**[Algorithm Gym Segment Trees](http://codeforces.com/blog/entry/15890)**

**Classic Segment Tree**

Classic, is the way I call it. This type of segment tree, is the most simple and common type. In this kind of segment trees, for each node, we should keep some simple elements, like integers or boolians or etc.

This kind of problems don't have update queries on intervals.

Example 1 (Online): Problem [380C - Sereja and Brackets](http://codeforces.com/contest/380/problem/C) :

For each node (for example *x*), we keep three integers : 1.t[x] = Answer for it's interval. 2. o[x] = The number of $($s after deleting the brackets who belong to the correct bracket sequence in this interval whit length t[x]. 3. c[x] = The number of $)$s after deleting the brackets who belong to the correct bracket sequence in this interval whit length t[x].

Lemma : For merging to nodes 2*x* and 2*x* + 1 (children of node 2*x* + 1) all we need to do is this :

*tmp = min(o[2 \* x], c[2 \* x + 1])*

*t[x] = t[2 \* x] + t[2 \* x + 1] + tmp*

*o[x] = o[2 \* x] + o[2 \* x + 1] - tmp*

*c[x] = c[2 \* x] + c[2 \* x + 1] - tmp*

So, as you know, first of all we need a *build* function which would be this : (as above) (C++ and [*l*, *r*) is inclusive-outclusive )

*void build(int id = 1,int l = 0,int r = n){*

*if(r - l < 2){*

*if(s[l] == '(')*

*o[id] = 1;*

*else*

*c[id] = 1;*

*return ;*

*}*

*int mid = (l+r)/2;*

*build(2 \* id,l,mid);*

*build(2 \* id + 1,mid,r);*

*int tmp = min(o[2 \* id],c[2 \* id + 1]);*

*t[id] = t[2 \* id] + t[2 \* id + 1] + tmp;*

*o[id] = o[2 \* id] + o[2 \* id + 1] - tmp;*

*c[id] = c[2 \* id] + c[2 \* id + 1] - tmp;*

*}*

For queries, return value of the function should be 3 values : *t*, *o*, *c* which is the values I said above for the intersection of the node's interval and the query's interval (we consider query's interval is [*x*, *y*) ), so in C++ code, return value is a pair<int,pair<int,int> >(pair<t, pair<o,c> >) :

*typedef pair<int,int>pii;*

*typedef pair<int,pii> node;*

*node segment(int x,int y,int id = 1,int l = 0,int r = n){*

*if(l >= y || x >= r)*

*return node(0,pii(0,0));*

*if(x <= l && r <= y)*

*return node(t[id],pii(o[id],c[id]));*

*int mid = (l+r)/2;*

*node a = segment(x,y,2 \* id,l,mid), b = segment(x,y,2 \* id + 1,mid,r);*

*int T, temp, O, C;*

*temp = min(a.y.x , b.y.y);*

*T = a.x + b.x + temp;*

*O = a.y.x + b.y.x - temp;*

*C = a.y.y + b.y.y - temp;*

*return node(T,pii(O,C));*

*}*

Example 2 (Offline): Problem [KQUERY](http://www.spoj.com/problems/KQUERY/)

Imagine we have an array *b*1, *b*2, ..., *bn* which,  and *bi* = 1 if an only if *ai* > *k*, then we can easily answer the query (*i*, *j*, *k*) in*O*(*log*(*n*)) using a simple segment tree (answer is *bi* + *bi*+ 1 + ... + *bj* ). We can do this ! We can answer the queries offline.

First of all, read all the queries and save them somewhere, then sort them in increasing order of *k* and also the array *a* in increasing order (compute the permutation  *p*1, *p*2, ..., *pn* where *ap*1 ≤ *ap*2 ≤ ... ≤ *apn* )

At first we'll set all array *b* to 1 and we will set all of them to 0 one by one. Consider after sorting the queries in increasing order of their *k*, we have a permutation *w*1, *w*2, ..., *wq* (of 1, 2, ..., *q*) where*kw*1 ≤ *kw*2 ≤ *kw*2 ≤ ... ≤ *kwq* (we keep the answer to the *i* - *th* query in *ansi* .

