PHENIKAA UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

PHENIKAA

UNIVERSITY

Project Report

Topic: "Segment Tree Data Structure in Competitive Programming"

Instructor: **Ph.D. Vũ Đức Minh**

Student: **Lê Hữu Trung (K13)**

Hanoi, December 2021

Table of Contents

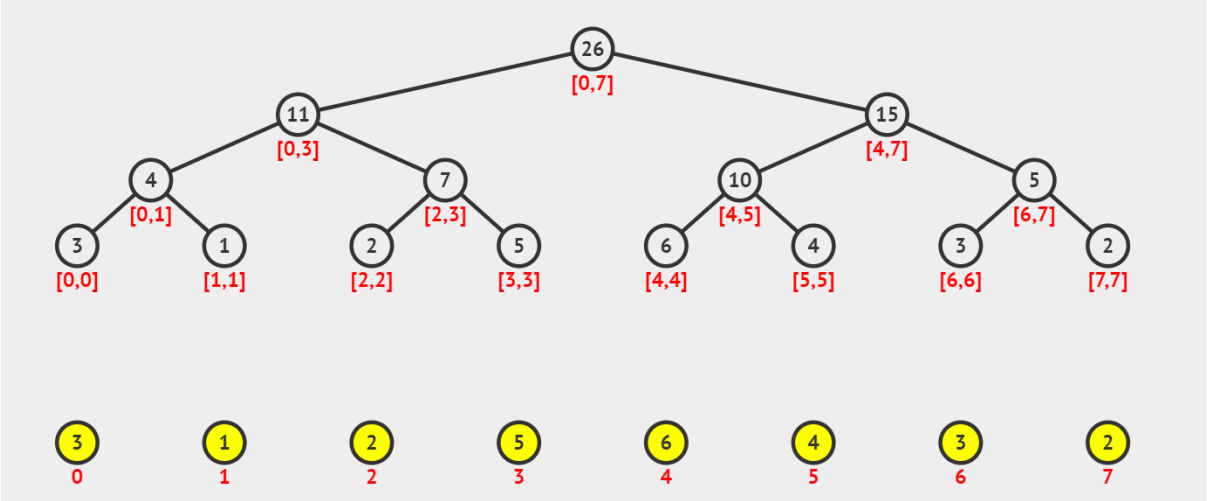
I. What is a Segment Tree?

A Segment Tree is a data structure that allows answering **range queries** over an array effectively, while still being flexible enough to allow modifying the array. This includes finding the sum of consecutive array elements ***arr[l...r]***, or finding the minimum element in such a range in ***O(logn)*** time.

Between answering such queries the Segment Tree allows modifying the array by replacing one element, or even changing the elements of a whole subsegment (e.g. assigning all elements ***arr[l...r]*** to any value, or adding a value to all elements in the subsegment).

In general, a Segment Tree is a very flexible data structure, and a huge number of problems can be solved with it. Additionally, it is also possible to apply more complex operations and answer more complex queries.

One important property of Segment Trees is that they require only a linear amount of memory. The standard Segment Tree requires maximum ***4n*** vertices for working on an array of size ***n***.



A basic Segment Tree for Sum.

II. Structure of a Segment Tree.

Let's consider the following problem to understand the structure of a Segment Trees.

Suppose we have an array of n elements, we should be able to:

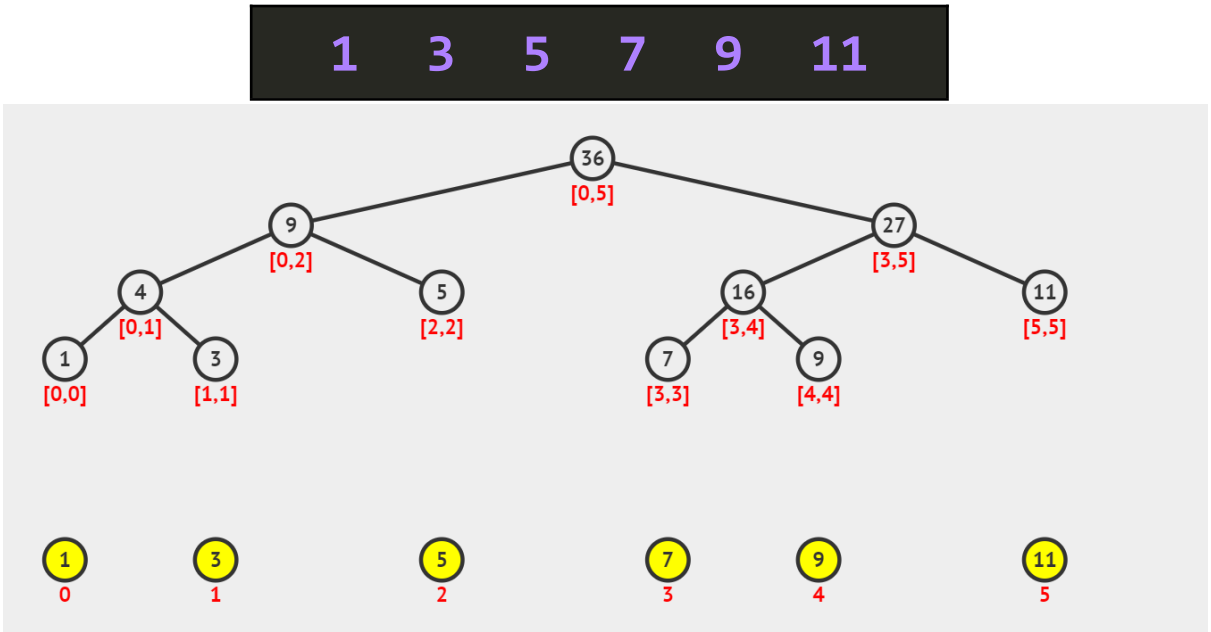
1. ***set(i, v):*** set the element with index i to v.
2. ***query(l, r)*:** find some information on the segment from l to $r - 1$.

1. Representation of Segment trees

- Leaf Nodes are the elements of the input array.
- Each internal node represents some merging of the leaf nodes. The merging may be different for different problems.

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

A Segment Tree for the following array:



Segment Tree for the sum of inputted array {1, 3, 5, 7, 9, 11}

Memory representation of Segment Tree for input array {1, 3, 5, 7, 9, 11}:

```
ST[] = {36, 9, 27, 4, 5, 16, 11, 1, 3, Dump, Dump, 7, 9,
Dump, Dump}
```

The dummy values are never accessed and have no use.

2. Construction of Segment Tree from the 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 merged information 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 into 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 ***2n – 1***. Note that this does not include dummy nodes.

3. 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 ***2x – 1*** where ***x*** is the smallest power of ***2*** greater than ***n***. For example, when ***n = 10***, then the size of the array representing the Segment Tree is ***2.16 - 1 = 31***.

An alternate explanation for size is based on height. The height of the Segment Tree will be $\lceil log_2 n \rceil$. Since the tree is represented using an array and relation between parent and child indexes must be maintained, the size of memory allocated for Segment Tree will be $2.\ 2^{\lceil log2n \rceil} - 1$.

III. Pseudo code.

1. Construct Segment Tree.

```
void construct(array, n, start, end, current){
    if the length of the array segment is 1:
        st[current] = array[start]
        return

    // Divide the array in two halves and call same procedure on
both halves
    construct(array, n, start, (start + end)/2, current*2 + 1)
    construct(array, n, (start + end)/2 + 1, end, current*2 + 2)

    // Store the information in the corresponding node
    st[current] = merge(st[current*2 + 1], st[current*2 + 2])
}
```

2. Query a given range.

Once the tree is constructed, how to query using the constructed segment tree. The following is the algorithm to query elements.

```
Node Query(currentNode, l, r) {
    Node answer = Null;
    if the range of the current node is within l and r:
        answer = currentNode
        return answer
    else if the range of the current node is completely outside l
and r:
        return answer
    else:
        Node left = Query(currentNode's left child, l, r)
        Node right = Query(currentNode's right child, l, r)
        answer = merge(left, right)
        return answer
```

```
}
```

