INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

CSE603

ADVANCED PROBLEM SOLVING

PERSISTENT SEGMENT TREE

Author:
Kshitij Paliwal
Sivangi Singh

Roll Number:
2018201063
2018201001

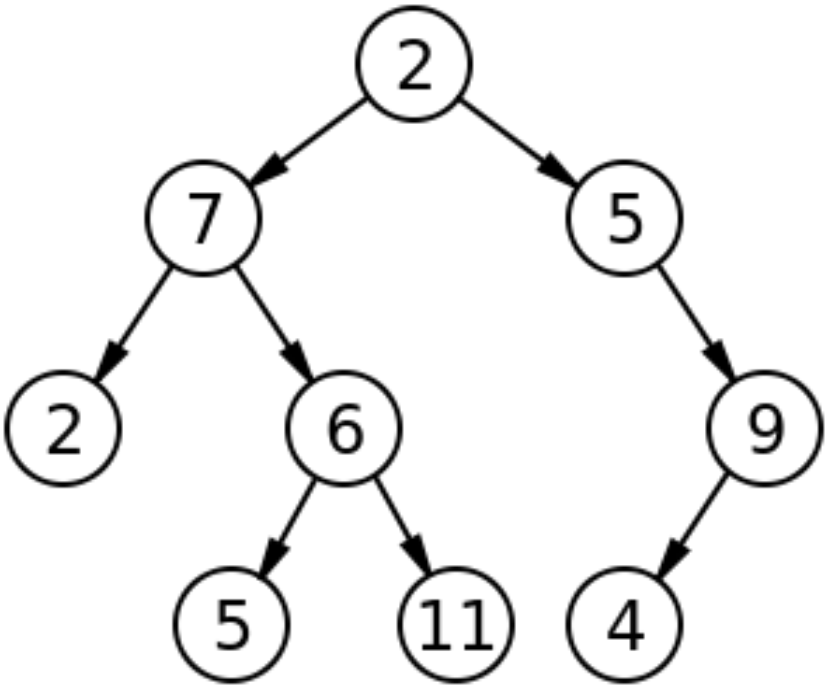
November 12, 2018

Contents

1 Introduction

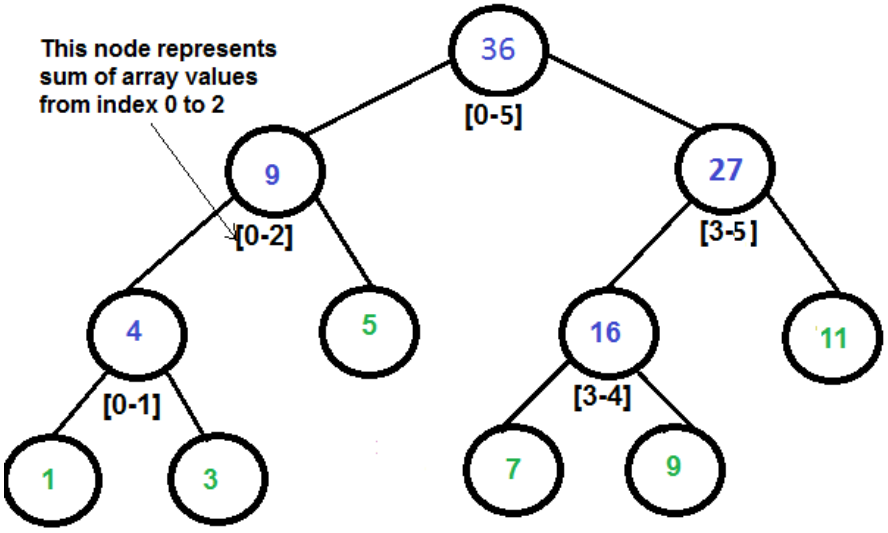
1.1 TREE :

Tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.



1.2 SEGMENT TREE :

A segment tree also known as a statistic tree is a tree data structure used for storing information about intervals, or segments. It allows querying which of the stored segments contain a given point. It is a height-balanced binary tree, usually built on top of an Array. Segment Trees can be used to solve Range Min/Max & Sum Queries and Range Update Queries in O(log n) time. It is, in principle, a static structure; that is, it's a structure that cannot be modified once it's built. A segment tree of n intervals uses O(n) storage and can be built in O(n) time. Segment trees support searching for all the intervals that contain a query point in O(log n + k), k being the number of retrieved intervals or segments.



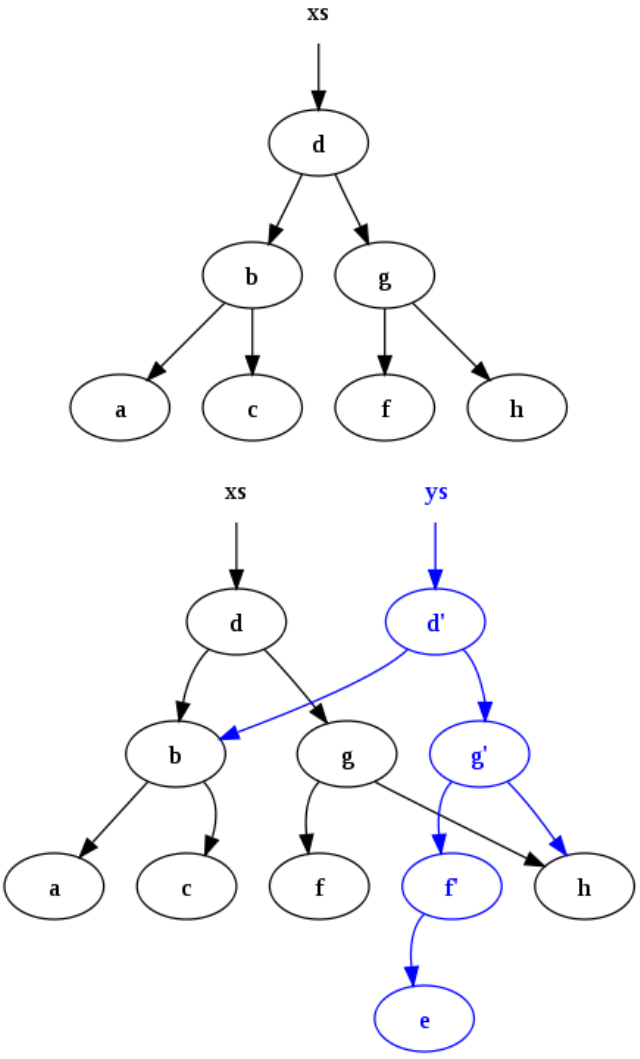
Segment Tree for input array {1, 3, 5, 7, 9, 11}

Definition of a Node of Segment Tree:

```
struct seg_tree{
    int value;
    struct seg_tree *left,* right;
};
```

1.3 PERSISTENT DATA STRUCTURE :

A persistent data structure is a data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure.These types of data structures are particularly common in logical and functional programming, and in a purely functional program all data is immutable, so all data structures are automatically fully persistent.Persistent data structures can also be created using in-place updating of data and these may, in general, use less time or storage space than their purely functional counterparts.



