BRAC University Competitive Programming Workshop

Handouts for Day 5

Arman Ferdous

Department of CSE
BRAC University
23 April 2025

1 Integer Compression

Consider the following problem -

> **Problem 1: Finding Maximum Covering**
>
> You manage a restaurant and you have the reservation list of $n \leq 10^5$ groups coming tomorrow. The ith group will arrive at time l_i and leave at time r_i where $1 \leq l_i \leq r_i \leq 10^5$.
>
> Your task is to print the maximum number of groups that will be present at any moment tomorrow.

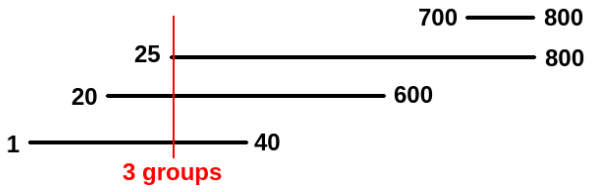
You should be able to solve this problem using the ideas of difference array discussed on Day 2. You may do something like this:

```
const int MAX = 100010;
int add[MAX]; // initialized to 0
for (int i = 0; i < n; ++i) {
    add[l[i]]++;     // Everything on [l[i], MAX) range gets +1
    add[r[i] + 1]--; // Everything on [r[i], MAX) range gets -1
}
// now take prefix sum of add[]
for (int i = 1; i < MAX; ++i) add[i] += add[i - 1];
// now add[i] gives #of groups present at time = i
```

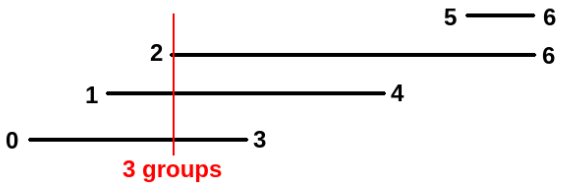
This works in $\mathcal{O}(n + MAX)$ time and space complexity, where MAX = highest possible value of l_i, r_i. This is good as long as the segments have small enough limits. So now we reach the following problem -

> **Problem 2: Finding Maximum Covering 2**
>
> You manage a restaurant and you have the reservation list of $n \leq 10^5$ groups coming tomorrow. The ith group will arrive at time l_i and leave at time r_i where $1 \leq l_i \leq r_i \leq 10^9$.
> Your task is to print the maximum number of groups that will be present at any moment tomorrow.

Now we can no longer take the `add[]` array as it will take $\sim$ 3.7GB of memory and $\sim$ 10s to finish. The key to solving this problem is to understand that the exact values l_i, r_i's don't matter rather the relation between them (less/equal/greater) matter. See the following picture -



If we simply compress all the co-ordinates/times like the following picture then observe that the answer does not change.



The crux of the trick is that with n groups, you have at most $2n$ unique start/end times. Hence only $2n$ values out of 10^9 only matter to us. If we can compress all those l_i, r_i's, we can use the solution of previous problem here as well.

This is known as co-ordinate/integer compression. It can be applied in other settings and is useful to decrease the domain of certain parameters if possible (very common with segment trees for example).

So how do we do it? First we need to store all the **unique** values that we wish to compress.

```
#define all(v) v.begin(),v.end()    // Useful macro to use

vector<int> compr;
for (int i = 0; i < n; ++i) {
    compr.push_back(l[i]);
    compr.push_back(r[i]);
}
sort(all(compr));
compr.erase(unique(all(compr)), compr.end());
// Now compr holds all values in sorted order with no duplicates
// Check the documentation of unique() function on your own
```

Following from the original uncompressed picture, we will now have the values {1, 20, 25, 40, 600, 700, 800} in `compr`.

Now we compress like this - for each value x which is some l_i or r_i: we map it to number of integers less than x in the array `compr`. Check yourself how this perfectly does the work for us.

```
for (int i = 0; i < n; ++i) {
    l[i] = lower_bound(all(compr), l[i]) - compr.begin();
    r[i] = lower_bound(all(compr), r[i]) - compr.begin();
}
// Now all l[i], r[i] < 2n and we are done
```

What if you were asked to print the time at which maximum number of groups were present? You can easily store the reverse mapping as well from the above snippet. So after finding the answer from the compressed values, you can use your reverse mapping to get actual value.

2 Interval Management

Problem 3

S is a binary string of length 10^9, initially filled with all 0s. You have to process two types of queries on it:

1. l r: apply $S[i] = 0$, where $l \leq i \leq r$.
2. l r: apply $S[i] = 1$, where $l \leq i \leq r$.

After each query print how many non-extendable intervals of 1 exist. For example if $S =$ 0001110101100, then there are 3 non-extendable intervals.

