***STRIVER CP SHEET***

## SQUARE ROOT DECOMPOSITION/ MO'S ALGORITHM:-

Mo's Algorithm is used for offline queries(where all there the queries are provided beforehand).

Square root decomposition is a technique used in algorithm design to optimize certain operations, particularly in problems involving range queries or updates on arrays. It's useful because it allows us to perform operations like range queries (e.g., finding the minimum or maximum element in a given range) or range updates (e.g., adding a value to all elements in a given range) efficiently.

One common application of square root decomposition is in problems involving dynamic programming over a range of values,

where the range can be subdivided into blocks whose sizes are related to the square root of the total size of the range. This approach helps reduce the overall complexity of the algorithm.

Additionally, square root decomposition is often used in conjunction with other data structures like segment trees or Fenwick trees to further optimize various types of range queries and updates.

**Where Do We Use It?**

Square root decomposition is used in problems that require efficient range queries and point updates. Common use cases include:

- Range sum queries
- Range minimum/maximum queries
- Range frequency queries
- Range gcd/lcm queries

It's particularly useful in competitive programming for problems where more sophisticated data structures might be overkill or too complex to implement within the time constraints of a competition.

## Difference between MO'S algorithm and Square Root Decomposition.

Mo's algorithm and square root decomposition are both techniques used to solve range query problems efficiently, but they differ in their approach, flexibility, and typical use cases. Here's a detailed comparison of the two:

### Mo's Algorithm

1. **Purpose**:
   - Specifically designed to handle multiple offline range queries efficiently.
2. **Approach**:
   - Sorts queries in a specific order to minimize the number of changes

needed when moving from the answer to one query to the next.
   - Uses two pointers (or sliding window) technique to adjust the current range to match each query.
3. **Time Complexity**:
   - Processes all queries in O((N+Q)N)O((N + Q) \sqrt{N})O((N+Q)N) time, where NNN is the size of the array and QQQ is the number of queries.
4. **Implementation Complexity**:
   - Requires careful management of the current range and updating the answer as the pointers move.
   - Generally considered more complex to implement compared to basic square root decomposition.
5. **Flexibility**:
   - Highly flexible and can handle complex queries like frequency counts, distinct elements, etc.

- Suitable for problems where queries are known in advance (offline queries).

6. **Use Cases**:
   - Best used when dealing with a large number of offline queries that need to be answered efficiently.
   - Ideal for competitive programming scenarios involving complex range queries.

**Square Root Decomposition**

1. **Purpose**:
   - Designed to handle both range queries and point updates efficiently.
2. **Approach**:
   - Divides the array into blocks of approximately N\sqrt{N}N size.
   - Precomputes auxiliary information (e.g., block sums) for each block to speed up query processing.
3. **Time Complexity**:

- Queries and updates typically take O(N)O(\sqrt{N})O(N) time each.

4. **Implementation Complexity**:
   - Simpler to implement compared to Mo's algorithm.
   - Less sophisticated, involving straightforward block management.
5. **Flexibility**:
   - Suitable for both online queries (queries that arrive in real-time) and offline queries.
   - Can handle simple point updates and range queries efficiently.
6. **Use Cases**:
   - Ideal for problems where both point updates and range queries are frequent.
   - Useful for simpler range query problems like range sum, range minimum/maximum, etc.

## Key Differences

1. **Query Handling**:

- Mo's Algorithm: Efficiently handles multiple offline queries by sorting and using a two-pointer technique.
- Square Root Decomposition: Handles both point updates and range queries using precomputed block information.

2. **Efficiency**:
   - Mo's Algorithm: Generally more efficient for a large number of queries, particularly offline queries, due to the O((N+Q)N)O((N + Q) \sqrt{N})O((N+Q)N) complexity.
   - Square Root Decomposition: Provides a balanced O(N)O(\sqrt{N})O(N) complexity for both queries and updates.
3. **Complexity**:
   - Mo's Algorithm: More complex to implement due to the need to manage and adjust query ranges dynamically.
   - Square Root Decomposition: Simpler and more straightforward to implement.

## Example Scenarios

- **Mo's Algorithm**:
  - Given an array, answer multiple queries asking for the sum of elements in a range, the number of distinct elements, or the frequency of a particular element in a range.
- **Square Root Decomposition**:
  - Given an array, efficiently update elements and answer queries asking for the sum or minimum/maximum in a range.

## Summary

Mo's algorithm is optimized for handling multiple offline range queries by minimizing adjustments between queries, while square root decomposition provides a balanced approach for both online and offline queries and updates. The choice between the two depends on the specific problem requirements, particularly whether the

problem involves more complex queries and if updates are needed.