2 PERSISTENT SEGMENT TREE :

Persistence is a concept in data structures, where we reuse the original state of data structure whenever some update is required. This results in a data structure where we can see the state of data structure before and after the update with minimal use of memory. Segment trees are interval trees over an array where we can store information about segments. It allows us to update the information regarding some segment of the array in O(log(n)) time complexity.

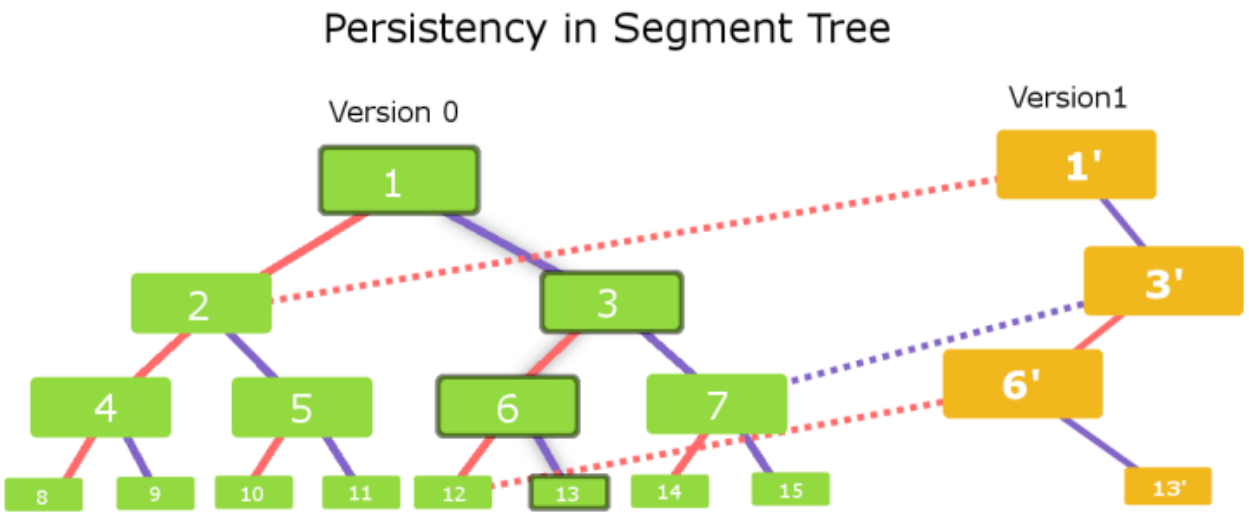
Consider a tree with N nodes labelled 1 to N. We need to perform queries that require us to create segment trees for each node. To build M such segment trees, we require O(N*M) time and memory. Suppose a segment tree is made for a node A. For any child B of A, the segment tree of B will be almost similar to that of A except for O(log(N)) nodes. We can reuse the remaining nodes from the segment tree of A and allocate memory for these O(log N) nodes. We allocate memory for O(log(N)) nodes per node in the tree. Memory and the time complexity are reduced to O(M*log(N)).

The new trees for the children differ from the parent's segment trees by O(log(N)) nodes. By taking advantage of this property, we can reduce the time required to build segment trees for all the nodes to O(M*log(N)).

Our aim is to apply persistency in segment tree and also to ensure that it does not take more than O(log n) time and space for each change.We will consider our initial version to be Version-0. Now, as we do any update in the segment tree we will create a new version for it and in similar fashion track the record for all versions. But , segment tree we need to create the whole tree for every version which will take O(n) extra space and O(n) time for each version.

BUT, At max logn nodes will be modified. So, our new version will only contain these log n new nodes and rest nodes will be the same as previous version. Therefore, it is quite clear that for each new version we only need to create these log n new nodes whereas the rest of nodes can be shared from the previous version.

Consider the below figure for better visualization :

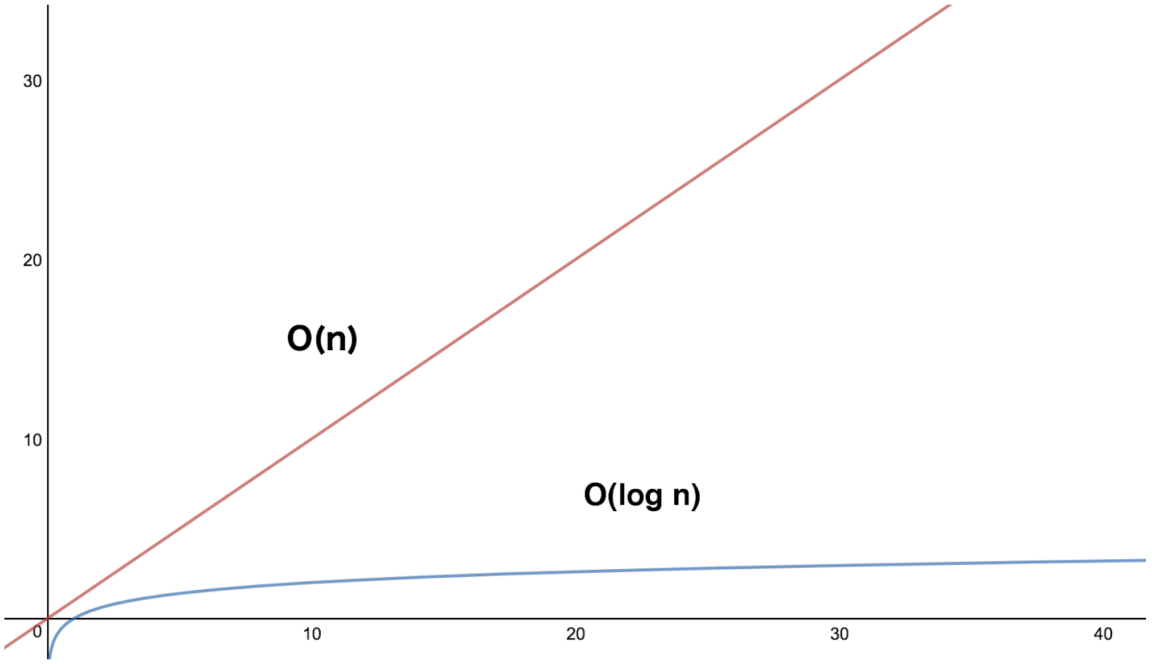


Consider the segment tree with green nodes . Let this segment tree BE version-0. The left child for each node is connected with solid red edge where as the right child for each node is connected with solid purple edge. Clearly, this segment tree consists of 15 nodes.

