Everything Under The Sun

A blog on CS concepts

A simple approach to segment trees

November 9, 2014 *by* *Kartik Kukreja*

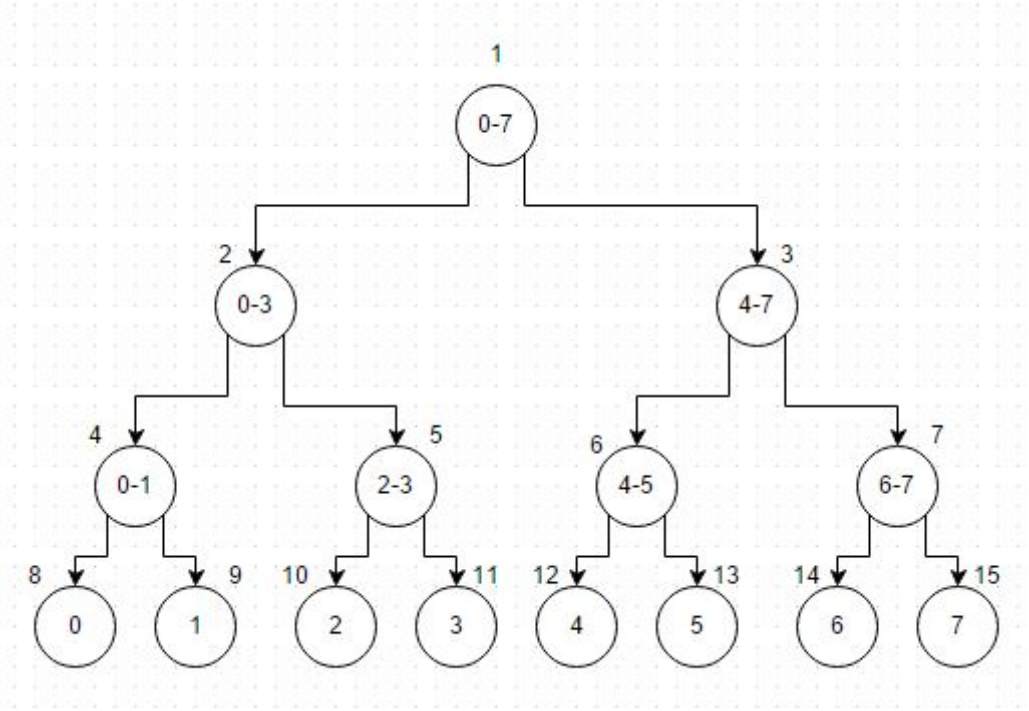
A segment tree is a tree data structure that allows aggregation queries and updates over array intervals in logarithmic time. As I see it, there are three major use cases for segment trees:

1. **Static segment trees:** This is probably the most common use case. We preprocess an array of N elements to construct a segment tree in O(N). Now, we can query aggregates over any arbitrary range/segment of the array in O(log N).
2. **Segment tree with point updates:** This allows us to update array values, one at a time in O(log N), while still maintaining the segment tree structure. Queries over any arbitrary range still occurs in O(log N).
3. **Segment tree with range updates:** This allows us to update a range of array elements at once in O(N) in the worst case, however problem specific optimizations and lazy propagation typically give huge improvements. Queries over any arbitrary range still occurs in O(log N).

In this post, I'll cover the first two use cases because they go together. Given a static segment tree, it is very easy to add point update capability to it. I'll leave the third use case as the subject matter of a future blog post. I intend this post to be a practical introduction to segment trees, rather than a theoretical description, so it will focus on how we can divide a segment tree into its components, the working of each component and how we can separate the problem specific logic from the underlying data structure. We'll build a template for a segment tree and then apply it to several problems to understand how problem specific logic can be cleanly separated from the template.

Structure of a segment tree

Let's understand what a segment tree looks like. Each node in a segment tree stores aggregate statistics for some range/segment of an array. The leaf nodes stores aggregate statistics for individual array elements. Although a segment tree is a tree, it is stored in an array similar to a heap. If the input array had 2^n elements (i.e., the number of elements were a power of 2), then the segment tree over it would look something like this:



(https://kartikkukreja.files.wordpress.com/2014/11/structure-of-a-segment-tree.jpg)
Each node here shows the segment of the input array for which it is responsible. The number outside a node indicates its index in the segment tree array. Clearly, if the array size N were a power of 2, then the segment tree would have 2*N-1 nodes. It is simpler to store the first node at index 1 in the segment tree array in order to simplify the process of finding indices of left and right children (a node at index i has left and right children at 2*i and 2*i+1 respectively). Thus, for an input array of size N, an array of size 2*N would be required to store the segment tree.

In practice, however, N is not usually a power of 2, so we have to find the power of 2 immediately greater than N, let's call it x, and allocate an array of size 2*x to store the segment tree. The following procedure calculates the size of array required to store a segment tree for an input array size N:

```
1  int getSegmentTreeSize(int N) {
2    int size = 1;
3    for (; size < N; size <<= 1);
4    return size << 1;
5  }
```

view raw Size of a segment tree.cpp hosted with ❤ by **GitHub**

We'll try to separate the implementation of the underlying data structure from the problem specific logic. For this purpose, let us define a structure for a segment tree node:

```
1  struct SegmentTreeNode {
2    // variables to store aggregate statistics and
3    // any other information required to merge these
4    // aggregate statistics to form parent nodes
5
6    void assignLeaf(T value) {
7      // T is the type of input array element
```

```
8     // Given the value of an input array element,
9     // build aggregate statistics for this leaf node
10   }
11
12   void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
13     // merge the aggregate statistics of left and right
14     // children to form the aggregate statistics of
15     // their parent node
16   }
17
18   V getValue() {
19     // V is the type of the required aggregate statistic
20     // return the value of required aggregate statistic
21     // associated with this node
22   }
23 };
```

view raw Segment tree node.cpp hosted with ❤ by **GitHub**

Building a segment tree

We can build a segment tree recursively in a depth first manner, starting at the root node (representative of the whole input array), working our way towards the leaves (representatives of individual input array elements). Once both children of a node have returned, we can merge their aggregate statistics to form their parent node.

```
1  void buildTree(T arr[], int stIndex, int lo, int hi) {
2    if (lo == hi) {
3      nodes[stIndex].assignLeaf(arr[lo]);
4      return;
5    }
6
7    int left = 2 * stIndex, right = left + 1, mid = (lo + hi) / 2;
8    buildTree(arr, left, lo, mid);
9    buildTree(arr, right, mid + 1, hi);
10   nodes[stIndex].merge(nodes[left], nodes[right]);
11 }
```

view raw Building a segment tree.cpp hosted with ❤ by **GitHub**

Here I've assumed that the type of input array elements is T. stIndex represents the index of current segment tree node in the segment tree array, lo and hi indicate the range/segment of input array this node is responsible for. We build the whole segment tree with a single call to buildTree(arr, 1, 0, N-1), where N is the size of input array arr. Clearly, the time complexity of this procedure is O(N), assuming that assignLeaf() and merge() operations work in O(1).

Querying the segment tree

Suppose we want to query the aggregate statistic associated with the segment [lo,hi], we can do this recursively as follows:

```
1  // V is the type of the required aggregate statistic
2  V getValue(int lo, int hi) {
3    SegmentTreeNode result = getValue(1, 0, N-1, lo, hi);
4    return result.getValue();
5  }
6
7  // nodes[stIndex] is responsible for the segment [left, right]
8  // and we want to query for the segment [lo, hi]
9  SegmentTreeNode getValue(int stIndex, int left, int right, int lo, int hi) {
10   if (left == lo && right == hi)
11     return nodes[stIndex];
12
13   int mid = (left + right) / 2;
14   if (lo > mid)
15     return getValue(2*stIndex+1, mid+1, right, lo, hi);
16   if (hi <= mid)
17     return getValue(2*stIndex, left, mid, lo, hi);
18
19   SegmentTreeNode leftResult = getValue(2*stIndex, left, mid, lo, mid);
20   SegmentTreeNode rightResult = getValue(2*stIndex+1, mid+1, right, mid+1, hi);
21   SegmentTreeNode result;
22   result.merge(leftResult, rightResult);
23   return result;
24 }
```

view raw Querying the segment tree.cpp hosted with ❤ by **GitHub**

This procedure is similar to the one used for building the segment tree, except that we cut off recursion when we reach a desired segment. The complexity of this procedure is O(log N).

