## Range Sum Query - Mutable

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.

The *update(i, val)* function modifies *nums* by updating the element at index *i* to *val*.

**Example:**

```
Given nums = [1, 3, 5]

sumRange(0, 2) -> 9
update(1, 2)
sumRange(0, 2) -> 8
```

**Note:**

1. The array is only modifiable by the *update* function.
2. You may assume the number of calls to *update* and *sumRange* function is distributed evenly.

## Solution 1

```
public class NumArray {

    class SegmentTreeNode {
        int start, end;
        SegmentTreeNode left, right;
        int sum;

        public SegmentTreeNode(int start, int end) {
            this.start = start;
            this.end = end;
            this.left = null;
            this.right = null;
            this.sum = 0;
        }
    }

    SegmentTreeNode root = null;

    public NumArray(int[] nums) {
        root = buildTree(nums, 0, nums.length-1);
    }

    private SegmentTreeNode buildTree(int[] nums, int start, int end) {
        if (start > end) {
            return null;
        } else {
            SegmentTreeNode ret = new SegmentTreeNode(start, end);
            if (start == end) {
                ret.sum = nums[start];
            } else {
                int mid = start  + (end - start) / 2;
                ret.left = buildTree(nums, start, mid);
                ret.right = buildTree(nums, mid + 1, end);
                ret.sum = ret.left.sum + ret.right.sum;
            }
            return ret;
        }
    }

    void update(int i, int val) {
        update(root, i, val);
    }

    void update(SegmentTreeNode root, int pos, int val) {
        if (root.start == root.end) {
           root.sum = val;
        } else {
            int mid = root.start + (root.end - root.start) / 2;
            if (pos <= mid) {
                 update(root.left, pos, val);
            } else {
                 update(root.right, pos, val);
            }
            root.sum = root.left.sum + root.right.sum;
```

```
        }
    }

    public int sumRange(int i, int j) {
        return sumRange(root, i, j);
    }

    public int sumRange(SegmentTreeNode root, int start, int end) {
        if (root.end == end && root.start == start) {
            return root.sum;
        } else {
            int mid = root.start + (root.end - root.start) / 2;
            if (end <= mid) {
                return sumRange(root.left, start, end);
            } else if (start >= mid+1) {
                return sumRange(root.right, start, end);
            }  else {
                return sumRange(root.right, mid+1, end) + sumRange(root.left, sta
rt, mid);
            }
        }
    }
}
```

written by 2guotou original link here

## Solution 2

The idea is using “buckets”. Assume the length of the input array is n, we can partition the whole array into m buckets, with each bucket having k=n/m elements. For each bucket, we record two kind of information: 1) a copy of elements in the bucket, 2) the sum of all the elements in the bucket.

For example: If the input is [0,1,2,3,4,5,6,7,8,9], and we partition it into 4 buckets, formatted as {[numbers], sum}:

- bucket0: {[0, 1, 2], 3}
- bucket1: {[3, 4, 5], 12}
- bucket2: {[6, 7, 8], 21}
- bucket3: {[9], 9};

Updating is easy. You just need to find the right bucket, modify the element value, and change the “sum” value in that bucket accordingly. The operation takes O(1) time.

Summation is a little complicated. In the above example, let’s say we want to compute the sum in range [1, 7]. We can see, the numbers we want to accumulate are in bucket0, bucket1, and bucket2. Specifically, we only need parts of numbers in bucket0 and bucket2, and all the numbers in bucket1. Because the summation of all numbers in bucket1 have already been computed, we don’t need to compute it again. So, instead of doing (1+2) + (3+4+5) + (6+7), we can just do (1+2) + 12 + (6+7). We save two “+” operations. If you change the size of buckets, the number of saved “+” operations will be different. The questions is:

**What is the best size that can save the most “+” operations?**

Here is my analysis, which might be incorrect.

We have:

- The number of buckets is **m**.
- The size of each bucket is **k**.
- The length of input array is **n**, and we have **mk=n**.

In the worst case (the query is [0, n-1]), we will first add all the elements in bucket0, then add from bucket1 to bucket(m-2), and finally add all the elements in bucket(m-1), so we do **2k+m-2** “+” operations. We want to find the minimum value of **2k+m**. Because **2km=2n**, when **2k=m**, **2k+m** reaches the minimum value. (Need proof?) So we have **m = sqrt(2n)** and **k=sqrt(n/2)**.

Therefore, in the worst case, the best size of bucket is **k=sqrt(n/2)**, and the complexity is **O(2k+m-2)=O(2m-2)=O(m)=O(sqrt(2n))=O(n^0.5)**;

Thank you for pointing out any mistake!

```
class NumArray {
public:

    struct Bucket
    {
```

```
    int sum;
    vector<int> val;
};

int bucketNum;
int bucketSize;
vector<Bucket> Bs;

NumArray(vector<int> &nums) {
    int size = nums.size();
    int bucketNum = (int)sqrt(2*size);
    bucketSize = bucketNum/2;
    while(bucketSize * bucketNum<size) ++bucketSize;

    Bs.resize(bucketNum);
    for(int i=0, k=0; i<bucketNum; ++i)
    {
        int temp = 0;
        Bs[i].val.resize(bucketSize);
        for(int j=0; j<bucketSize && k<size; ++j, ++k)
        {
            temp += nums[k];
            Bs[i].val[j] = nums[k];
        }
        Bs[i].sum = temp;
    }
}

void update(int i, int val) {
    int x = i / bucketSize;
    int y = i % bucketSize;
    Bs[x].sum += (val - Bs[x].val[y]);
    Bs[x].val[y] = val;
}

int sumRange(int i, int j) {
    int x1 = i / bucketSize;
    int y1 = i % bucketSize;
    int x2 = j / bucketSize;
    int y2 = j % bucketSize;
    int sum = 0;

    if(x1==x2)
    {
        for(int a=y1; a<=y2; ++a)
        {
            sum += Bs[x1].val[a];
        }
        return sum;
    }

    for(int a=y1; a<bucketSize; ++a)
    {
        sum += Bs[x1].val[a];
    }
    for(int a=x1+1; a<x2; ++a)
    {
```

```
        {
            sum += Bs[a].sum;
        }
        for(int b=0; b<=y2; ++b)
        {
            sum += Bs[x2].val[b];
        }
        return sum;
    }
};
```

written by TTester original link here

## Solution 3

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]);
```

```
            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);
```

written by rikimberley original link here