# **Explaining the Binary Indexed Tree**

<https://medium.com/@edison.cy.yang/explaining-the-binary-indexed-tree-34f27ad0a513>

In the last article, we talked about the trie data structure and how it can be used to implement a dictionary for fast prefix searches. Today we will talk about another type of tree — the Binary Indexed Tree, or the Fenwick Tree. This tree also has something to do with a prefix! It allows for efficient update and calculation of prefix sums.

# **Why?**

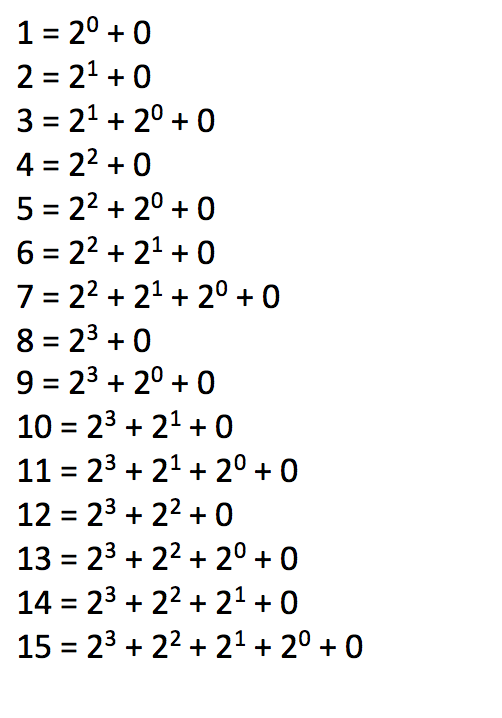
Before starting the introduction of the binary indexed tree, let’s see what kind of problem is it targeting. Let’s say we have an array [2, 3, -1, 0, 6] and we want to calculate the sum of the first 3 elements. We just add the numbers starting from index 0. Simple, right? The time to calculate the sum is proportional to the length of the array. Alternatively, we can also have another array that keeps track of the sum “so far”. The above array would correspond to [2, 5, 4, 4, 10]. This makes the lookup time of sum starting from 0 to input index O(1). What if we update the number at index 2? We will need to go through the sum array starting at index 2, until the end of the array. This becomes a hassle when we need to constantly update the array. This is where the binary indexed tree comes to the rescue!

# **Binary Representation of Numbers**

To understand how BIT works, we need to understand the binary numbers first. Binary, or base 2 numerical system, which is often used in mathematics and electronic systems, represents numbers with 0s and 1s. Each digit is referred to as a bit. Binary is used due to its straightforward implementation in electronic circuits using logic gates. The numbers we use on the daily are base 10 representation — numbers 0 to 9. Base 2 representation follows the same procedure as base 10, except there are only 0s and 1s. After a digit reaches 1, an increment resets it to 0 and also causes an increment of the next digit to the left. Number 1, 2, 3, 4, 5 correspond to 1, 10, 11, 100, 101 in binary representation.[INSERT MORE DEFINITION FROM WIKIPEDIA]

# **Each Integer is Sum of Powers of Two**

Every integer can be represented by the sum of powers of two. Below are examples of number 1 to 16:



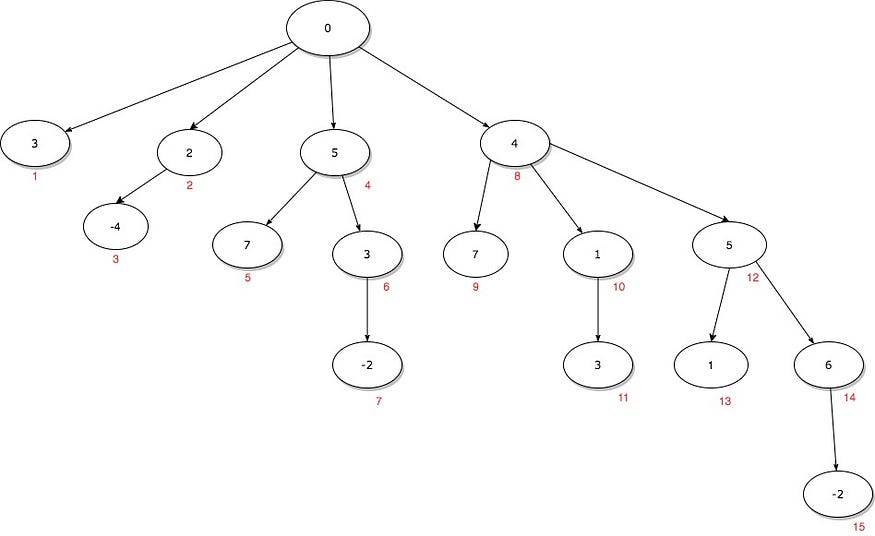
This information will be important for us in a little bit, stay tuned.