Pseudo code : (all 0-based)

*po = 0*

*for j = 0 to q-1*

*while po < n and a[p[po]] <= k[w[j]]*

*b[p[po]] = 0, po = po + 1*

So, build function would be like this (*s*[*x*] is the sum of *b* in the interval of node *x*) :

*void build(int id = 1,int l = 0,int r = n){*

*if(r - l < 2){*

*s[id] = 1;*

*return ;*

*}*

*int mid = (l+r)/2;*

*build(2 \* id, l, mid);*

*build(2 \* id + 1, mid, r);*

*s[id] = s[2 \* id] + s[2 \* id + 1];*

*}*

et An update function for when we want to st b[p[po]] = 0 to update the segment tree:

*void update(int p,int id = 1,int l = 0,int r = n){*

*if(r - l < 2){*

*s[id] = 0;*

*return ;*

*}*

*int mid = (l+r)/2;*

*if(p < mid)*

*update(p, 2 \* id, l, mid);*

*else*

*update(p, 2 \* id + 1, mid, r);*

*s[id] = s[2 \* id] + s[2 \* id + 1];*

*}*

Finally, a function for sum of an interval

*int sum(int x,int y,int id = 1,int l = 0,int r = n){* // [x, y)

*if(x >= r or l >= y)*

*return 0;* // [x, y) intersection [l,r) = empty

*if(x <= l && r <= y)* // [l,r) is a subset of [x,y)

*return s[id];*

*int mid = (l + r)/2;*

*return sum(x, y, id \* 2, l, mid) +*

*sum(x, y, id\*2+1, mid, r) ;*

*}*

So, in main function instead of that pseudo code, we will use this :

*build();*

*int po = 0;*

*for(int y = 0;y < q;++ y){*

*int x = w[y];*

*while(po < n && a[p[po]] <= k[x])*

*update(p[po ++]);*

*ans[x] = sum(i[x], j[x] + 1); // the interval [i[x], j[x] + 1)*

*}*

**Lazy Propagation**

I told you enough about lazy propagation in the last lecture. In this lecture, I want to solve ans example .

Example : Problem [POSTERS](http://www.spoj.com/problems/POSTERS/).

We don't need all elements in the interval [1, 107]. The only thing we need is the set *s*1, *s*2, ..., *sk* where for each *i*, *si* is at least *l* or *r* in one of the queries. We can use interval 1, 2, ..., *k* instead of that (each query is running in this interval, in code, we use 0-based, I mean [0, *k*) ). For the *i* - *th*query, we will paint all the interval [*l*, *r*] whit color *i* (1-based). For each interval, if all it's interval is from the same color, I will keep that color for it and update the nodes using lazy propagation. So,we will have a value *lazy* for each node and there is no any build function (if *lazy*[*i*] ≠ 0 then all the interval of node *i* is from the same color (color *lazy*[*i*]) and we haven't yet shifted the updates to its children. Every member of *lazy* is 0 at first).

A function for shifting the updates to a node, to its children using lazy propagation :

*void shift(int id){*

*if(lazy[id])*

*lazy[2 \* is] = lazy[2 \* id + 1] = lazy[id];*

*lazy[id] = 0;*

*}*

Update (paint) function (for queries) :

//painting the interval [x,y) whith color "color"

*void upd(int x,int y,int color, int id = 0,int l = 0,int r = n){*

*if(x >= r or l >= y)*

*return ;*

*if(x <= l && r <= y){*

*lazy[id] = color;*

*return ;*

*}*

*int mid = (l+r)/2;*

*shift(id);*

*upd(x, y, color, 2 \* id, l, mid);*

*upd(x, y, color, 2\*id+1, mid, r);*

*}*

So, for each query you should call *upd*(*x*, *y* + 1, *i*) (*i* is the query's 1-base index) where *sx* = *l* and *sy* = *r* . At last, for counting the number of different colors (posters), we run the code below (it's obvious that it's correct) :

*set <int> se;*

*void cnt(int id = 1,int l = 0,int r = n){*

*if(lazy[id]){*

*se.insert(lazy[id]);*

//there is no need to see the children,because all the interval is from the same color

*return ;*

*}*

*if(r - l < 2)*

*return ;*

*int mid = (l+r)/2;*

*cnt(2 \* id, l, mid);*

*cnt(2\*id+1, mid, r);*

*}*

And answer will be se.size() .

**Segment tree with vectors**

In this type of segment tree, for each node we have a vector (we may also have some other variables beside this) .

Example : Online approach for problem [KQUERYO](http://www.spoj.com/problems/KQUERYO/) (I added this problem as the online version of KQUERY):

It will be nice if for each node, with interval [*l*, *r*) such that *i* ≤ *l* ≤ *r* ≤ *j* + 1 and this interval is maximal (it's parent's interval is not in the interval [*i*, *j* + 1) ), we can count the answer.