3. Update a value.

```
void Update(position, value, start, end, current){
    if the position of the current node is completely outside
start and end:
        return

    if the length of the segment is 1:
        st[current] = value
        return

    // Divide the tree into two halves and call the same procedure
for the in-range half
    if position <= mid:
        Update(position, value, start, mid, current * 2 + 1)
    else:
        Update(position, value, mid + 1, end, current * 2 + 2)

    // Update information for internal node
    ST[current] = merge(ST[current * 2 + 1], ST[current * 2 + 2])
}
```

IV. Implementation.

The problems in this section may have other solutions, but to demonstrate the power of Segment Tree I won't mention it here.

1. Sum on a segment.

Suppose we have an array of n elements, and we want to be able to do two operations with it:

- ***set(i, v):*** set the element with index i to v.
- ***sum(l, r):*** find the sum on the segment from l to $r - 1$.

For this problem, the internal node should store the sum of its left and right child.

```
struct Node{
    int value;

    void merge(Node left, Node right){
        value = left.value + right.value;
    }
};
```

Therefore, the root node may store the sum of the segment it manages.

Reference code: *Sum on a segment*

2. Minimum value on a segment.

Suppose we have an array of n elements, and we want to be able to do two operations with it:

- ***set(i, v):*** set the element with index i to v.
- ***min(l, r):*** find the minimum value on the segment from l to $r - 1$.

For this problem, the internal node should store the minimum value of its left and right child.

```
struct Node{
    int value;

    void merge(Node left, Node right){
        value = min(left.value, right.value);
    }
};
```

Therefore, the root node may store the minimum value of the segment it manages.

Reference code: *Minimum value on a segment*

3. Count the number of minimum values on a segment.

Suppose we have an array of n elements, and we want to be able to do two operations with it:

- ***set(i, v):*** set the element with index i to v.
- ***countMin(l, r):*** find the minimum value on the segment from l to $r - 1$.

For this problem, the internal node should store two pieces of information:

- The minimum value of its left and right child.
- The number of elements equal to the minimum on this segment.

```
struct Node {
    // Each node store the value of the element
    // and the number of elements equal to the minimum on this
segment.
    int value, count;

    // A method for setting the internal node.
    void merge(Node left, Node right) {
        // If the value of the left and right child is equal, sum
up the count attribute.
        if (left.value == right.value) {
```

```
            value = left.value;
            count = left.count + right.count;
        }
        // Else get the child with smaller value's count
        else {
            Node tmp = min(left, right);
            value = tmp.value;
            count = tmp.count;
        }
    }
};
```

Therefore, the root node may store the minimum value and the number of elements equal to the minimum on the segment it manages.

Reference code: *Number of minimum on a segment*

4. Segment with the maximum sum.

Now we consider the problem of finding a segment with a maximum sum. Our data structure must support two operations on the array:

- ***set(i, v):*** set the element with index i to v.
- ***max_segment():*** find the segment of the array with the maximum sum.

For this problem, the internal node should store 4 pieces of information as below:

- Max prefix sum on a segment.
- Max suffix sum on a segment.
- Total sum on a segment.
- Max subarray sum on a segment.

```
struct Node {
    // Each node contain 4 pieces of information:
    long long maxPrefixSum;
    long long maxSuffixSum;
```

```
    long long totalSum;
    long long maxSubArraySum;

    // A method for setting the internal node.
    void merge(Node left, Node right) {
        maxPrefixSum = max(left.maxPrefixSum, left.totalSum +
right.maxPrefixSum);
        maxSuffixSum = max(right.maxSuffixSum, right.totalSum +
left.maxSuffixSum);
        totalSum = left.totalSum + right.totalSum;
        maxSubArraySum = max(max(left.maxSubArraySum,
right.maxSubArraySum), left.maxSuffixSum + right.maxPrefixSum);
    }
};
```

Therefore, the root node may store the maximum subarray sum of the segment it manages.

