Discussing Problems

Array & subarrays

Given
↳ array size N

Find number of subarrays whose sum = X
↳ TARGET SUM.

Also if X = 0
↳ You don't count empty sub-arrays.

When you deal with sub-arrays
↳ you usually deal w/ PREFIX SUM

Example:

let say A = { 1, 2, 3, 2, 3, 2 }
(0 1 2 3 4 5)

↓

P = { 1, 3, 6, 8, 11, 13 }

If we want sum of A[1] to A[3] = 7

↓

P[3] − P[1−1] = 7

Hence for A[i] to A[j] sum =

P[j] − P[i−1] = X
↳ target sum.

So the number of (i, j) pairs which satisfy this above cond^n GIVES ↓

No. of sub-arrays whose sum = X

THIS IS THE IDEA !

We will implement this using maps.

WHY A MAP?

↳ Purpose of the map is to store freq of the prefix sums encountered so far
This allows to quickly check if particular sub-array can be formed using current prefix sum.

Comider our prev example,

A = {1, 2, 3, 2, 3, 2}

P = {1, 3, 6, 8, 11, 13} Target sum = 5

↓

mp = {0:1, 1:1, 3:1, 6:1, 8:1, 11:1, 13:1}

ans = 1 + 1 + 1 + 1

6-5 = 1 8-5 = 3 11-5 = 6 13-5 = 8

already present

Basically we are checking the equality i.e.

↓

prefix[j] - sum = prefix[i-1]

↳ It means if prefix[j] - sum = some value → which is present in prefix[] then counter ↑

INSHORT: We create map for one value & check the existence of other

↳ in this case mp[prefix[j] - sum]

Check code for implementation.

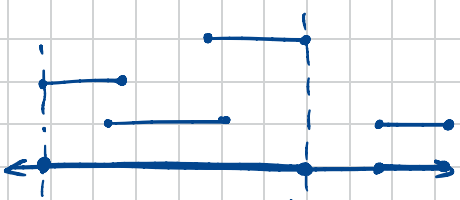
```cpp
void solve(){
  int n;
  cin>>n;
  // Resizing vectors for input size
  arr.resize(new_size: n);
  prefix.resize(new_size: n);

  // Adding input to arr and simul creating prefix sum
  for(int i=0; i<n; i++){
    cin>>arr[i];
    prefix[i]+=arr[i];
    // and if i>0 we also add previous sums
    if (i>0) {
      prefix[i]+=prefix[i-1];
    }
  }

  // Taking input for the target sum
  cout<<"Enter Target Sum:"<<endl;
  lli sum;
  cin>>sum;
  // Creating a map to save the number of time the prefix sum has been seen
  lli ans=0; //Basically this is our counter
  map<lli, lli> mp;
  mp[0]++;
  for (int j=0; j<n; j++) {
    ans+=mp[prefix[j]-sum]; // Checking the existence of prefix sum in the map
    mp[prefix[j]]++;
  }

  // Outputing the ans
  cout<<"Number of subarrays: "<<ans;
}
```

accessing & inserting takes O(logN)

So for N-times,
TC → O(NlogN)

Segments - segments everywhere

↳ Overview



Consider a number line

↓ Query

↳ In this case = 2

Find the no. of continuous line segments on no. line.

In terms of code

↳

Insert → (l, r)

Query → returns no. of continuous segments.

↓

query needs to be maintained

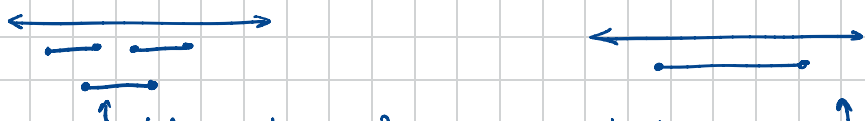
Implementation

↓

We need to maintain start & end of pairs according to the insertion.

Solut^n

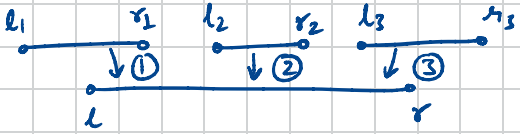
↳ What we do is



↑ let say this is inserted } → so we check accordingly & merge the reqd segments

Code ↓

When coding this we need to keep in mind 3-cond^{n}s




```
void insert(lli l, lli r){
  // In a multiset, it's sorted on the basis of the 1 of the pair
  auto it =mset.lower_bound({l, 0}); // This is to find whether a line seg with an l smaller than i/p l is present or not
  // If the itt doesn't return begin we check for the conditions i.e where does the input l,r segment lies and we place it accordingly
  if (it!=mset.begin()) {
    it--;

    if (it->second >= r) {
      // Basically this means that input line segment is consumed by the present line segment
      return;
    }
    if (it->second >= l) {
      // then we merge it
      l = it->first;
      mset.erase(position: it);
    }
  }

  it = mset.upper_bound({r, 0});
  if (it!=mset.begin()) {
    it--;

    if (it->second >= r) {
      // then we merge it
      r = it->second;
      mset.erase(position: it);
    }
  }
```

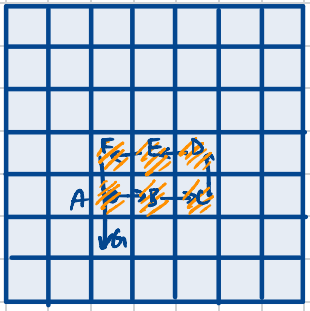
① ② ③

But as you see in this case we will still have $\{l_2, r_2\}$ so using a loop we need to erase them.

Ice-Board

Given : (N,M) board & You start at a point (x,y)

Basically if you travel from one cell to another your prev cell becomes ice



L, R, U, D

↓

Set of instruct^{n}s

→ Let say you start from A → Once you go to B A → ice → Now you follow your path ABCDEF → When you encounter A again

WORKING

If you hit a boundary you fall of the board

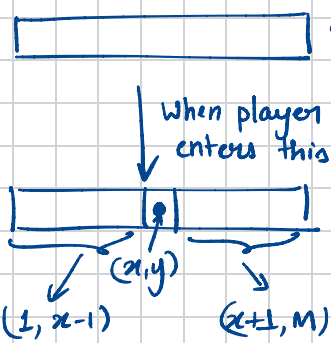
You slip from F to G directly & not to A ∵ A was ice.

Input : Queries like {L, R, L, U}

Output : Our final posit^n.

Sol Proposed sol^n : Maintain the free cells (cells not turned to ice) as segments

Eg:



→ For any row let say when player has never reached there it can be represented as, (1, M)

→ Since (x,y) → ice ⇒ row will be represented as (1, x-1), (x+1, M)

Similarly we can maintain segments for colums

So for every move → 2 log N + 2 log M

For Q queries → TC → O(Q log(M×N))

basically adding a multiset & editing one so, log N + log N