Updating the segment tree

The above two procedures, building the segment tree and querying it, are sufficient for the first use case: a static segment tree. It so happens that the second use case: point updates, doesn't require many changes. In fact, we don't have to change the problem specific logic at all. No changes in the structure SegmentTreeNode are required.
We just need to add in a procedure for updating the segment tree. It is very similar to the buildTree() procedure, the only difference being that it follows only one path down the tree (the one that leads to the leaf node being updated) and comes back up, recursively updating parent nodes along this same path.

```
1  // We want to update the value associated with index in the input array
2  void update(int index, T value) {
3    update(1, 0, N-1, index, value);
4  }
5
6  // nodes[stIndex] is responsible for segment [lo, hi]
7  void update(int stIndex, int lo, int hi, int index, T value) {
```

```
8   if (lo == hi) {
9     nodes[stIndex].assignLeaf(value);
10    return;
11  }
12
13  int left = 2 * stIndex, right = left + 1, mid = (lo + hi) / 2;
14  if (index <= mid)
15    update(left, lo, mid, index, value);
16  else
17    update(right, mid+1, hi, index, value);
18
19  nodes[stIndex].merge(nodes[left], nodes[right]);
20  }
```

view raw Updating the segment tree.cpp hosted with ❤ by **GitHub**

Clearly, the complexity of this operation is O(log N), assuming that assignLeaf() and merge() work in O(1).

Segment Tree template

Let's put all this together to complete the template for a segment tree.

```
1   // T is the type of input array elements
2   // V is the type of required aggregate statistic
3   template<class T, class V>
4   class SegmentTree {
5     SegmentTreeNode* nodes;
6     int N;
7
8   public:
9     SegmentTree(T arr[], int N) {
10      this->N = N;
11      nodes = new SegmentTreeNode[getSegmentTreeSize(N)];
12      buildTree(arr, 1, 0, N-1);
13    }
14
15    ~SegmentTree() {
16      delete[] nodes;
17    }
18
19    V getValue(int lo, int hi) {
20      SegmentTreeNode result = getValue(1, 0, N-1, lo, hi);
21      return result.getValue();
22    }
23
```

```
24    void update(int index, T value) {
25      update(1, 0, N-1, index, value);
26    }
27
28    private:
29    void buildTree(T arr[], int stIndex, int lo, int hi) {
30      if (lo == hi) {
31        nodes[stIndex].assignLeaf(arr[lo]);
32        return;
33      }
34
35      int left = 2 * stIndex, right = left + 1, mid = (lo + hi) / 2;
36      buildTree(arr, left, lo, mid);
37      buildTree(arr, right, mid + 1, hi);
38      nodes[stIndex].merge(nodes[left], nodes[right]);
39    }
40
41    SegmentTreeNode getValue(int stIndex, int left, int right, int lo, int hi) {
42      if (left == lo && right == hi)
43        return nodes[stIndex];
44
45      int mid = (left + right) / 2;
46      if (lo > mid)
47        return getValue(2*stIndex+1, mid+1, right, lo, hi);
48      if (hi <= mid)
49        return getValue(2*stIndex, left, mid, lo, hi);
50
51      SegmentTreeNode leftResult = getValue(2*stIndex, left, mid, lo, mid);
52      SegmentTreeNode rightResult = getValue(2*stIndex+1, mid+1, right, mid+1, hi);
53      SegmentTreeNode result;
54      result.merge(leftResult, rightResult);
55      return result;
56    }
57
58    int getSegmentTreeSize(int N) {
59      int size = 1;
60      for (; size < N; size <<= 1);
61      return size << 1;
62    }
63
64    void update(int stIndex, int lo, int hi, int index, T value) {
65      if (lo == hi) {
```

```
66      nodes[stIndex].assignLeaf(value);
67      return;
68      }
69
70      int left = 2 * stIndex, right = left + 1, mid = (lo + hi) / 2;
71      if (index <= mid)
72        update(left, lo, mid, index, value);
73      else
74        update(right, mid+1, hi, index, value);
75
76      nodes[stIndex].merge(nodes[left], nodes[right]);
77    }
78  };
```

