Segment trees, simple, dynamic, and persistent

M.Besher Massri
Jožef Stefan International Postgraduate School, Slovenia
besher.massri@ijs.si

ABSTRACT

In this seminar, we present the segment tree data structure. This data structure is mainly used to answer range query tasks over an array of elements. It is one of the common data structures used in competitive programming environments. We will describe the simple segment trees, and cover two variations of it; namely dynamic and persistent segment trees.

KEYWORDS

Segment trees, Data structures, Competitive programming

1 INTRODUCTION

Query tasks are a common type of problems in competitive programming. The essence of such problems is this: given an array of numbers, and a list of questions to be that updates or query on these list. Such problems can have multiple variations like operating with empty array and adding and removing element as we go, having single item query or range queries, querying about the status of the array at an old point in time, etc.

2 SEGMENT TREES

Segment tree is a binary tree where each node is either a leaf or has exactly two children. Each node in the tree represents a range of consecutive items in the array, and holds some pre-calculated values representing the answer of a certain operation(s) along with possible other information. In figure 1, we see an example of a segment tree. Each leaf node is responsible for a single element, or a range of length one. For each internal node and the root, it's responsible for the combination of its left and right child ranges. Since by design each node is either a leaf or has exactly two children, the height of such tree is at most $\lceil log_2 N \rceil$ where N is the number of the items in the tree. If N is a power of two, the segment tree becomes a perfect binary tree. The $\lceil log_2 N \rceil$ height of the tree is what allows for efficient logarithmic complexity of most operations.

The essence of the segment tree is by pre-calculating the target value for carefully selected ranges of the array, we can operate efficiently and find solution to queries that cover any ranges, as well as enabling updates of the array items without changing a lot of the pre-calculations. The segment tree can answer queries like sum of range, or other operations like max,min,avg,etc.

The object for the a single node is as follows

```
struct Node{
    int val;
    int lzy;
};
```

Where *val* represent the value of the segment tree node, and the *lzy*, represent the cached value for the lazy update (more on that later). The type of the variable is defined by the type of problem, which could be "float" or "double" (in C++) if the problem deals with decimal numbers, or "long long" if the range of numbers exceeds 4-bytes integers. Furthermore, the number of *val* and *lzy* variables could be more than one in some scenarios if the target operation requires more than one variable, or the tree is built to support multiple operations at the same time.

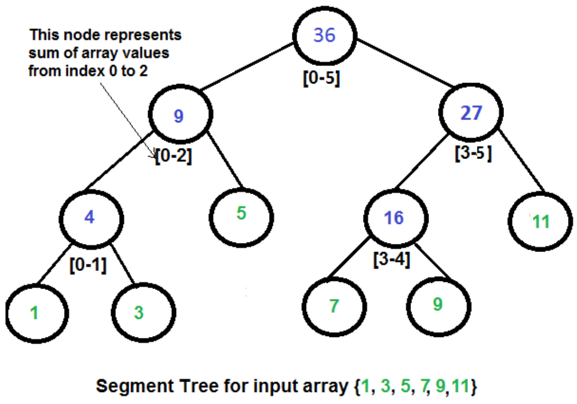
This node represents sum of array values from index 0 to 2
36
[0-5]
9
[0-2]
27
[3-5]
4
5
16
11
[0-1]
[3-4]
1
3
7
9
Segment Tree for input array {1, 3, 5, 7, 9,11}

Figure 1: Example of segment tree data structures, from [4]