For that propose, we can keep all elements of *al*, *al*+ 1, ..., *ar* in increasing order and use binary search for counting. So, memory will be*O*(*n*.*log*(*n*)) (each element is in *O*(*log*(*n*)) nodes ). We keep this sorted elements in verctor *v*[*i*] for *i* - *th* node. Also, we don't need to run sort on all node's vectors, for node *i*, we can merge *v*[2 \* *i*] and *v*[2 \* *id* + 1] (like merge sort) .

So, build function is like below :

*void build(int id = 1,int l = 0,int r = n){*

*if(r - l < 2){*

*v[id].push\_back(a[l]);*

*return ;*

*}*

*int mid = (l+r)/2;*

*build(2 \* id, l, mid);*

*build(2\*id+1, mid, r);*

*merge(v[2 \* id].begin(), v[2 \* id].end(), v[2 \* id + 1].begin(), v[2 \* id + 1].end(), back\_inserter(v[id]));*

// read more about back\_inserter in http://www.cplusplus.com/reference/iterator/back\_inserter/

*}*

And function for solving queries :

// solve the query (x,y-1,k)

*int cnt(int x,int y,int k,int id = 1,int l = 0,int r = n){*

*if(x >= r or l >= y)*

*return 0;*

*if(x <= l && r <= n)*

*return v[id].size() - (upper\_bound(v[id].begin(), v[id].end(), k) - v[id].begin());*

*int mid = (l+r)/2;*

*return cnt(x, y, k, 2 \* id, l, mid) +*

*cnt(x, y, k, 2\*id+1, mid, r) ;*

*}*

Another example : [Component Tree](http://codeforces.com/gym/100513/problem/C)

Segment tree with sets

In this type of segment tree, for each node we have a set or multiset or hash\_map ([here](http://codeforces.com/blog/entry/15869)) or unorderd\_map or etc (we may also have some other variables beside this) .

Consider this problem :

We have *n* vectors, *a*1, *a*2, ..., *an* and all of them are initially empty. We should perform *m* queries on this vectors of two types :

1. *A* *p* *k* Add number *k*at the end of *ap*
2. *C* *l* *r* *k* print the number  where *count*(*ai*, *k*) is the number of occurrences of *k* in *ai* .

For this problem, we use a segment tree where each node has a multiset, node *i* with interval [*l*, *r*) has a multiset *s*[*i*] that contains each number *k* exactly  times (memory would be *O*(*q*.*log*(*n*)) ) .

For answer query *C* *x* *y* *k*, we will print the sum of all *sx*.*count*(*k*) where if the interval of node *x* is [*l*, *r*), *x* ≤ *l* ≤ *r* ≤ *y* + 1 and its maximal (its parent doesn't fulfill this condition) .

We have no build function (because vectors are initially empty). But we need an add function :

*void add(int p,int k,int id = 1,int l = 0,int r = n){* // perform query A p k

*s[id].insert(k);*

*if(r - l < 2) return ;*

*int mid = (l+r)/2;*

*if(p < mid)*

*add(p, k, id \* 2, l, mid);*

*else*

*add(p, k, id\*2+1, mid, r);*

*}*

And the function for the second query is :

*int ask(int x,int y,int k,int id = 1,int l = 0,int r = n){* // Answer query C x y-1 k

*if(x >= r or l >= y) return 0;*

*if(x <= l && r <= y)*

*return s[id].count(k);*

*int mid = (l+r)/2;*

*return ask(x, y, k, 2 \* id, l, mid) +*

*ask(x, y, k, 2\*id+1, mid, r) ;*

*}*

**Segment tree with other data structures in each node**

From now, all the other types of segments, are like the types above.

**2*D* Segment trees**

In this type of segment tree, for each node we have another segment tree (we may also have some other variables beside this) .

**Segment trees with tries**

In this type of segment tree, for each node we have a trie (we may also have some other variables beside this) .

**Segment trees with DSU**

In this type of segment tree, for each node we have a disjoint set (we may also have some other variables beside this) .

Example : Problem [76A - Gift](http://codeforces.com/contest/76/problem/A), you can read my source code ([8613428](http://codeforces.com/contest/76/submission/8613428)) with this type of segment trees .

