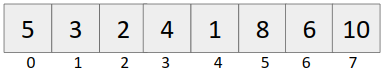
**Segment trees**

Need of segment trees

Let us take an example of returning and updating the sum of the subarray a[i….j] of an array of size n.

Example



Query: Output the sum from i=1 to i=5.

Update: Update the element at ith index. Example: put a[4] = 13.

Approach 1

For query: Iterate from i=1 to i=5 and calculate the sum.

Time complexity: O(n)

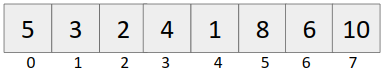
For update: Update the ith index, simply put a[i] = updated\_element

|  |  |
| --- | --- |
| Query | Update |
| O(n) | O(1) |

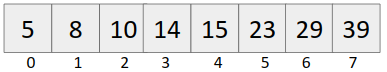
Approach 2 (Prefix Sum Approach)

Build the prefix sum array

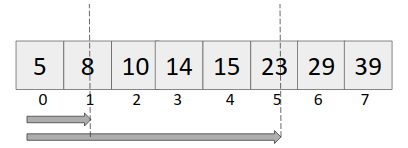
Given array:



Prefix sum array:



For query: Output sum from i to j (0 <= i <= j < n) (say i=1 to i=5)



Sum[i....j] = pref[j]-pref[i-1] {if i!=0}

pref[j] {if i = 0}

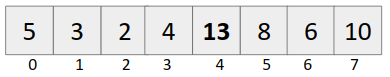
Time complexity: O(1)

For update: Put a[i] = updated\_value

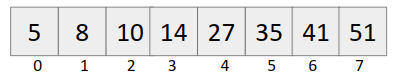
To update in the prefix array, we need to change all pref[i1] {i1 >= i}

Example: Update the 4th indexed element to 13.

Original Array becomes:



Prefix sum array becomes:



Time complexity: O(n)

Time complexity of this approach

|  |  |
| --- | --- |
| Query | Update |
| O(1) | O(n) |

If we want both the operations to be in reasonable time, we use segment trees.

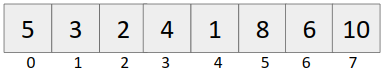
Time Complexity comparison table

|  |  |  |
| --- | --- | --- |
| Approach | Query | Update |
| Approach 1 | O(n) | O(1) |
| Approach 2 | O(1) | O(n) |
| Segment tree | O(log(n)) | O(log(n)) |

Requirement of log(n) time complexity: Many a times, number of queries and number of updates are of the order of 105-106, we will get tle if we use Approach 1 or Approach 2.

**Segment tree construction**

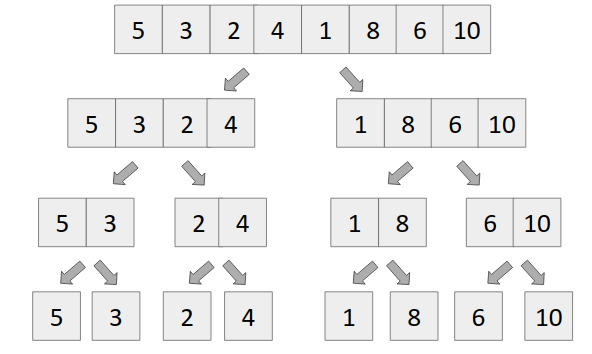
Given array:



Power of number 2 in programming

1. Binary Representation of numbers - All operations be it sum / subtraction/ product, all are accomplished in O(1).
2. Division of array (Divide and conquer)

We can divide the above array as



Number of nodes = n + n/2 + n/4 + . . . . + 2 + 1 , which is geometric progression

Let number of terms in the above G.P. be x, which denotes the height of the segment tree.

We know,

*arx-1 = n*

*putting a = 1, r = 2, we get*

*(2)x-1 = n*

*log2(2x-1) = log2n*

*x = 1 + log2n*

Number of nodes = *1 + 2 + 4 + . . . . + n/4 + n/2 + n.*

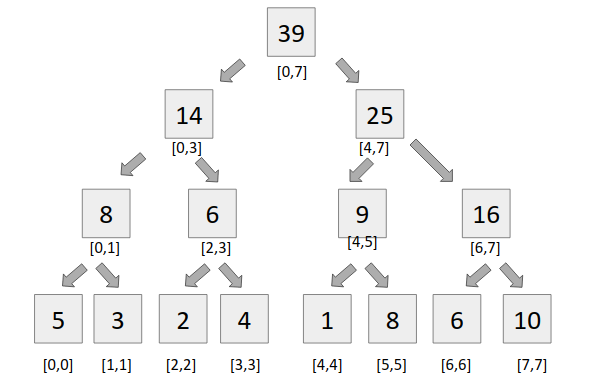
*= 1((2)1+log(n) - 1) / 2-1*

*= 2.2log(n)-1*

*= 2n-1*

For safety, we make segment tree of size 4\*n.

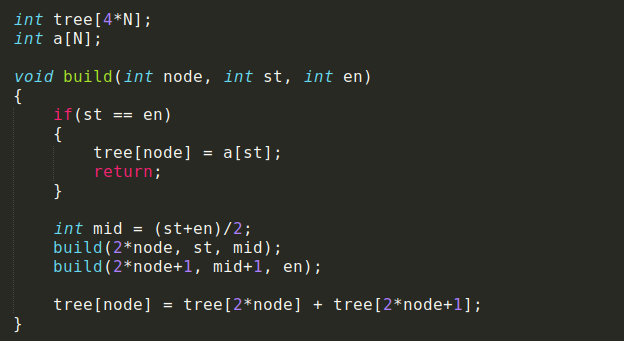
**Structure of segment tree**



**Building a segment tree**

It is very simple to build a segment tree, we use divide and conquer approach to build the segment tree.

Code:

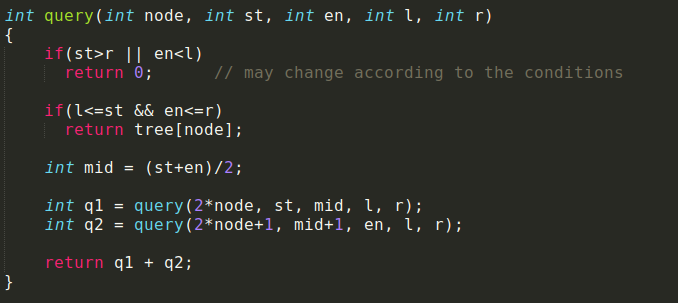


**Query**

For query, we see two types of segments

* Complete overlapping segments - When our st Partial overlapping segments and en lies completely in the range [l,r], it is called complete overlapping segment.
* Partial overlapping segments - When our st and en does not lie completely in the range [l,r], it is called partial overlapping segment.

Code:



**Update**

Updating an element in the segment tree is very similar to binary search.

We find out mid, and compare our index with mid and two conditions arise

1. Idx <= mid, then we recursively call the left child of the tree’s node.
2. Idx > mid, then we recursively call the right child of the tree’s node.

Code:

