Pentru ca nu toate problemele se pot rezolva (nu usor cel putin) cu arbori de intervale 2D iar algoritmul de baleiere reprezinta intuitiv o reducere a unei probleme 2D la o problema 1D considerand doar evenimentele care se intampla pe baleiera si nu in tot planul.

In cadrul problemei noastre, baleiera va reprezenta latura din stanga a fermei pe care vrem sa o amplasam. Vom tine asadar un arbore de intervale pe baleiera ce va retine pentru fiecare pozitie p, care ar fi costul amplasarii fermei cu coltul din stanga jos pe linia p, la coloana reprezentata de pozitia actuala a baleierei. Vom tine si linia pentru care s-a obtinut acest cost minim. O data cu mutarea baleierei, vom intalni marginile din stanga ale unor ferme ce trebuie demolate, respectiv vom intalni marginea din dreapta a unor ferme care nu mai e necesar sa le demolam ca sa putem amplasa ferma noastra. La fiecare amplasare a baleierei, vom cauta costul minim pentru a amplasa ferma noastra stiind ca are marginea din stanga pe baleiera noastre.

Atentie! Cand avem o baleiera la o anumita coloana p, trebuie sa ne uitam la fermele care apar la coloana $p + (DX - 1)$ deoarece acestea vor intersecta marginea din dreapta a fermei noastre. Altfel spus, baleiera la noi este o 'fasie verticala' de lungime DX. Fiecare dreptunghi $[(x1, y1)\ x\ (x2, y2)]$ va aparea la momentul $x1 - (DX - 1)$ si va disparea la momentul $x2$. La noi evenimentele sunt de forma:

```
struct event {
  int x;
  int y1;
  int y2;
  int cost;
  bool start_rectangle;
};
```

unde x este coloana, $y1$ si $y2$ reprezinta intervalul de pe baleiera afectat de acest eveniment, *cost* e self explanatory si start_rectangle e $true/false$ in functie daca dreptunghiul apare(true) sau dispare(false).

Evenimentele generate asadar de dreptunghi sunt:

```
events.push_back({x1 - (dx - 1), y1, y2, cost, true});
events.push_back({x2, y1, y2, cost, false});
```

pe care o sa le sortam dupa x inainte sa incepem sa iteram cu baleiera.

Codul sursa aici

10 Aplicatii

10.1 problema Parcele

Observatia esentiala este ca oricare 2 dreptunghiuri care nu se intersecteaza si care au laturile paralele cu axele de coordonate, trebuie sa fie separate printr-o dreapta verticala sau orizontala. Vom trata mai intai cazul in care aceasta dreapta e verticala. Parcurgem punctele cu o baleiera verticala de la stanga la dreapta. Fiecare punct de pe baleiera reprezinta numarul de copaci din interiorul unei parcele care are coltul din stanga jos aici.

Avem cate 2 evenimente (unul cand intalnim si unul cand lasam in urma) de forma $(x, y1, y2, true/false)$ pentu fiecare fiecare punct (p, q), evenimente care marcheaza toata regiunea in care daca am pune un dreptunghi cu coltul din stanga jos acolo, atunci acesta va contine punctul in interior. Cu alte cuvinte, evenimentele generate de un punct (p, q) sunt $(x - DX, y - DY, y, true)$ si $(x + 1, y - DY, y, false)$. Pentru fiecare x, dupa ce adaugam in arborele de intervale toate evenimentele de la pozitia x, vom interoga arborele de intervale pentru a afla numarul maxim de puncte dintr-o parcela cu latura din stanga pe aceasta baleiera.

Dupa ce obtinem asadar pentru fiecare x cea mai buna parcela cu latura din stanga pe coloana x, vom fixa dreapta de demarcatie dintre cele 2 parcele si vom selecta cele 2 parcele a.i acestea sa nu taie aceasta dreapta, lucru realizabil cu 2 vectori care tin maximele partiale pe prefixe si sufixe.

La final, vom inversa dx cu dy si cele 2 coordonate ale fiecarui punct si vom aplica din nou acelasi algoritm pentru a trata si cazul in care dreapta de demarcatie este orizontala.

Codul sursa aici (ia 95 puncte pentru ca trebuie optimizat)

10.2 problema EasyQuery

Solutia este foarte asemanatoare cu cea din editorial, doar ca are mai multe explicatii.