2.1 Data representation

While the tree the can be constructed using pointers to connect each node with its children/parent. A more efficient representation is to store them in an array in compact form. Each node in index x, has two children, the left is stored at $x * 2$ and the right is stored at index $x * 2 + 1$, with the root set at index 1. Based on this representation, the parent of a node at index x will be $\lfloor \frac{x}{2} \rfloor$. This representation allows the navigation from a node to its children or parent without explicitly storing this information. Furthermore, all data can be fit in array of length 4N at most. Moreover, we saw that each node represents a range of numbers in the array, (like [0-5] for the root node, and [0-2] for its left child in figure 1. However, the boundaries of such range also does not need to be stored and can be deduced on the fly as follows: starting from the root, which is responsible for all elements of the array (a[0..n-1]) hence $l = 0, r = n - 1$. The left child is responsible for the first half, precisely from 1 to $(l + r)/2$, and right child from the element after that, $(l + r)/2 + 1$ till the end of the range, i.e. r, and the rest of the ranges follows recursively.

2.2 Merge function

For the segment tree to work, a key function is the merge function. The merge function is responsible for calculating the node value based on the combination of the two child nodes. The combining method depends on the operation to be performed which is problem-dependent. For example, the merge function for a sum query is as simple as this:

```
void merge (Node& x, Node& a, Node& b){
    x.val=a.val+b.val;
}
```

As the merge is problem-dependent, we will discuss further the merge function for each of the chosen experiments/problems.

2.3 Build

The build function is for initializing the values of each node based on the range it represents. The tree is initiated from bottom to top, where the value of the leaf nodes is the value answer of the query for the single element that the node is responsible for, which is equal to the same number in most of the cases, like in sum/min/max functions. However, it could be different sometime, e.g. if the query is to count the longest valid sequence of brackets, in which we would need to store three variables for the matched, right open unmatched, and left open unmatched brackets. a typical build function is as follows:

```
void build (int x,int l,int r){
    if (l==r){
        node[x].val=a[l];
    }
    build(x*2,l,(l+r)/2);
    build(x*2+1,(l+r)/2+1,r);
    merge(node[x],node[x*2],node[x*2]+1);
}
```

2.4 Query

The query method is responsible for generating the answer to the target function for the given sub array. The query function, usually implemented recursively, works in this way, starting from the root node, which is responsible for the entire range, at each stage, it examine the range covered by the node, if it's completely covered by the target range, then it returns the value of the node. If not then we have one of three cases: the target range is completely in the left node range, the target range is completely in the right node range, or it's partially covered by both the left and the right node. In the first and second case, the answer at the current node is given by the answer of the left and right children, respectively. However, in the third case, the answer returned is a combined answer using the merge function from the answer return from the left child (which will be covered partially by it, hence its value might be different from the value of the left child's node), and the right child.

An example of the range query with operation as sum is illustrated in figure 2, where the query is asking for the elements from the third till the fifth, i.e. $\sum_{2<=i<=4} a_i$. As explained before, we start with the root, we see that it covers items with index from 0 to 4, which doesn't fall entirely within the target range ([2,4]), hence we check the the left and right children's range, we see that the target range is covered partially by both children. Hence, we start by going to the left child. Again the range is partially intersected with the target, but in this time, the left child (the node responsible for range [0,1]) is totally out of the range, and the right node falls under it, hence we move to the right node (a[2..2]). In this node, we see it entirely falls within the target range ([2,4]) hence, we return its value, which is -2 in this case. Going back up, the value return from node a[0..2] is the same of the value of the right node since its left child was out of range. By now, we have the value from the left child of the root node, which is -2, we we visit the right child of the root (node a[3..4]), we observe that it entirely falls within the target range, hence, we immediately return the value stored in the node, 1, without going further. Finally, the two values return from both children are combined using the merge function, which in this case is just a simple addition of values. Hence, the final value returned from the root is $2 + 1 = 3$ which is the answer to the query.