Now consider we need to make change in the leaf node 13 of version-0. So, the affected nodes will be – node 13 , node 6 , node 3 , node 1. Therefore, for the new version (Version-1) we need to create only these 4 new nodes.Now, lets construct version-1 for this change in segment tree. We need a new node 1 as it is affected by change done in node 13. So , we will first create a new node $1'$. The left child for node $1'$ will be the same for left child for node 1 in version-0. So, we connect the left child of node $1'$ with node 2 of version-0. Now, examine the right child for node $1'$ in version-1. We need to create a new node as it is affected . So we create a new node called node $3'$ and make it the right child for node $1'$.

Similarly, now examine node $3'$. The left child is affected , So we create a new node called node $6'$ and connect it with node $3'$, where as the right child for node $3'$ will be the same as right child of node 3 in version-0. So, we will make the right child of node 3 in version-0 as the right child of node $3'$ in version-1.
Same procedure is done for node $6'$ and the left child of node $6'$ will be the left child of node 6 in version-0 and right child is newly created node called node $13'$. Each yellow color node is a newly created node and dashed edges are the inter-connection between the different versions of the segment tree.

Graph of N V/s LogN:



3 ALGORITHMS

3.1 SEGMENT TREE:

Below is the Algorithm to create a segment tree

```
Insert( arr[], i, j):
 1. if i=j
        create_node(arr[i])
        return
 2. mid:=(i+j)/2
 3. left:=insert( arr, i, mid )
 4. right:=insert( arr, mid+1, j )
 5. val:=funct( left->value, right->value )
 6. node:=create_node(val)
 7. node->left:=left
 8. node->right:=right
```

To keep all the versions of segment tree by not using persistence , every time whole segment tree needs to be created which is time as well as space inefficient.Hence , Persistence segment tree can used.

Code snippet of Creation of Segment Tree:

```
seg_tree * insert_value(int arr[],int i ,int j,int (*functocall)(int,int)){
    seg_tree * node,* left , * right;
    if(i==j){
        node=create_node(arr[i]);
        return node;
    }
    int mid=(i+j)/2,val;
    left=insert_value(arr,i,mid,functocall);
    right=insert_value(arr,mid+1,j,functocall);
    val=functocall(left->value,right->value);
    node=create_node(val);
    node->left=left;
    node->right=right;
    return node;
}
```

3.2 PERSISTENT SEGMENT TREE:

To create first version of persistent segment tree, Same Algorithm of Insertion is used as in Normal Segment Tree, and then for every other version Update Algorithm is used as described below:

Update Algorithm-

```
update( value, l, r, i, j, root ):
 1. mid:=(i+j)/2
 2. if i=j and i=l and r=l
      create_node(value)
      return
 3. if l<=mid
      left:=update( value, l, r, i, mid, root->left )
 4. else
      left:=root->left
 5. if r>=mid+1
      right:=update( value, l, r, mid+1, j, root->right )
 6. else
      right:=root->right
 7. val:=funct( left->value, right->value )
 8. node:=create_node( val)
 9. node->left:=left
10. node->right:=right
```

Code snippet of Update of Persistent Segment Tree:

```
seg_tree *update_seg(int value,int l ,int r,int i ,int j
    ,seg_tree * previous_root,int (*functocall)(int ,int )){
    seg_tree * node,* left , * right;
    node=(seg_tree *)malloc(sizeof(struct seg_tree));
    int mid=(i+j)/2,val;
    //cout<<i<<" "<<j<<l<<" "<<r<<" "<<mid<<"\n";
    if(i==j && i==l && r==j){
        node=create_node(value);
        return node;
    }
    if(l<=mid){
    left=update_seg(value,l,r,i,mid,previous_root->left,functocall);
    }
    else{
    left=previous_root->left;
    }
    if(r>=mid+1){
    right=update_seg(value,l,r,mid+1,j,previous_root->right,functocall);
    }
    else{
    right=previous_root->right;
    }
    val=functocall( left->value,right->value);
    node=create_node(val);
    node->left=left;
    node->right=right;
    return node;
}
```

Hence, on every update O(logn) time and space will be utilised making it efficient and now we have all the versions stored and are easily accesible.
In the above Algorithms, function **funct** is the type of segment tree that needs to be created , funct **performs the specified operation, such as min , max or addition.**

Code snippet of Function to be Performed:

```
int sum (int a, int b)
{
    return a+b;
}

int max (int a, int b)
{
    if(a>b)
        return a;
    return b;
}

int min_m(int a ,int b){
    if(a>b)
        return b;
    return a;
}
```

```
int (*functocall)(int ,int);
if(operationToPreform.compare("sum")==0){
    functocall=sum;
}
else if(operationToPreform.compare("min")==0){
    functocall=min_m;
}
else if(operationToPreform.compare("max")==0){
    functocall=max;
}
else{
    cout<<"Invalid Operation\n";
    exit(0);
}
```

4 Graph of Persistent Segment Tree and Segment tree

4.1 Query v/s Time Graph:

Keeping Size of Array constant i.e. 10000 and then applying various Test Cases by Varying the Query size and then Calculating the time taken by both Persistent Segment Tree and Normal Segment Tree (which has all the versions).