## Q1 Range Minimum Query (SQUARE ROOT DECOMPOSITION)?

## Ans:- solved by using Square root Decomposition.

You are given a list of **N** numbers and **Q** queries. Each query is specified by two numbers *i* and *j*; the answer to each query is the minimum number between the range [i, j] (inclusive).

*Note:* the query ranges are specified using 0-based indexing.

### Input

The first line contains **N**, the number of integers in our list (**N <= 100,000**). The next line holds **N** numbers that are guaranteed to fit inside an integer. Following the list is a number **Q** (**Q <= 10,000**). The next **Q** lines each contain two numbers **i** and **j** which specify a query you must answer (**0 <= i, j <= N-1**).

### Output

For each query, output the answer to that query on its own line in the order the queries were made.

### Example

```
Input:
3
1 4 1
2
1 1
1 2

Output:
4
1
```

Code:-

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long int
#define take(x) int x; cin>>x;

#define getv(v, n)  vector<int> v;for (int i =
0; i < n; i++){ take(x) v.push_back(x);}

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

#define allr(v) v.rbegin(), v.rend()

#define sortv(v) sort(all(v))

#define sortvr(v) sort(allr(v))

#define gcd(a,b)  __gcd(a,b)

#define mem1(a) memset(a,-1,sizeof(a))

#define mem0(a) memset(a,0,sizeof(a))

#define pb push_back

#define pob pop_back

#define yes cout <<YES<< endl

#define no cout <<NO<< endl
```

```cpp
#define ff first

#define ss second

#define MOD 1000000007

#define INF 1e18

#define ppc(x) __builtin_popcountll(x)

int main()
{
    int n;
    cin >> n;
    vector<int>v;
    for (int i = 0; i < n; i++)
    {
        take(x);
        v.pb(x);
    }

    int q;
    cin>>q;
    vector<pair<int,int>>vq;
    for (int i = 0; i < q; i++)
    {
        int a,b;
        cin>>a>>b;
        vq.pb({a,b});
    }
```

```cpp
    vector<int>v1(ceil(sqrt(n)),INT_MAX);
    int len=sqrt(n);
    // cout<<len<<endl;
    for(int i=0;i<n;i++){
        if(i%len==0){
           v1[i/len]=v[i];
        }
        else{
            v1[i/len]=min(v1[i/len],v[i]);
        }
    }

    for (int i = 0; i < q; i++)
    {
        int l=vq[i].first;
        int r=vq[i].second;
         int ans=1e9;
        for (int j = l; j <=r; j++)
        {
            if(j%len==0 && j+len-1<=r){
               ans=min(ans,v1[j/len]);
               j+=len-1;
            }
            else{
                ans=min(ans,v[j]);
            }
        }
        cout<<ans<<endl;
    }

    return 0;
```

```
}
```

Brief description of using square root decomposition in this question.

Initially, for every question, we have to create a box of size (square root(n)) where n is the size of the given array. And the no. of boxed we need is (ceil(squareroot(n)).

Create another vector v1 whose size is equal to no. of boxes we need and each index of v1 will store the cumulative result of certain indexes from the original array and the total no. of indexes that take part to give cumulative result is less than or equal to the size of the box.

So it means v1[0] will store the required result(as given in the question here it Is given to find the minimum so we will store the minimum) from index 0 to (box size -1) index of the original array (v) and so on…..

Now we will do the precomputation thing where we will store the required ans in v1

vector which is the cumulative result from a certain index from the original array.

Working of precomputation:-

```
for(int i=0;i<n;i++){
        if(i%len==0){
           v1[i/len]=v[i];
        }
        else{
            v1[i/len]=min(v1[i/len],v[i]);
        }
    }
```

Here n is the size of the original array, len is the box size we need and v1 is the vector of size (ceil(square-root (n)) which stores the cumulative result.

Here we are taking i/len because we want to store the results of certain indexes of v into the single index of v1.

For example here, i%len==0 it means we entered the new box of v1, where we are going to store the cumulative result from now on, as because , the boxes has size (square-root(n)), and each box of v1 will store the

(len) elements. Lets say we are starting from 1st box of v1 and 1st index of v, so i==0 and lets say len ==2 and total no. of box=2;

So at first i==0, and i%2==0, it means there is new box and clearly I mention it is the 1st box, so 1st element of v is updated at the index of i/2 of v1 which is v1[0]. Now further for i==1, as i%len!=0 so it is not the new box, it is just the previous box on which we are working.

So, the result will still be stored in i/len==0 means in v1[0], as you see this is how we store the cumulative result of index 1 and 2 from v into the single index of v1. Then comes i==2, where i%len==0, it means there is a beginning of a new box, so in the new box, we can update v1[i/len] with the value of v[i] as this v[i] is the first element for the new box.

And so on we fill our entire v1 vector with the cumulative results which is taken from the original array.

Now we will see how we compute the answer:-

