Working of Binary Indexed Tree

Let the input array be A and the Binary Indexed array be BIT.

- **getSum(): returns the sum of the subarray [0, .., x] from BIT[] arr.**

  Algorithm:

  1. Initialize the sum as 0, and the curIdx as x+1.
  2. Do the following if curIdx is greater than 0
     a. Add the BIT[curIdx] to the sum.
     b. Update curIdx as

        parentIdx = curIdx - (curIdx & (-curIdx))
        curIdx = parentIdx

     c. Repeat step 2.
  3. Return sum.

- **update(): updates the BIT array if we change the original array as arr[curIdx] += val.**

  Algorithm:

  1. Initialize curIdx as x+1.
  2. Do the following if curIdx is less than or equal to n
     a. Add the value to BIT[curIdx] i,e,

        BIT[curIdx] += val

     b. Update curIdx to child index where childIndex is calculated as.

        childIndex = curIdx + (curIdx & (curIdx))
        curIdx = childIndex

     c. Repeat step 2.
  3. Return sum.

Implementation

- **Update()**

```
function update(n, curIdx, val, BIT[]){
        curIdx = curIdx + 1

        while curIdx <= n
                BIT[curIdx] = BIT[curIdx] + val
                childIndex = curIdx + (curIdx & (-curIdx))
                curIdx = childIndex

}
```

- **construction()**

Initially, all values in BIT[] are equal to 0. Then, we call the update() function for each element of the given array A to construct the Fenwick Tree.

```
function construction(n, A[]) {

        //  initialize a new array BIT[] equal to size of array A + 1
        i = 1
        while i <= n
                BIT[i] = 0
                i++

        i = 0
        //  call update function for every element of the arr
        while i <= n
                update(n, i, A[i], BIT)
                i++
}
```

- **getSum()**

```
function getSum(n, curIdx, BIT[]){
        sum = 0

        while curIdx > 0
                sum = sum + BIT[curIdx]
                parentIndex = curIdx - (curIdx & (-curIdx))
                curIdx = parentIndex
        return sum
}
```

Time Complexity: The maximum number of set bits in the binary representation of a number n is O(Logn). Therefore, we traverse at-most O(Logn) nodes in both getSum() and update() operations. The time complexity of the construction is O(nLogn) as it calls update() for all n elements.

Now for calculating the sum in some specific range (l,r), we can say that

rangeSum(l, r) = getSum(r) – getSum(l-1).