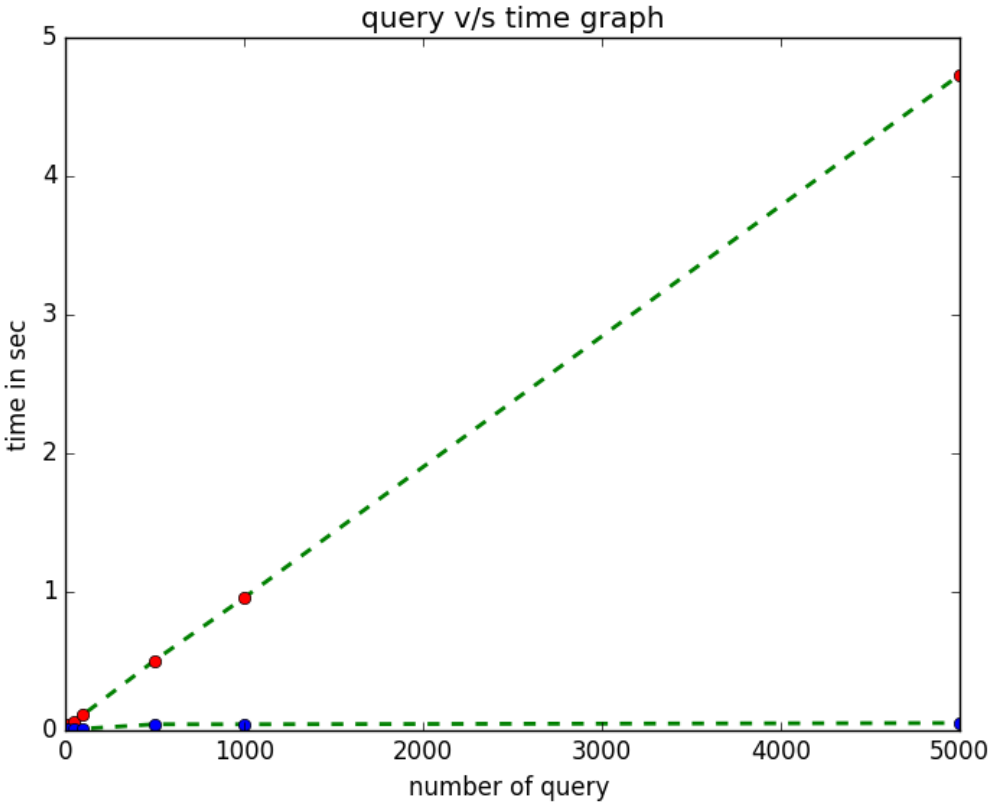
The time complexity of creation in Normal Segment Tree is O(n) , it will take O(n) to create each version for every update making the graph grow linearly with increase in number of queries.

whereas, The time complexity of creation of each version in Persistent Segment Tree is O(logn) , it will take O(logn) to create each version for every update making the graph will be similar as logn curve with increase in number of queries.

Table of Various Test Cases:

queries	Persistent Seg tree	Segment Tree
10	0.008414	0.044289
50	0.009445	0.063789
100	0.012028	0.1134489
500	0.0442206	0.504911
1000	0.044017	0.957499
5000	0.053172	4.730650

Graph:



Blue dots: Persistent Segment Tree
Red dots: Normal Segment Tree

4.2 Query v/s Space Graph:

Keeping Size of Array constant i.e. 10000 and then applying various Test Cases by Varying the Query size and then Calculating the size taken by both Persistent Segment Tree and Normal Segment Tree (which has all the versions).

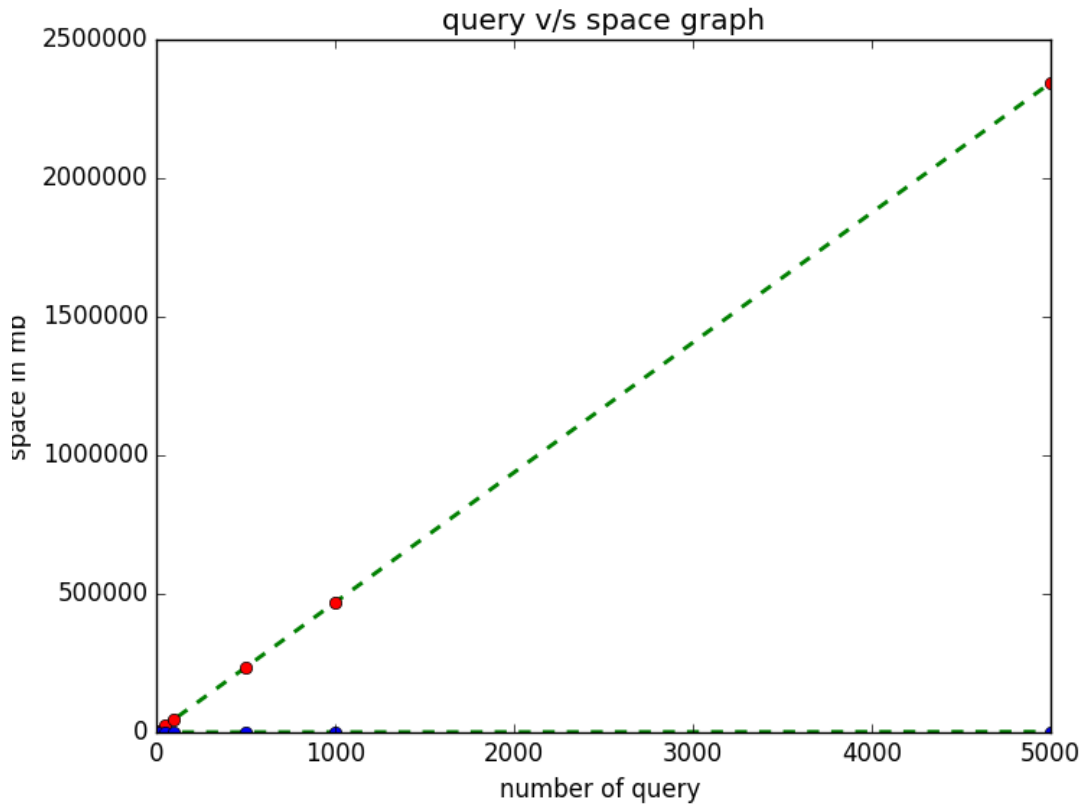
The size complexity of creation in Normal Segment Tree is O(n) , it will take O(n) to create each version for every update making the graph grow linearly with increase in number of queries.

whereas, The size complexity of creation of each version in Persistent Segment Tree is O(logn) , it will take O(logn) to create each version for every update making the graph will be similar as logn curve with increase in number of queries.