```
for (int i = 0; i < q; i++)
    {
        int l=vq[i].first;
        int r=vq[i].second;
         int ans=1e9;
        for (int j = l; j <=r; j++)
        {
            if(j%len==0 && j+len-1<=r){
               ans=min(ans,v1[j/len]);
               j+=len-1;
            }
            else{
                ans=min(ans,v[j]);
            }
        }
        cout<<ans<<endl;
    }
```

Here our main objective is to use v1 vector index answer so that we do not have to traverse the entire v vector, and some cumulative answers is already stored in the v1 vector, and these cumulative results can directly be fetched from v1 in O(1) time. But we need to check whether we can fetch or

not for a given l and r, we know how we divide our boxes, and we also know from which index to which index of v our box has stored the result. So if there is any box that stores the result of indexes greater than or equal to l and less than equal to r of v, then directly fetch the value from v1, otherwise traverse manually.

Then print the final answer.

Q2 DQUERY (MO'S ALGORITHM)?

Ans:- solve using MO'S Algorithm

# DQUERY - D-query

#sorting #tree

English Vietnamese

Given a sequence of n numbers $a_1, a_2, ..., a_n$ and a number of d-queries. A d-query is a pair (i, j) ($1 \le i \le j \le n$). For each d-query (i, j), you have to return the number of distinct elements in the subsequence $a_i, a_{i+1}, ..., a_j$.

## Input

- Line 1: n ($1 \le n \le 30000$).
- Line 2: n numbers $a_1, a_2, ..., a_n$ ($1 \le a_i \le 10^6$).
- Line 3: q ($1 \le q \le 200000$), the number of d-queries.
- In the next q lines, each line contains 2 numbers i, j representing a d-query ($1 \le i \le j \le n$).

## Output

- For each d-query (i, j), print the number of distinct elements in the subsequence $a_i, a_{i+1}, ..., a_j$ in a single line.

## Example

```
Input
5
1 1 2 1 3
3
1 5
2 4
3 5

Output
3
2
3
```

# Code:-

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long int
#define take(x) int x; cin>>x;

#define getv(v, n)  vector<int> v;for (int i = 0; i < n; i++){ take(x)
v.push_back(x);}

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

#define allr(v) v.rbegin(), v.rend()

#define sortv(v) sort(all(v))

#define sortvr(v) sort(allr(v))

#define gcd(a,b)  __gcd(a,b)

#define mem1(a) memset(a,-1,sizeof(a))

#define mem0(a) memset(a,0,sizeof(a))

#define pb push_back

#define pob pop_back

#define yes cout <<YES<< endl
```

```cpp
#define no cout <<NO<< endl

#define ff first

#define ss second

#define MOD 1000000007

#define INF 1e18

#define ppc(x) __builtin_popcountll(x);

int len;
class query{
    public:
    int l,r,idx;
};

bool comp(query a,query b){
    if(a.l/len!=b.l/len){
        return a.l<b.l;
    }
    else{
        return a.r<b.r;
    }

}

int main()
{

    int n;
    cin >> n;

    vector<int>v;

    for (int i = 0; i < n; i++)
    {
        take(x);
        v.pb(x);
    }
    int q;
    cin>>q;
    vector<query>que(q+1);
    vector<int>v1(q+1,0);
    for (int i = 0; i < q; i++)
    {
        cin>>que[i].l>>que[i].r;
```

```
        que[i].idx=i;
        que[i].l--;
        que[i].r--;
    }
     len =sqrt(n);
     unordered_map<int,int>m;
    sort(que.begin(),que.begin()+q,comp);
    for (int i = 0; i < q; i++)
    {
        // cout<<que[i].l<<" "<<que[i].r<<" "<<que[i].idx<<endl;
    }
    int ML=0,MR=-1;
    int cnt=0;
    for (int i = 0; i < q; i++)
    {
        int L=que[i].l;
        int R=que[i].r;

        while(MR<R){
            MR++;
            m[v[MR]]++;
               if(m[v[MR]]==1){
                cnt++;
            }

        }
        while(ML<L){
            m[v[ML]]--;
            if(m[v[ML]]==0){
                cnt--;
            }
            ML++;
        }
        while(ML>L){
            ML--;
            m[v[ML]]++;
            if(m[v[ML]]==1){
                cnt++;
            }

        }
        while(MR>R){
            m[v[MR]]--;
            if(m[v[MR]]==0){
                cnt--;
            }
            MR--;
        }
```

```
        v1[que[i].idx]=cnt;
    }
    for (int i = 0; i < q; i++)
    {
        cout<<v1[i]<<endl;
    }

    return 0;
}
```

Let's crack this code segment by segment, so everytime we use mo's algorithm, we first create a class named as a query which stores 3 variables l=>left index, r=>right index, idx=index of original query.