# **Example**

Let us have an array:

[3, 2, -4, 5, 7, 3, -2, 4, 7, 1, 3, 5, 1, 6, -2]  
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15//Indices

The above array can be represented in the binary indexed tree as illustrated below:

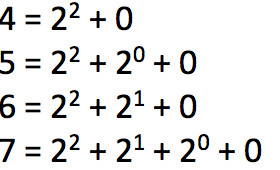


* Each node contains a value, which is its corresponding value in the array
* The number in red represents the node’s corresponding ***position*** in the array, note that this is not its 0 based index

Now you may be wondering, why is the tree structured this way?

Let’s look at the first level of the tree — their ***positions*** are 1, 2, 4, 8, which are all powers of two. We will pick a subtree, say the subtree starting from the node that represents position 4. The following are some observations:

* Nodes with positions 5, 6, 7 are reachable

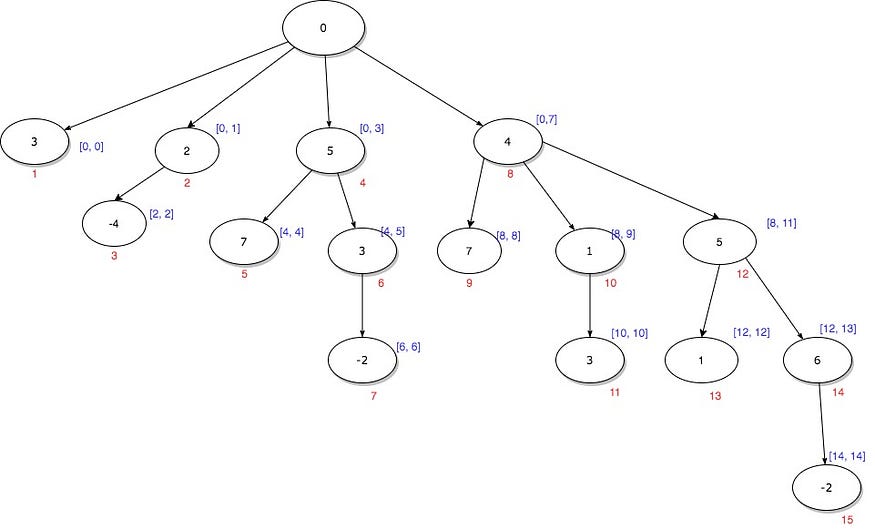


The 4 nodes in the subtree

* These 4 nodes can all be computed starting from 2²
* If we want to get to the node for position 6, we no longer have to start looping from the beginning of the array

# **The Prefix Range of Each Node**

I hope this is starting to make some sense. However, it’s not quite clear how this can help us solve the prefix sum problem. Let’s look at the diagram below with some additional information:



The original tree diagram, with added blue text

* The text in blue represents the information this node will contain from

[start\_index, end\_index]

* For example, the node that represents *position 4*will contain information from index 0 to index 3 in the original array
* The node that represents*position 5,*which we can reach from the node in the last bullet point,will contain information from:

[0, 3] and [4, 4] => [0, 4]

With the above observations, we can see that instead of storing values of each array element in each node, we should store the ***prefix sum of the range this node corresponds to***.

