Segment Tree | Set 1 (Sum of given range)

Let us consider the following problem to understand Segment Trees.

We have an array arr[0 . . . n-1]. We should be able to
1 Find the sum of elements from index l to r where 0 <= l <= r <= n-1

2 Change value of a specified element of the array to a new value x. We need to do arr[i] = x where 0 <= i <= n-1.

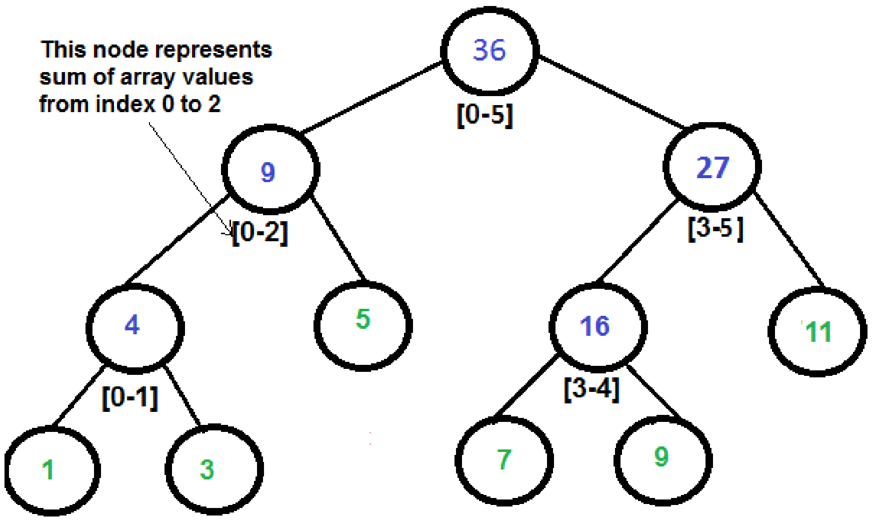
A **simple solution** is to run a loop from l to r and calculate the sum of elements in the given range. To update a value, simply do arr[i] = x. The first operation takes O(n) time and the second operation takes O(1) time.

Another solution is to create another array and store sum from start to i at the ith index in this array. The sum of a given range can now be calculated in O(1) time, but update operation takes O(n) time now. This works well if the number of query operations is large and very few updates.

What if the number of query and updates are equal? **Can we perform both the operations in O(log n) time once given the array?** We can use a Segment Tree to do both operations in O(Logn) time.

Representation of Segment trees
1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is sum of leaves under a node.

An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index 2*i+1, right child at 2*i+2 and the parent is at $\lfloor (i-1)/2 \rfloor$.



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

How does above segment tree look in memory?
Like Heap, the segment tree is also represented as an array. The difference here is, it is not a complete binary tree. It is rather a full binary tree (every node has 0 or 2 children) and all levels are filled except possibly the last level. Unlike Heap, the last level may have gaps between nodes. Below are the values in the segment tree array for the above diagram.

Below is memory representation of segment tree for input array {1, 3, 5, 7, 9, 11}
st[] = {36, 9, 27, 4, 5, 16, 11, 1, 3, DUMMY, DUMMY, 7, 9, DUMMY, DUMMY}

The dummy values are never accessed and have no use. This is some wastage of space due to simple array representation. We may optimize this wastage using some clever implementations, but code for sum and update becomes more complex.

Construction of Segment Tree from given array
We start with a segment arr[0 . . . n-1]. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the sum in the corresponding node.
All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree because we always divide segments in two halves at every level. Since the constructed tree is always a full binary tree with n leaves, there will be n-1 internal nodes. So the total number of nodes will be 2*n – 1. Note that this does not include dummy nodes.

What is the total size of the array representing segment tree?
If n is a power of 2, then there are no dummy nodes. So the size of the segment tree is 2n-1 (n leaf nodes and n-1) internal nodes. If n is not a power of 2, then the size of the tree will be 2*x – 1 where x is the smallest power of 2 greater than n. For example, when n = 10, then size of array representing segment tree is 2*16-1 = 31.
An alternate explanation for size is based on heignt. Height of the segment tree will be

$\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for Sum of given range
Once the tree is constructed, how to get the sum using the constructed segment tree. The following is the algorithm to get the sum of elements.

```
int getSum(node, l, r)
{
   if the range of the node is within l and r
        return value in the node
   else if the range of the node is completely outside l and r
        return 0
   else
    return getSum(node's left child, l, r) +
           getSum(node's right child, l, r)
}
```

Update a value
Like tree construction and query operations, the update can also be done recursively. We are given an index which needs to be updated. Let *diff* be the value to be added. We start from the root of the segment tree and add *diff* to all nodes which have given index in their range. If a node doesn't have a given index in its range, we don't make any changes to that node.

Time Complexity:
Time Complexity for tree construction is O(n). There are total 2n-1 nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is O(Logn). To query a sum, we process at most four nodes at every level and number of levels is O(Logn).

The time complexity of update is also O(Logn). To update a leaf value, we process one node at every level and number of levels is O(Logn).

Segment Tree | Set 2 (Range Minimum Query)

We have an array arr[0 . . . n-1]. We should be able to efficiently find the minimum value from index *qs* (query start) to *qe* (query end) where *0 <= qs <= qe <= n-1.*

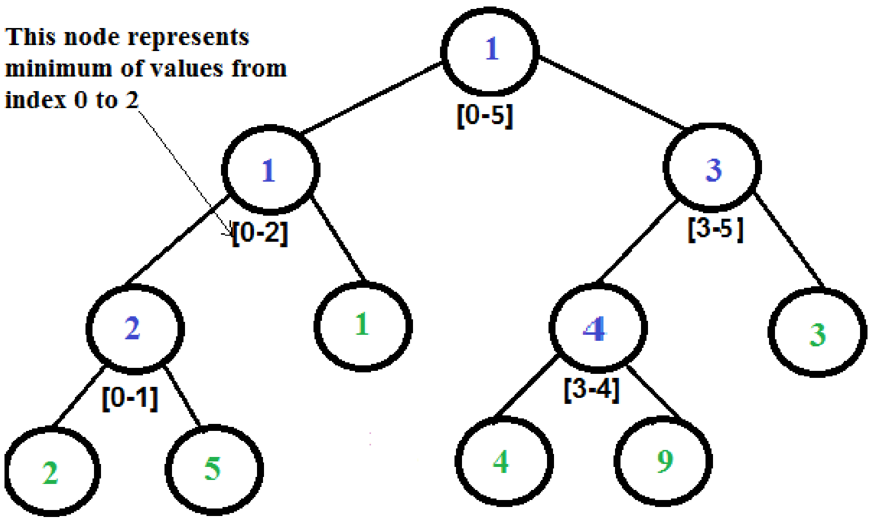
A **simple solution** is to run a loop from *qs* to *qe* and find minimum element in given range. This solution takes O(n) time in worst case.

Another solution is to create a 2D array where an entry [i, j] stores the minimum value in range arr[i..j]. Minimum of a given range can now be calculated in O(1) time, but preprocessing takes O(n^2) time. Also, this approach needs O(n^2) extra space which may become huge for large input arrays.

Segment tree can be used to do preprocessing and query in moderate time. With segment tree, preprocessing time is O(n) and time to for range minimum query is O(Logn). The extra space required is O(n) to store the segment tree.

Representation of Segment trees
1. Leaf Nodes are the elements of the input array.
2. Each internal node represents minimum of all leaves under it.

An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index 2*i+1, right child at 2*i+2 and the parent is at $\lfloor (i-1)/2 \rfloor$.



Segment Tree for input array {2, 5, 1, 4, 9, 3 }

Construction of Segment Tree from given array
We start with a segment arr[0 . . . n-1]. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the minimum value in a segment tree node. All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be n-1 internal nodes. So total number of nodes will be 2*n – 1.
Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for minimum value of given range
Once the tree is constructed, how to do range minimum query using the constructed segment tree. Following is algorithm to get the minimum.

```
// qs --> query start index, qe --> query end index
int RMQ(node, qs, qe)
{
   if range of node is within qs and qe
        return value in node
   else if range of node is completely outside qs and qe
        return INFINITE
   else
    return min( RMQ(node's left child, qs, qe), RMQ(node's right child, qs,
qe) )
}
```

Time Complexity:
Time Complexity for tree construction is O(n). There are total 2n-1 nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is O(Logn). To query a range minimum, we process at most two nodes at every level and number of levels is O(Logn).

--

Segment Tree | Set 3 (XOR of given range)

We have an array arr[0 . . . n-1]. There are two type of queries

1. Find the XOR of elements from index l to r where 0 <= l <= r <= n-1
2. Change value of a specified element of the array to a new value x. We need to do arr[i] = x where 0 <= i <= n-1.

There will be total of q queries.

Input Constraint

```
 n <= 10^5, q <= 10^5
```

Solution 1
A simple solution is to run a loop from l to r and calculate xor of elements in given range. To update a value, simply do arr[i] = x. The first operation takes O(n) time and second operation takes O(1) time. Worst case time complexity is O(n*q) for q queries
which will take huge time for n ~ 10^5 and q ~ 10^5. Hence this solution will exceed time limit.

Solution 2
Another solution is to store xor in all possible ranges but there are O(n^2) possible ranges hence with n ~ 10^5 it wil exceed space complexity, hence without considering time complexity, we can state this solution will not work.

Solution 3 (Segment Tree)

We build a segment tree of given array such that array elements are at leaves and internal nodes store XOR of leaves covered under them.