We can also store these 3 indexes in a vector<pair<pair<int,int>,int>>que; but class is a convenient way. We also declare the global variable which stores the value of block size (square-root(n(size of given array)), this block is the same as we use in square root decomposition.

Secondly, we sort all the queries so that there will be fewer iterations for ML and MR.(sorting of queries is necessary in MO's Algorithm).

Sorting is done by using a comparator.

```
bool comp(query a,query b){
    if(a.l/len!=b.l/len){
        return a.l<b.l;
    }
    else{
        return a.r<b.r;
    }
}
```

Here this comparator takes two values present in the query and checks whether the two queries' left indexes are in the same block(this block is the same as we define in square-root decomposition) or not.

Same block means a.l/len==b.l/len, total no. of the block are n/square-root(n). and each block will have less than equal to square root (n) elements. Block[0] means it will store those elements/indexes that gives (index/len==0). If len=2, It will store or we can confirm which indexes belong to this block. So block[0] will consist of index 0 and 1. So if we get (a.l=0 and b.l=0) then we can say they (are in/ belong to) the same block.

So if they are the same block then whose (right index) is small should be stored first.

Else if their blocks are different then, whose (left index) is small should be stored first.

After sorting the queries our main functions start.

```
int ML=0,MR=-1;
   int cnt=0;
   for (int i = 0; i < q; i++)
   {
       int L=que[i].l;
       int R=que[i].r;

       while(MR<R){
           MR++;
           m[v[MR]]++;
              if(m[v[MR]]==1){
               cnt++;
           }
```

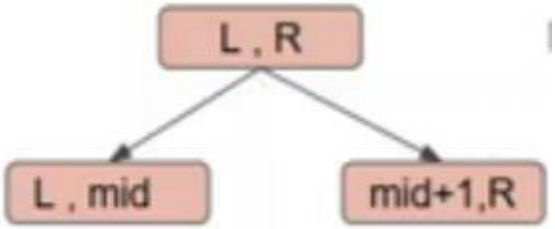
```
        }
        while(ML<L){
            m[v[ML]]--;
            if(m[v[ML]]==0){
                cnt--;
            }
            ML++;
        }
        while(ML>L){
            ML--;
            m[v[ML]]++;
            if(m[v[ML]]==1){
                cnt++;
            }
        }
        while(MR>R){
            m[v[MR]]--;
            if(m[v[MR]]==0){
                cnt--;
            }
            MR--;
        }
        v1[que[i].idx]=cnt;
    }
```

This above code template is the same for every MO'S Algorithm, these 4 while loops and increment/decrement of MR and ML are constant for every MO'S algorithm, in last v1[que[i].idx]=cnt is also fixed. After these constants rest thing depends from question to question here we increment or decrement (cnt) according to conditions (these conditions vary from question to question).

# SEGMENT TREE

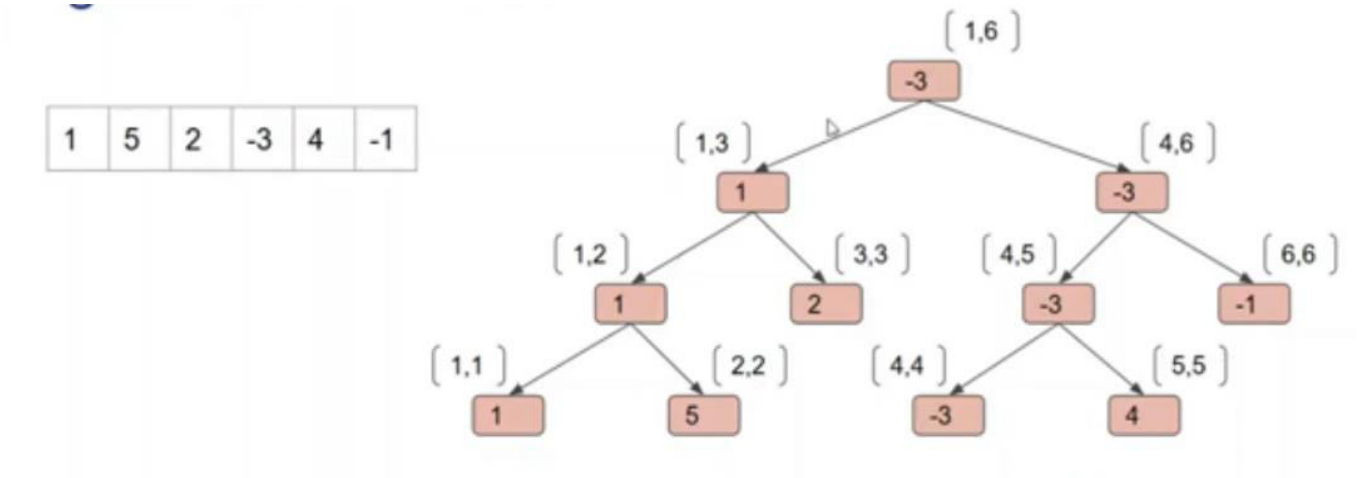
1. Introduction To Segment Tree
2. Implementation (Build & Query functions)
3. Problem 1: RMQSQ (SPOJ)
4. Problem 2: GSS1 (SPOJ)
5. Point Update : Introduction & Implementation
6. Problem 3: Help Ashu (HackerEarth)
7. Problem 4: GSS3 (SPOJ)
8. Problem with Range Update
9. Solution of Range Update(Lazy Propagation)
10. Problem 5: HORRIBLE (SPOJ)
11. Problem 6: SEGSQRSS (SPOJ)

