<https://leetcode.com/problems/count-of-range-sum/description/>

Given an integer array nums and two integers lower and upper, return the number of range sums that lie in [lower, upper] inclusive.

Range sum S(i, j) is defined as the sum of the elements in nums between indices i and j inclusive, where i <= j.

**Example 1:**

**Input:** nums = [-2,5,-1], lower = -2, upper = 2

**Output:** 3

**Explanation:** The three ranges are: [0,0], [2,2], and [0,2] and their respective sums are: -2, -1, 2.

**Example 2:**

**Input:** nums = [0], lower = 0, upper = 0

**Output:** 1

**Constraints:**

1 <= nums.length <= 10^5

-2^31 <= nums[i] <= 2^31 - 1

-10^5 <= lower <= upper <= 10^5

The answer is **guaranteed a** **32-bit** integer

**Attempt 1: 2023-12-25**

**Solution 1: Binary Indexed Tree (720min)**

class Solution {

    public int countRangeSum(int[] nums, int lower, int upper) {

        int n = nums.length;

        long[] presum = new long[n + 1];

        for(int i = 0; i < n; i++) {

            presum[i + 1] = presum[i] + (long)nums[i];

        }

        long[] arr = new long[n \* 3 + 3];

        for(int i = 0, j = 0; i <= n; i++, j += 3) {

            arr[j] = presum[i];

            arr[j + 1] = presum[i] - (long)lower;

            arr[j + 2] = presum[i] - (long)upper;

        }

        TreeSet<Long> set = new TreeSet<>();

        for(long num : arr) {

            set.add(num);

        }

        long[] discretization = set.stream().mapToLong(i -> i).toArray();

        BIT bit = new BIT(discretization.length);

        int result = 0;

        for(long sum : presum) {

            int leftIndex = binarySearch(discretization, sum - upper);

            int rightIndex = binarySearch(discretization, sum - lower);

            result += bit.query(rightIndex) - bit.query(leftIndex - 1);

            bit.update(binarySearch(discretization, sum), 1);

        }

        return result;

    }

    private int binarySearch(long[] discretization, long val) {

        int lo = 0;

        int hi = discretization.length - 1;

        while(lo <= hi) {

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

            if(discretization[mid] >= val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

}

class BIT {

    int[] tree;

    int n;

    public BIT(int len) {

        n = len;

        tree = new int[n + 1];

    }

    public int query(int index) {

        int count = 0;

        // Change 'index' to 1-based

        index++;

        while(index > 0) {

            count += tree[index];

            index -= index & -index;

        }

        return count;

    }

    public void update(int index, int val) {

        // Change 'index' to 1-based

        index++;

        while(index <= n) {

            tree[index] += val;

            index += index & -index;

        }

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Step by Step**

e.g nums = {-2, 5, -1}

presum = {0, -2, 3, 2}

We calculate all prefix sums of nums and store them in 'presum'. The prefix sum up to index i is the sum of all elements from the beginning of the array up to and including i. The initial prefix sum, S(0, -1), is 0. This is useful because a subarray sum S(i, j) can be easily calculated as presum[j] - presum[i - 1].

the definition of presum[i] is the prefix sum of nums[..i], ith number inclusive -> after adding one extra slot at index = 0 for 0, which cause the shift of all the indexes to 1 right position, the definition change to presum[i] is the prefix sum of nums[..i), ith number exclusive, which is nums[..i - 1], so take example from nums = {-2, 5, -1}, we get presum[0] = nums[-1] = empty array = 0, presum[1] = nums[0] = -2, presum[2] = nums[0] + nums[1] = -2 + 5 = 3, presum[3] = nums[0] + nums[1] + nums[2] = 2

(1) range sum S(0, -1) -> means empty array, range is [] empty

Theoretically, presum[-1] = 0 but index cannot be -1, so the actual mapping is presum[0] = 0, we create one extra slot at index = 0 for 0, which cause the shift of all the indexes to 1 right position

(2) range sum S(0, 0) -> i = j = 0

Theoretically, it suppose we should have "presum[0] - presum[0 - 1] = presum[0] - presum[-1] = nums[0] - 0 = nums[0]", since we create one extra slot for 0, which cause the shift of all the indexes to 1 position right, we actually have "presum[1] - presum[1 - 1] = presum[1] - presum[0] = -2 - 0 = nums[0] = 2"