* The node with position 4 should store sum from index 0 to 3, which is

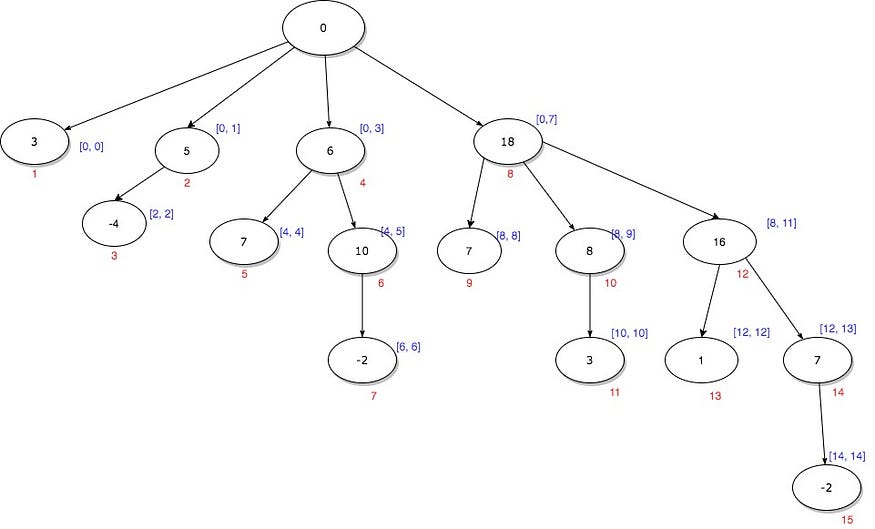
3 + 2 + (-4) + 5 = 6

* The node with position 5 will store sum from index 4 to 4, which is just 7.
* When we want to get the sum from index 0 to 4, simply use the information from the node with position 4:

6 + 7 = 13

# **Prefix Sum of the Range of Each Node**

Let's modify the tree diagram to reflect what we just mentioned — ****Each node will store the prefix sum of the index range this node is responsible for****



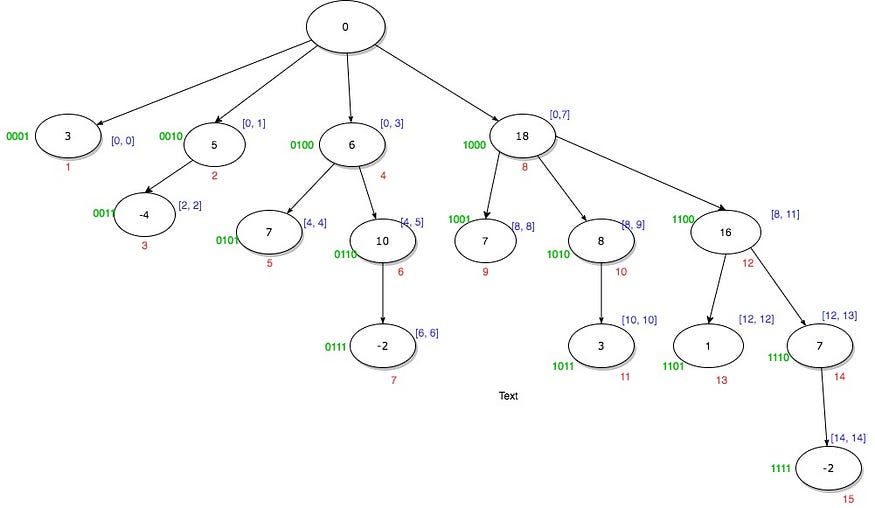
Elaborated from the previous diagram, except each node contains the prefix sum of a range

* When we are looking for the prefix sum at an index, simply start from the node itself, and traverse up to the root, while adding up the numbers.
* Let’s try to get the prefix sum at position 14, or index 13. Traverse up while adding the numbers: 7 →16 →18

7 + 16 + 18 = 41

# **Traversing the Tree Using Binary Index**

Now that we have simplified the process of finding prefix sum, we need to know how to “*traverse*” the tree. It is easy when we have a visual diagram like this. To achieve this, we will add some more information to the existing diagram — the binary representation of the position numbers:



Elaborated from the previous diagram, with binary representation of each number

* The green text represents the binary representation of each number (the “*position*” in red) of the node
* Let’s look at getting the prefix sum at *position*14 again, here are the binary representations of their position numbers as we traverse up the tree:

1110 -> 1100 -> 1000 -> 0000

* Notice something? We are flipping the ****rightmost 1 bit (also called the set bit)****as we go up the tree. This is true for any path.

