Математичка гимназија у Београду

МАТУРСКИ РАД

из програмирања и програмских језика

Сегментно стабло

Ученик:
Марко Илић

Ментор:
Снежана Јелић

Београд, мај 2021.

Садржај

1. Увод

Сермeнтно стабло је први пут представио Бентли 1979. године. Моћ ове структуре је у томе што нам омогућава да ефикасно рачунамо неке упите на интревалима низа. Сегментно стабло данаса има широку примену, оно представља једну од најкориснијих структура у такмичарском програмирању, а има примена и у развоју база података и различитих апликација.

Покушаћу да објасним како функционише сегментно стабло и неке основне идеје које се користе у такмичарском програмирању везане за ову структуру.

У раду ће прво бити приказана структура сегментог стабла, а након тога неке напредне технике и врсте. За крају сам оставио три задатка која сам радио током припрема за такмичење. Идеја је да они не буду претешки, али да на довољно добрар начин прикажу моћ и разноврсност ове структуре.

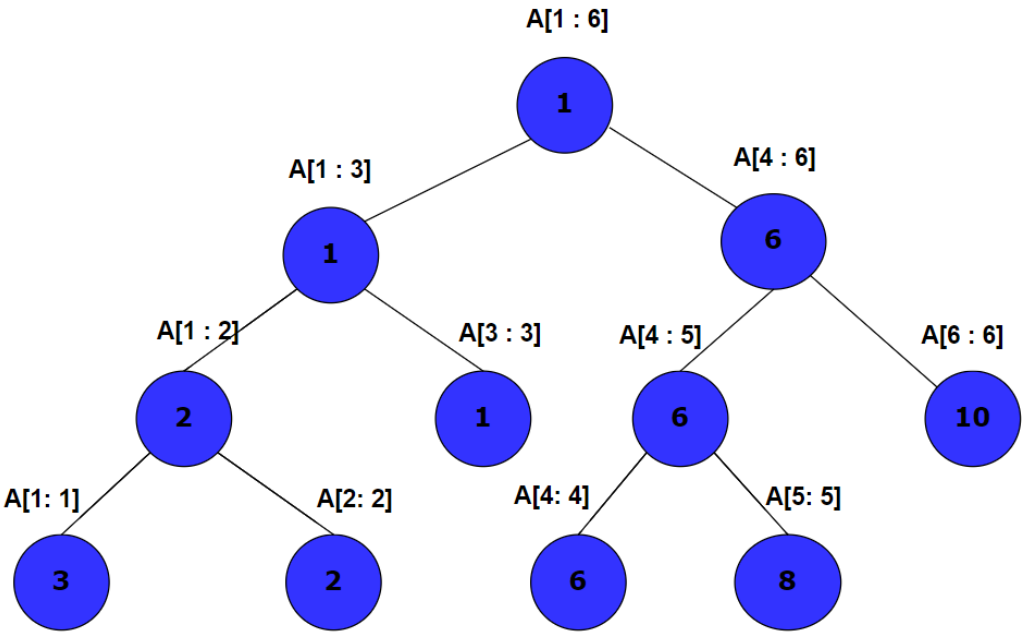
Сви кодови у раду су у програмском језику C++.

2. Структура сегментног стабла

2.1. Основне особине

Дефиницаја 1: Бинарно стабло је стабло чији сваки чвор има највише 2 детета.

Сегментно стабло представља врсту бинарног стабла, где сваки чвор представља одређени интервал низа. Стабло је изграђено тако да ако тренутни чвор представља интервал до l до r (l различито од r), онда његова деца представљају интревале [l, (l + r) / 2], [(l + r) / 2 + 1, r]. Чворови код којих је l једнако r немају деце и они се називају листови. Корен стабла представља цео низ. Пошто је ово стабло и бинарно, можемо га представити у низу. Ако посматрамо чвор са индексом i у низу, онда његова деца имају индексе 2 * i и 2 * i + 1. Ако низ који посматрамо има дужину n, онда се обично за дужину низа који представља стабло узима 4 * n.



За даљи опис структуре, посматраћемо проблем у коме се тражи да на одређеном интервалу нађемо суму елемената низа.

