<https://leetcode.com/problems/range-sum-query-immutable/>

Given an integer array nums, handle multiple queries of the following type:

Calculate the **sum** of the elements of nums between indices left and right **inclusive** where left <= right.

Implement the NumArray class:

* NumArray(int[] nums) Initializes the object with the integer array nums.
* int sumRange(int left, int right) Returns the **sum** of the elements of nums between indices left and right **inclusive** (i.e. nums[left] + nums[left + 1] + ... + nums[right]).

**Example 1:**

Input

["NumArray", "sumRange", "sumRange", "sumRange"]

[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]

Output

[null, 1, -1, -3]

Explanation

NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);

numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1

numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1

numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 + (-1) = -3

**Constraints:**

* 1 <= nums.length <= 104
* -105 <= nums[i] <= 105
* 0 <= left <= right < nums.length
* At most 104 calls will be made to sumRange.

**Attempt 1: 2023-06-25**

**Solution 1:  preSum array (10 min)**

**Style 1: preSum pad a '0' for preSum[0], then start recording cumulative sum from preSum[1] till preSum[len]**

class NumArray {

// nums = {-2, 0, 3, -5, 2, -1}

// preSum = {0, -2, -2, 1, -4, -2, -3}

int[] preSum;

public NumArray(int[] nums) {

int len = nums.length;

// preSum pad a '0' for preSum[0], then start recording

// cumulative sum from preSum[1] till preSum[len]

// e.g

// preSum[0] = 0

// preSum[1] = preSum[0] + nums[0]

// preSum[2] = preSum[0] + nums[0] + nums[1] = preSum[1] + nums[1]

// preSum[3] = preSum[0] + nums[0] + nums[1] + nums[2] = preSum[2] + nums[2]

// ... etc.

preSum = new int[len + 1];

for(int i = 1; i <= len; i++) {

preSum[i] = preSum[i - 1] + nums[i - 1];

}

}

public int sumRange(int left, int right) {

return preSum[right + 1] - preSum[left];

}

}

/\*\*

\* Your NumArray object will be instantiated and called as such:

\* NumArray obj = new NumArray(nums);

\* int param\_1 = obj.sumRange(left,right);

\*/

**Style 2: No pad, but initialize preSum[0] as nums[0]**

class NumArray {

// nums = {-2, 0, 3, -5, 2, -1}

// preSum = {-2, -2, 1, -4, -2, -3}

int[] preSum;

public NumArray(int[] nums) {

int len = nums.length;

// preSum do not pad a '0' for preSum[0], just recording

// cumulative sum from preSum[0] till preSum[len - 1]

// e.g

// preSum[0] = nums[0]

// preSum[1] = nums[0] + nums[1] = preSum[0] + nums[1]

// preSum[2] = nums[0] + nums[1] + nums[2] = preSum[1] + nums[2]

// ... etc.

preSum = new int[len];

preSum[0] = nums[0];

for(int i = 1; i < len; i++) {

preSum[i] = preSum[i - 1] + nums[i];

}

}

public int sumRange(int left, int right) {

if(left == 0) {

return preSum[right];

}

return preSum[right] - preSum[left - 1];

}

}

/\*\*

\* Your NumArray object will be instantiated and called as such:

\* NumArray obj = new NumArray(nums);

\* int param\_1 = obj.sumRange(left,right);

\*/

**Solution 2:  Segment Tree (10 min)**

**Style 1: Iterative**

class NumArray {

int[] tree;

int n;

public NumArray(int[] nums) {

n = nums.length;

tree = new int[n \* 2];

for(int i = n, j = 0; i < n \* 2; i++, j++) {

tree[i] = nums[j];

}

for(int i = n - 1; i > 0; i--) {

tree[i] = tree[i \* 2] + tree[i \* 2 + 1];

}

}

public int sumRange(int left, int right) {

int sum = 0;

left += n;

right += n;

while(left <= right) {

if(left % 2 == 1) {

sum += tree[left];

left++;

left /= 2;

} else {

left /= 2;

}

if(right % 2 == 0) {

sum += tree[right];

right--;

right /= 2;

} else {

right /= 2;

}

}

return sum;

}

}

/\*\*

\* Your NumArray object will be instantiated and called as such:

\* NumArray obj = new NumArray(nums);

\* int param\_1 = obj.sumRange(left,right);

\*/

Complexity Analysis

- Time complexity : O(log⁡n)

Time complexity is O(log⁡n) because on each iteration of the algorithm we move one level up, either to the parent of the current node or to the next sibling of parent to the left or right direction till the two boundaries meet. In the worst-case scenario this happens at the root after log⁡n iterations of the algorithm.

- Space complexity : O(1).