As we traverse up or down the tree, we need to add to or subtract from the rightmost set bit with our current binary number. To isolate the rightmost set bit, with number 14(1110) as example:

const x = 14; ///1110  
const rightMostSetBit = x & -x; /// 2 = 0010  
x - rightMostSetBit; /// 12 = 1100

Great! Now we know how to traverse the tree using some binary magic. Below is the small program in JavaScript that will build a binary indexed tree, get the prefix sum of a given index in the array, and get the range sum of two given indices in the array.

class BIT {

constructor(nums) {

this.nums = nums;

this.n = nums.length;

this.BIT = new Array(this.n + 1).fill(0);

for (let i = 0; i < this.n; i++) {

this.init(i, nums[i]);

}

}

init(i, val) {

i++;

while (i <= this.n) {

this.BIT[i] += val;

i += (i & -i);

}

}

update(i, val) {

const diff = val - this.nums[i];

this.nums[i] = val;

this.init(i, diff);

}

getSum(i) {

let sum = 0;

i++;

while (i > 0) {

sum += this.BIT[i];

i -= (i & -i);

}

return sum;

}

sumRange(i, j) {

return this.getSum(j) - this.getSum(i - 1);

}

}

const nums = [1,3,5];

const bit = new BIT(nums);

console.log(bit.sumRange(0, 2));

bit.update(1, 2);

console.log(bit.sumRange(0, 2));

* The constructor will build the tree by traversing the tree from the root while calculating the sum. We can achieve this by

i += (i & -i)

* To get the prefix sum of a given index, we traverse up the tree, we can achieve this by

i -= (i & -i)

Feel free to clone the code from [https://github.com/edison-cy-yang/range-sum-query](https://github.com/edison-cy-yang/range-sum-query" \t "https://medium.com/@edison.cy.yang/_blank) and play around. You could plug in a large dataset and benchmark the performance, compare that with iterating through the whole array to get the sum.

# **Performance of Binary Indexed Tree**

* Prefix sum calculation: O(logN) since we only need to traverse the height of the tree, instead of the whole array
* Update: O(logN)
* Space: O(N) since we need to initialize an array of size N+1 to hold the binary indexed tree

# **Wrap Up**

We looked at the binary indexed tree and how it can be used to obtain large performance gain when calculating prefix sum. The idea of using binary numbers to traverse the tree is difficult to understand at the beginning, but once you read it over a few times you will have that “gotcha” moment. Here are some key takeaways:

* Each position in BIT can be represented by its binary number
* Each node is responsible for storing the prefix sum of a range in the tree
* To traverse up and down the tree, subtract/add the rightmost set bit to the current node

If you are interested, below are some videos that provide a good explanation of how the binary indexed tree works:

<https://www.youtube.com/watch?v=CWDQJGaN1gY>

<https://www.youtube.com/watch?v=v_wj_mOAlig>

**Practical Example: L307.Range Sum Query - Mutable**

**Based on above document, the most critical part is how to build up the Binary Indexed Tree with given nums array, below is the detail steps for the same input nums array shared on document**

======================================================================

Basic operation for i & -i

00000001 = 1

11111111 = -1

00000001 = 1 (1&-1)

00000010 = 2

11111110 = -2

00000010 = 2 (2&-2)

00000011 = 3

11111101 = -3

00000001 = 1 (3&-3)

00000100 = 4

11111100 = -4

00000100 = 4 (4&-4)

00000101 = 5

11111011 = -5

00000001 = 1 (5&-5)

00000110 = 6

11111010 = -6

00000010 = 2 (6&-6)

00000111 = 7

11111001 = -7

00000001 = 1 (7&-7)

etc...