(3) range sum S(0, 1) -> i = 0, j = 1

Theoretically, it suppose we should have "presum[1] - presum[0 - 1] = presum[1] - presum[-1] = nums[0] + nums[1] - 0 = nums[0] + nums[1]", since we create one extra slot for 0, which cause the shift of all the indexes to 1 position right, we actually have "presum[2] - presum[1 - 1] = presum[2] - presum[0] = 3 - 0 = nums[0] + nums[1] = 3"

(4) range sum S(0, 2) -> i = 0, j = 2

Theoretically, it suppose we should have "presum[2] - presum[0 - 1] = presum[2] - presum[-1] = nums[0] + nums[1] + nums[2] - 0 = nums[0] + nums[1] + nums[2]", since we create one extra slot for 0, which cause the shift of all the indexes to 1 position right, we actually have "presum[3] - presum[1 - 1] = presum[3] - presum[0] = 2 - 0 = nums[0] + nums[1] + nums[2] = 2"

(5) range sum S(1, 2) -> i = 1, j = 2

Theoretically, it suppose we should have "presum[2] - presum[1 - 1] = presum[2] - presum[0] = nums[0] + nums[1] + nums[2] - nums[0] = nums[1] + nums[2]", since we create one extra slot for 0, which cause the shift of all the indexes to 1 position right, we actually have "presum[3] - presum[2 - 1] = presum[3] - presum[1] = 2 - (-2) = nums[0] + nums[1] + nums[2] = 4"

And the concern about how the extra one slot add ahead in presum array impact on BIT array ? Will the query() on index hit index = 0 in BIT array like BIT[0] ?

In simple, the extra one slot add ahead in presum array magically match to the extra one slot add on BIT array, its a coincidence but make the two "extra one slot" working in harmony.

The purpose of extra one slot in presum array is handling presum calculation subarray start from index = 0 element in given array.

The purpose of extra one slot in BIT array is make sure index start with 1-based, because bit manipulation as x & -x cannot work properly on 0-based.

So the extra one slot on BIT will guarantee the index left boundary of query of index result will >= 1, it never hits index = 0, and the query of index result guaranteed >= 1 is what we hope for presum usage, because in range sum formula "range num [i, j] = presum[j] - presum[i - 1]" requires its left boundary i >= 1, as 'i - 1' must >= 0

This choice allows for consistent handling of ranges and simplifies the implementation of the Binary Indexed Tree (BIT) and the search for range sums in the main logic of the solution. It ensures that the prefix sum of any subarray (including the entire array) can be calculated using presum[rightIndex] - presum[leftIndex - 1]. Having the extra 0 at the start makes this calculation straightforward and avoids special cases for the first index.

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

Why we have to use BIT and how query logic build up ?

https://leetcode.com/problems/count-of-range-sum/solutions/78026/an-o-n-log-n-solution-via-fenwick-tree/

Let presum[i] be the prefix sum of nums[..i]. Then the range-sum of [i, j] is equal to presum[j] - presum[i - 1], and after extra 0 adding ahead in presum array, presum[i] change to be the prefix sum of nums[..i), ith number change from inclusive to exclusive, as nums[..i) or nums[..i - 1], the range-sum of [i, j] get update accordingly and equal to presum[j + 1] - presum[i]. We enumerate all i's. For any fixed i, we need to query the count of j's satisfying lower ≤ presum[j + 1] - presum[i] ≤ upper, i.e., lower + presum[i] ≤ presum[j + 1] ≤ upper + presum[i], for all i ≤ j < n. Hence for any fixed i, we find a pair [lower + presum[i], upper + presum[i]] first, and the difference between these two values's mapping count query out from BIT structure is the count of j's satisfying lower ≤ presum[j + 1] - presum[i] ≤ upper. This kind of query can be answered, in O(log n) time, by using data structures like Fenwick Tree or Segment Tree. Therefore, the total runtime is O(n log n).