Table of Various Test Cases:

queries	Persistent Seg tree	Segment Tree
10	472.125	5155.9921875
50	485.5078125	23905.0546875
100	502.359375	47341.328125
500	637.59375	234832.0078125
1000	805.546875	469195.2890625
5000	2152.546875	2344101.5390625

Graph:



Blue dots: Persistent Segment Tree
Red dots: Normal Segment Tree

5 BENCHMARK

Given an array a[1 ... n] of different integer numbers, It is required to generate a program that answer a series of queries Q(i, j, k) in the form: "What would be the k-th number in a[i ... j] segment, if this segment was sorted?"

5.1 Naive Approach:

This problem can be solved by taking elements from the range of index from i to j (elements from i to j including i and j = m elements), sorting them in some data structure and then taking the kth element from the sorted structure. Repeating this for every query i.e. q times.

For example, In the array a = (1, 5, 2, 6, 3, 7, 4). Let the question be Q1(2, 5, 3) and Q2(4, 6, 2). For Q1, The segment a[2 ... 5] is (5, 2, 6, 3). If we sort this segment, we get (2, 3, 5, 6), the third number is 5, and therefore the answer to the query is 5.
Similarly, for Q2, The segment a[4 ... 6] is (6, 3, 7). If we sort this segment, we get (3, 6, 7), the second number is 6, and therefore the answer to the query is 6.

```
Naivekthnum( i, j, k ):
1. for l from i to j do:
       push arr[l] to vector v
2. sort v
3. return v[k]
```

This is the Naive approach and will take time O(n) to first take the range elements then O(nlogn) time to sort them and O(1) to extract kth element. If there are q queries then this n + nlogn + 1 will be repeated q times i.e. O(q*nlogn) time.

But, This time can be reduced by using **Persistent Segment Tree**

5.2 Persistent Segment Tree Approach:

We will make persistent segment tree for count according to the index from original array taking elements in ascending order.

Take another array vec other than Original array arr which is equal to arr,Sort it. Now, Consider an array count with all elements as 0. Find the index of first element of arr in vec , make that index value of count array as 1 and make segment tree of this count array with the function used as sum. Push the root of this segment tree in a vector root_arr.

From now take one element of arr at a time ,find its index in vec and update that index value of count array as 1 and hence make persistent segment tree of this updated array ,pushing the new root to the root_arr vector.

Now, for every query Q(i,j,k) take the roots of tree i and tree j from the root_arr vector, according to the difference of both nodes values and k , maintaining the index range values ,traverse to the required leaf. If the value of k is less than the difference then going to the left child of both trees else going to the right child.

Here, Sorting is required only once which is O(nlogn) . Since, for every Query we are only traversing the tree till the leaf, taking logn time. For q queries O(qlogn) time is required. Hence making it efficient and better than the naive Approach as the total time complexity would be O(nlogn + qlogn) and for large value of q it will eventually be O(qlogn).

```
insert_value( arr[], vec[]):

1. count[arr_size]={}
2. root:=insert( count, 0, n-1 )
3. push root in vector root_arr
4. for i from 0 to n :
5.    find index of arr[i] in vec
6.    count[index]:=1
7.    root:= update( 1, index, 0, n-1, root )
8.    push root in vector root_arr
```

Above is the Algorithm to build the persistent segment tree of count array, so that the queries can be made on it.

Code snippet of Insertion in Persistent Kthnum Segment Tree:

```
vector< seg_tree * > insert(int arr[],std::vector<int> v,int arr_size){
    int count[arr_size]={};
    int index=0;
    vector <seg_tree * >root_arr;
    seg_tree * root=insert_value(count,0,arr_size-1);
    root_arr.push_back(root);
    for(int i=0;i<arr_size;i++){
        index=lower_bound(v.begin(),v.end(),arr[i])-v.begin();
        count[index]=1;
        root=update_seg(1,index,0,arr_size-1,root);
        root_arr.push_back(root);
    }
    return root_arr;
}
```

the Algorithm for running queries is-

```
query_knum( root1, root2, i, j, k ):

1. if root1 and root2 are leaf
     return j
2. mid:=(i+j)/2
3. val:=root1->left->value-root2->left->value
4. if val>=k
     return query_knum( root1->left, root2->left, i, mid, k )
5. return query_knum( root1->right, root2->right, mid+1, j, k-val )
```

Code snippet of Query Handling in Persistent Kthnum Segment Tree:

```
int knum(seg_tree * root1,seg_tree * root2,int i,int j,int k){
    if(root1->left==NULL && root1->right==NULL)
    {
        return j;
    }
    int mid=(i+j)/2;
    int val=(root1->left->value-root2->left->value);
    if(val>=k){
        return knum(root1->left,root2->left,i,mid,k);
    }
    return knum(root1->right,root2->right,mid+1,j,k-val);
}
```

In the same example with array a=(1, 5, 2, 6, 3, 7, 4).Storing its Sorted form in an array v as v=(1, 2, 3, 4, 5, 6, 7).Initially count array will be created with all n elements as 0(n=7 here) i.e. count=(0, 0, 0, 0, 0, 0, 0). Creating segment tree of this count array. Now taking elements of array v at a time starting from v[1] i.e. 1 and finding its index in array a (index of 1 in a is 1 and index of 2 is 3) and updating that index value of count array as 1 ,like for 1 count[1]=1 , for 2 count[3]=1 and so on, Each time creating persistent segment tree of the updated value in count and hence taking logn time and space for this operation. Root of every segment tree hence created are kept in a vector root_arr.
The value of count array for every version will be-

version 0 - 0, 0, 0, 0, 0, 0, 0
version 1 - 1, 0, 0, 0, 0, 0, 0
version 2 - 1, 0, 0, 0, 1, 0, 0
version 3 - 1, 1, 0, 0, 1, 0, 0
version 4 - 1, 1, 0, 0, 1, 1, 0
version 5 - 1, 1, 1, 0, 1, 1, 0
version 6 - 1, 1, 1, 0, 1, 1, 1
version 7 - 1, 1, 1, 1, 1, 1, 1

The post order traversal of persitent segment tree for the different versions are-

version 0 - 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
version 1 - 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1
version 2 - 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 2
version 3 - 1, 1, 2, 0, 0, 0, 2, 1, 0, 1, 0, 1, 3
version 4 - 1, 1, 2, 0, 0, 0, 2, 1, 1, 2, 0, 2, 4
version 5 - 1, 1, 2, 1, 0, 1, 3, 1, 1, 2, 0, 2, 5

version 6 - 1, 1, 2, 1, 0, 1, 3, 1, 1, 2, 1, 3, 6
version 7 - 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 3, 7