Генерално решавање проблема овом структуром се дели на три функције:

1. Изградња стабла (build)
2. Мењање елемената стабла (update)
3. Решавање самог упита (get_sum)

2.2 Изградња стабла

Овај део се као и већина ствари везаних за сегментно стабло решава рекурзивно. Нека чвор који посматрамо представља интервал од l до r (l је различито од r), тада његова деца престављају интервале [l, (l + r) / 2] и [(l + r) / 2 + 1, r]. Циљ нам је да у овом чвору имамо суму свих елемената од l до r. Ако у чворовима деце држимо одговарајуће суме, онда вредност у почетном чвору можемо добити као суму вредности у чворовима деце. Сада нам само остаје тривијалан случај када се границе интервала посматрају. Тада је вредност у том чвору једнака елемену низа на тој позицији. Сложеност овога је O(n).

```
void build(int node, int l, int r)
{
   if(l==r)
   {
      seg[node] = niz[l];
      return;
   }
   int mid = (l+r)/2;
   build(node*2, l, mid);
   build(node*2+1, mid+1, r);
   seg[node] = seg[node*2] + seg[node*2+1];
}
```

2.3. Мењање елемента низа

Идеја је врло слична грађењу самог стабла. Једина разлика је што овог пута треба да обиђемо само чворове који садрже индекс који ажурирамо, у интервалу који представљају.

```
void update(int node, int l, int r, int pos, int x)
{
   if(l==r)
```

```
    {
        niz[l] = x;
        return;
    }
    int mid = (l+r)/2;
    if(pos>mid) update(node*2+1, mid+1, r, pos, x);
    else update(node*2, l, mid, pos, x);
    niz[node] = niz[node*2] + niz[node*2+1];
}
```

2.4. Сума на интервалу

Претпоставимо да желимо да нађемо суму елемената на интервалу од l до r. Ако тренутно посматрамо чвор који представља интервал од L до R, имамо три могућности:

1. Интервали [L, R] и [l, r] су дисјунктни. У овом случају треба врати нулу.
2. Интервал [L, R] је потпуно садржан у [l, r]. Овде треба вратити вредност у посматраном чвору.
3. Интервали [L, R] и [l, r] се секу. Сада можемо интервал [L, R] поделити на [L, (L+ R) / 2] и [(L + R) / 2 + 1, R] и решавати одвојено њих. На крају, као резлтат треба вратити суму решења за ове две половине.

```
int get_sum(int node, int L, int R, int l, int r)
{
    if(L>r||R<l) return 0;
    if(L>=l&&R<=r) return niz[node];
    int mid = (L+R)/2;
    return get_sum(node*2, L, mid, l, r) + get_sum(node*2+1, mid+1, R, l, r);
}
```

2.5. Анализа сложености

Дефиниција 2: Дубина чвора у стабу представља његово растојање до корена стабла.

Лема 1: Ако посматрамо низ дужине n, онда је максимална дубина чвора у сегментном стаблу $\lceil \log_2 n \rceil$.

Доказ: Стим да свако дете чвор представља половину родитељевог интервала, након $\lceil \log_2 n \rceil$ дељења, максимална дужина интервала који чвор може да представља је $\frac{n}{2^{\lceil \log_2 n \rceil}} \geq 1$, што значи да је даље дељење непотребно.

Лема 2: Максималан број посећених чворова на истом нивоу током упита је четири.

Доказ: Доказ ћемо урадити индукцијом по дубини нивоа.

Базни случај: Ако је посматрана дубина један, стим да је само корен стабла на дубини један, свакако ћемо обићи мање од четири чвора на овом нивоу.

Индуктивна хипотеза: Претоставимо да смо у к-том нивоу посетили максимлано четири чвора. Посматрајмо два случаја:

1. Број посећених чворова на к-том нивоу је један или два. Онда ће број посећених чворова на (к+1)-вом нивоу бити до четири јер сваки од чворова посећених на к-том нивоу посећују до два чвора на (к+1)-вом, па је максимум четири.
2. Број посећених чворова на к-том нивоу је три или четири. Тада чворови у "средини" морају потпуно бити подинтервали интервала на коме тражимо упит, јер иначе би интервал на коме тражимо суму био са прекидном. Дакле, из њих можемо само вратити вредност коју садрже. Остају нам само чворови лево и десно. Како их је двоје, они могу да рекурзивно обраде максимало четири чвора.

Сада, из базе и идуктивне претпоставке следи да је максималан број обрађених чворова по нивоу четири.

Како је број нивоа до $\lceil \log_2 n \rceil$ и број посећених чворова по нивоу максимално четири, видимо да је максимални број операција по упиту 4 * $\lceil \log_2 n \rceil$, односно сложеност упита је O($\log_2 n$).