La fel, trebuie sa determinam un set minim de informatii pe care trebuie sa le tinem pentru a reuni usor 2 intervale intr-un interval mai mare. Vom tine asadar in nodul arborelui de intervale corespunzator subsecventei $[x_l, ..., x_{right}]$ urmatoarele valori:

- x_max = $max(x_p | p \in [l, r])$
- x_min = $min(x_p | p \in [l, r])$
- x_max_max = $max(x_p + x_q | p \leq q; p, q \in [l, r])$
- x_max_min = $min(x_p - x_q | p \leq q; p, q \in [l, r])$
- x_min_max = $max(x_p - x_q | p \leq q; p, q \in [l, r])$
- x_min_min = $min(x_p + x_q | p \leq q; p, q \in [l, r])$
- y_max = $max(x_p - x_q + x_r | p \leq q, p \leq r; p, q, r \in [l, r])$
- z_min = $min(x_p - x_q + x_r | p \leq q, p \leq r; p, q, r \in [l, r])$

Intuitia din spatele acestor date este ca noua ne trebuie 'triplete' de valori pentru a calcula y_max si z_min, dar ca sa obtinem informatia pentru triplete, ne trebuie sa calculam mai intai informatia pentru perechi de valori, care la randul lor se calculeaza folosind informatia din x_max si x_min. Cum calculam aceste valori pentru un interval $[x_{left}, ..., x_{right}]$?

- x_max: trebuie sa luam maximul din cei 2 fii.

```
answer.x_max = max(left_node.x_max,
                   right_node.x_max);
```

- x_min: trebuie sa luam minimul din cei 2 fii.

```
answer.x_min = min(left_node.x_min,
                   right_node.x_min);
```

- x_max_max: avem 3 cazuri: ambele pozitii sunt in stanga, ambele pozitii sunt in dreapta, respectiv p este in stanga si q in dreapta.

```
answer.x_max_max = max(left_node.x_max_max,
                       right_node.x_max_max,
                       left_node.x_max + right_node.x_max);
```

- x_max_min: ca mai sus.

```
answer.x_max_min = min(left_node.x_max_min,
                       right_node.x_max_min,
                       left_node.x_min - right_node.x_max);
```

- x_min_max: ca mai sus.

```
answer.x_min_max = max(left_node.x_min_max,
                       right_node.x_min_max,
                       left_node.x_max - right_node.x_min);
```

- x_min_min: ca mai sus.

```
answer.x_min_min = min(left_node.x_min_min,
                       right_node.x_min_min,
                       left_node.x_min + right_node.x_min);
```

- y_max: avem 5 cazuri, in ordine: p,q,r sunt in stanga; p,q,r sunt in dreapta; doar p si q in stanga; doar p si r in stanga; doar p in stanga

```
answer.y_max = max(
    left_node.y_max,
    right_node.y_max,
    left_node.x_max_max - right_node.x_min,
    left_node.x_min_max + right_node.x_max,
    left_node.x_max - right_node.x_min + right_node.x_max);
```

- z_min: ca mai sus

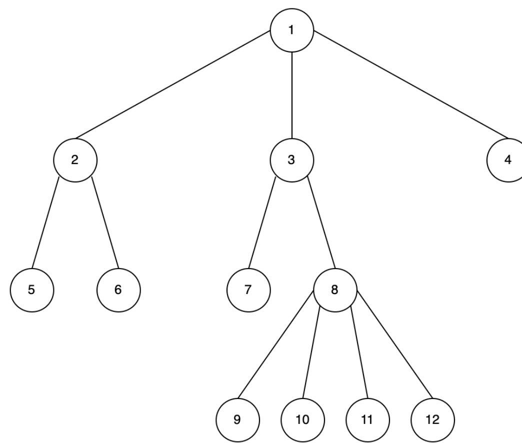
```
    answer.z_min = min(
        left_node.z_min,
        right_node.z_min,
        left_node.x_max_min + right_node.x_min,
        left_node.x_min_min - right_node.x_max,
        left_node.x_min - right_node.x_max + right_node.x_min);
```

Avand toate aceste informatii, putem asadar combina 2 intervale adiacente intr-unl singur.
Codul sursa aici