Now taking one query at a time , on the basis of the values of i and j take versions i-1 and j from the root_arr as root2 and root1 and passing them to the function query_num which takes two root nodes of persistent segment tree and according to the value of k traverse the trees by taking the difference of values of both roots and comparing it with value of k and hence deciding weather to next traverse right or left. when leaf is reached the desired index which contains the kth element is known and hence required element is known.
For query Q1(2, 5, 3) the roots of version 5 and version 1 will be taken and 3rd element is to be found.

```
iteration 1-query_num(root of version 5, root of version 1, 1, 7, 3)
                root1 and root2 are not leaf
                mid=4      (as (1+7)/2=4)
                val=2      (root2(value of left child)=3,root1(value)=1, 3-1=2)
                k!<=val      (4>2)
                query_knum( root1->right, root2->right, 4+1, 7, 1 )
iteration 2-query_num(root2, root1, 5, 7, 1)
                root1 and root2 are not leaf
                mid=6      (as (5+7)/2=6)
                val=2      (root2(value of left child)=2,root1(value)=0, 2-0=2)
                k<=val      (1<2)
                    query_num(root1->left, root2->left, 5, 6, 1 )
iteration 3-query_num(root2, root1, 5, 6, 1)
                root1 and root2 are not leaf
                mid=5      (as (5+6)/2=5)
                val=1      (root2(value of left child)=1,root1(value)=0, 1-0=1)
                k<=val      (1=1)
                    query_num(root1->left, root2->left, 5, 5, 1 )
iteration 4-query_num(root2, root1, 5, 5, 1)
                root1 and root2 are leaf
                return 5
```

This returned value is the index of array v and the required answer will be v[index] i.e 5. Similarly, Q2(4, 6, 2) can be found by following the same procedure answer will be 6.

6 Graphs of Persistent Kthnum and Naive Kthnum

6.1 Query v/s Space Graph:

Keeping the size of array as Constant i.e. 101000 and varying the number of Queries , the behaviour of space taken by both Persistent Kthnum Segment Tree and Naive Kthnum with various Test Cases were Observed.

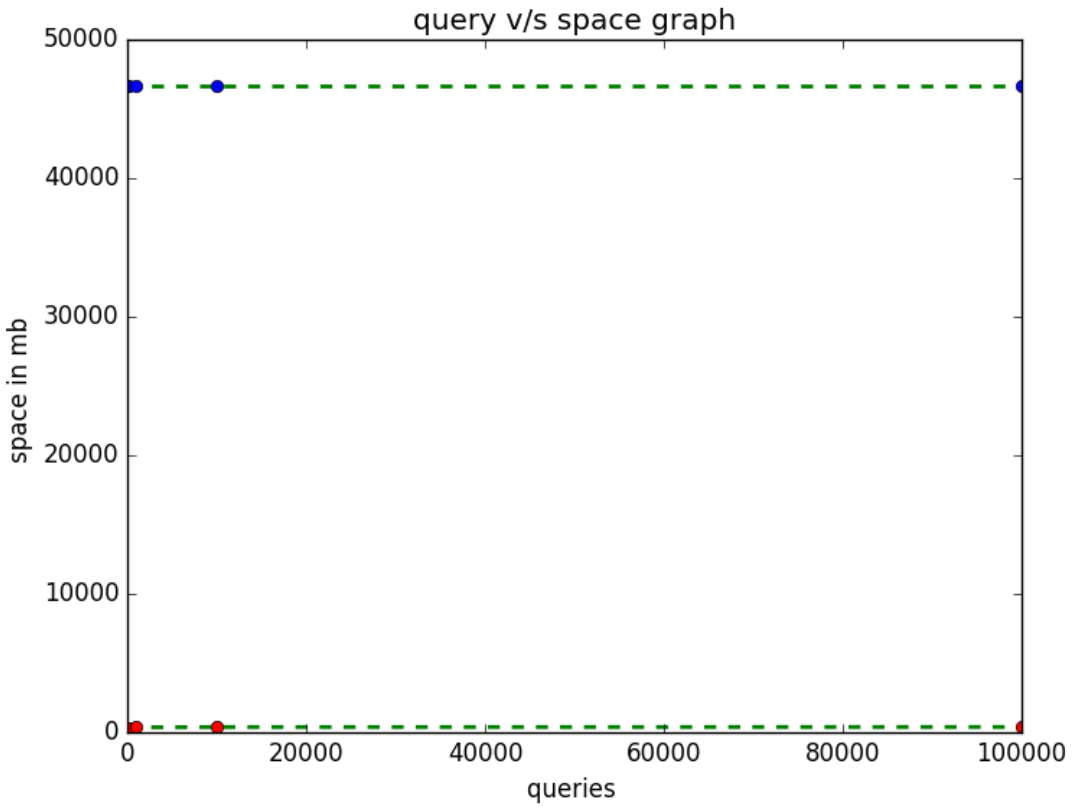
For Persistent Kthnum Segment Tree, the versions were already created and will be same not depending on the number of Queries and for every Query only Traversal is to be done.The size taken here does not depend on the number of Queries hence, the graph will be constant with varying number of queries.
For Naive Kthnum, the maximum length for a query can be n which is constant ,hence it will also show almost constant behaviour of size with increase in number of queries.

Both the Constant values for Persistent Kthnum Segment Tree as well as Naive Kthnum are different and Persistent one will take much larger space as it creates version for every element in array, each version taking logn extra space and Naive Approach doesn't do this.

Table of Various Test Cases:

queries	Kthnum	Naive
10	46638.9140625	392.6640625
100	46638.9140625	353.65625
1000	46638.9140625	366.53515625
10000	46638.9140625	386.98828125
100000	46638.9140625	392.6640625

Graph:



blue dots: Persistent Kthnum Segment Tree
red dots: Naive Kthnum

6.2 Query v/s Time Graph:

Keeping the size of array as Constant i.e. 101000 and varying the number of Queries , the behaviour of Time taken by both Persistent Kthnum Segment Tree and Naive Kthnum with various Test Cases were Observed.