**Style 2: Recursive**

class NumArray {

int[] tree;

int n;

public NumArray(int[] nums) {

n = nums.length;

tree = new int[4 \* n];

buildTree(nums, 0, 0, n - 1);

}

private void buildTree(int[] nums, int treeIndex, int lo, int hi) {

if(lo == hi) {

tree[treeIndex] = nums[lo];

return;

}

int mid = lo + (hi - lo) / 2;

buildTree(nums, 2 \* treeIndex + 1, lo, mid);

buildTree(nums, 2 \* treeIndex + 2, mid + 1, hi);

tree[treeIndex] = tree[treeIndex \* 2 + 1] + tree[treeIndex \* 2 + 2];

}

public int sumRange(int left, int right) {

return sumRangeHelper(0, 0, n - 1, left, right);

}

private int sumRangeHelper(int treeIndex, int lo, int hi, int left, int right) {

if(lo > right || hi < left) {

return 0;

}

if(lo >= left && hi <= right) {

return tree[treeIndex];

}

int mid = lo + (hi - lo) / 2;

if(left > mid) {

return sumRangeHelper(treeIndex \* 2 + 2, mid + 1, hi, left, right);

} else if(right <= mid) {

return sumRangeHelper(treeIndex \* 2 + 1, lo, mid, left, right);

} else {

int leftSum = sumRangeHelper(treeIndex \* 2 + 1, lo, mid, left, mid);

int rightSum = sumRangeHelper(treeIndex \* 2 + 2, mid + 1, hi, mid + 1, right);

return leftSum + rightSum;

}

}

}

/\*\*

\* Your NumArray object will be instantiated and called as such:

\* NumArray obj = new NumArray(nums);

\* int param\_1 = obj.sumRange(left,right);

\*/

Both the read and update queries now take logarithmic O(log2(n))time, which is what we desired.

Complexity Analysis

- Time complexity : O(log⁡n)

- Space complexity : O(log⁡n)

**Step by Step explain Segment Tree construction in recursive way:**

nums=[-2, 0, 3, -5, 2, -1]

===>

tree=[-3, 1, -4, -2, 3, -3, -1, -2, 0, 0, 0, -5, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

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

tree[treeIndex] = nums[lo];

lo=hi=0

tree[7] = nums[0]

lo=hi=1

tree[8] = nums[1]

tree[treeIndex] = tree[treeIndex \* 2 + 1] + tree[treeIndex \* 2 + 2];

tree[3] = tree[7] + tree[8]

lo=hi=2

tree[4] = nums[2]

tree[1] = tree[3] + tree[4]

lo=hi=3

tree[11] = nums[3]

lo=hi=4

tree[12] = nums[4]

tree[5] = tree[11] + tree[12]

lo=hi=5

tree[6] = nums[5]

tree[2] = tree[5] + tree[6]

tree[0] = tree[1] + tree[2]

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

All initial element in nums will be leaf node of the tree when use recursive way to construct segment tree

tree[0]

/ \

tree[1] tree[2]

/ \ / \

tree[3] tree[4] tree[5] tree[6]

/ \ (nums[2]) / \ (nums[5])

tree[7] tree[8] tree[11] tree[12]

(nums[0]) (nums[1]) (nums[3]) (nums[4])

**Step by Step explain Segment Tree sumRange query in recursive way:**

nums=[-2, 0, 3, -5, 2, -1]

===>

tree=[-3, 1, -4, -2, 3, -3, -1, -2, 0, 0, 0, -5, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

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

tree[0]

/ \

tree[1] tree[2]

/ \ / \

tree[3] tree[4] tree[5] tree[6]

/ \ (nums[2]\*) / \ (nums[5])

tree[7] tree[8] tree[11] tree[12]

(nums[0]) (nums[1]\*) (nums[3]\*) (nums[4])

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

sumRange(1, 3) -> expected sum up by tree[8](nums[1]) + tree[4](nums[2]) + tree[11](nums[3]) based on above tree structure

===>

sumRangeHelper(int treeIndex, int lo, int hi, int left, int right)

=sumRangeHelper(0, 0, 5, 1, 3)

Start recursion with mid = 2

-> calling

int leftSum = sumRangeHelper(treeIndex \* 2 + 1, lo, mid, left, mid)

= sumRangeHelper(1, 0, 2, 1, 2)

mid = 1

-> calling again

int leftSum = sumRangeHelper(treeIndex \* 2 + 1, lo, mid, left, mid)

= sumRangeHelper(3, 0, 1, 1, 1)

mid = 0

left(= 1) > mid(= 0)

-> calling

return sumRangeHelper(treeIndex \* 2 + 2, mid + 1, hi, left, right);

return sumRangeHelper(8, 1, 1, 1, 1);

-> return tree[treeIndex] = tree[8] (= nums[1] = 0) --> found tree[8]