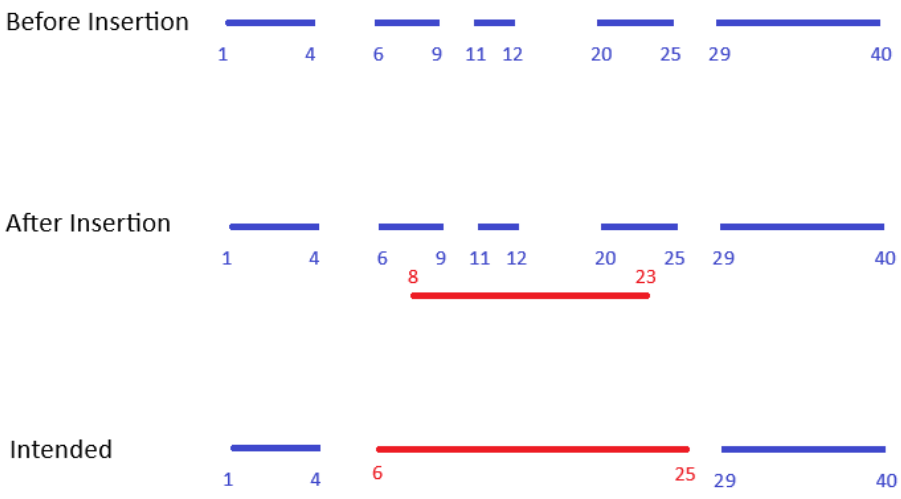
CSES1188 is a similar problem which was discussed in the workshop.

Solution: Suppose $S = 0001110101100$. Write all the (start index, end index) of non-extendable 1 segments in a set: $I = \{(4, 6), (8, 8), (10, 11)\}$. (1-indexed).

We will maintain this set I through our series of queries. Then the answer to our problem will be $|I|$ after each query.

```
set<pair<int, int>> I;  // Declare I
// Why is this a set? Why can't we simply use an array?
// We need to keep all our intervals sorted, to be able to apply
// our queries quickly. You will see shortly.
```

So we now write a function that will insert an interval $[L, R]$ in our set I. Let us first see what kind of events happen if we simply insert the interval pair.



We need to process a lot of merging, and the original interval $[L, R]$ sometimes even grows larger by combining with its neighbors. So we process these events in the following 3 steps.

1. Find all existing intervals that have $l_i \leq R+1$ and delete them. Also update R in the process if needed. (In the figure these are intervals $[11, 12]$ and $[20, 25]$ and they make our R go from 23 to 25)

2. Find the left neighbor. If it has $r_j + 1 \geq L$, then delete the left neighbor and update L. (In the figure this is the interval $[6, 9]$ which decreases our L from 8 to 6)

3. Finally after all the necessary removals from I, we can safely insert our current $[L, R]$, which is guaranteed to not merge with other intervals. (In the figure we finally insert $[6, 25]$ not $[8, 23]$)

Using a set will keep all the intervals sorted for us even after insertions/deletions. Which means we can run a binary search for the first interval in the set which has $L \leq l_i$. If this location also has $l_i \leq R+1$ satisfying, then it merges with our current $[L, R]$, otherwise no interval is mergeable in the first step (convince yourself).

Using a set also means that the subsequent intervals to be removed in Step 1 will all be in front of that point. So I hope now you realize why we want to use a set.

```
// we will actually return where the interval got inserted too
// why? to help with delete later on
set<pair<int,int>>::iterator addInterval (int L, int R) {
    auto it = I.lower_bound({L, R}); // find first location to merge
    // STEP 1
    while (it != I.end() && it->first <= R + 1) {
        R = max(R, it->second);
        it = I.erase(it); // erase deletes and returns the next element too
    }
    // STEP 2
    // Remember to always do boundary checks first. Otherwise you will get RTE.
    if (it != I.begin() && (--it)->second + 1 >= L) {
        L = min(L, it->first);
        R = max(R, it->second); // can you design a case where this is important?
        I.erase(it);
    }
    // STEP 3
    return I.insert({L, R}).first;
    // set::insert() returns a pair, where .first is the iterator to it
    // and .second is a bool denoting whether insert was successful
}
```

So, how do we remove a segment $[L, R]$? You can try drawing multiple scenarios about how some intervals get cut from the middle and some get totally erased. But we can actually reduce a lot of hassle by simply inserting the interval $[L, R]$! We get the newly inserted $[L', R']$ back from the insert function. From there we check the following -

1. If $L' < L$ we change this segments $R' = L-1$, because we are not supposed to delete $[L', L-1]$. Otherwise $L' = L$ and we completely delete it.