view raw Segment tree template.cpp hosted with ❤ by **GitHub**

We shall now see how this template can be used to solve different problems, without requiring a change in the tree implementation, and how the structure SegmentTreeNode is implemented differently for different problems.

The **first problem** we'll look at it is GSS1 (http://www.spoj.com/problems/GSS1/). This problem asks for a solution to maximum subarray problem (http://en.wikipedia.org/wiki/Maximum_subarray_problem) for each range of an array. My objective here is not to explain how to solve this problem, rather to demonstrate how easily it can be implemented with the above template at hand.
As it turns out, we need to store 4 values in each segment tree node to be able to merge child nodes to form a solution to their parent's node:

1. Maximum sum of a subarray, starting at the leftmost index of this range
2. Maximum sum of a subarray, ending at the rightmost index of this range
3. Maximum sum of any subarray in this range
4. Sum of all elements in this range

The SegmentTreeNode for this problem looks as follows:

```
1   struct SegmentTreeNode {
2     int prefixMaxSum, suffixMaxSum, maxSum, sum;
3
4     void assignLeaf(int value) {
5       prefixMaxSum = suffixMaxSum = maxSum = sum = value;
6     }
7
8     void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
9       sum = left.sum + right.sum;
10      prefixMaxSum = max(left.prefixMaxSum, left.sum + right.prefixMaxSum);
11      suffixMaxSum = max(right.suffixMaxSum, right.sum + left.suffixMaxSum);
12      maxSum = max(prefixMaxSum, max(suffixMaxSum, max(left.maxSum, max(right.maxSum, left.suffixM
13    }
```

```
14
15    int getValue() {
16      return maxSum;
17    }
18  };
```

view raw GSS1 segment tree node.cpp hosted with ❤ by **GitHub**

The complete solution for this problem can be viewed here (https://github.com/kartikkukreja/blog-codes/blob/master/src/spoj_GSS1.cpp).

The **second problem** we'll look at is GSS3 (http://www.spoj.com/problems/GSS3/), which is very similar to GSS1 with the only difference being that it also asks for updates to array elements, while still maintaining the structure for getting maximum subarray sum. Now, we can understand the advantage of separating problem specific logic from the segment tree implementation. This problem requires no changes to the template and even uses the same SegmentTreeNode as used for GSS1. The complete solution for this problem can be viewed here (https://github.com/kartikkukreja/blog-codes/blob/master/src/spoj_GSS3.cpp).

The **third problem**: BRCKTS (http://www.spoj.com/problems/BRCKTS/), we'll look at is very different from the first two but the differences are only superficial since we'll be able to solve it using the same structure. This problem gives a string containing parenthesis (open and closed), requires making updates to individual parenthesis (changing an open parenthesis to closed or vice versa), and checking if the whole string represents a correct parenthesization.
As it turns out, we need only 2 things in each segment tree node:

1. The number of unmatched open parenthesis in this range
2. The number of unmatched closed parenthesis in this range

The SegmentTreeNode for this problem looks as follows:

```
1   struct SegmentTreeNode {
2     int unmatchedOpenParans, unmatchedClosedParans;
3
4     void assignLeaf(char paranthesis) {
5       if (paranthesis == '(')
6         unmatchedOpenParans = 1, unmatchedClosedParans = 0;
7       else
8         unmatchedOpenParans = 0, unmatchedClosedParans = 1;
9     }
10
11    void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
12      int newMatches = min(left.unmatchedOpenParans, right.unmatchedClosedParans);
13      unmatchedOpenParans = right.unmatchedOpenParans + left.unmatchedOpenParans - newMatches;
14      unmatchedClosedParans = left.unmatchedClosedParans + right.unmatchedClosedParans - newMatches;
15    }
16
```

```
17    bool getValue() {
18      return unmatchedOpenParans == 0 && unmatchedClosedParans == 0;
19    }
20  };
```

view raw BRCKTS segment tree node.cpp hosted with ♥ by **GitHub**

The complete solution for this problem can be viewed here (https://github.com/kartikkukreja/blog-codes/blob/master/src/spoj_BRCKTS).

The **final problem** we'll look at in this post is KGSS (http://www.spoj.com/problems/KGSS/). This problem asks for the maximum pair sum in each subarray and also requires updates to individual array elements. As it turns out, we only need to store 2 things in each segment tree node:

1. The maximum value in this range
2. The second maximum value in this range

The SegmentTreeNode for this problem looks as follows:

```
1   struct SegmentTreeNode {
2     int maxNum, secondMaxNum;
3
4     void assignLeaf(int num) {
5       maxNum = num;
6       secondMaxNum = -1;
7     }
8
9     void merge(SegmentTreeNode& left, SegmentTreeNode& right) {
10      maxNum = max(left.maxNum, right.maxNum);
11      secondMaxNum = min(max(left.maxNum, right.secondMaxNum), max(right.maxNum, left.secondMaxN
12    }
13
14    int getValue() {
15      return maxNum + secondMaxNum;
16    }
17  };
```

view raw KGSS segment tree node.cpp hosted with ♥ by **GitHub**

The complete solution for this problem can be viewed here (https://github.com/kartikkukreja/blog-codes/blob/master/src/spoj_KGSS.cpp).

I hope this post presented a gentle introduction to segment trees and I look forward to feedback for possible improvements and suggestions for a future post on segment trees with lazy propagation.

Continue to Part 2 (https://kartikkukreja.wordpress.com/2015/01/10/a-simple-approach-to-segment-trees-part-2/)…

This entry was posted in Algorithms, Data Structures, Spoj and tagged aggregate statistics, BRCKTS Spoj, C++ implementation, GSS1 Spoj, GSS3 Spoj, KGSS Spoj, maximum subarray problem, query, segment tree, template, update. Bookmark the permalink.

68 thoughts on “A simple approach to segment trees”

1. Anonymous says:
   July 28, 2015 at 3:42 pm
   Thanks bro

   Reply
2. Ido Hadanny says:
   September 5, 2015 at 10:23 pm

Hey Kartik – great article.
Question – Why do you input “left, right” as parameters to the query and update methods?
Wouldn’t it be more encapsulated to put them inside SegmentTreeNode on buildTree?
I created a python version of your ST, and in buildTree I’m doing:

st[st_index] = {“s”: arr_start, “e”: arr_end, “v”: v}

So query and update have less parameters to move around (also less code duplication). What’s the disadvantages of this approach?

Reply

- kartik kukreja says:
  September 5, 2015 at 10:31 pm
  Yeah, that’s a good alternative. I was just trying to separate the problem specific code in SegmentTreeNode from the problem invariant data structure implementation in SegmentTree. There isn’t much code duplication. Even if we use a structure to represent the arguments, they still are different for each method call but I agree, it’d be more elegant this way. There are no disadvantages, it’s mostly a matter of choice.