10.3 problema Omizi

In primul rand vom face 2 liniarizari ale arborelui: una pentru omizile de stanga si una pentru omizile de dreapta. Fie un nod oarecare nod cu lista sa de fii $[son_1, son_2, ..., son_k]$. O omida de stanga ar alege nodurile in ordinea: $[lin_le(son_1), lin_le(son_2), ..., lin_le(son_k), nod]$ si o omida de dreapta ar alege nodurile in ordinea $[lin_ri(son_k), lin_ri(son_{k-1}), ..., lin_ri(son_1), nod]$ (am folosit notatia $lin_le(nod)$ = liniarizarea de stanga a lui nod si $lin_ri(nod)$ = liniarizarea de dreapta a lui nod).

In exemplul de mai jos, normalizarea de stanga este $[5, 6, 2, 7, 9, 10, 11, 12, 8, 3, 4, 1]$ si normalizarea de dreapta este $[4, 12, 11, 10, 9, 8, 7, 3, 6, 5, 2, 1]$



Pentru fiecare liniarizare vom tine si vectorul invers (pentru fiecare nod, pe ce pozitie se afla in liniarizare), precum si cate un arbore de intervale. Vom face un dfs pe arbore si incercam sa calculam pozitia fiecarei omizi incepand de la frunze spre radacina. Dupa ce o omida isi gaseste pozitia finala, nodul respectiv va deveni ocupat. Presupunem ca am pozitionat deja toate omizile subarborilor unui nod si ca nodul curent are o omida de stanga (similar pentru o omida de dreapta). Observam ca in subsecventa care reprezinta liniarizarea de stanga a subarborelui lui nod, noi avem nevoie practic de prima pozitie neocupata din cadrul liniarizarii de stanga. Dupa ce o determinam, nodul acesta va deveni ocupat si implicit va trebui sa marcam in cei 2 arbori de intervale acest nod.

Sa luam liniarizarea de stanga. Avem nevoie deci de urmatoarele 2 operatii:

- $set(pos)$: marcheaza ca ocupata pozitia poz
- $query(left, right)$: gaseste prima pozitie din intervalul $[left, right]$ care este neocupata.

Cum putem realiza acest lucru? In arborele de intervale vom tine pentru fiecare nod numarul de pozitii din acest interval care sunt ocupate. Acum cand avem un interval de query $[left, right]$ il vom sparge in mai multe intervale din arbore de forma $[left, pos_1], [pos_1 + 1, pos_2], [pos_2 + 1, pos_3], ..., [pos_k + 1, right]$. Vom lua primul interval care nu are toate pozitiile ocupate si vom cauta prima pozitie libera in acesta. Sa zicem ca suntem intr-un interval din arbore de forma $[p, q]$ pentru care stim ca exista cel putin o pozitie neocupata. Fie $m = [\frac{p+q}{2}]$. Daca intervalul $[p, m]$ contine cel putin o pozitie neocupata, cautam in acest interval. Altfel, intervalul $[m + 1, q]$ va contine cel putin o pozitie neocupata si vom cauta in acesta.

```
int query(int type, int node, int left, int right,
          int query_left, int query_right)
{
  if (query_left <= left and right <= query_right) {
    if (segment_tree[type][node] == right - left + 1)
```

```
      return -1;
    if (left == right) {
      return left;
    } else {
      int middle = (left + right) / 2,
          left_son = node * 2,
          right_son = node * 2 + 1;

      if (segment_tree[type][left_son] != middle - left + 1)
        return query(type, left_son, left, middle,
                     query_left, query_right);
      else
        return query(type, right_son, middle + 1, right,
                     query_left, query_right);
    }
  } else {
    int middle = (left + right) / 2,
        left_son = node * 2,
        right_son = node * 2 + 1;

    int answer = -1;

    if (query_left <= middle)
      answer = query(type, left_son, left, middle,
                     query_left, query_right);
    if (answer != -1)
      return answer;
    if (middle + 1 <= query_right)
      answer = query(type, right_son, middle + 1, right,
                     query_left, query_right);

    return answer;
  }
}
```

Codul sursa aici

11 Bibliografie

https://www.infoarena.ro/arbori-de-intervale
https://www.infoarena.ro/algoritmi-de-baleiere
https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree
https://cp-algorithms.com/data_structures/segment_tree.html