A typical implementation of the query function is below. Note that the lazy_update function call will be explained later in range updates section.

```
Node query(int x,int l,int r,int s,int e){
    lazy_update(x, l, r);
    if (s<=l && e>=r)
        return node[x];
    const int mid=(r+l)/2;
    if (mid<s)
        return query(x*2+1,mid+1,r,s,e);
    if (mid+1>e)
        return query(x*2,l,mid,s,e);
    Node a=query(x*2,l,mid,s,e);
    Node b=query(x*2+1,mid+1,r,s,e);
    Node c;
    merge(c,a,b);
    return c;
}
```

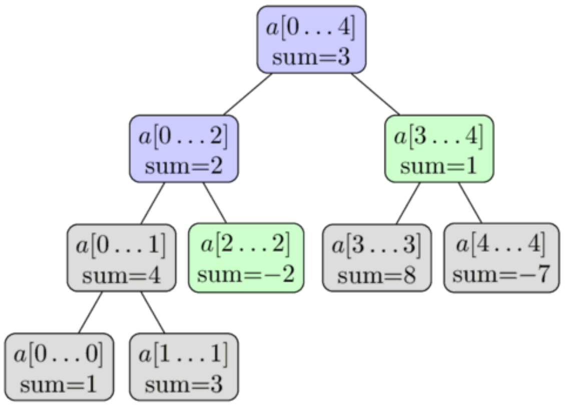

Figure 2: Example of a range sum query. From [2]

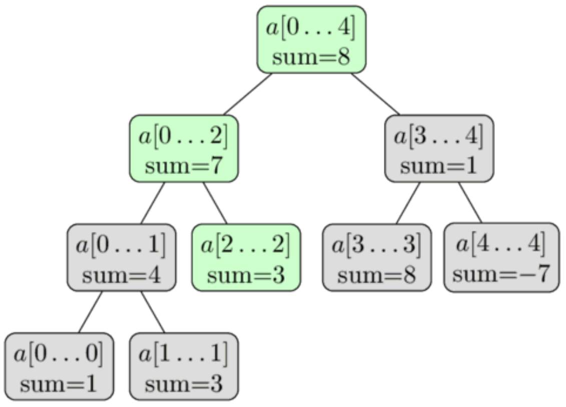

Figure 3: Illustration of a single point update on a sum operation. From [2]

2.5 Single point update

When we query over a large number of items, we can answer the query efficiently by pre-calculating the answers of selective sub-ranges, as seen in the previous subsection. However, if we want to update the value of a single element in the array, some of the pre-calculated nodes has to be updated as well. For example, in figure 3, support we want to modify the third element (a[2]), the nodes to be updated is the node responsible for this single element, along with all of its ancestors up to the root. Because these nodes and only these nodes cover the third item in its range, hence only these needs to be updated. The order of updating start from the leaf node, where the value is updated as appropriate, depending on the target problem. Then going upward, the merge function is call on each of the ancestor to recalculate the value of the node, based on the new updated data.

Since the number of nodes updated is one in each level. The number of nodes updates is $\lceil log_2 N \rceil$ nodes. The complexity of the update operation is O(log N)* cost of merge function. Which is constant in most cases.

A typical implementation of single point update is seen below.

```
void add(int x,int l,int r,int idx,int v){
    lazy_update(x, l, r);
    if (l == r ){
        node[x].lzy=v;
        lazy_update(x, l, r);
        return;
    }
    const int mid=(r+l)/2;
    if (mid<idx){
        update(x*2+1,mid+1,r,s,e);
    }
    else{
        update(x*2,l,mid,s,e);
    }
    merge(node[x],node[x*2],node[x*2+1]);
}
```