Note: i, j refer to the index from original given nums, have i ≤ j < n, n is nums.length

Still take the look on same input:

nums = {-2, 5, -1}

presum = [0, -2, 3, 2]

sortedValues = [-4, -2, 0, 1, 2, 3, 4, 5...]

initial status

bit = [0, 0, 0, 0, 0, 0, 0, 0, 0]

After discretization unique sorted presum array mapping to BIT array

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

bit = [0, 0, 0, 0, 0, 0, 0, 0, 0] -> Round 1 presum = 0 -> bit.query(5) - bit.query(2 - 1) / bit.update(0)

bit = [0, 0, 0, 1, 1, 0, 0, 0, 1] -> Round 2 presum = -2 -> bit.query(3) - bit.query(1 - 1) / bit.update(-2)

bit = [0, 0, 1, 1, 2, 0, 1, 0, 3] -> Round 3 presum = 3 -> bit.query(8) - bit.query(4 - 1) / bit.update(3)

bit = [0, 0, 1, 1, 2, 1, 2, 0, 4] -> Round 4 presum = 2 -> bit.query(7) - bit.query(3 - 1) / bit.update(2)

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

Round 1: sum(presum[0]) = 0 -> for Round 1 will definitely return 0 because that's the query on extra slot = 0 of presum, the update on count in BIT array for this extra slot = 0 only happen after query done in Round 1, that will contribute to query in Round 2 only.

leftIndex = search(sortedValues, uniqueCount, sum - upper)

-> leftIndex = search(sortedValues, uniqueCount, 0 - 2)

-> leftIndex = 2 (1-based)

rightIndex = search(sortedValues, uniqueCount, sum - lower)

-> rightIndex = search(sortedValues, uniqueCount, 0 - (-2))

-> rightIndex = 5 (1-based)

result += bit.query(rightIndex) - bit.query(leftIndex - 1)

-> result += bit.query(5) - bit.query(2 - 1)

-> result += 0 - 0 = 0

update for presum = 0

bit.update(search(sortedValues, uniqueCount, 0), 1)

-> bit.update(3, 1)

-> bit = [0, 0, 0, 1, 1, 0, 0, 0, 1]

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

Round 2: sum(presum[1]) = -2 -> Round 2 is the actual first round query we expect a count return, because the query result based on previous Round 1 update result in range [leftIndex, rightIndex] calculated in Round 2

leftIndex = search(sortedValues, uniqueCount, sum - upper)

-> leftIndex = search(sortedValues, uniqueCount, -2 - 2)

-> leftIndex = 1 (1-based)

rightIndex = search(sortedValues, uniqueCount, sum - lower)

-> rightIndex = search(sortedValues, uniqueCount, -2 - (-2))

-> rightIndex = 3 (1-based)

result += bit.query(rightIndex) - bit.query(leftIndex - 1)

-> result += bit.query(3) - bit.query(1 - 1)

-> result += 1 - 0 = 0 + 1 = 1

update for presum = -2

bit.update(search(sortedValues, uniqueCount, -2), 1)

-> bit.update(3, 1)

-> bit = [0, 0, 1, 1, 2, 0, 0, 0, 2]

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

Round 3: sum(presum[2]) = 3

leftIndex = search(sortedValues, uniqueCount, sum - upper)

-> leftIndex = search(sortedValues, uniqueCount, 3 - 2)

-> leftIndex = 4 (1-based)

rightIndex = search(sortedValues, uniqueCount, sum - lower)

-> rightIndex = search(sortedValues, uniqueCount, 3 - (-2))

-> rightIndex = 8 (1-based)

result += bit.query(rightIndex) - bit.query(leftIndex - 1)

-> result += bit.query(8) - bit.query(4 - 1)

-> result += 2 - 2 = 1 + 0 = 1

update for presum = 3

bit.update(search(sortedValues, uniqueCount, 3), 1)

-> bit.update(3, 1)

-> bit = [0, 0, 1, 1, 2, 0, 1, 0, 3]

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

Round 4: sum(presum[3]) = 2

leftIndex = search(sortedValues, uniqueCount, sum - upper)

-> leftIndex = search(sortedValues, uniqueCount, 2 - 2)

-> leftIndex = 3 (1-based)

rightIndex = search(sortedValues, uniqueCount, sum - lower)

-> rightIndex = search(sortedValues, uniqueCount, 2 - (-2))

-> rightIndex = 7 (1-based)

result += bit.query(rightIndex) - bit.query(leftIndex - 1)

-> result += bit.query(7) - bit.query(3 - 1)

-> result += 3 - 1 = 1 + 2 = 3

update for presum = 2

bit.update(search(sortedValues, uniqueCount, 2), 1)

-> bit.update(2, 1)

-> bit = [0, 0, 1, 1, 2, 1, 2, 0, 4]

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

Why in discretization we have 'presum[i] - lower' and 'presum[i] - upper' ?