**Segment trees with Fenwick**

In this type of segment tree, for each node we have a Fenwick (we may also have some other variables beside this) . Example :

Consider this problem :

We have *n* vectors, *a*1, *a*2, ..., *an* and all of them are initially empty. We should perform *m* queries on this vectors of two types :

1. *A* *p* *k* Add number *k*at the end of *ap*
2. *C* *l* *r* *k* print the number  for each *j* ≤ *k* where *count*(*ai*, *k*) is the number of occurrences of *k* in *ai* .

For this problem, we use a segment tree where each node has a vector, node *i* with interval [*l*, *r*) has a set *v*[*i*] that contains each number *k* if and only if  (memory would be *O*(*q*.*log*(*n*)) ) (in increasing order).

First of all, we will read all queries, store them and for each query of type *A*, we will insert *k* in *v* for all nodes that contain *p* (and after all of them, we sort these vectors using merge sort and run unique function to delete repeated elements) .

Then, for each node *i*, we build a vector *fen*[*i*] with size |*s*[*i*]| (initially 0).

Insert function :

*void insert(int p,int k,int id = 1,int l = 0,int r = n){* //perform query A p k

*if(r - l < 2){*

*v[id].push\_back(k);*

*return ;*

*}*

*int mid = (l+r)/2;*

*if(p < mid)*

*insert(p, k, id \* 2, l, mid);*

*else*

*insert(p, k, id\*2+1, mid, r);*

*}*

Sort function (after reading all queries) :

*void SORT(int id = 1,int l = 0,int r = n){*

*if(r - l < 2)*

*return ;*

*int mid = (l+r)/2;*

*SORT(2 \* id, l, mid);*

*SORT(2\*id+1, mid, r);*

*merge(v[2 \* id].begin(), v[2 \* id].end(), v[2 \* id + 1].begin(), v[2 \* id + 1].end(), back\_inserter(v[id]));*

*v[id].resize(unique(v[id].begin(), v[id].end()) - v[id].begin());*

*fen[id] = vector<int> (v[id].size() + 1, 0);*

*}*

Then for all queries of type *A*, for each node *x* containing *p* we will run :

*for(int i = a + 1;i < fen[x].size(); i += i & -i)*

*fen[x][i] ++;*

Where *v*[*x*][*a*] = *k* . Code :

*void upd(int p,int k, int id = 1,int l = 0,int r = n){*

*int a = lower\_bound(v[id].begin(), v[id].end(), k) - v[id].begin();*

*for(int i = a + 1; i < fen[id].size(); i += i & -i )*

*fen[id][i] ++ ;*

*if(r - l < 2) return;*

*int mid = (l+r)/2;*

*if(p < mid)*

*upd(p, k, 2 \* id, l, mid);*

*else*

*upd(p, k, 2\*id+1, mid, r);*

*}*

And now we can easily compute the answer for queries of type *C* :

*int ask(int x,int y,int k,int id = 1,int l = 0,int r = n){// Answer query C x y-1 k*

*if(x >= r or l >= y) return 0;*

*if(x <= l && r <= y){*

*int a = lower\_bound(v[id].begin(), v[id].end(), k) - v[id].begin();*

*int ans = 0;*

*for(int i = a + 1; i > 0; i -= i & -i)*

*ans += fen[id][i];*

*return ans;*

*}*

*int mid = (l+r)/2;*

*return ask(x, y, k, 2 \* id, l, mid) +*

*ask(x, y, k, 2\*id+1, mid, r) ;*

*}*

**Segment tree on a rooted tree**

As you know, segment tree is for problems with array. So, obviously we should convert the rooted tree into an array. You know DFS algorithm and starting time (the time when we go into a vertex, starting from 1). So, if *sv* is starting time of *v*, element number *sv* (in the segment tree) belongs to the vertex number *v* and if *fv* = *max*(*su*) + 1 where *u* is in subtree of *v*, the interval [*sv*, *fv*) shows the interval of subtree of *v* (in the segment tree) .

Example : Problem [396C - On Changing Tree](http://codeforces.com/contest/396/problem/C)

Consider *hv* height if vertex *v* (distance from root).

For each query of first of type, if *u* is in subtree of *v*, its value increasing by *x* + (*hu* - *hv*) ×  - *k* = *x* + *k*(*hv* - *hu*) = *x* + *k* × *hv* - *k* × *hu*. So for each *u*, if *s* is the set of all queries of first type which *u* is in the subtree of their *v*, answer to query 2 *u* is , so we should calculate two values  and , we can answer the queries. So, we for each query, we can store values in all members of its subtree ( [*sv*, *fv*) ).