2.6 Range update and lazy propagation

We saw in the last section that if we want to update the value of a single element, we update all the nodes in the tree that cover this element and hence affected by it. However, if the update command is perform on a range of items not just a single element, we would need to modify far mode nodes, in the worst case all node if we want to modify the entire array. However, since we don't need to know the most-up-to date value of a node until we reach to it in the query method. We can perform some sort of lazy execution of the update in some areas or nodes in the tree, and only update them for real when we need them. To give an example, let's take the existing tree in figure 2 and suppose we want to add the value 5 to each element in the array from index 2 to index 4. In this scenario, we see that wee need to modify a total of 6 nodes, namely a[2..2], a[0..2], a[3..3], a[4..4], a[3..4], and finally a[0..4]. However, we know that for each node where ALL of its elements are covered with this update (a[3..4] for example) that we are adding the value 5 to each of its elements. We can calculate the new value of this node directly without the need to go down and update all of the nodes in its sub tree. In this example, since a[3..4] is totally covered within the update range (2..4), we know that every item that this node covers will be added 5 to its value. Hence, the new value of this item is the old value + 5 * the number of items that this node covers (in this case 2), hence the new value of this node is 1 + 2*5 = 11.

However, after this change, the value of the children are not changes. So if for a given query, we ended up in one of the leaf nodes a[3..3] or a[4..4]. Their value is still not updated and hence it is incorrect. Therefore, such cached or unprocessed operations has to be update in what is known as "lazy propagation" or "lazy update" since we only pushes the updates downstream when we need to. However, we also do the push in a lazy fashion such that each node cached updates are only push to its direct children and their values are updated accordingly, but not its grand-children and so on.

One might think that if we can calculate the new value for fully covered nodes, we can also calculate the new value for partially covered; by checking how many items were covered and updating accordingly. This is a valid approach except that in this case, we will need to store the range of items that were covered in the update, since it is not the entire range, so when more than one update is recorded in the same node, such updates have to stored separately and can't be combined (since each covers different partial range). an then each range has to be propagated to each child down the stream, which makes the lazy update process non-constant and the overall cost of operation is more than logarithmic.

Following all of this, the query update behaves in the same logic as the query function. Except that when we reach a full covered section, we update the value of this node only, and we store in a separate variable (the lzy variable in our notation) the information about the update.

```
void update(int x, int l, int r, int s, int e, int v) {
  lazy_update(x, l, r);
  if (s <= l && e >= r) {
    node[x].lzy = v;
    lazy_update(x, l, r);
    return;
  }
  const int mid = (r + l) / 2;
  if (mid < s) {
    update(x * 2 + 1, mid + 1, r, s, e, v);
    lazy_update(x * 2, l, mid);
  } else if (mid + 1 > e) {
    update(x * 2, l, mid, s, e, v);
    lazy_update(x * 2 + 1, mid + 1, r);
  } else {
    update(x * 2, l, mid, s, e, v);
    update(x * 2 + 1, mid + 1, r, s, e, v);
  }
  merge(node[x], node[x * 2], node[x * 2 + 1]);
}
```

In the lazy propagation function, first you need to update the value of the current node, based on the lazy value. For example, if lzy value of a node x is 7, then that means that all elements in the range are updated with this value, then depending on the operation, the value of the node gets updated. For example if the operation is sum, then the value of the node is increased by the length of the range times the value. Afterward, the lazy value is pushed down to the the direct children, and the lazy value of the current node is reset. An implementation of the lazy update or propagation function over the the range sum operation is shown below.

```
void lazy_update(int x, int l, int r) {
  if (node[x].lzy == 0)
    return;
  node[x].val += 1ll * (r - l + 1) * node[x].lzy;
  if (l != r) {
    node[x * 2].lzy += node[x].lzy;
    node[x * 2 + 1].lzy += node[x].lzy;
  }
  node[x].lzy = 0;
}
```

2.7 High-dimensional segment trees

In previous sections, we saw the segment trees applied over an array of numbers. However, the data structure is flexible and can be applied on higher dimensions. For example, we can build a 2D segment tree over a matrix, and allow efficient operations over a sub-rectangle of the matrix. However, with more dimensions, more space and more complexity for calculation are required.

3 DYNAMIC SEGMENT TREES

3.1 Offline vs online queries

In competition scenarios, both the arrays and the queries are given as an input, and the user is asked to return the answer to all queries as an output. In this scenario, queries can be processed all at once, in an offline fashion, rather than having to solve them one by one. However, in real-life systems, the system is required to answer queries as they came in an online fashion.