2. If $R < R'$ we insert a new segment $[R+1, R']$. Otherwise $R = R'$ and we don't do anything extra.

```
void removeInterval (int L, int R) {
    auto it = addInterval(L, R);
    int R2 = it->second; // This is the R'

    // STEP 1
    if (it->first < L) (int &)it->second = L - 1;
    // The above line hard assigns the value of it->second
    // Only do this if you are sure it doesn't break the ordering of the set
    else I.erase(it);

    // STEP 2
    if (R < R2) I.insert({R+1, R2});
}
```

Complexity Analysis: The insert function has two $\mathcal{O}(\lg(|I|))$ operations, the `lower_bound()` at the start and the `insert()` at the end (it is possible to make that insert $\mathcal{O}(1)$ if you want). But the biggest issue is the loop in the middle...

How many times will that loop run? The answer depends on the number of intervals currently in I. But you should notice that any interval from all of our queries (add/remove) will only be removed from the set I at most once. Which means the total number of operations `addInterval()` will do throughout the whole run will be bounded by $\mathcal{O}(Q \lg(Q))$, where Q is the number of queries we need to process.

So we can say running `addInterval()` once has an "average" runtime of $\mathcal{O}(\lg(Q))$ complexity. This is called an **amortized time complexity**, because it can behave much worse at times, but when you consider the whole picture it is good.

In the same way `deleteInterval()` has $\mathcal{O}(\lg(Q))$ amortized time complexity.

3 Arbitrary Deletion From Heap

"Wait!!! But we were taught you can't delete anything but the top element from a heap."

Yes, you are right. But you can support arbitrary deletion with a simple tweak ;)

> **Problem 4**
>
> You have an empty set H. Support the following queries on it Q times -
>
> 1. push(x): Insert x in H. Duplicates allowed.
> 2. erase(x): Delete a single copy of x from H.
> 3. top(): Return the largest element in H.

This problem is solvable using `std::set` too. But we will do it with `std::priority_queue` because it has a faster performance than set (due to heap data structure being less heavy than a red-black-tree which set/map uses).

The idea is we keep two heaps. One to maintain the values in it, the other to maintain the values we deleted.

```
priority_queue<int> H, D;
```

When we push(x), we just insert into H.

```
void push (int x) { H.push(x); }
```

But when we erase(x), we rather push x into D.

```
void erase (int x) { D.push(x); }
```

Now we can access the highest value from H using `H.top()`. But if this value was deleted we will know it by looking at `D.top()`. The code looks like this -

```
int top () {
    while (!H.empty() && !D.empty()) {
        if (H.top() == D.top()) { H.pop(); D.pop(); }
        else break;
    }
    if (!H.empty()) return H.top();
    return -1; // Otherwise H is empty, so top() is invalid
}
```

All functions work in $\mathcal{O}(\lg(Q))$ time where the `top()` function's complexity is amortized.

4 STL Quiz

Answer each question independently. Try to find good solutions. What do you use when you need to support:

1. insert(x): insert x (duplicates allowed).
   erase(x): erase a single copy of x.
   erase_global(x): erase all copies of x.
   unique(): get number of distinct elements. (Also consider the situation without this query)
   lower_bound(x): find smallest value greater or equal to x.

2. insert(x): insert x (duplicates allowed).
   get_min(): get the minimum value entered so far.
   get_max(): get the maximum value entered so far.
   erase_min(): erase the minimum value.
   erase_max(): erase the maximum value.

3. append(x): push x at the end of current list.
   prepend(x): push x at the start of the list.
   pop_back(): remove last element.
   pop_front(): remove the first element.
   get(i): access the ith element.

4. insert(x): insert x (duplicates allowed).
   get_median(): find the median value. [1]

5. Problem 3 from above. But now you need to report the maximum length of any non-extendable 1 segment. Example: $S = 0011111110011$, max length = 7.

6. insert(x): insert x (duplicates allowed).
   erase(x): erase a single copy of x.
   add_all(v): add v to all elements currently inserted.
   get_min(): find the minimum value.

7. insert(x): insert x (no duplicates).
   erase(x): erase a single copy of x.
   find_kth(k): find the kth smallest element.
   count_smaller(x): count #of elements smaller than x.

[1] If odd number of elements are present then it is the middle value after sorting, otherwise it is the average of the two middle values.