## Segment Tree : Idea?




the root node will store the result(whatever question demands) of the array from index L(including) to index R(including). The left node will store the result(whatever question demands) of the array from index L(including) to index mid(including) and the right node will store the result(whatever question demands)

of the array from index mid+1(including) to index R(including). Similarly, further left and right nodes will also have two child nodes each and rules for storing the results are the same as the root node. Like consider the left node as the root node, then the left child of the left node will store the results from L to (l+mid)/2 th index and the right child of the left node will store the results from (l+mid)/2+1 th to mid index. The same kind of rule will also be applied to the right node and its children.
for example:-




Here the required result is to store the minimum for a range. As we know, the root node stores the result from L to R so the root node has a value of -3.

So each node stores the result according to the range assigned to it, like the range assigned to the root node from which to which it is allowed to store the result is from index L to R.

Observing the question, we will see how the Build and query functions work in the segment tree.

Q1 Range Minimum Query (Segment Tree)?

Ans:-

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long int
#define take(x) int x; cin>>x;

#define getv(v, n)  vector<int> v;for (int i = 0; i < n;
i++){ take(x) v.push_back(x);}

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

#define allr(v) v.rbegin(), v.rend()

#define sortv(v) sort(all(v))

#define sortvr(v) sort(allr(v))

#define gcd(a,b)  __gcd(a,b)

#define mem1(a) memset(a,-1,sizeof(a))

#define mem0(a) memset(a,0,sizeof(a))

#define pb push_back

#define pob pop_back

#define yes cout << "YES" << endl
```

```cpp
#define no cout << "NO" << endl

#define ff first

#define ss second

#define MOD 1000000007

#define INF 1e18

#define ppc(x) __builtin_popcountll(x)

void build_segment_tree(int si, int ss, int se,
vector<int>& v, vector<int>& segt) {
    if (ss == se) {
        segt[si] = v[ss];
        return;
    }
    int mid = (ss + se) / 2;
    build_segment_tree(2 * si, ss, mid, v, segt);
    build_segment_tree(2 * si + 1, mid + 1, se, v,
segt);
    segt[si] = min(segt[2 * si], segt[(2 * si) + 1]); //
only this line will change depending upon the questions.
}

int ansquery(int si, int ss, int se, int l, int r,
vector<int>& segt) {
    if (l > se || r < ss) {
        return 1e9;
    }
    if (l <= ss && se <= r) {
        return segt[si];
    }
    int mid = (ss + se) / 2;
```

```
    int left = ansquery(2 * si, ss, mid, l, r, segt);
    int right = ansquery(2 * si + 1, mid + 1, se, l, r,
segt);
    return min(left, right);
}

int main() {
    int n;
    cin >> n;
    vector<int> v(n + 1);
    for (int i = 1; i <= n; i++) {
        take(x);
        v[i] = x;
    }

    int q;
    cin >> q;
    vector<pair<int, int>> vp(q);
    for (int i = 0; i < q; i++) {
        cin >> vp[i].first >> vp[i].second;
    }

    vector<int> segt(4*n, 1e9);
    build_segment_tree(1, 1, n, v, segt);

    // Output the segment tree for debugging
    // for (int i = 1; i < 2 * n; i++) {
    //     cout << segt[i] << " ";
    // }
    // cout << endl;

    // Process each query
    for (int i = 0; i < q; i++) {
        int l = vp[i].first;
        int r = vp[i].second;
```

```
        cout << ansquery(1, 1, n, l+1, r+1, segt) <<
endl;
    }

    return 0;
}
```

The segment tree array should be 4 times the size of the given array. In the segment tree, we follow 1 base Indexing. The segment tree stores the result in a node (segment tree array index).
we first build the segment tree array:-