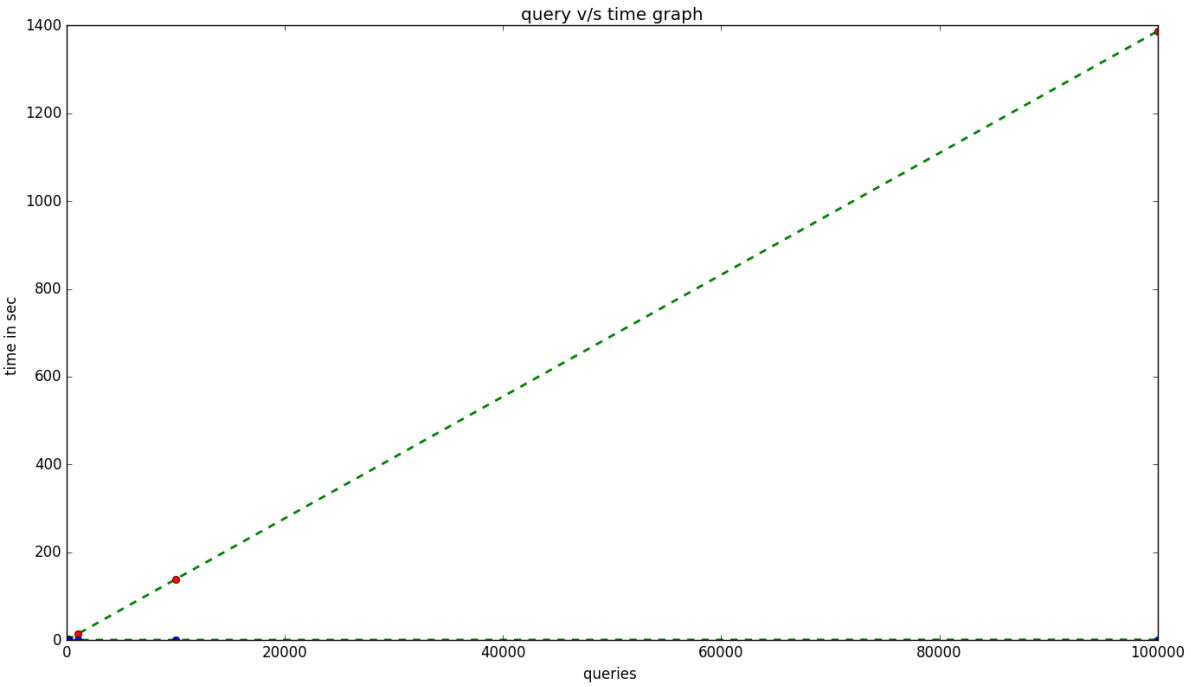
The time complexity of handling one query in Naive Kthnum is O(nlogn) , it will take total O(q*nlogn) for all the q queries.Hence, the graph will vary accordingly with the increase in number of queries.

whereas, The time complexity of handling one query in Persistent Kthnum Segment Tree is O(logn) , it will take total O(q*logn) for all the q queries.Hence, the graph will vary accordingly with the increase in number of queries.

Table of Various Test Cases:

queries	Kthnum	Naive
10	0.513133	0.274032
100	0.505762	1.528225
1000	0.513299	14.128664
10000	0.570361	139.143845
100000	1.099634	1386.652832

Graph:



blue dots: Persistent Kthnum Segment Tree
red dots: Naive Kthnum

6.3 Size of Array v/s Space Graph:

Keeping the number of queries as Constant i.e. 100 and varying the size of Array , the behaviour of Space taken by both Persistent Kthnum Segment Tree and Naive Kthnum with various Test Cases were Observed.

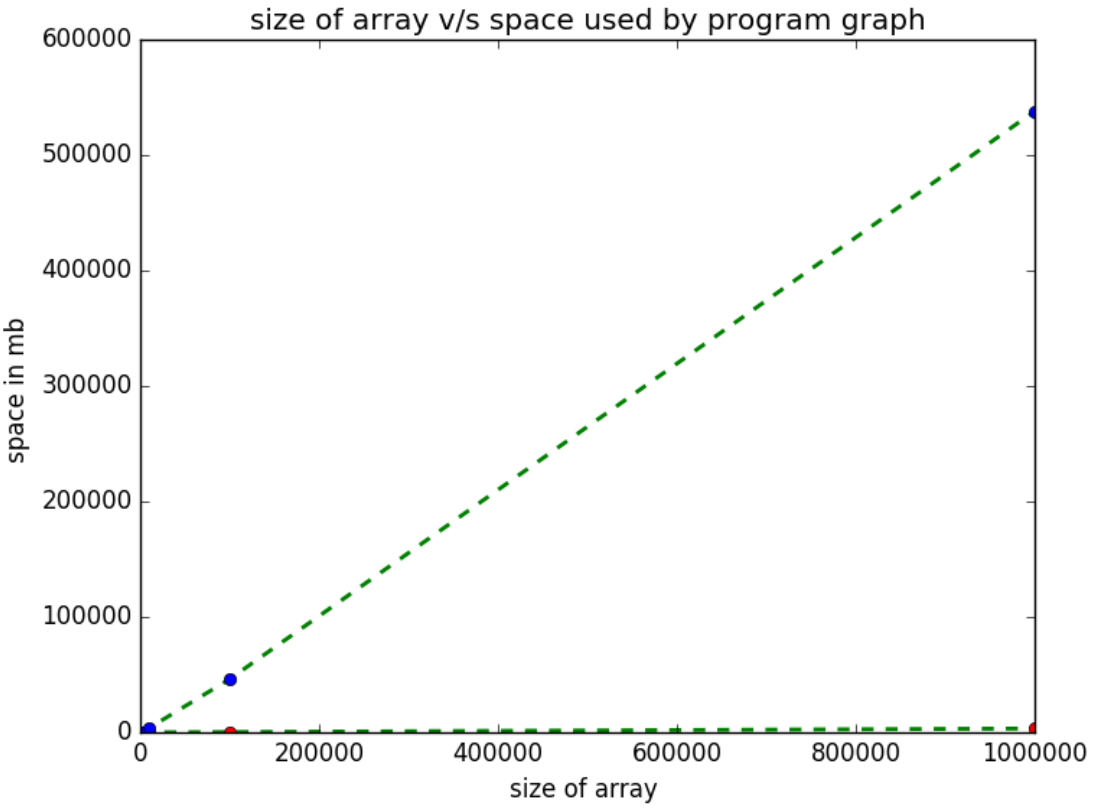
The space complexity of handling one query in Naive Kthnum is O(n) , it will take total O(q*n) for all the q queries , q here is constant i.e. 100 therefore it eventually becomes O(n).Hence, the graph will vary accordingly with the increase in number of elements.

whereas, The space complexity of handling one query in Persistent Kthnum Segment Tree is O(logn) , it will take total O(q*logn) for all the q queries,q here is constant i.e. 100 therefore it eventually becomes O(logn).Hence, the graph will vary accordingly with the increase in number of queries.

