Maximum Subarray Sum in a given Range

Given an array of n numbers, the task is to answer the following queries:

```
maximumSubarraySum(start, end) : Find the maximum
subarray sum in the range from array index 'start'
to 'end'.
```

Also see : Range Query With Update Required

Examples:

```
Input : arr[] = {1, 3, -4, 5, -2}
        Query 1: start = 0, end = 4
        Query 2: start = 0, end = 2
Output : 5
         4
Explanation:
For Query 1, [1, 3, -4, 5] or ( [5] )
represent the maximum sum sub arrays
with sum = 5.

For Query 2, [1, 3] represents the
maximum sum subarray in the query range
with sum = 4
```

Segment Trees can be used to solve this problem.Here, we need to keep information regarding various cumulative sums.At every Node we store the following:
1) Maximum Prefix Sum,

4) Maximum Subarray Sum

A classical Segment Tree with each Node storing the above information should be enough to aswer each query. The only focus here is on how the left and the right Nodes of the tree are merged together. Now, we will discuss how each of the information is constructed in each of the segment tree Nodes using the information of its left and right child.

Constructing the Maximum Prefix Sum using Left and Right child

There can be two cases for maximum prefix sum of a Node:

1. The maximum prefix sum occurs in the left child,

Case 1: Maximum Prefix Sum = 3

```
In this Case,
Maximum Prefix Sum = Maximum Prefix Sum of Left Child
```

2. The maximum prefix sum contains every array element of the left child and the elements contributing to the maximum prefix sum of the right child,

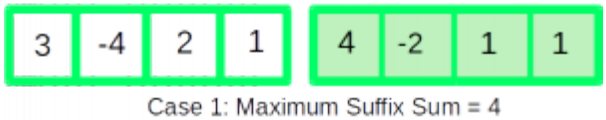
Case 2: Maximum Prefix Sum = 3 + (-4) + 2 + 1 + 4 = 6

```
In this Case,
Maximum Prefix Sum = Total Sum of Left Child +
                     Maximum Prefix Sum of Right Child
```

Constructing the Maximum Suffix Sum using Left and Right child

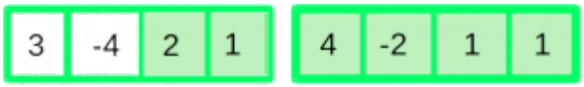
There can be two cases for maximum suffix sum of a Node:

1. The maximum suffix sum occurs in the right child,



Case 1: Maximum Suffix Sum = 4

2. The maximum suffix sum contains every array element of the Right child and the elements contributing to the maximum suffix sum of the left child,



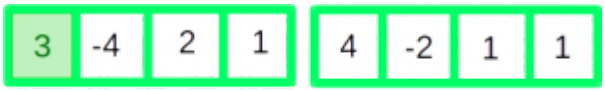
Case 2: Maximum Suffix Sum = 2 + 1 + 4 + (-2) + 1 + 1 = 7

```
In this Case,
Maximum Suffix Sum = Total Sum of Right Child +
                     Maximum Suffix Sum of Left Child
```

Constructing the Maximum Subarray Sum using Left and Right child

There can be three cases for the maximum sub-array sum of a Node:

1. The maximum sub-array sum occurs in the left child,



Case 2: Maximum Subarray Sum = 3

```
In this Case,
Maximum Sub-array Sum = Maximum Subarray Sum of Left Child
```

2. The maximum sub-array sum occurs in the right child,

Case 1: Maximum Subarray Sum = 4

```
In this Case,
Maximum Sub-array Sum = Maximum Subarray Sum of Right Child
```

3. The maximum subarray sum, contains array elements of the right child contributing to the maximum prefix sum of the right child, and the array elements of the Left child contributing to the maximum suffix sum of the left child,