3.2 Dynamic segment trees

In some problems, instead of operating on an initial array of number, we start with an empty set of numbers, and add, remove, update, and query on such set of numbers. If we were to solve this problem using segment tree data structure, we would need to have a the tree built on hypothetical array of range MIN...MAX, where MIN/MAX is the minimum/maximum value the elements can take. However, if the range of values is large, having a full segment tree on top of it is unfeasible. One way to resolve that is by reading all queries at once, and then apply a numbers compression technique, i.e. ordering all mentioned numbers/indices in queries, and then replace the queries with queries on an artificial array of length DISITINCT with elements corresponds to the original values of the numbers, where DISTINCT is the number of different values mentioned, which makes the size of the array more manageable. However, applying such tricks can not be accomplished in an online settings, since we have to parse queries one by one and answer promptly.

To solve this problem, we use a dynamic version of segment trees, which is based on the assumption that nodes are only created when needed. At first, only the root exists, which is responsible for the range of 0...MAX, but has no left or right child. Then for every element added, we move in tree in the same way that we would move in a single-point update, and if in the direction we are going it has no node created yet, we create it and go further. Once we reach the leaf node, we modify the value accordingly, then going up, we apply the merge function going upward with handling the case that one of the children is a null. For query functions, it starts similar to the original query, but if we reached a empty node, we return the value from the other child only, or 0 (or whatever default value for the problem). Regarding data representation, one way we can implement this is throughout adding left and right pointers for each node to connect it to its children. However, it can be also implemented in compact mode, by having a dynamic list of nodes (vector in C++) and allocating nodes as necessary. This would still require having left and right variables, but will be indices instead of pointers for non-pointer implementation.

While the implementation of the dynamic version is almost the same, we highlight the main different functions, where the rest is implemented similarly. Below is an implementation for single number insertion and range query.

```
struct Node{
    int val;
    int l;
    int r;
};
int node_counter=2; // 1 for root
....
void add(int x, int l, int r, int idx, int v) {
  if (l == r) {
    node[x].val = v;
    return;
  }
```

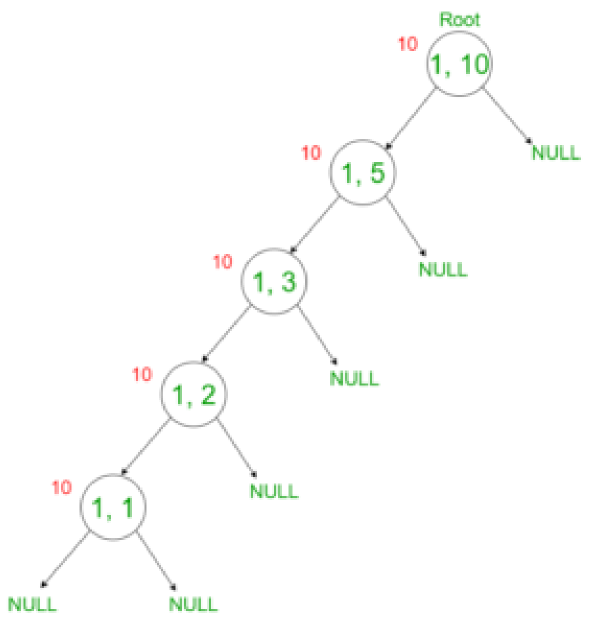

Figure 4: An example of dynamic segment tree after adding an element at position one. From [5]