Table of Various Test Cases:

Number of Elements	PersistentKthnum	Naivekthnum
200	50.2265625	0.6171875
1000	304.1015625	3.20703125
10000	3834.7065625	36.37109375
100000	46146.7265625	311.171875
1000000	537923.9765625	3369.8203125

Graph:



blue dots: Persistent Kthnum Segment Tree
red dots: Naive Kthnum

6.4 Size of Array v/s Time Graph:

Keeping the number of queries as Constant i.e. 100 and varying the size of Array , the behaviour of Time taken by both Persistent Kthnum Segment Tree and Naive Kthnum with various Test Cases were Observed.

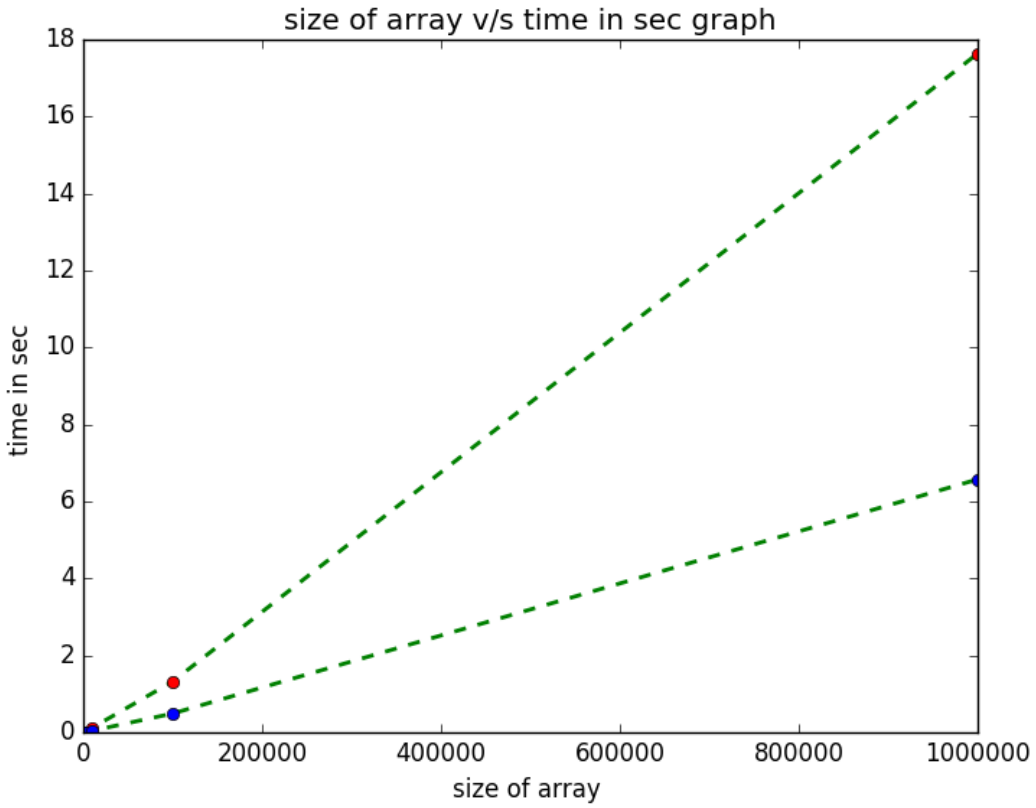
The time complexity of handling one query in Naive Kthnum is O(nlogn) , it will take total O(q*nlogn) for all the q queries , q here is constant i.e. 100 therefore it eventually becomes O(nlogn).Hence, the graph will vary accordingly with the increase in number of elements.

whereas, The time complexity of handling one query in Persistent Kthnum Segment Tree is O(logn) , it will take total O(q*logn) for all the q queries,q here is constant i.e. 100 therefore it eventually becomes O(logn).Hence, the graph will vary accordingly with the increase in number of queries.

Table of Various Test Cases:

Number of Elements	PersistentKthnum	NaiveKthnum
200	0.001958	0.004301
1000	0.005211	0.013897
10000	0.043403	0.119289
100000	0.490195	1.320655
1000000	6.574135	17.635662

Graph:



blue dots: Persistent Kthnum Segment Tree
red dots: Naive Kthnum

7 END USER DOCUMENTATION

1.For running Persistent Segement Tree Complie And Run persistent_segment_tree.cpp and then provide the proper input according to prompted by program.

2.For running Segement Tree Complie And Run segment_tree.cpp and then provide the proper input according to prompted by program.

3.For running navie Knthnum Complie And Run navieKNthNum.cpp and provide input as follow:

- First provide number of elements of array N
- Then Provide number of Queries Q
- Then Provide N elements of array
- Then Provide i j and k for each query

4.For running Persitent Segment Tree KnthNum Complie And Run knthnum.cpp and provide input as follow:

- First provide number of elements of array N
- Then Provide number of Queries Q
- Then Provide N elements of array
- Then Provide i j and k for each query

5.For running Genralized Persitent Segment Tree KnthNum (which can handle both max and min)Complie And Run knthnum_min_max.cpp and provide input as follow:

- First provide number of elements of array N
- Then Provide number of Queries Q
- Then Provide 1 for max kth Number and 2 for min kth Number
- Then Provide N elements of array
- Then Provide i j and k for each query

6.For Test Cases look into testcase folder.

7.For Output Related to testcases look into output Folder.

8.For Graphs Created through the produced output look into graph folder.

GitHub Link

https://github.com/kshitij02/Persistent_Segment_Tree

8 REFERENCES

1.https://en.wikipedia.org/wiki/Segment_tree
2.https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/
3.https://www.geeksforgeeks.org/persistent-segment-tree-set-1-introduction
4.https://www.youtube.com/watch?v=m82htYhBQy