5 STL Quiz Solutions

Check solutions only if you tried :)

1. If we didn't have the unique() query, we could use a multiset.

```
multiset<int> st;
st.insert(x); // can insert duplicates
st.erase(x); // will delete all copies of x

// but if you want to erase only a single copy
// first find one location of it -
auto it = st.find(x);
// then erase it specifically, if found
if (it != st.end()) st.erase(it);

auto it = st.lower_bound(x); // for lower_bound query
```

For the full problem, we can use a map.

```
map<int, int> mp; // mp[x] will store the current frequency of x

void insert (int x) {
    mp[x]++; // default value is always 0 if didn't exist
}
void erase (int x) {
    // Slow but easy to understand:
    mp[x]--; // now deleting it completely if it has value/frequency <= 0
    if (mp[x] <= 0)
        mp.erase(x);
    // Don't use this, all 3 lines take O(lg n) time

    // Faster & ideal approach, only find() costs O(lg n) here
    auto it = mp.find(x);
    it->second--; // O(1)
    if (it->second <= 0) mp.erase(it); // O(1) as we specify the iterator
}
void erase_global (int x) {
    auto it = mp.find(x);
    if (it != mp.end()) mp.erase(it); // delete entirely
}
int unique () {
    return (int)mp.size(); // since all existing keys have frequency > 0
}
map<int, int>::iterator lower_bound (int x) {
    return mp.lower_bound(x); // returns iterator to the lower_bound
}
```

2. A messy way to do it is by creating two heaps (one min, one max), where you use the arbitrary deletion trick as show previously. But I like the following:

```
multiset<int> st; // that's all we need

st.insert(x); // insert is simple

// accessing min/max is O(1)
int minim = *st.begin(); // st.begin() points to the least element
int maxim = *st.rbegin(); // check out reverse iterators
// for example second max can be found like this:
int scmx = ((int)st.size() > 1 ? *(++st.rbegin()) : -1);

// erasing min/max is simple too
if (!st.empty()) st.erase(st.begin());
if (!st.empty()) st.erase(--st.end());
// or st.erase(prev(st.end()));
```

3. What you are looking for is a double-ended queue, deque in short.

```
deque<int> dq;
dq.push_back(x);
dq.push_front(y);
dq.pop_back();
dq.pop_front();

dq[i] = 5; // yes you can index it like an array :)
```

4. The idea is to use two heaps, L = max heap and R = min heap.

```
priority_queue<int> L; // max heap
priority_queue<int, vector<int>, greater<int>> R; // min heap
```

   Whenever you insert an element x, you insert it into the L heap. If after the insertion $|L| > |R|$, you pop the top (max) value from L and push it into R instead.

   Doing this ensures that L and R will have an even split of all the elements. For example if there are a total of 7 inserts, then 3 of them will be in L and 4 in R.

   Now getting the median is easy. If an odd number of elements are inserted so far, then your median is the top (min) element of R. Otherwise it is the average of `L.top()` and `R.top()`.

5. We will reuse the whole solution from problem 3. So we have a set I which contains all the $[L, R]$ pairs which correspond to a non-extendable 1 segment. Now we need to find the maximum value of $R - L + 1$ among them. How to do it without iterating through the whole set?

   Keep a multiset J which contains the length $(R - L + 1)$ of all active segments in I. You will need to modify the `addInterval()` and `deleteInterval()` functions. Whenever you insert anything new into I you need to insert its length into J, whenever you take something out of

I you will remove that corresponding length from J. Everything will still be $\mathcal{O}(\lg(Q))$, and you can find the maximum length by accessing `*J.rbegin()`.

6. This can be solved using a multiset and some global update counter. You can read more about it here.

7. You can't solve it with simple sets, because it can't find the kth smallest element or count elements smaller than another. The DS that can support this is called an **order statistic tree** and in many languages you need to write large amount of code to support these operations.

   However in C++ you have some policy based data structures in STL. Which when modified can give you some powerful functionality. You should read it from here.

```cpp
#include <bits/extc++.h>
using namespace __gnu_pbds;

template<class T>
using Tree = tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;

void example() {
    // Tree can do anything a set can do
    Tree<int> t, t2; t.insert(8);
    auto it = t.insert(10).first; // t = {8, 10} now
    assert(it == t.lower_bound(9));

    // order_of_key: returns #of elements less that it
    assert(t.order_of_key(10) == 1);
    assert(t.order_of_key(11) == 2);

    // find_by_order: returns the kth element (0 indexed)
    assert(*t.find_by_order(0) == 8);

    t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}
```

   This will not keep duplicate values. You can try to figure out how you can keep duplicates (like multiset) on your own, or read from that blog.

   Also keep in mind, this ordered set is VERY SLOW. Be wary of every single $\mathcal{O}(\log(n))$ operation that you call, as it comes with a huge constant factor (from my experience around 8-10x). It is a good practice to benchmark how many ordered set operations you can support within 1s in the mock contests, so that you have a good idea of what will get TLE or not. If ordered set gets TLE, then you need to implement your own (e.g. with fenwick/segment tree, or treap).