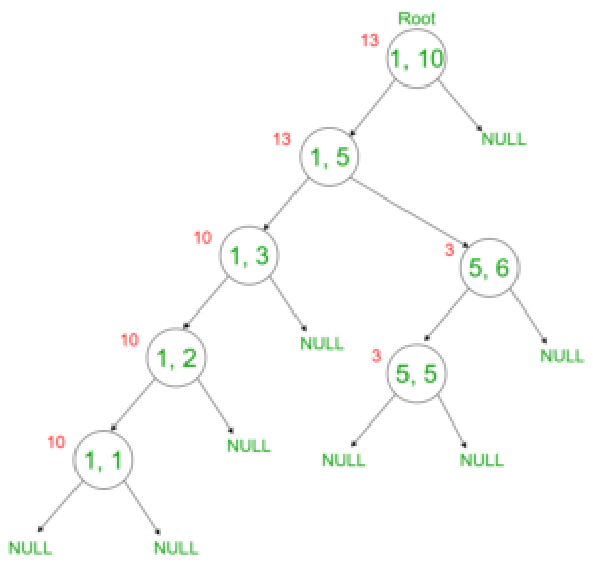

Figure 5: An example of dynamic segment tree after adding another element at position 5. From [5]

```
  const int mid = (r + l) / 2;
  if (mid < idx) {
    if (node[x].r == 0) {
      node[x].r = node_counter++;
    }
    add(node[x].r, mid + 1, r, idx, v);
  } else {
    if (node[x].l == 0) {
      node[x].l = node_counter++;
    }
    add(node[x].l, l, mid, idx, v);
  }
  if (node[x].l == 0) {
    node[x].val = node[node[x].r].val;
  } else if (node[x].r == 0) {
    node[x].val = node[node[x].l].val;
  } else {
   merge(node[x], node[node[x].l], node[node[x].r]);
  }
}


Node query(int x, int l, int r, int s, int e) {
  if (x == 0) {//empty node
    return Node();
  }
  if (s <= l && e >= r)
    return node[x];
  const int mid = (r + l) / 2;
  if (mid < s)
    return query(node[x].r, mid + 1, r, s, e);
  if (mid + 1 > e)
    return query(node[x].l, l, mid, s, e);
  Node a = query(node[x].l, l, mid, s, e);
  Node b = query(node[x].r, mid + 1, r, s, e);
  Node c;
  merge(c, a, b);
  return c;
}
```

4 PERSISTENT SEGMENT TREES

Persistence data structures are a type of data structures where history can be accessed, i.e. different states of the data structures before/after certain operations can be accessed. A famous usage of persistent data structure is the git version control system, where the status of the versioned code at any commit can be accessed. Persistent data structures can be either fully persistent or partially persistent. A partially persistent data structure is a data structure where all versions can be read/queried, and only the latest version can be modified. On the other hand, in fully persistent data structures, all versions can be read/queried and modified.

Persistent segment trees are fully persistent variant of segment tree, where we keep track of the state of the tree after each update. The core idea is that, as single-point update in the original version, we update the values of the nodes responsible for that item from the leaf up towards the root. However, in the persistent version, instead of modifying directly, we make a new root for that updated version, then we make a copy of each node we want to update, and update on the new node accordingly. For example in figure 6, we see an example of an update of the sixth item (represented by node 13). At first, we make a copy of the root item, then as we go down, since the target node is to the right, we make another copy of the right child, and we link the left pointer of the root tree to the original left child. As we go down, we keep track of the original node as well, so we can link or create another copy of each child accordingly, at node 3' and 3, we see we have to go to the left, hence we link the right child of 3' to 7, and we create a new node 6'. Finally we create another copy of 13, namely 13', and make a link from 6' to 12. Once we reach the leaf, we update the value accordingly, then going up in the newly created path of nodes, we apply the merge function based on the new left and right children. The implementation also requires keeping track of which are the right and left children, which can be done in a similar fashion to dynamic segment trees with pointers, or with indices in the compact version. A typical implementation of the persistent segment tree build and update function can be found below, the query function is similar to that of dynamic segment tree.