======================================================================

Take example nums array to see how Binary Indexed Tree build up:

public NumArray(int[] nums) {

this.nums = nums;

n = nums.length;

BIT = new int[n + 1];

for (int i = 0; i < n; i++)

init(i, nums[i]);

}

public void init(int i, int val) {

i++;

while (i <= n) {

BIT[i] += val;

i += (i & -i);

}

}

nums = {3, 2, -4, 5, 7, 3, -2, 4, 7, 1, 3, 5, 1, 6, -2}

i must <= 15

----------------------------------------------------------------------

i=0 -> i++=1 -> 1+(1&-1)=2 -> 2+(2&-2)=4 -> 4+(4&-4)=8 -> 8+(8&-8)=16>15[discard]

which means the 1st round as i=0 will impact on BIT index as 1, 2, 4, 8 when building tree by adding nums[0]=3 to these indexes

Reverse thinking, after round i=0, BIT[1]=nums[0], BIT[2]=nums[0], BIT[4]=nums[0], BIT[8]=nums[0]

BIT=[0, 3, 3, 0, 3, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=1 -> i++=2 -> 2+(2&-2)=4 -> 4+(4&-4)=8 -> 8+(8&-8)=16>15[discard]

which means the 2nd round as i=1 will impact on BIT index as 2, 4, 8 when building tree by adding nums[1]=2 to these indexes

Reverse thinking, after round i=1, BIT[2]=nums[0]+nums[1], BIT[4]=nums[0]+nums[1], BIT[8]=nums[0]+nums[1]

BIT=[0, 3, 5, 0, 5, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=2 -> i++=3 -> 3+(3&-3)=4 -> 4+(4&-4)=8 -> 8+(8&-8)=16>15[discard]

which means the 3rd round as i=2 will impact on BIT index as 3, 4, 8 when building tree by adding nums[2]=-4 to these indexes

Reverse thinking, after round i=2, BIT[3]=nums[2], BIT[4]=nums[0]+nums[1]+nums[2], BIT[8]=nums[0]+nums[1]+nums[2]

BIT=[0, 3, 5, -4, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=3 -> i++=4 -> 4+(4&-4)=8 -> 8+(8&-8)=16>15[discard]

which means the 4th round as i=3 will impact on BIT index as 4, 8 when building tree by adding nums[3]=5 to these indexes

Reverse thinking, after round i=3, BIT[4]=nums[0]+nums[1]+nums[2]+nums[3], BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]

BIT=[0, 3, 5, -4, 6, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=4 -> i++=5 -> 5+(5&-5)=6 -> 6+(6&-6)=8 -> 8+(8&-8)=16>15[discard]

which means the 5th round as i=4 will impact on BIT index as 5, 6, 8 when building tree by adding nums[4]=7 to these indexes

Reverse thinking, after round i=4, BIT[5]=nums[4], BIT[6]=nums[4], BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]+nums[4]

BIT=[0, 3, 5, -4, 6, 7, 7, 0, 13, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=5 -> i++=6 -> 6+(6&-6)=8 -> 8+(8&-8)=16>15[discard]

which means the 6th round as i=5 will impact on BIT index as 6, 8 when building tree by adding nums[5]=3 to these indexes

Reverse thinking, after round i=5, BIT[6]=nums[4]+nums[5], BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]+nums[4]+nums[5]

BIT=[0, 3, 5, -4, 6, 7, 10, 0, 16, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=6 -> i++=7 -> 7+(7&-7)=8 -> 8+(8&-8)=16>15[discard]

which means the 7th round as i=6 will impact on BIT index as 7, 8 when building tree by adding nums[6]=-2 to these indexes

Reverse thinking, after round i=6, BIT[7]=nums[6], BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]+nums[4]+nums[5]+nums[6]

BIT=[0, 3, 5, -4, 6, 7, 10, -2, 14, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

i=7 -> i++=8 -> 8+(8&-8)=16>15[discard]

which means the 8th round as i=7 will impact on BIT index as 8 when building tree by adding nums[7]=4 to these indexes

Reverse thinking, after round i=7, BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]+nums[4]+nums[5]+nums[6]+nums[7]

BIT=[0, 3, 5, -4, 6, 7, 10, -2, 18, 0, 0, 0, 0, 0, 0, 0]

----------------------------------------------------------------------

etc...