-> leftSum = 0

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

return to previous level recursion

mid = 1

-> calling

int rightSum = sumRangeHelper(treeIndex \* 2 + 2, mid + 1, hi, mid + 1, right)

= sumRangeHelper(4, 2, 2, 2, 2)

-> return tree[treeIndex] = tree[4] (= nums[2] = 3) --> found tree[4]

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

return to previous level recursion

mid = 2

-> return leftSum + rightSum = 0 + 3 = 3

-> leftSum = 3

so left branch under mid = 2 is 3

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

mid = 2

-> calling

int rightSum = sumRangeHelper(treeIndex \* 2 + 2, mid + 1, hi, mid + 1, right)

= sumRangeHelper(2, 3, 5, 3, 3)

mid = 4

right(= 3) < mid(= 4)

-> calling

return sumRangeHelper(treeIndex \* 2 + 1, lo, mid, left, right);

return sumRangeHelper(5, 3, 4, 3, 3)

mid = 3

-> calling again

return sumRangeHelper(treeIndex \* 2 + 1, lo, mid, left, right);

return sumRangeHelper(11, 3, 3, 3, 3)

-> return tree[treeIndex] = tree[11] (nums[3] = -5) --> found tree[11]

-> rightSum = -5

so right branch under mid = 3 is -5

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

return to previous level recursion

close recursion with mid = 2

-> return leftSum + rightSum = 3 + (-5) = -2

-> final result = -2

**Refer to**

<https://leetcode.com/problems/range-sum-query-immutable/solutions/75314/solutions-using-binary-indexed-tree-and-segment-tree/>

Segment Tree, O(n) build, O(lgn) modify, O(lgn) query

public class NumArray {

SegmentTreeNode root;

public NumArray(int[] nums) {

if (nums.length == 0) return;

root = buildTree(nums, 0, nums.length - 1);

}

public int sumRange(int i, int j) {

return query(root, i, j);

}

private int query(SegmentTreeNode node, int start, int end) {

int mid = node.start + ((node.end - node.start) >> 1);

if (start <= node.start && end >= node.end) {

return node.sum;

} else if (end <= mid) {

return query(node.left, start, end);

} else if (start > mid) {

return query(node.right, start, end);

} else if (start <= mid && end > mid) {

return query(node.left, start, mid) + query(node.right, mid + 1, end);

}

return 0;

}

private SegmentTreeNode buildTree(int[] nums, int l, int r) {

if (l == r) {

return new SegmentTreeNode(l, r, nums[l]);

}

int mid = l + ((r - l) >> 1);

SegmentTreeNode leftNode = buildTree(nums, l, mid);

SegmentTreeNode rightNode = buildTree(nums, mid + 1, r);

SegmentTreeNode node = new SegmentTreeNode(l, r, leftNode.sum + rightNode.sum);

node.left = leftNode;

node.right = rightNode;

return node;

}

class SegmentTreeNode {

int start, end, sum;

SegmentTreeNode left, right;

public SegmentTreeNode(int s, int e, int val) {

start = s;

end = e;

sum = val;

}

}

}

**Solution 3:  Binary Indexed Tree (10 min)**

**Style 1: Iterative**

class NumArray {

int[] BIT;

int n;

public NumArray(int[] nums) {

n = nums.length;

BIT = new int[n + 1];

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

init(i, nums[i]);

}

}

private void init(int i, int val) {

i++;

while(i <= n) {

BIT[i] += val;

i += (i & -i);

}

}

public int sumRange(int left, int right) {

return getSum(right) - getSum(left - 1);

}

private int getSum(int i) {

int sum = 0;

i++;

while(i > 0) {

sum += BIT[i];

i -= (i & -i);

}

return sum;

}

}

/\*\*

\* Your NumArray object will be instantiated and called as such:

\* NumArray obj = new NumArray(nums);

\* int param\_1 = obj.sumRange(left,right);

\*/

- Time complexity : O(log⁡n)

- Space complexity : O(log⁡n)

**Refer to**

<https://leetcode.com/problems/range-sum-query-immutable/solutions/75314/solutions-using-binary-indexed-tree-and-segment-tree/>

Binary Indexed Tree, O(lgn) modify, O(lgn) query

public class NumArray {

int[] tree;

public NumArray(int[] nums) {

tree = new int[nums.length + 1];

for (int i = 0; i < nums.length; i++) {

int j = i + 1;

while (j < nums.length + 1) {

tree[j] += nums[i];

j += lowBit(j);

}

}

}

public int sumRange(int i, int j) {

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

}

private int getSum(int i) {

int sum = 0;

while (i > 0) {

sum += tree[i];

i -= lowBit(i);

}

return sum;

}

private int lowBit(int x) {

return x&(-x);

}

}