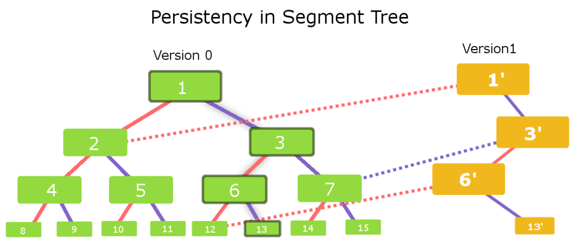

Figure 6: An example of update operation in persistent segment tree. From [3]

```
void build(int x,int l,int r){
    if (l==r){
        return;
    }
    node[x].l=cnt++,node[x].r=cnt++;
    build(node[x].l,l,mid);
    build(node[x].r,mid+1,r);
}
void update(int cur,int x,int l,int r,int t){
    if (l==r){
      node[x].val=1;//or other value, depending on the problem
        return;
    }
    if (mid>=t){
        node[x].r=node[cur].r,node[x].l=cnt++;
        update(node[cur].l,node[x].l,l,mid,t);
    }
    else{
        node[x].l=node[cur].l,node[x].r=cnt++ ;
        update(node[cur].r,node[x].r,mid+1,r,t);
    }
    merge(node[x],node[node[x].l],node[node[x].r]);
}
```

5 EXPERIMENTS SETUP

For experimentation, we will use the implementation of the simple, dynamic and persistent trees on three tasks, one for each. The sources of the problems are Sphere Online Judge (SPOJ) [6] and CodeChef [1]. Both are popular platform for competitive programming.

5.1 Task 1: Range Queries, Range Updates (SPOJ: HORRIBLE)

The horrible queries problem on SPOJ is a classical data structures problem where there are both range updates and range queries. The task prompt is this: Given an array of N integers $1 <= N <= 100,000$. perform a set of C commands $1 <= C <= 100,000$, where each command is either of the following types:

- **0 p q v**: you have to add v to all elements a_i of the array, where $p <= i <= q$.
- **1 p q**: calculate the sum of a_i of the array, where $p <= i <= q$.

5.2 Task 2: CODECHEF=LCSTSUB

In the second task, we have 2 arrays A and B each of them consists of N numbers, for every K such that 1 <= k <= N, you have to find the length of the longest possible subsequence in the first k elements of array A such that for each two consecutive elements i then j it should hold that |a[i]-a[j]| <= b[i]. Moreover, the input is encrypted, where the new input has to be decoded based on the answer to the previous query, by applying an xor operation on the answer with each of the numbers in the query. For limits, N is up to 100,000. A[i],b[i] is up to 1000,000,000.

5.3 Task 3: SPOJ=MKTHNUM

In the third task we have a sequence of N integers A indexed from 1 to N on which we have to answer several queries online. the answer of query(l, r, k) is the value of k-th number after sorting numbers in the sub-array from l to r in non-decreasing order. The number of operations is up to 10^5 and the time limit for this task is 1 second.

6 RESULTS

6.1 Task 1 solution

To solve this task, simple segment tree with lazy propagation can be used. node[x] will contain the sum of values of its range so our merge function will be a sum function. Lazy propagation is used to perform range updates. The methods for query and update are the ones used in explaining in the previous section. Finally, the last thing to remember, is to use 8-bytes integers (long long in c++) instead of 4-bytes integers as the numbers can get out of the 4-bytes integer rate.

6.2 Task 2 solution

Since the input is encrypted we are forced to solve it online, in other words answering for K before knowing the k+1-th element. Let ans[i] = the length of the longest subsequence in the first i elements that satisfies the pre-described conditions. So $ans[i] = Max\{1 + ans[j] : 1 <= j < i|a[i]\check{}a[j]| < b[i]\}$ The straightforward implementation for this would be to iterate for all possible values of (i, j) which runs in a time complexity of $O(N^2)$ which is too slow for N up to 10^5. So we need to use a segment tree get rid of the looping over j. If we define an array **Longest_Subsequence_Ending_at** such that **Longest_Subsequence_Ending_at**[x] = the length of longest subsequence that ends with an element of a value x. We can build a segment tree on top of it, with max operation. Since the numbers are up to a billion, we can't store such array so we need to use a dynamic segment tree instead.