  Reply

3. Ido Hadanny says:
   September 5, 2015 at 10:40 pm
   thanks! btw, all your help still didn’t help me to beat KGSS on SPOJ. Probably time limits are set with C++ in mind… But I did learn the principle 🙂

   Reply
4. chhavi gupta says:
   October 15, 2015 at 7:25 pm
   Very good tutorial especially for absolute beginners. Best i found anywhere. Just need a few more pointers on how to maintain node where pre-processing on array is required…for example:
   http://www.spoj.com/problems/DQUERY/en/

   Reply

   - kartik kukreja says:
     October 15, 2015 at 10:54 pm
     Since the problem-specific logic is separated out from the tree implementation, you can be as creative with how your input is represented as you want. In this problem, I believe we need to maintain the index of each element in sorted order. You can build the segment tree over the array of these indices. Basically you can do whatever preprocessing you want and build the segment tree over its output.

     Reply
5. Kushal Saharan says:
   November 13, 2015 at 10:16 am
   Great Article!!

   Reply
6. rabiul_awal says:
   December 20, 2015 at 11:34 am
   Great Article

   Reply

7. Ayushi says:
   December 31, 2015 at 4:02 pm
   Hey Kartik, thanks for the great article. One of the best articles explaining segTrees. I learnt the basic structure of segTree. Thanks a lot. 🙂
   For GSS1, I have a problem specific question. How do you decide on the values the nodes need to store to be able to merge children?
   (4 values in this case :
   1. Maximum sum of a subarray, starting at the leftmost index of this range
   2. Maximum sum of a subarray, ending at the rightmost index of this range
   3. Maximum sum of any subarray in this range
   4. Sum of all elements in this range.
)

   Reply
   - kartik kukreja says:
     December 31, 2015 at 5:21 pm
     That's the entire problem solving logic, isn't it? You have to understand the problem and decide what all you need to store in each node so that you can compute those values for a node by using only the values of its children nodes. For this problem, we want only (3) maximum sum of any subarray in this range, but to be able to compute this from children values, we need to introduce (1) prefixMaxSum and (2) suffixMaxSum. Then, to be able to compute (1) and (2), we also need to store (4) sum. So we can take a top-down approach from what we want to what we need to compute that.

     Reply
8. hemanth kumar tirupati says:
   February 8, 2016 at 8:02 am
   Hey I guess you have to use return size; in getSegmentTreeSize function

   Reply
   - kartik kukreja says:
     February 8, 2016 at 9:08 am
     No. Please read the text before the getSegmentTreeSize() function. In short, if we want to create segment tree over N elements and N is a power of 2, we need 2*N-1 nodes in the segment tree. If N is not a power of 2, we need 2*x-1 nodes, where x is the power of 2 immediately greater than N. To simplify implementation, we are ignoring the first node and using 1-based indexing so we need 2*x nodes. I hope that answers your concern.

     Reply
9. Neil DSouza says:
   March 10, 2016 at 7:47 pm
   Hi Kartik. Thank you for the excellent explanation on Segment trees.
   There is one thing I wanted to clarify in the application to GSS1 where we are computing the merge.
   Lets say we have 2 intervals [a,b] and [c,d]
   Could I say, that another way of writing maxSum would be
   max(a, b, c, d, [a-b], [b-c], [c-d], [a-c], [b-d] and [a-d])?
   So the earlier prefixMaxSum and suffixMaxSum and their combinations handle cases for certain combinations of the above set ranges?

   Thanks very much.

Reply

- kartik kukreja says:
  March 10, 2016 at 10:51 pm
  Please define what do you mean by max of a range [a-b]?

  Reply

10. Neil DSouza says:
    March 10, 2016 at 8:09 pm
    I tried leaving a comment, but it didnt appear here when I clicked “Post Comment”. So, sorry if this is a duplicate.

    Let’s say the Left node is interval [a-b] and the right node is interval [b-c]
    Can I say that prefixMax sum is equivalent to max of ([a], [a-b], [a-c])
    Can I say that postfixMax sum is equivalent to max of ([b-d], [c-d], [d])
    and maxSum is
    max of ([a], [b], [c], [d], [a-b], [b-c], [c-d], [a-c], [b-d], [a-d]) ?

    Reply

    - kartik kukreja says:
      March 10, 2016 at 10:51 pm
      What do you mean by max of a range [a-b]? Maximum element in that range? Then, no. PrefixMaxSum of a range [a-b] is the maximum sum of a prefix of that interval i.e., sum([a,x]) where a <= x = sum([a,y]) where a <= x < y <= b.