3. Различите врсте сегментних стабала и технике над њима

3.1 Лења пропагација

До сада смо видели начин да мењамо само један елемент у низу, али сегментно стабло нам може омогућити мењање и целих интервала, тачније повећавање или смањивање сваког елемента на интервалу за одређену вредност. Наивно, ово бисмо могли да урадимо у сложеност O(дужина интервала * $\log_2 n$), тако што бисмо мењали сваки елемент посебно. Како бисмо и ово могли да урадимо у O($\log_2 n$), потребно нам је још једно стабло у коме ћемо у чворовима чувати колико треба додати на сваки чвор у подстаблу посматраног чвора.

Да бисмо добили оптималу сложеност, идеја је да не ажурирамо цело стабло одмах, већ тек у моменту када нам одређени чвор затреба. Претпоставимо да желимо да на интервалу [l, r] додамо свим елементима x. Поново ћемо се кретати од корена стабла ка деци. Ако посматрамо неки чвор који нам представља интерал [L, R] имаћемо три случаја:

1. [L, R] и [l, r] су дисјунктни. Тада не треба ништа урадити.
2. [L, R] је подскуп [l, r]. Тада можемо само повећати вредност у обичном стаблу за (R – L + 1) * x. Оно што треба урадити у лењом стаблу је да децу овог чвора (ако постоје) повећамо за x, јер су и ти одговарјући интервали подскупи [l, r].
3. Ако се [L, R] и [l, r] секу, али [L, R] није подскуп [l, r]. Тада само треба рекурзивно позвати функцију за његову децу, па након њиховог извршења ажурирати тренутни чвор.

```
void update(int node, int L, int R, int l, int r, int x)
{
   if(lazy[node])
   {
      seg[node] += (R-L+1)*lazy[node];
      if(L!=R)
      {
         lazy[node*2] += lazy[node];
         lazy[node*2+1] += lazy[node];
      }
      lazy[node] = 0;
   }
   if(L>r||R<l) return;
```

```
    if(L>=l&&R<=r)
    {
        seg[node] += (R-L+1)*x;
        if(L!=R)
        {
            lazy[node*2] += x;
            lazy[node*2+1] += x;
        }
        return;
    }
    int mid = (L+R)/2;
    update(node*2, L, mid, l, r, x);
    update(node*2+1, mid+1, R, l, r, x);
    seg[node] = seg[node*2] + seg[node*2+1];
}
```

Када желимо да добијемо суму на неком интервалу, радићемо скоро исто као кад немамо лењу пропагацију. Разлика која се јавља је то што сада када посматрамо неки чвор, прво ћемо помоћу лењог стабла проверити да ли тај чвор треба да се ажурира.

```
int get_sum(int node, int L, int R, int l, int r)
{
    if(lazy[node])
    {
        seg[node] = (R-L+1)*lazy[node];
        if(L!=R)
        {
            int mid = (L+R)/2;
            lazy[node*2] += lazy[node];
            lazy[node*2+1] += lazy[node];
```

```
        }
        lazy[node] = 0;
    }
    if(L>r||R<l) return 0;
    if(L>=l&&R<=r) return seg[node];
    int mid = (L+R)/2;
    return get_sum(node*2, L, mid, l, r) + get_sum(node*2+1, mid+1, R, l, r);d
}
```

3.2. Шетња по сегментном стаблу

Размотримо проблем проналажења к-те нуле низу. Ако посматрамо сегментно стабло где бисмо чували број нула у одговарајућим интервалима, онда бинарном претрагом можемо наћи одговор. Сложеност овога би била O($(\log_2 n)^2$). Ефикасност можемо поправити тако што ћемо ићи кроз стабло и симулирати бинарну претрагу. Када се налазимо у корену, можемо видети колико имамо нула у првој половини. Ако их је x, онда у случају да је x веће или једнако к, само можемо рекурзивно позвати функцију која тражи к-ту нулу у првој половини. Са друге стране, ако је x мање од к, можемо рекурзивно тражити (к-x)-ту нулу у другој половини низа.

3.3. Перзистентно стабло

Перзистентно сегментно стабло чува историју стабла. Практично, оно нам омогућава да и након одређених ажурирања знамо какво је стабло било пре. Идеја је у томе што када мењамо неки елемент мењамо само чворове на путу од његовог листа, до корена, односно O($\log_2 n$) чворова. Онда, можемо само да сваки пут направимо потребан број нових чворова, а остатак стабла “накачимо” на њих.