presum[i] + lower <= presum[j + 1] <= presum[i] + upper

=> lower ≤ presum[j + 1] - presum[i] ≤ upper

as we have below:

presum[j + 1] = x

presum[i] = l or r

then relationship is:

x - upper <= l -> l represents for presum[i], we need to find its last right with Binary Search template

x - lower >= r -> r represents also for presum[i], we need to find its first left with Binary Search template

to get l and r we use below:

discretization[j + 1] = presum[i] - lower;

discretization[j + 2] = presum[i] - upper;

**Refer to**

<https://algo.monster/liteproblems/327>

**Problem Description**

The problem presents an integer array nums and asks to find the quantity of subarray sums that fall within the range [lower, upper] inclusive. A subarray sum, S(i, j), is the sum of elements from the i-th to the j-th element, where i and j are indices of the array with the condition that i <= j.

**Intuition**

To solve this problem, a direct approach would involve checking all possible subarrays and their sums, which would be inefficient and lead to a high time complexity. Instead, we consider using data structures like Binary Indexed Trees (BIT) or Segment Trees to optimize the process.

The intuition behind using a Binary Indexed Tree in this solution is that we can efficiently update frequencies and query cumulative frequencies of different sums that we encounter.

Here's what we do step-by-step, simplified:

We first create the prefix sums of the array. By doing this, we can find the sum of any subarray using subtraction of two prefix sums.

Then we need to discretize the data, meaning we map the prefix sums to a new set of integers that are tightly packed. This is necessary because Binary Indexed Trees work with indices and we need to use indices to represent the prefix sums and differences like x - lower and x - upper.

We initialize a Binary Indexed Tree to keep track of the number of times a particular prefix sum appears.

We iterate through the prefix sums, and for each sum, we find the boundaries (l and r) of sums that satisfy the lower and upper requirements by searching the discretized values.

We query BIT by these boundaries to find out how many prefix sums fall within our desired range.

Finally, we update the BIT with the current prefix sum to include it in our future queries.

The final answer is the total number of times we found subarray sums that fell within [lower, upper], accumulated throughout the iteration.

By doing this, we greatly reduce the number of operations needed as compared to the brute force approach. Each update and query in the BIT takes logarithmic time to complete, which makes the solution much more efficient.

**Solution Approach**

The solution employs a Binary Indexed Tree (BIT), also known as a Fenwick Tree, to efficiently compute the frequencies of the cumulative sums encountered. A BIT provides two significant operations: update and query, both of which operate in O(log n) time, where n is the number of elements. Understanding how BIT works are essential to grasp the solution's approach.

The update function is used to add a value to an element in the binary indexed tree, which in this context represents incrementing the count of a particular prefix sum. The query function is used to get the cumulative frequency up to an index, which helps us count the number of sums that fall within our desired range.

Here's a closer look at the implementation steps:

We calculate all prefix sums of nums and store them in s. The prefix sum up to index i is the sum of all elements from the beginning of the array up to and including i. The initial prefix sum, S(0, -1), is 0. This is useful because a subarray sum S(i, j) can be easily calculated as s[j] - s[i-1].

Then, we discretize our sums because BITs handle operations based on indices. We create an arr that consists of the unique values that our prefix sum might be subtracted by lower and upper (as x - lower and x - upper), as well as the prefix sums themselves.

We create a Binary Indexed Tree tree based on the length of the arr which now contains all the unique values that are relevant for querying the number of sums within the [lower, upper] range.

As we iterate through each prefix sum x in s, we determine l and r such that x - upper <= l and x - lower >= r. We identify the indices l and r in arr which the sums that fall in [lower, upper] will be between. We use binary search (bisect\_left) since arr is sorted.

Then, we query the Binary Indexed Tree between l and r to find out how many prefix sums we’ve encountered fall within our desired range. This step essentially tells us the count of subarray sums ending at the current index which are within the range [lower, upper].

After querying, we increment the frequency count of the current prefix sum in the BIT so it could be used to find the range counts of subsequent subarray sums.

We continue to sum these counts into an accumulator, ans, that will ultimately contain the total count of valid range sums that satisfy the problem's condition.

Finally, we return ans which represents the number of subarray sums inside the range [lower, upper].

Through the usage of BIT and an ordered set of values, the implemented solution optimizes from a brute-force approach with potentially quadratic time complexity O(n^2) to a more efficient O(n log n) time complexity, where n is the length of the input list nums.