Time and Space Complexity:
Time Complexity for tree construction is O(n). There are total 2n-1 nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is O(log n).
The time complexity of update is also O(log n).

Total time Complexity is : O(n) for construction + O(log n) for each query = O(n) + O(n * log n) = O(n * log n)

`Time Complexity` `O(n * log n)`
`Auxiliary Space` `O(n)`

Iterative Segment Tree (Range Minimum Query)

The iterative version of the segment tree basically uses the fact, that for an index i, left child = 2 * i and right child = 2 * i + 1 in the tree. The parent for an index i in the segment tree array can be found by parent = i / 2. Thus we can easily travel up and down through the levels of the tree one by one. At first we compute the minimum in the ranges while constructing the tree starting from the leaf nodes and climbing up through the levels one by one. We use the same concept while processing the queries for finding the minimum in a range. Since there are (log n) levels in the worst case, so querying takes log n time. For update of a particular index to a given value we start updating the segment tree starting from the leaf nodes and update all those nodes which are affected by the updation of the current node by gradually moving up through the levels at every iteration. Updation also takes log n time because there we have to update all the levels starting from the leaf node where we update the exact value at the exact index given by the user.

Time Complexity :(n log n)
Auxiliary Space : (n)

Segment tree | Efficient implementation

(*using bitwise operators)*

This is the most optimized approach out of all discussed.

Complete implementation of segment tree including the query and update functions in such a less number of lines of code than the previous recursive one. Let us now understand about how each of the function is working:

1. The picture makes it clear that the leaf nodes are stored at i+n, so we can clearly insert all leaf nodes directly.
2. The next step is to build the tree and it takes O(n) time. The parent has always it's index less then its children so we just process all the nodes in decreasing order calculating the value of parent node. If the code inside the build function to calculate parents seems confusing then you can see this code, it is equivalent to that inside the build function.
3. `tree[i]=tree[2*i]+tree[2*i+1]`
4. Updating a value at any position is also simple and the time taken will be proportional to the height of the tree. We only update values in the parents of the given node which is being changed. so for getting the parent , we just go up to the parent node , which is p/2 or p>>1, for node p. p^1 turns (2*i) to (2*i + 1) and vice versa to get the second child of p.
5. Computing the sum also works in O(log(n)) time .if we work through an interval of [3,11), we need to calculate only for nodes 19,26,12 and 5 in that order.

The idea behind the query function is that whether we should include an element in the sum or we should include its parent. Let's look at the image once again for proper understanding. Consider that L is the left border of an interval and R is the right border of the interval [L,R). It is clear from the image that if L is odd then it means that it is the right child of it's parent and our interval includes only L and not it's parent. So we will simply include this node to sum and move to the parent of it's next node by doing L = (L+1)/2. Now, if L is even then it is the left child of it's parent and interval includes it's parent also unless the right borders interferes. Similar conditions is applied to the right border also for faster computation. We will stop this iteration once the left and right borders meet.

The theoretical time complexities of both previous implementation and this implementation is same but practically this is found to be much more efficient as there are no recursive calls. We simply iterate over the elements that we need. Also this is very easy to implement.

Time Complexities:

- Tree Construction : O(n)
- Query in Range : O(Log n)
- Updating an element : O(Log n).

Persistent Segment Tree | Set 1 (Introduction)

Segment Tree is itself a great data structure that comes into play in many cases. In this post we will introduce the concept of Persistency in this data structure. Persistency, simply means to retain the changes. But obviously, retaining the changes cause extra memory consumption and hence affect the Time Complexity.

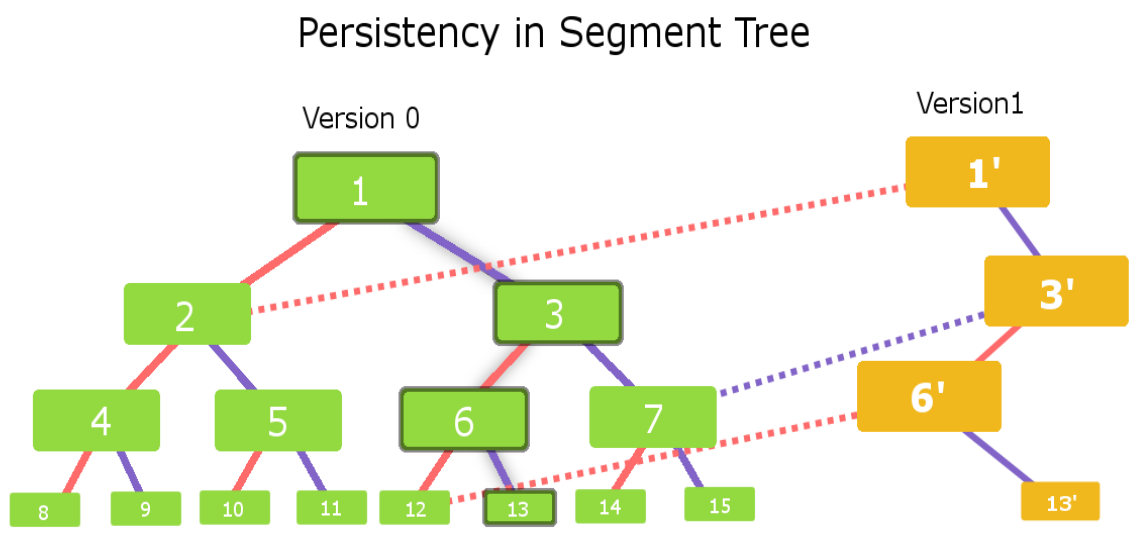
Our aim is to apply persistency in segment tree and also to ensure that it does not take more than **O(log n) time and space** for each change.