Најпознатији проблем код перзистентног стабла је да се нађе к-ти по величини број на интервалу. За почетак, нека су сви елементи мањи или једнаки величини низа. Можемо направити перзистентно стабло где ако чвор представља интервал [L, R], онда ћемо у њему чувати колико има елемената који су већи или једнаки L, а мањи или једнаки R. Претпоставимо да тражимо к-ти елемент по величини на интервалу [0l, r]. Ако су ми ажурирања перзистентног стабла заправо додавање елемената низа са лева на десно, онда одузимањем вредности чвора након r-тог и (l-1)-ог додавања, добијамо бројаче само за тражени интервал. Сада к-ти елемент по величини можемо наћи помоћу шетње. Ако елементи могу бити већи од дужине низа, прво можемо урадити компресију елемената, па онда урадити исти поступак као у првом случају.

3.4. Имплицитно стабло

Некада су елементи јако велики, па није могуће направити цело стабло. То можемо решити коришћењем имплицитног сегментног стабла. Идеја је, као и код перзистентног, да не правимо све чворове, већ само оне који су нам потребни, а остатак стабла можемо "слепити". Сложеност ће овог пута бити O($\log_2 x$), где је x елемент који додајемо. Свакако, то је и даље јако ефиксано стим да је за x = 10^9, број операција приближно 30.

4. Примери

Волео бих да кроз наредна три примера покажем колика је примена сегментног стабла у такмичарском програмирању и истовременом укажем на задатке који су мени били интересантни.

4.1. Watering can, Пољска олимпијада 2012/2013, фаза друга, дан други, задатак први.

Имамо низ бројева, број к, и 2 типа упита:

1. Повећати све елементе низа на интервалу за један.
2. Одредити колико бројева на интервалу је веће или једнако к.

Овај провлем можемо једноставно решити у O(n^2), где је n величина низа, тако што ћемо само симулирати промене. Оптималну сложеност можемо добити коришћењем 2 сегмента стабла, једно класично за добијање суме, а друго стабло у коме чувамо максимуме на интервалима. На почетку, у сегментном стаблу са сумом, у листовима стабла ће бити 0 ако је одговарајући елемент мањи од к, а један у супротном. Дакле, ако бисмо успели да након сваког упита типа 1 имали овакво стабло, сумом на интервалу бисмо могли добити колико елемената је веће или једнако к. То можемо достићи тако што ћемо радити лењу пропагацију на стаблу са максимум, и ако неки чвор постане већи или једнак к, можемо ићи по стаблу и наћи који је тај елемент. Када га пронађемо, на његово место поставићемо неку вредност која ће нам представљати минус бесконачно, како не бисмо поново дошли у ситуацију да тај елемент дође до к, а у стабло са сумом поставићемо тај елемент на један. Стим да ћемо стабло обилазити само када је неки елемент већи или једнак к, а за промену тог елемента на минус бесконачно нам треба O($\log_2 n$) промена на чворовима, сложеност овога ће бити O(n$\log_2 n$ + q$\log_2 n$), где је q број упита.

4.2. Патуљци, Хрватско отворено такмичење из програмирања '09, треће такмичење, пети задатак.

Дат је низ бројева дужине n, и дато је q интервала. Ако је неки интервал од l до r, проверити да ли се неки елемент налази бар (r – l + 1) / 2 пута, и ако да који.

Овај проблем се може решити и рандомизацијом, тако што ћемо уз довољан број погађања елемента вероватноћу да не погодимо решење свести на минимум, али коришћењем сегментог стабла ипак можемо доћи до мало бржег решења. Приметимо да ако се неки елемент налази више од половине пута у низу, онда ако бисмо избацивали по 2 различита елемента из низа, на крају би остао само тај елемент. То нас доводи до тога да можемо да правимо сегментно стабло у коме у чворовима ћемо чувати "победнике" ове игре на одговарајућим интервалима, и колико је тих елемената остало. Сада када посматрамо одређени интервал, можемо помоћу стабла добити потенцијалног победника ове игре. Када га добијемо, треба још проверити да ли се он заиста налази добољан број пута у том интервалу. То можемо урадити тако што ћемо за сваку вредност елемента имати низ у коме ћемо чувати позиције његовог појављивања и онда бинарном претрагом проверити да ли је то заиста решење. Слошеност решења је O(q$\log_2 n$).

4.3. Sterilizing spray, JOI 15

Дат је низ елемената дужине n, q упита и број к.

Упити су типа:

1. Постави елемент на позицији l, на вредност x.
2. Целобројно подели све елементе на интервалу [l, r] са к.
3. Исписати суму елемената на интервалу [l, r].