**Example Walkthrough**

Let's walk through a small example to illustrate the solution approach. Consider the array nums = [1, 2, -1] and we want to find the number of subarray sums within the range [1, 2].

Calculate Prefix Sums: We begin by calculating the prefix sums s:

idx 0 1 2 3

s 0 1 3 2

The prefix sums have an extra 0 at the start to handle subarrays that begin at the start of the array.

Discretize Sums: We prepare to discretize the data by creating an array arr which includes all prefix sums, and each prefix sum subtracted by lower and upper:

original sums (s): 0, 1, 3, 2

subtract lower (1): -1, 0, 2, 1

subtract upper (2): -2, -1, 1, 0

combined (arr): -2, -1, 0, 1, 2, 3

After sorting and removing duplicates, we get arr = [-2, -1, 0, 1, 2, 3].

Initialize BIT: A Binary Indexed Tree tree is initialized to track frequencies of these sums.

Iterate Prefix Sums: For each x in s, we find the range [l, r] for sums between [lower, upper] as discrete indices in arr:

For x = 0: l = index of 0 - 2 (-2) = 0, r = index of 0 - 1 (-1) = 1

For x = 1: l = index of 1 - 2 (-1) = 1, r = index of 1 - 1 (0) = 2

For x = 3: l = index of 3 - 2 (1) = 3, r = index of 3 - 1 (2) = 4

For x = 2: l = index of 2 - 2 (0) = 2, r = index of 2 - 1 (1) = 3

BIT Queries: Initially, our BIT is empty so querying it gives 0, which is expected as no subarrays have been processed.

BIT Updates: We update BIT with each prefix sum as we iterate:

Update with 0: tree now stores frequency of sum 0 as index 2.

Update with 1: tree now stores frequency of sum 1 as index 3.

Update with 3: tree now stores frequency of sum 3 as index 5.

Update with 2: tree now stores frequency of sum 2 as index 4.

Query and Update: With each iteration, after querying, the BIT is updated:

When processing x = 0, there are no sums [1, 2] yet. Query result is 0 for this prefix. Update BIT with 0.

When processing x = 1, the previous sum 0 is within [1-2]=[1,0], so we count 1. Update BIT with 1.

When processing x = 3, the previous sums (0, 1) contributes to count, as both are within [3-2, 3-1]=[1,2], we count 1 more for the sum (1) + the existing count 1 = 2. Update BIT with 3.

When processing x = 2, sums (0, 1, 3) contributes to count, as (3) is the only within [2-2, 2-1]=[0,1], which is our initial sum, we just count 1 more for a total of 3. Update BIT with 2.

Accumulate Answer: As we iterate, we keep a running sum ans:

ans = 0 + 1 + 1 (from prefix sum 3) + 1 (from prefix sum 2) = 3

This gives us our final answer: 3. There are three subarrays whose sums are within the range [1, 2]: [1], [2], and [1, 2, -1].

Thus, the steps of calculating prefix sums, discretizing the data, initializing and updating the BIT, querying for count of subarray sums within [lower, upper], and maintaining the cumulative count lead to an efficient computation of the desired result.

**Java Solution**

import java.util.Arrays;

// A class representing a Binary Indexed Tree (also known as a Fenwick Tree).

class BinaryIndexedTree {

    private final int size;  // Size of the array

    private final int[] tree;  // The tree represented as an array

    // Constructor to initialize the tree array with a given size.

    public BinaryIndexedTree(int size) {

        this.size = size;

        this.tree = new int[size + 1];

    }

    // Update the tree with a value at a specific index.

    public void update(int index, int value) {

        while (index <= size) {

            tree[index] += value;

            index += index & -index;

        }

    }

    // Query the cumulative frequency up to a given index.

    public int query(int index) {

        int sum = 0;

        while (index > 0) {

            sum += tree[index];

            index -= index & -index;

        }

        return sum;

    }

}

// A solution class containing the method to count the range sum.

class Solution {

    // Main method to count the number of range sums that lie in [lower, upper].

    public int countRangeSum(int[] nums, int lower, int upper) {

        int n = nums.length;

        long[] prefixSums = new long[n + 1];  // Array to hold prefix sums.

        for (int i = 0; i < n; i++) {

            prefixSums[i + 1] = prefixSums[i] + nums[i];

        }

        // Create a sorted array that will hold unique values needed for the Binary Indexed Tree.

        long[] sortedValues = new long[n \* 3 + 3];

        for (int i = 0, j = 0; i <= n; i++, j += 3) {

            sortedValues[j] = prefixSums[i];

            sortedValues[j + 1] = prefixSums[i] - lower;

            sortedValues[j + 2] = prefixSums[i] - upper;

        }

        Arrays.sort(sortedValues);

        int uniqueCount = 0;  // Count of unique elements.