So for each node of segment tree, we will have two variables  and  (we don't need lazy propagation, because we only update maximal nodes).

Source code of update function :

*void update(int x,int k,int v,int id = 1,int l = 0,int r = n){*

*if(s[v] >= r or l >= f[v]) return ;*

*if(s[v] <= l && r <= f[v]){*

*hkx[id] = (hkx[id] + x) % mod;*

*int a = (1LL \* h[v] \* k) % mod;*

*hkx[id] = (hkx[id] + a) % mod;*

*sk[id] = (sk[id] + k) % mod;*

*return ;*

*}*

*int mid = (l+r)/2;*

*update(x, k, v, 2 \* id, l, mid);*

*update(x, k, v, 2\*id+1, mid, r);*

*}*

Function for 2nd type query :

*int ask(int v,int id = 1,int l = 0,int r = n){*

*int a = (1LL \* h[v] \* sk[id]) % mod;*

*int ans = (hkx[id] + mod - a) % mod;*

*if (r - l < 2) return ans;*

*int mid = (l+r)/2;*

*if (s[v] < mid)*

*return (ans + ask(v, 2 \* id, l, mid)) % mod;*

*return (ans + ask(v, 2\*id+1, mid, r)) % mod;*

*}*

**Persistent Segment Trees**

In the last lecture, I talked about this type of segment trees, now I just want to solve an important example.

Example : Problem [MKTHNUM](http://www.spoj.com/problems/MKTHNUM/)

First approach : *O*((*n* + *m*).*log*2(*n*))

I won't discuss this approach, it's using binary search an will get TLE.

Second approach : *O*((*n* + *m*).*log*(*n*))

This approach is really important and pretty and too useful :

Sort elements of *a* to compute permutation *p*1, *p*2, ..., *pn* such that *ap*1 ≤ *ap*2 ≤ ... ≤ *apn* and *q*1, *q*2, ..., *qn* where, for each *i*, *pqi* = *i*.

We have an array *b*1, *b*2, ..., *bn* (initially 0) and a persistent segment tree on it.

Then *n* step,for each *i*, starting from 1, we perform *bqi* = 1 .

Lest *sum*(*l*, *r*, *k*) be *bl* + *bl*+ 1 + ... + *br* after *k* - *th* update (if *k* = 0, it equals to 0)

As I said in the last lecture, we have an array *root* and the root of the empty segment tree, *ir* . So for each query *Q*(*x*, *y*, *k*), we need to find the first *i* such that *sum*(1, *i*, *r*) - *sum*(1, *i*, *l* - 1) > *k* - 1 and answer will be *api*. (I'll explain how in the source code) :

Build function (*s* is the sum of the node's interval):

*void build(int id = ir,int l = 0,int r = n){*

*s[id] = 0;*

*if(r - l < 2)*

*return ;*

*int mid = (l+r)/2;*

*L[id] = NEXT\_FREE\_INDEX ++;*

*R[id] = NEXT\_FREE\_INDEX ++;*

*build(L[id], l, mid);*

*build(R[id], mid, r);*

*s[id] = s[L[id]] + s[R[id]];*

*}*

Update function :

*int upd(int p, int v,int id,int l = 0,int r = n){*

*int ID = NEXT\_FREE\_INDEX ++;* // index of the node in new version of segment tree

*s[ID] = s[id] + 1;*

*if(r - l < 2)*

*return ID;*

*int mid = (l+r)/2;*

*L[ID] = L[id], R[ID] = R[id];* // in case of not updating the interval of left child or right child

*if(p < mid)*

*L[ID] = upd(p, v, L[ID], l, mid);*

*else*

*R[ID] = upd(p, v, R[ID], mid, r);*

*return ID;*

*}*

Ask function (it returns *i*, so you should print *api* :

*int ask(int id, int ID, int k, int l = 0,int r = n){*

// id is the index of the node after l-1-th update (or ir) and ID will be its index after r-th update

*if(r - l < 2) return l;*

*int mid = (l+r)/2;*

*if(s[L[ID]] - s[L[id]] >= k)* // answer is in the left child's interval

*return ask(L[id], L[ID], k, l, mid);*

*else*

*return ask(R[id], R[ID], k - (s[L[ID]] - s[L[id]] ), mid, r);*

// there are already s[L[ID]] - s[L[id]] 1s in the left child's interval

*}*

As you can see, this problem is too tricky.