```
void build_segment_tree(int si, int ss, int se, vector<int>& v,
vector<int>& segt) {
    if (ss == se) {
        segt[si] = v[ss];
        return;
    }
    int mid = (ss + se) / 2;
    build_segment_tree(2 * si, ss, mid, v, segt);
    build_segment_tree(2 * si + 1, mid + 1, se, v, segt);
    segt[si] = min(segt[2 * si], segt[(2 * si) + 1]); // only this line
will change depending upon the questions.
}
```

Here segt is the segment tree array, v is the given array, si is the segment tree array index (node), ss is the starting index and se is the ending index for a node si, ss, and se denotes that si stores the required cumulative result from ss index of v to se index of v in si index of segt. If condition denotes that if there is a node where its ss and se is same, means si has a range

to store only a single element, so we will store the v[ss] or v[se] in segt[si], for that si node, v[ss] is the answer.

But if the given range for the node is  more than 1, then it means we have to explore more ranges so that we will bifurcate the range.

```
int mid = (ss + se) / 2;
    build_segment_tree(2 * si, ss, mid, v, segt);
    build_segment_tree(2 * si + 1, mid + 1, se, v, segt);
```

here mid we have defined to bifurcate the given range.

In the build_segment_tree, I have written 2*si, basically, it denotes the left child node of the current node in segment tree/any tree and (2*si)+1 denotes the right child node of the current node in segment tree/any tree. The left child node of the current node will store the results from ss index of the current node to the mid index. Similarly, The Right child node of the current node will store the results from mid+1 index of the current node to se index.

In segment tree apart from storing the results we also assigning the range from which to which index of v it will store the result.

The building method of segment tree is same for any segment tree, only this line will change:-

```
segt[si] = min(segt[2 * si], segt[(2 * si) + 1]); // only this line
will change depending upon the questions.
```

Now we will see how we implement query functions:-

```
for (int i = 0; i < q; i++) {
        int l = vp[i].first;
        int r = vp[i].second;
        cout << ansquery(1, 1, n, l+1, r+1, segt) << endl;
    }
```

As we have used 1 base indexing in the segment tree, and queries are given in 0 base indexing, we have to add 1 here in l and r.

```
int ansquery(int si, int ss, int se, int l, int r, vector<int>& segt) {
    if (l > se || r < ss) {
        return 1e9;
    }
    if (l <= ss && se <= r) {
        return segt[si];
    }
    int mid = (ss + se) / 2;
    int left = ansquery(2 * si, ss, mid, l, r, segt);
    int right = ansquery(2 * si + 1, mid + 1, se, l, r, segt);
    return min(left, right);// only this line will differ according to
the questions, otherwise rest of the method for ansquery in segment
tree will remain same for every segment tree query implementation.
}
```

Hume query me l aur r dia hoga jisme hame l se leke r tk ka minimum batana h. maanlo l h 3 aur r h 6. to hum apne node me search krna shuru krenge, aur hr node ka apna range h. ab maanlo hum kisi node me gae(maanlo root node se bifurcation krte krte idhar aae) aur uska range hua 1 to 4, but hame to 3 to 6 ka chahiye, pr dekho 1 to 4 me 3 aur 4 cover horha h for 3 to 6, to kmse km 3 aur 4 ka to  nikal lo, to hum 1 to 4 wale node ka left aur right child me dhundenge, to jb maine 1 to 4 ka left child dekha to usne 1 to 2 (index of

v) ka result store kia h, and ye result 3 to 6 wale se completely bhr h aur hame iski zaroorat nhi h, to humne 1e9 return kia, because ye baad me neglect hojaega(because here we are solving for minimums) because khin na khin se valid ans aaega hi. to idhar left child ne to 1e9 return kia. Now for right child node whose index in segment tree is(2*si+1) and range is(mid+1,se(it the ending range index of parent node)).

It stores the result from 3 to 4 and this range lies completely inside the given l (3) and r (6) so we do not have to bifurcate further so the result of this node is applicable, so right child node return segt[si(the index of right child node)]. 3 to 6 mese  3 to 4 ka direct result mila jo ki segment tree ke node me store, aur agr segment tree ne 3 to 4  ka result store kia h to wo sbse optimal hoga, iske alawa khin bhi isse better answer nhi milega, beause building hi aisi hui h segment tree ki.

Ab hame chahye 5 to 6 wala result. So similarly we get 5 to 6 result too, now left child node of the root node return the result for 3 to 4 and right child node return the result of 5 to 6 , then over all result will be the min of left and right , which denotes the overall answer for 3 to 6.

# POINT UPDATE AND IMPLEMENTATION

## Continue.. later

For range update we use LAZY PROPAGATION

## LAGY PROPAGATION

### HORRIBLE - Horrible Queries

#tree #binary-search

World is getting more evil and it's getting tougher to get into the Evil League of Evil. Since the legendary Bad Horse has retired, now you have to correctly answer the evil questions of Dr. Horrible, who has a PhD in horribleness (but not in Computer Science). You are given an array of **N** elements, which are initially all 0. After that you will be given **C** commands. They are -

* **0 p q v** - you have to add **v** to all numbers in the range of **p** to **q** (inclusive), where **p** and **q** are two indexes of the array.

* **1 p q** - output a line containing a single integer which is the sum of all the array elements between **p** and **q** (inclusive)

#### Input

In the first line you'll be given **T**, number of test cases.

Each test case will start with **N** (**N** <= 100 000) and **C** (**C** <= 100 000). After that you'll be given **C** commands in the format as mentioned above. 1 <= **p**, **q** <= **N** and 1 <= **v** <= 10^7.

#### Output

Print the answers of the queries.

#### Example