        // Filter out duplicates to retain only unique elements.

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

            if (i == 0 || sortedValues[i] != sortedValues[i - 1]) {

                sortedValues[uniqueCount++] = sortedValues[i];

            }

        }

        // Creating a Binary Indexed Tree for the range of unique elements.

        BinaryIndexedTree bit = new BinaryIndexedTree(uniqueCount);

        int result = 0;  // Initialize the result.

        for (long sum : prefixSums) {

            int leftIndex = search(sortedValues, uniqueCount, sum - upper);

            int rightIndex = search(sortedValues, uniqueCount, sum - lower);

            result += bit.query(rightIndex) - bit.query(leftIndex - 1);

            bit.update(search(sortedValues, uniqueCount, sum), 1);

        }

        return result;

    }

    // Binary search helper method to find the index of a given value.

    private int search(long[] values, int right, long target) {

        int left = 0;

        while (left < right) {

            int mid = (left + right) >> 1;  // Equivalent to dividing by 2.

            if (values[mid] >= target) {

                right = mid;

            } else {

                left = mid + 1;

            }

        }

        return left + 1;  // Returns the index in the BIT, which is 1-based.

    }

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the countRangeSum function is determined by several factors:

Calculating the prefix sums of the nums array: The accumulate function is used to calculate the prefix sums which takes O(n) time where n is the length of the nums array.

Preparing the arr array for the Binary Indexed Tree (BIT): The set comprehension and sorting takes O(n log n) time, as it creates a set with potentially 3n elements (in the worst case, every expression in the set comprehension is unique) and then sorts it.

Insertions into the BIT: The loop runs for each element of the calculated prefix sums array s. For each element, two operations take place: update and query. The update operation has a complexity of O(log m) where m is the length of the arr array, which represents the number of unique values among all the prefix sums and their respective lower and upper sums. The same goes for the query operation. Since there are n elements, the overall complexity of this part is O(n log m).

Combining these together, the total time complexity is O(n log n) + O(n log m). Since m is derived from n, the order of growth is bounded by the number of operations linked to n, which results in O(n log n).

**Space Complexity**

The space complexity is determined by:

The s array for storing the prefix sums, which takes O(n) space.

The arr array for storing unique values for BIT, which in the worst case can store 3n elements (if all prefix sums and the respective lower and upper sums are unique), taking O(n) space (since they are values derived from the original input nums).

The c array inside the BIT instance, which also takes O(m) space. As with time complexity, m is related to n and the space required is proportional to the number of unique elements that can be from the original array.

Therefore, the total space complexity of the entire operation is O(n) as it scales linearly with the input size.

**Solution 2: Segment Tree (1440min)**

**Segment Tree without Discretization**

**Style 1: ArrayList Segment Tree**

**Style 1-1: ArrayList implementation without Discretization and build / update / query Segment Tree start with treeIndex = 1**

class Solution {

public int countRangeSum(int[] nums, int lower, int upper) {

int n = nums.length;

//List<Pair<Long, Integer>> v = new ArrayList<>();

List<Long[]> list = new ArrayList<>();

// v.add(new Pair<>((long) nums[0], 0));

list.add(new Long[]{(long)nums[0], (long)0});

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

//v.add(new Pair<>(v.get(i - 1).getKey() + nums[i], i));

list.add(new Long[]{list.get(i - 1)[0] + nums[i], (long)i});

}

//v.sort(Comparator.comparingLong(Pair::getKey));

Collections.sort(list, (a, b) -> a[0].compareTo(b[0]));

//List<Long> u = new ArrayList<>(Collections.nCopies(n, 0L));

List<Long> tmp = new ArrayList<>(Collections.nCopies(n, 0L));

//Map<Integer, Integer> uMap = new HashMap<>();

Map<Integer, Integer> map = new HashMap<>();

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

//u.set(i, v.get(i).getKey());

tmp.set(i, list.get(i)[0]);