Reference code: *Segment with the maximum sum*

5. Kth one.

Consider the problem of finding the Kth one. Our data structure must support two operations on the array:

- ***set(i, v):*** set element i to $v \in \{0, 1\}$.
- ***find(k):*** find the index of the Kth one.

For this problem, a basic Segment Tree for the sum is enough. In other words, the internal node will store the sum of its left and right child.

```
struct Node{
    int value;

    void merge(Node left, Node right){
        value = left.value + right.value;
    }
```

```
};
```

Therefore, the root node may store the sum of the segment it manages. But to get the Kth one, we have to modify the Query method a little bit.

Suppose we need to find the Kth one on the segment $[l, r)$. If $r = l$, then we found the desired one. Otherwise, we look at the sum s on the left subsegment. If $k < s$, then the Kth one is in the left subtree, otherwise, we need to start the search for the one with index $k - s$ in the right subtree.

Reference code: *Kth One*

6. First element at least X.

Consider the task of finding the first element greater than or equal to x. Our structure should support two operations on the array:

- ***set(i, v):*** set element i to v.
- ***first_above(x):*** find the first item greater than or equal to x.

For this problem, the idea is to construct a Segment Tree for the max. In other words, the internal node should store the maximum value of its left and right child.

```
struct Node{
    int value;

    void merge(Node left, Node right){
        value = min(left.value, right.value);
    }
};
```

Therefore, the root node may store the maximum value of the segment it manages.

To search for an element in a segment, we go down to the left subtree, if there is a maximum of at least x, otherwise, we go down to the right subtree.

Reference code: *First element at least X*

7. First element at least X from an index.

Let's complicate the previous task: we need to respond to requests ***first_above(x, l)***: find the first element to the right of ***l*** greater than or equal to ***x***.

To process this request, we will act recursively as follows. Suppose we want to find such an element on the segment.

- If the maximum on the left subsegment is greater than or equal to x and the segment intersects with [l,n), then we recursively start from the left subsegment.
- If we did not find an element in the left subsegment, then we start from the right subsegment.

Reference code: *First element at least X from an index*

8. Inversions.

A permutation p of n elements is given. You need to find for each element the number of inversions: the number of elements to the left, which are greater.

For this problem, construct an empty Segment Tree of size n with the operation sum.

Now go through the permutation elements from left to right. A one in a leaf of a tree of segments will mean that such an element was already visited. When moving to the i-th element of p[i], we will make a request to calculate the sum of [p[i],n] in the segment tree: it will just count the number of elements to the left which are greater than p[i]. And finally, put one in position p[i].

Reference code: *Inversions*

9. Inversions 2.

Array of inversions is given. You need to restore the permutation p. Let's build a segment tree of size n, which is filled with ones, and is able to respond to the request k-th one from the end.

Now, let's process the elements from right to left. A one in a leaf of a segment tree will mean that an element with this value is to the left.

When moving to the i-th element, we will look for the a[i]-th one from the right: this just means that there will a[i] greater elements to the left of the i-th position. Finally, put 0 in the found position, i.e. remove the element from consideration.

Reference code: *Inversions 2*

V. Other problems from competitive programming contests.

1. Get the greatest common divisor on a segment.

Problem Link: *https://oj.vnoi.info/problem/olp_kc19_seq*

Reference code: *ST for GCD*

2. Count the number of formable triangles on a segment.

Problem Link: *https://oj.vnoi.info/problem/olp_kc19_tri*

Reference code: *Count formable triangles*

VI. Conclusion.

In short, Segment Tree is a powerful data structure for competitive programming. Segment tree, hard as it is to be installed, can address a wide variety of problems once you have mastered it.

Although this report is not perfect, to some extent it has demonstrated most of the basic aspects of Segment Tree. In some way it will be helpful for competitive programming beginners.

VII. References.

All code: *https://github.com/huutrungle2001/DSA4CP*

1. Codeforces edu.

2. Visual Algo.

3. VNOI.

4. Geeksforgeeks.

5. CP-Algorithm.