```
Input:
1
8 6
0 2 4 26
0 4 8 80
0 4 5 20
1 8 8
0 5 7 14
1 4 8

Output:
80
508
```

Code:-

```cpp
#include <bits/stdc++.h>
using namespace std;
#define ll long long int
```

```
#define take(x) int x; cin>>x;

#define getv(v, n)  vector<int> v;for (int i = 0; i < n;
i++){ take(x) v.push_back(x);}

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

#define allr(v) v.rbegin(), v.rend()

#define sortv(v) sort(all(v))

#define sortvr(v) sort(allr(v))

#define gcd(a,b)  __gcd(a,b)

#define mem1(a) memset(a,-1,sizeof(a))

#define mem0(a) memset(a,0,sizeof(a))

#define pb push_back

#define pob pop_back

#define yes cout <<YES<< endl

#define no cout <<NO<< endl

#define ff first

#define ss second

#define MOD 1000000007

#define INF 1e18
```

```
#define ppc(x) __builtin_popcountll(x);

void build(int si,int ss,int
se,vector<ll>&segt,vector<ll>&v){
    if(ss==se){
        segt[si]=v[ss];
        return;
    }
    else{
        int mid=(ss+se)/2;
        build(2*si,ss,mid,segt,v);
        build((2*si)+1,mid+1,se,segt,v);
        segt[si]=segt[2*si]+segt[(2*si)+1];
        return;
    }
}

ll query(int si,int ss,int se,int l,int
r,vector<ll>&segt,vector<ll>&lazy){
    if(lazy[si]!=0){
        ll d=lazy[si];
        lazy[si]=0;
        segt[si]+=d*(se-ss+1);
        if(ss!=se){
        lazy[2*si]+=d;
        lazy[(2*si)+1]+=d;
        }
    }
    if(ss>r || se<l){
        return 0;
    }
    if(l<=ss && r>=se){
        return segt[si];
    }
   int mid=(ss+se)/2;
```

```
   ll l1=query(2*si,ss,mid,l,r,segt,lazy);
   ll r1=query(2*si+1,mid+1,se,l,r,segt,lazy);
   return l1+r1;
   }

   void update(int si,int ss,int se,int l,int
r,vector<ll>&segt,vector<ll>&lazy,ll val)
   {
    if(lazy[si]!=0){
              ll d=lazy[si];
        lazy[si]=0;
        segt[si]+=d*(se-ss+1);
         if(ss!=se){
        lazy[2*si]+=d;
        lazy[(2*si)+1]+=d;
        }
    }
    if(ss>r || se<l){
        return ;
    }
    if(l<=ss && r>=se){
        segt[si]+=val*(se-ss+1);
           if (ss != se) {
            lazy[2 * si] += val;
            lazy[2 * si + 1]+= val;
        }
        return ;
    }

    int mid=(ss+se)/2;
    update(2*si,ss,mid,l,r,segt,lazy,val);
    update((2*si)+1,mid+1,se,l,r,segt,lazy,val);
    segt[si] = segt[2 * si] + segt[2 * si + 1];
   }
```

```cpp
int main()
{
    int n;
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        int k,c;
        cin>>k>>c;
        vector<ll>v(k+1,0);
        vector<ll>segt(4*k);
        vector<ll>lazy(4*k);
         build(1,1,k,segt,v);
        for (int i = 0; i < c; i++)
        {
            int b;
            cin>>b;


            if(b==0){
                int l,r,val;
                cin>>l>>r>>val;
                //  cout<<b<<" "<<l<<" "<<r<<"
"<<val<<endl;
                update(1,1,k,l,r,segt,lazy,val);
                     }
            else if(b==1){
                int l,r;
                cin>>l>>r;
                //  cout<<b<<" "<<l<<" "<<r<<endl;
                cout<<query(1,1,k,l,r,segt,lazy)<<endl;
            }
             }
    }
    return 0;
}
```

Here we will learn the solution step by step:-
we will see how I make the build:-

```
void build(int si,int ss,int se,vector<ll>&segt,vector<ll>&v){
    if(ss==se){
        segt[si]=v[ss];
        return;
    }
    else{
        int mid=(ss+se)/2;
        build(2*si,ss,mid,segt,v);
        build((2*si)+1,mid+1,se,segt,v);
        segt[si]=segt[2*si]+segt[(2*si)+1];// only this line differs
for making build of segment tree. here sum of the node will be the sum
of left and right childs as agr 3 to 6 ki range ka sum batana ho to ,
to hume 3 to 4 and 5 to 6 ka batana hoga.segment tree me kisi bhi node
ka result chahe sum ho ya minimum uske child nodes pe depend krta h.
        return;
    }
}
```

Now we see for query:-

```
ll query(int si,int ss,int se,int l,int
r,vector<ll>&segt,vector<ll>&lazy){
    if(lazy[si]!=0){
        ll d=lazy[si];
        lazy[si]=0;// iska calculation krlia , to ab iss node me add
krnr ka h nhi kuch to lazy[si]=0 krdenge.
        segt[si]+=d*(se-ss+1);// jis node pe query krne aae ho, pehle
check krlo ki usme huch to nhi krna tha, agr add krna tha to iska matlb
h si node ko root mante hue uske jitne bhi child h un sab pe lazy[i]
add krna padega, aur si node ke kitne node h jisme wo lazy[i] add krna
h to wo h (se-ss+1), because hum ek range me hi to add krenge na, aur
us node ka range h se-ss+1, to hame basically itne elements me plus
krna h,to mtlb uss node(is node me sum of values which starts from
index ss of v to se index of v should be stored) me jo total aad hua wo
h segt[si]+=d*(se-ss+1). d*(se-ss+1) extra add hua h.
        if(ss!=se){
        lazy[2*si]+=d;
        lazy[(2*si)+1]+=d;
        // si node ne apna calcultion krdia ye maan ke ki usne apne
child nodes me pehle hi calulation krdia h, becoz tabhi usne ye
calculation likhi h segt[si]+=d*(se-ss+1), to si apne child nodes ko
```

```
bolega lazy[si] add krde ko, waise lazy[si] waise bhi uske sare child
nodes me add hota hi.
        // ab child node ke paas info aa chuka h ki ab un child nodes
ko apne sare(grand child,child , ultra grand child or many more) me kya
aur add krna h.
        }
    }
    if(ss>r || se<l){
        return 0;
    }
    if(l<=ss && r>=se){
        return segt[si]; // baithe bithae ans mil jae to bdia h
    }
   int mid=(ss+se)/2;
   ll l1=query(2*si,ss,mid,l,r,segt,lazy);
   ll r1=query(2*si+1,mid+1,se,l,r,segt,lazy);
   return l1+r1; // ye bus searching method h apne ans tk pohonch ne ke
liye, same approach h jaise minimum nikala tha , isme add krna padh rha
h , usme min(l,r) krna padta tha.
   }
```

Now we will see for update:-

```
void update(int si,int ss,int se,int l,int
r,vector<ll>&segt,vector<ll>&lazy,ll val)
   {
    if(lazy[si]!=0){
              ll d=lazy[si];
        lazy[si]=0;
        segt[si]+=d*(se-ss+1);
         if(ss!=se){
        lazy[2*si]+=d;
        lazy[(2*si)+1]+=d;
        // jo reason query me update krne ka tha whi reason iska bhi h.
        }
    }
    if(ss>r || se<l){
        return ;
    }
    if(l<=ss && r>=se){
        segt[si]+=val*(se-ss+1);// value add krna h , to value bhi usi
procedure se add hoga jos process se lazy[si] add hua tha.
          if (ss != se) {
           lazy[2 * si] += val;
           lazy[2 * si + 1]+= val;
```

```
            // si node ko apne sare child node me add krna tha hi, pr
si ne calculation pehle hi krli ki sb me add krne ke baad kya value
hoga segt[si] ka, to ab wo apne child node ko bol rha h ki unko apne
sare child node me val wali value aur add krni h.
        }
        return ;
    }

    int mid=(ss+se)/2;
    update(2*si,ss,mid,l,r,segt,lazy,val);
    update((2*si)+1,mid+1,se,l,r,segt,lazy,val);
    segt[si] = segt[2 * si] + segt[2 * si + 1];
    // sara update krne ke baad new values ye aaengi.
   }
```

## Purpose of the arrays taken:-

```
vector<ll>v(k+1,0);//orignal array.

        vector<ll>segt(4*k);// segment tree array.

        vector<ll>lazy(4*k);// ye wala array/lazy[i] btaega ki uss
node(ith node) ko root mante hue uske sare child nodes pe lazy[i] value
addd krni h.
```