======================================================================

After 8 rounds, we can see below relation build up between original input nums array and output BIT array

BIT[1]=nums[0]

BIT[2]=nums[0]+nums[1]

BIT[3]=nums[2]

BIT[4]=nums[0]+nums[1]+nums[2]+nums[3]

BIT[5]=nums[4]

BIT[6]=nums[4]+nums[5]

BIT[7]=nums[6]

BIT[8]=nums[0]+nums[1]+nums[2]+nums[3]+nums[4]+nums[5]+nums[6]+nums[7]

======================================================================

Update with relation between BITs to see range coverage

BIT[1]=nums[0]

BIT[2]=BIT[1]+nums[1]

BIT[3]=nums[2]

BIT[4]=BIT[2]+BIT[3]+nums[3]

BIT[5]=nums[4]

BIT[6]=BIT[5]+nums[5]

BIT[7]=nums[6]

BIT[8]=BIT[4]+BIT[6]+BIT[7]+nums[7]

======================================================================

Coverage relation

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*

\_\_\_\_\_\_\*

\_\_\* \_\_\*

\* \* \* \*

BIT indexes: 0 1 2 3 4 5 6 7 8

arr indexes: 0 1 2 3 4 5 6 7

======================================================================

**Refer to**

<https://leetcode.com/problems/range-sum-query-mutable/solutions/75753/java-using-binary-indexed-tree-with-clear-explanation/>

Java using Binary Indexed Tree with clear explanation

This is to share the explanation of the BIT and the meaning of the bit operations.

public class NumArray {

/\*\*

\* Binary Indexed Trees (BIT or Fenwick tree):

\* https://www.topcoder.com/community/data-science/data-science-

\* tutorials/binary-indexed-trees/

\*

\* Example: given an array a[0]...a[7], we use a array BIT[9] to

\* represent a tree, where index [2] is the parent of [1] and [3], [6]

\* is the parent of [5] and [7], [4] is the parent of [2] and [6], and

\* [8] is the parent of [4]. I.e.,

\*

\* BIT[] as a binary tree:

\* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\*

\* \_\_\_\_\_\_\*

\* \_\_\* \_\_\*

\* \* \* \* \*

\* indices: 0 1 2 3 4 5 6 7 8

\*

\* BIT[i] = ([i] is a left child) ? the partial sum from its left most

\* descendant to itself : the partial sum from its parent (exclusive) to

\* itself. (check the range of "\_\_").

\*

\* Eg. BIT[1]=a[0], BIT[2]=a[1]+BIT[1]=a[1]+a[0], BIT[3]=a[2],

\* BIT[4]=a[3]+BIT[3]+BIT[2]=a[3]+a[2]+a[1]+a[0],

\* BIT[6]=a[5]+BIT[5]=a[5]+a[4],

\* BIT[8]=a[7]+BIT[7]+BIT[6]+BIT[4]=a[7]+a[6]+...+a[0], ...

\*

\* Thus, to update a[1]=BIT[2], we shall update BIT[2], BIT[4], BIT[8],

\* i.e., for current [i], the next update [j] is j=i+(i&-i) //double the

\* last 1-bit from [i].

\*

\* Similarly, to get the partial sum up to a[6]=BIT[7], we shall get the

\* sum of BIT[7], BIT[6], BIT[4], i.e., for current [i], the next

\* summand [j] is j=i-(i&-i) // delete the last 1-bit from [i].

\*

\* To obtain the original value of a[7] (corresponding to index [8] of

\* BIT), we have to subtract BIT[7], BIT[6], BIT[4] from BIT[8], i.e.,

\* starting from [idx-1], for current [i], the next subtrahend [j] is

\* j=i-(i&-i), up to j==idx-(idx&-idx) exclusive. (However, a quicker

\* way but using extra space is to store the original array.)

\*/

int[] nums;

int[] BIT;

int n;

public NumArray(int[] nums) {

this.nums = nums;

n = nums.length;

BIT = new int[n + 1];

for (int i = 0; i < n; i++)

init(i, nums[i]);

}

public void init(int i, int val) {

i++;

while (i <= n) {

BIT[i] += val;

i += (i & -i);

}

}

void update(int i, int val) {

int diff = val - nums[i];

nums[i] = val;

init(i, diff);

}

public int getSum(int i) {

int sum = 0;

i++;

while (i > 0) {

sum += BIT[i];

i -= (i & -i);

}

return sum;

}

public int sumRange(int i, int j) {

return getSum(j) - getSum(i - 1);

}

}

// Your NumArray object will be instantiated and called as such:

// NumArray numArray = new NumArray(nums);

// numArray.sumRange(0, 1);

// numArray.update(1, 10);

// numArray.sumRange(1, 2);