      Reply

11. Neil DSouza says:
    March 10, 2016 at 11:01 pm
    I meant it exactly the way you have meant it.
    For prefix sum – max of the ranges starting at a
    for suffix sum max of the ranges ending at d.

    Reply

    - kartik kukreja says:
      March 10, 2016 at 11:16 pm
      The notation does not indicate that. If you read max([a], [a-b], [a-c]), you wouldn’t know what it should mean? what does [a] mean btw?

      Reply

      - Neil DSouza says:
        March 10, 2016 at 11:36 pm
        I am getting where Im going wrong. Since since the range [a-a] is contained in the range [a-b] and in this particular case, I should have said PrefixMaxSum [a-b] there is no need to mention [a] again. What I was trying to do was relate PrefixMaxSum/SuffixMaxSum to the primitive intervals [a-b] and [c-d] that are going to make up the new range [a-d].

        Reply

12. Anonymous says:
    March 24, 2016 at 11:36 pm
    Great article Kartik. It helped me a lot. Thanks. Keep up the good work.

Reply

13. Allan Boll says:
March 26, 2016 at 1:10 am
Great explanation, thanks! Inspired by your post I wrote a version that stores all data in the segment node class rather than in a separate array, as I find it easier to comprehend this way. Maybe you'd be interested: http://allanrbo.blogspot.com/2016/03/segment-tree-implementation-in-java.html

Reply

14. Rahul Jha says:
April 6, 2016 at 11:04 pm
How do I find sum of all subsets of a range in an array?
Given an array A of N elements and Q queries of type [l, r]. Print the sum of each subset in the range { A[l], A[r] }.

For example: A[]= { 1, 2, 3, 4 } and [l,r]= [1, 3] then

print sum(1), sum (2), sum(3), sum(1,2), sum(1,3), sum(2,3), sum(1,2,3)

Reply

- Kartik Kukreja says:
April 7, 2016 at 5:32 am
I think I've got it but do verify. I've made an assumption which I think would be part of the problem specification: The array A[] has distinct elements.

Each segment tree node will need to store only 1 thing: sum of all subsets in that segment. The size of a segment can be calculated from its range and need not be stored.

For leaf nodes, the value is simply the element itself. For any internal node, we combine the values of the two children segments, having sizes x and y, and values a and b respectively, as follows:
a*2^y + b*2^x

I think that should do it.

Reply

15. Anonymous says:
April 6, 2016 at 11:13 pm
Thanks a lot! Nice Tutorial !!

Reply

16. rash007blog says:
April 10, 2016 at 1:23 pm
I think the sizeofsegmenttree function should return size, because when I am giving input 10 it returns 32 while it should return 32/2 and it is also true in general.

Reply

- Kartik Kukreja says:
April 12, 2016 at 12:46 am
No. It should return 32. Size of segment tree is not the next power of 2 but twice that. Please look at this comment: https://kartikkukreja.wordpress.com/2014/11/09/a-simple-approach-to-segment-trees/comment-page-2/#comment-6222

Reply

17. piyank sarawagi says:
May 5, 2016 at 7:14 pm
best article on segment tree 😀

Reply

- Kartik Kukreja says:
May 5, 2016 at 8:27 pm
Thank you

Reply

18. Xueyang Liu says:
July 19, 2016 at 9:39 pm
In the third problem BRCKTS. Can I represent '(' as -1, ')' as 1, then in the segment tree node I store a sum of the current range, then if the sum value of the root node is 0, it shows that the bracket is correct. Is it correct? I got WA. But I did not where is wrong in this idea.

Reply

- Kartik Kukreja says:
July 19, 2016 at 10:44 pm
Addition on integers is symmetric but concatenation on brackets is not. -1 + 1 or 1 + -1 are both 0, but () is balanced and)(is not.

Reply

19. Pingback: Problem of the day: Queue with min operation | Everything Under The Sun