Овај задатак можемо лако решити у O(n^2), симулирајући промене. Идеја за оптимално решење је та да ће се елементи низа дељењем са к брзо смањити до нуле. Дакле, можемо чувати сет "живих" елемената и други упит решити тако што ћемо проћи кроз елементе на траженом интервалу и поделити их. Такође, имаћемо обично сегментно стабло са сумом, којим ћемо решавати трећи упит. За први упит, само ћемо променити вредност у сегментном и додати елемент у сет, ако већ није. Сложеност овог решења је O(q$\log_2 n + c * n$), где је с нека мала константа.

```
#include <bits/stdc++.h>

using namespace std;

#define ll long long

#define endl "\n"

#define INF 1000000000

#define LINF 1000000000000000LL

#define pb push_back

#define all(x) x.begin(), x.end()

#define len(s) (int)s.size()
```

```
#define test_case { int t; cin>>t; while(t--)solve(); }
#define input(n, v) {for(int i = 0;i<n;i++) cin>>v[i];}
#define output(n, v) {for(int i = 0;i<n;i++) cout<<v[i]<<" "; cout<<endl;}
#define single_case solve();
#define line cout<<"------------"<<endl;
#define ios { ios_base::sync_with_stdio(false); cin.tie(NULL); }
using namespace std;
int n, q, k;
const int N = 1e5 + 5;
ll sum[4*N];
vector<ll> a(N);

ll get_sum(int node, int l, int r, int i, int j)
{
   if(i>r||j<l) return 0;
   if(i>=l&&j<=r) return sum[node];
   int mid = (i+j)/2;
   return get_sum(node*2+1, l, r, i, mid) + get_sum(node*2+2, l, r, mid+1, j);
}

void update(int node, int l, int r, int pos, int x)
{
   if(l==r)
   {
      sum[node] = x;
      return;
   }
```

```
    int mid = (l+r)/2;
    if(pos>mid)
        update(node*2+2, mid+1, r, pos, x);
    else
        update(node*2+1, l, mid, pos, x);
    sum[node] = sum[node*2+1] + sum[node*2+2];
}

ll build_sum(int node, int l, int r)
{
    if(l==r)
    {
      return sum[node] = a[l];
    }
    int mid = (l+r)/2;
    return sum[node] = build_sum(node*2+1, l, mid) + build_sum(node*2+2, mid+1, r);
}



int main()
{
    ios
    cin>>n>>q>>k;
    for(int i = 0;i<n;i++)
        cin>>a[i];
    set<ll> alive;
    for(ll i = 0;i<n;i++)
```

```
    if(a[i])
    alive.insert(i);
build_sum(0, 0, n-1);
while(q--)
{
    ll s, t, u;
    cin>>s>>t>>u;
    t--;
    u--;
    if(s==1)
    {
        // a-ti tanjir na b
        u++;
        if(!a[t]&&u)
        alive.insert(t);
        if(a[t]&&!u)
            alive.erase(t);
        a[t] = u;
        update(0, 0, n-1, t, u);
    }
    else if(s==2)
    {
        // sprej
        if(k==1) continue;
        vector<int> s;
        auto w = alive.lower_bound(t);
        while(w!=alive.end())
```

```
        {
            if((*w)>u) break;
            int e = *w;
            a[e]/=k;
            if(!a[e]) s.pb(e);
            update(0, 0, n-1, e, a[e]);
            w++;
        }
        for(int x : s)
            alive.erase(x);
    }
    else{
        // ispisi sumu od l do r
        cout<<get_sum(0, t, u, 0, n-1);
            cout<<endl;
    }
  }

  return 0;
}
```

5. Закључак

Овај рад је описао основе сегментог стабла које су данас неопходне за успешно такмичење. Надам да ћу некога успети да мотивишем да уђе у свет такмичарског програмирања и да ће читањем рада моћи нешто ново и да се научи. Неке ствари нису толико детаљно описане, јер је идеја била да се прикажу само основне и занимљиве ствари како би се тема што више приближила читаоцу.

На крају, волео бих да се захвалим професорки Снежани Јелић на помоћи око израде матурског рада, а и због тога што ми је помогла да увидим лепоту програмирања и лакше одаберем своју будућу професију. Захвалио бих се и Младену Пузићу на помоћи у избору задатака.

6. Литература

1. Давид Милићевић, Сегментно и фенвиково стабло и њихове примене у такмичарским задацима, https://www.mg.edu.rs/uploads/files/images/stories/dokumenta/maturski/david-milicevic.pdf
2. Segment tree, https://cp-algorithms.com/data_structures/segment_tree.html