Case 3: Maximum Subarray Sum = 2 + 1 + 4 = 7

Maximum Suffix Sum of Left Child

```
// C++ Program to Implement Maximum Sub-Array Sum in a range
#include <bits/stdc++.h>
using namespace std;

#define inf 0x3f3f

/* Node of the segment tree consisting of:
1. Maximum Prefix Sum,
2. Maximum Suffix Sum,
3. Total Sum,
4. Maximum Sub-Array Sum */
struct Node {
    int maxPrefixSum;
    int maxSuffixSum;
    int totalSum;
    int maxSubarraySum;

    Node()
    {
        maxPrefixSum = maxSuffixSum = maxSubarraySum = -inf;
        totalSum = -inf;
    }
};

// Returns Parent Node after merging its left and right child
Node merge(Node leftChild, Node rightChild)
{
    Node parentNode;
    parentNode.maxPrefixSum = max(leftChild.maxPrefixSum,
                                  leftChild.totalSum +
                                  rightChild.maxPrefixSum);

    parentNode.maxSuffixSum = max(rightChild.maxSuffixSum,
                                  rightChild.totalSum +
                                  leftChild.maxSuffixSum);

    parentNode.totalSum = leftChild.totalSum +
                          rightChild.totalSum;

    parentNode.maxSubarraySum = max({leftChild.maxSubarraySum,
                                     rightChild.maxSubarraySum,
                                     leftChild.maxSuffixSum +
                                     rightChild.maxPrefixSum});

    return parentNode;
```

```
                                        int end, int index)
{

    /* Leaf Node */
    if (start == end) {

        // single element is covered under this range
        tree[index].totalSum = arr[start];
        tree[index].maxSuffixSum = arr[start];
        tree[index].maxPrefixSum = arr[start];
        tree[index].maxSubarraySum = arr[start];
        return;
    }

    // Recursively Build left and right children
    int mid = (start + end) / 2;
    constructTreeUtil(tree, arr, start, mid, 2 * index);
    constructTreeUtil(tree, arr, mid + 1, end, 2 * index + 1);

    // Merge left and right child into the Parent Node
    tree[index] = merge(tree[2 * index], tree[2 * index + 1]);
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructTreeUtil() to fill the allocated
   memory */
Node* constructTree(int arr[], int n)
{
    // Allocate memory for segment tree
    int x = (int)(ceil(log2(n))); // Height of the tree

    // Maximum size of segment tree
    int max_size = 2 * (int)pow(2, x) - 1;
    Node* tree = new Node[max_size];

    // Fill the allocated memory tree
    constructTreeUtil(tree, arr, 0, n - 1, 1);

    // Return the constructed segment tree
    return tree;
}

/* A Recursive function to get the desired
   Maximum Sum Sub-Array,
The following are parameters of the function-

tree     --> Pointer to segment tree
index --> Index of the segment tree Node
ss & se  --> Starting and ending indexes of the
```

```
                                    int qe, int index)
{
    // No overlap
    if (ss > qe || se < qs) {

        // returns a Node for out of bounds condition
        Node nullNode;
        return nullNode;
    }

    // Complete overlap
    if (ss >= qs && se <= qe) {
        return tree[index];
    }

    // Partial Overlap Merge results of Left
    // and Right subtrees
    int mid = (ss + se) / 2;
    Node left = queryUtil(tree, ss, mid, qs, qe,
                                        2 * index);
    Node right = queryUtil(tree, mid + 1, se, qs,
                               qe, 2 * index + 1);

    // merge left and right subtree query results
    Node res = merge(left, right);
    return res;
}

/* Returns the Maximum Subarray Sum between start and end
   It mainly uses queryUtil(). */
int query(Node* tree, int start, int end, int n)
{
    Node res = queryUtil(tree, 0, n - 1, start, end, 1);
    return res.maxSubarraySum;
}

int main()
{
    int arr[] = { 1, 3, -4, 5, -2 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Construct Segment Tree
    Node* Tree = constructTree(arr, n);
    int start, end, maxSubarraySum;

    // Answering query 1:
    start = 0;
    end = 4;
    maxSubarraySum = query(Tree, start, end, n);
    cout << "Maximum Sub-Array Sum between "
```

```
    start = 0;
    end = 2;
    maxSubarraySum = query(Tree, start, end, n);
    cout << "Maximum Sub-Array Sum between "
         << start << " and " << end
         << " = " << maxSubarraySum << "\n";

    return 0;
}
```

Output:

```
Maximum Sub-Array Sum between 0 and 4 = 5
Maximum Sub-Array Sum between 0 and 2 = 4
```

Time Complexity: O(logn) for each query.

Recommended Posts:

Maximum length of subarray such that sum of the subarray is even

XOR of a subarray (range of elements)

Range query for Largest Sum Contiguous Subarray

Range queries to count 1s in a subarray after flip operations

Size of The Subarray With Maximum Sum

Maximum Product Subarray | Set 3

Longest subarray having maximum sum

Maximum circular subarray sum

Maximum subarray sum in O(n) using prefix sum

Maximum sum bitonic subarray

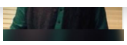
Maximum Product Subarray

Maximum sum subarray having sum less than or equal to given sum

Maximum Subarray Sum after inverting at most two elements

Maximum Product Subarray | Set 2 (Using Two Traversals)

Find the maximum subarray XOR in a given array



Check out this Author's contributed articles.

If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please Improve this article if you find anything incorrect by clicking on the "Improve Article" button below.

Improved By : atulim

Article Tags : Advanced Data Structure, Arrays, Segment-Tree

Practice Tags : Arrays, Segment-Tree

2

To-do Done

4

Based on **9** vote(s)

Feedback/ Suggest Improvement | Add Notes | Improve Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

A computer science portal for geeks

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org

COMPANY
About Us
Careers
Privacy Policy
Contact Us

LEARN
Algorithms
Data Structures
Languages
CS Subjects
Video Tutorials

PRACTICE
Courses
Company-wise
Topic-wise
How to begin?

CONTRIBUTE
Write an Article
Write Interview Experience
Internships
Videos