Other notes:

[CP Notes - Data Structures](https://docs.google.com/document/d/1tTXgVgZPAMS8JcBV1RYRxCQg29BpLxbO01GGrl0hfRI/edit#)

[CP Notes - Graph Algorithms](https://docs.google.com/document/d/1SWrZTmICx4TP19zjUqZlj1YU3KfsybsMZRp0aWdHVGY/edit)

[CP Notes - Tree](https://docs.google.com/document/d/1TBky0mONxJ8Tvg7JfI4LHr99fnd710d1KvNG9kIPbyg/edit)

[CP Notes - Dynamic Programming](https://docs.google.com/document/d/1fYKZF_SaslzR01AgfwuduaceRGsHhEtlVoqLY1gGPR4/edit)

[CP Notes - Miscellaneous](https://docs.google.com/document/d/1TOQj29vF6n_bER7s94du4VPDHKgeYP4RhGla3lq37kw/edit)

Legend:

**Bold** = have code below

List of useful algorithm codes to keep in notes:

* Graph Algorithms
  + Shortest Path
    - Dijkstra
    - Floyd Warshall
    - ~~A\* Search~~
  + Trees
    - Preorder (never use post-order before so yah)
    - Lowest Common Ancestor (and hence 2^k decomp)
    - SET MERGING (impt for dp)
    - Heavy-light decomposition
    - Centroid decomposition
    - Euler tour tree?
    - ~~Line tree~~
  + Minimum Spanning Tree
  + Articulation Points
  + Bridges
  + SCC and Conversion to DAG
  + MCBM
  + Floyd’s
* Number Theory and Computation
  + Fast Exponentiation
  + Bignum Operation
  + Base conversion
  + Sieve of eratosthenes
  + Prime factorisation/integer factorisation
* Data Structures
  + UFDS
  + Segment Tree
    - Lazy node
    - Lazy propagation
    - **Persistent**
    - Array segtree?
  + Fenwick Tree
    - RUPQ
    - PURQ
    - RURQ
  + **PBDS**
  + Trie
  + Sparse table
  + Treap
  + ~~K-d tree~~
  + ~~AVL tree~~
  + ~~Red-black tree~~
  + ~~KMP tree~~
* Dynamic Programming
  + Divide and Conquer
  + Convex hull
  + Slope trick
* Miscellaneous
  + Super fast I/O
  + **Mo’s algorithm**
  + Bitwise functions

**Cancer Code:**

Persistent rangemax with lazy propagation segtree:

#include<bits/stdc++.h>

using namespace std;

#define ONE LLONG\_MIN

typedef long long ll;

struct node{

ll v=0,lazy=0;

node \*l=NULL,\*r=NULL;

node(){}

ll query(ll s,ll e,ll a,ll b){

if(a<=s && e<=b)return v;

ll lv=ONE,rv=ONE;

if(a<=((s+e)/2)){//left child involved

if(l==NULL){

lv=lazy;

}else{

lv=l->query(s,(s+e)/2,a,b)+lazy;

}

}

if((s+e)/2<b){

if(r==NULL){

rv=lazy;

}else{

rv=r->query((s+e)/2 + 1,e,a,b)+lazy;

}

}

return max(lv,rv);

}

void update(ll s,ll e,ll a,ll b,ll add){

if(a<=s && e<=b){

if(s!=e)lazy+=add;

v+=add;

return;

}

if(a<=((s+e)/2)){//left child involved

if(l==NULL){

l=new node();

}

l->update(s,(s+e)/2,a,b,add);

}

if((s+e)/2<b){

if(r==NULL){

r=new node();

}

r->update((s+e)/2 + 1,e,a,b,add);

}

v=ONE;

if(l!=NULL)v=max(v,l->v);

else v=0ll;

if(r!=NULL)v=max(v,r->v);

else v=max(v,0ll);

v+=lazy;

}

void copy(node\* n,ll s,ll e,ll a,ll b,ll add){

v=n->v;lazy=n->lazy;

l=n->l;r=n->r;

if(a<=s && e<=b){

if(s!=e)lazy+=add;

v+=add;

return;

}

if(a<=((s+e)/2)){//left child involved

if(l==NULL){

l=new node();

l->update(s,(s+e)/2,a,b,add);

}else{

l=new node();

l->copy(n->l,s,(s+e)/2,a,b,add);

}

}

if((s+e)/2<b){

if(r==NULL){

r=new node();

r->update((s+e)/2 + 1,e,a,b,add);

}else{

r=new node();

r->copy(n->r,(s+e)/2 + 1,e,a,b,add);

}

}

v=ONE;

if(l!=NULL)v=max(v,l->v);

else v=0ll;

if(r!=NULL)v=max(v,r->v);

else v=max(v,0ll);

v+=lazy;

}

} \*st[100005];//1 root node for each version...

void init(int L, int M, int Q){st[0]=new node();}//meh...

void proposal(int n, int k, int a, int b, int c){

st[n]=new node();

st[n]->copy(st[k],0,1e9,a,b,c);

}

long long max\_height(int P, int X, int Y) {

return st[P]->query(0,1e9,X,Y);

}

Template for Mo’s:

void addl(int x) {

// remove leftmost element (index at x)

}

void addr(int x) {

// add rightmost element (index at x)

}

void reml(int x) {

// remove leftmost element (index at x)

}

void remr(int x) {

// remove rightmost element (index at x)

}

/\*

Comparator

auto lp = make\_pair(lhs.first / S, (lhs.first / S & 1) ? -lhs.second : lhs.second), rp = make\_pair(rhs.first / S, (rhs.first / S & 1) ? -rhs.second : rhs.second);

return lp < rp;

\*/

main() {

for (int i = L[1]; i <= R[1]; i++) addr(i);

for (int i = 2; i <= Q; i++)

if (R[i] >= R[i - 1]) {

for (int j = R[i - 1] + 1; j <= R[i]; j++) addr(j);

if (L[i] <= L[i - 1]) for (int j = L[i - 1] - 1; j >= L[i]; j--) addl(j);

else for (int j = L[i - 1]; j < L[i]; j++) reml(j);

} else {

if (L[i] <= L[i - 1]) for (int j = L[i - 1] - 1; j >= L[i]; j--) addl(j);

else for (int j = L[i - 1]; j < L[i]; j++) reml(j);

for (int j = R[i - 1]; j > R[i]; j--) remr(j);

}

}

### **Policy Based Data Structures:**

Add the following lines to the top of your program to use PBDS.

#include <ext/pb\_ds/assoc\_container.hpp>

#include <ext/pb\_ds/tree\_policy.hpp>

using namespace \_\_gnu\_pbds;

(constructor) ordered\_set (change less<T> to less\_equal<T> for a multiset):

**template** <**typename** T>

**using** ordered\_set = tree<T, null\_type, less<T>, rb\_tree\_tag, tree\_order\_statistics\_node\_update>;

ordered\_set<**int**> s; //ordered\_set<T> name;

*Note: On older compilers, use* null\_mapped\_type *instead of* null\_type

(constructor) ordered\_map:

**template** <**typename** K, **typename** V>

**using** ordered\_map = tree<K, V, less<K>, rb\_tree\_tag, tree\_order\_statistics\_node\_update>;

ordered\_map<**string**, **int**> m; //ordered\_map<K,V> name;

An STL set or map with two extra functions:

**find\_by\_order(x)** returns an iterator pointing to the *x*th element in the set (0-indexed).

**order\_of\_key(k)** returns the number of elements strictly less than k.