6.3 Task 3 solution

There are solutions that work in $O((N + M) * LogN^3)$ or $O((N + M) * LogN^2)$ that use merge sort trees, but here we will describe the $O((N + M) * LogN)$ solution using Persistent Segment Trees.

Let's consider that we have a segment tree for all $1 <= i, j <= N$ ranges we have a segment tree on the frequency array of the sub-array defined by that range. A frequency array is defined as frequency[x] = number of times x occurred in the array.

Having these segment trees we can answer the query by doing a binary search on the answer. For a value X we want to calculate the number of elements <= X. To get this number we use the segment tree for the range (i,j) querying the sum of the range (1,X). After getting this value we continue the binary search by decreasing X if it is more than K and increasing it otherwise. Using this method, we can answer each query in a time complexity of $O(LogN * LogN)$. To optimize it more, we do a binary search inside the segment tree as follows: starting with the segment tree at the root, we check the sum of elements in the left subtree(which means the least numbers) if it s greater than k, then the answer lies in the left sub tree, otherwise it lies in the lies in the right sub-tree. We continue doing this until we reach a leaf which its index marks our answer. This will run each query in a time complexity of O(LogN)

We see that we can get the answer of the query in O(Log N). But building those segment trees will take $O(N^3)$ memory and $O(N^3 * LogN)$ time. However, there are two observations that reduce our time and memory: First, in segment tree for range (i, j) each node tree(i,j)[node] = tree(1,j)[node] – tree(1,i-1)[node], in other words, can be calculated from respective nodes in segment trees for ranges (1, i-1) and (1, j). Therefore, it is sufficient to build only N segment trees one for each prefix (1,i). With this observation we reduced our time and memory complexity to $O(N^2 * LogN)$ The second: the segment for range (1,i) is the same as segment tree for range (1,i-1) except for $O(logN)$ nodes that will change along the path from the root to the leaf where we update the i-the element. So we can create the segment for range (1,i) by adding only $O(logN)$ to the segment tree for range (1,i-1), i.e. by using persistent segment trees. With this observation combined with the previous one, we reduce our time and memory complexity to $O((N + M) * LogN)$ which is sufficient to solve the problem.

7 CONCLUSION

In summary, we presented the segment tree data structure, a flexible binary tree data structure than can support a wide range of operations. We started with the general structure of the segment tree and the data representation of it. Then we explained how the basic functions like query, build, and merge works. For updating elements, we discussed both the single-point update and the range-update with lazy propagation. Furthermore, we explored two different variation of the segment tree, namely the dynamic segment tree, which is used in problems where we operate on a set of items with operations like insert/delete/update/query, as well as the persistent segment tree, which enables keeping track of different states of the tree after each update. We evaluated the algorithms by solving 3 problems covering the simple segment tree, the dynamic segment tree, and the persistent segment tree cases.

REFERENCES

[1] 2022. Code chef. https://codechef.com/. (2022).

[2] e-maxx eng. 2022. Segment tree. https://cp-algorithms.com/data_structures/segment_tree.html. (2022).

[3] GeeksforGeeks. 2022. Persistent segment tree | set 1 (introduction). https://www.geeksforgeeks.org/persistent-segment-tree-set-1-introduction/. (2022).

[4] GeeksforGeeks. 2021. Segment tree | set 1 (sum of given range). https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range. (2021).

[5] Akash Sharma. 2021. Dynamic segment trees : online queries for range sum with point updates. https://www.geeksforgeeks.org/dynamic-segment-trees-online-queries-for-range-sum-with-point-updates/. (2021).

[6] 2022. Sphere online judge. https://spoj.com/. (2022).