Let’s think in terms of versions i.e. for each change in our segment tree we create a new version of it.
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 creating the whole tree for every version will take O(n log n) extra space and O(n log n) time. So, this idea runs out of time and memory for large number of versions.

Let’s exploit the fact that for each new update(say point update for simplicity) in segment tree, 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 . Lets call this segment tree as **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′**(yellow color) . 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(red dashed line in figure). Let's 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′(solid purple edge connection).

In the similar fashion we will now examine for **node 3′**. The left child is affected , So we create a new node called **node 6′** and connect it with solid red edge 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(see the purple dash edge.)

Same procedure is done for node 6′ and we see that the left child of node 6′ will be the left child of node 6 in version-0(red dashed connection) and right child is newly created node called **node 13′**(solid purple dashed edge).

Each **yellow color node** is a newly created node and dashed edges are the inter-connection between the different versions of the segment tree.

Now, the Question arises : **How to keep track of all the versions?**
– We only need to keep track the first root node for all the versions and this will serve the purpose to track all the newly created nodes in the different versions. For this purpose we can maintain an array of pointers to the first node of segment trees for all versions.

Let's consider a very basic problem to see how to implement persistence in segment tree

```
Problem : Given an array A[] and different point update
operations.Considering
each point operation to create a new version of the array. We need to
answer
the queries of type
Q v l r : output the sum of elements in range l to r just after the v-th
update.
```

We will create all the versions of the segment tree and keep track of their root node.Then for each range sum query we will pass the required version's root node in our query function and output the required sum.

Note : The above problem can also be solved by processing the queries offline by sorting it with respect to the version and answering the queries just after the corresponding update.

Time Complexity : The time complexity will be the same as the query and point update operation in the segment tree as we can consider the extra node creation step to be done in O(1). Hence, the overall Time Complexity per query for new version creation and range sum query will be **O(log n)**.

Applications

Segment Tree(ST) is data structure for range query processing. It has a divide and conquer approach.

We preprocess the the list/array, so that the queries are optimised. There are two types of preprocessing : on-line and off-line. We can visualise ST as a tree like we do in merge sort and like in the merge sort algorithm the nodes in the ST also store an aggregate value for a range in the given array; the aggregate value in the node in merge-sort is the sorted values in the given range [L, …R]. Using clever approach we can also sort an array using ST.

In segment we can use a custom/user-defined aggregator, and not just simple aggregator like MAX, SUM, MIN, PRODUCT etc, for pre-processing of the nodes. The aggregator should run on disjoint ranges. After pre-processing the range the corresponding nodes store the answer for that particular range for the given array. You can define you own mathematically function to preprocess the nodes; this part is the most interesting one!

There is also a similar data structure which is also used in range queries - **The Fenwick Tree(FT) or Binary Index Tree(BIT).** When Segment Tree is an overkill Binary Index Tree comes for the rescue. It is not as versatile as the ST but it is quicker to process and optimised in certain problems.

1. FT takes less space than ST. Two times less.
2. FT does not support any user defined merge function in general like ST does. You can find range sum form both data structure in LogN time, but if you want to find MAX in a range, then ST is one should should be looking at, as you cannot only find MAX in [0..i] using FT. You mathematical function has the formula to aggregate on contiguous range, but **BIT** stores does not stores aggregated value for every range like the ST does, that is why it is more space optimised.
3. FT is faster than ST. Per query time(LogN) is same for both of them but ST has to traverse LogN depth for query query and can also have a complicated merge function, so it takes a little more time than FT. In BIT, we have aggregated value for the range [0, …16] and [0, …8] and we need answer for [9, ..16], then there should be a mathematical function/algorithm to calculate it; MAX function have this problem.
4. ST is more powerful and versatile than FT. So far, I have seen all the problem that can be solved using BIT can also be solved using ST.