//uMap.put(v.get(i).getValue(), i);

map.put(list.get(i)[1].intValue(), i);

}

long Lower = lower;

long Upper = upper;

//SegmentTree sgTree = new SegmentTree(u);

SegmentTree segmentTree = new SegmentTree(tmp);

int count = 0;

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

//count += sgTree.query(1, 0, n - 1, Lower, Upper);

count += segmentTree.query(1, 0, n - 1, Lower, Upper);

Lower += nums[i];

Upper += nums[i];

//sgTree.update(1, 0, n - 1, uMap.get(i));

segmentTree.update(1, 0, n - 1, map.get(i));

}

return count;

}

}

// The Segment Tree build / update / query method start with treeIndex = 1,

// not treeIndex = 0, then child = 2 \* i & 2 \* i + 1, if start with

// treeIndex = 0, then child = 2 \* i + 1 & 2 \* i + 2, refer L307

class SegmentTree {

private List<Long> list; // a copy of the input vector

// The node tree[treeIndex] over the range [left, right] represents the number of valid tmp[k]'s in this range,

// for left <= k <= right. tree[treeIndex] is just (right - left + 1) at the beginning.

private List<Integer> tree;

private int length; // 4\*N

private int N; // size of the input vector

public SegmentTree(List<Long> tmp) {

N = tmp.size();

length = 4 \* N;

tree = new ArrayList<>(Collections.nCopies(length, 0));

list = new ArrayList<>(Collections.nCopies(N, 0L));

build(tmp, 1, 0, N - 1);

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

list.set(i, tmp.get(i));

}

}

private void build(List<Long> tmp, int treeIndex, int left, int right) {

if (left == right) {

tree.set(treeIndex, 1);

return;

}

int mid = (left + right) / 2;

build(tmp, 2 \* treeIndex, left, mid);

build(tmp, 2 \* treeIndex + 1, mid + 1, right);

tree.set(treeIndex, tree.get(2 \* treeIndex) + tree.get(2 \* treeIndex + 1));

}

// Update tmp[index] as a removed entry

public void update(int treeIndex, int left, int right, int index) {

if (left == right) {

tree.set(treeIndex, 0);

return;

}

int mid = (left + right) / 2;

if (index <= mid) {

update(2 \* treeIndex, left, mid, index);

} else {

update(2 \* treeIndex + 1, mid + 1, right, index);

}

tree.set(treeIndex, tree.get(2 \* treeIndex) + tree.get(2 \* treeIndex + 1));

}

// Return the number of valid leaf nodes of which values are between Lower and Upper, both ends inclusive.

// If the node tree[treeIndex] represents the range [left, right] on the input array 'tmp',

// then the maximum value of 'tmp' over [left, right] = tmp[right]

// and the minimum value of 'tmp' over [left, right] = tmp[left]

// because 'tmp' is sorted increasingly.

public int query(int treeIndex, int left, int right, long Lower, long Upper) {

if (Lower <= list.get(left) && list.get(right) <= Upper) {

return tree.get(treeIndex);

}

if (list.get(right) < Lower || list.get(left) > Upper) {

return 0;

}

int mid = (left + right) / 2;

return query(2 \* treeIndex, left, mid, Lower, Upper) + query(2 \* treeIndex + 1, mid + 1, right, Lower, Upper);

}

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Step by Step**

e.g

nums = {-2, 5, -1}, lower = -2, upper = 2

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

    int n = nums.length;

    List<Long[]> list = new ArrayList<>();

    list.add(new Long[]{(long)nums[0], (long)0});

    for (int i = 1; i < n; i++) {

        list.add(new Long[]{list.get(i - 1)[0] + nums[i], (long)i});

    }

    Collections.sort(list, (a, b) -> a[0].compareTo(b[0]));

    List<Long> tmp = new ArrayList<>(Collections.nCopies(n, 0L));

    Map<Integer, Integer> map = new HashMap<>();

    for (int i = 0; i < n; i++) {

        tmp.set(i, list.get(i)[0]);

        map.put(list.get(i)[1].intValue(), i);

    }

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

(1) external sorted list = {{-2, 0}, {2, 2}, {3, 1}}

it represents as key value pair {presum, index of the presum}

the same thing as below presum array

For input nums = {-2, 5, -1} -> presum = {-2, 3, 2}

             presum  -2  3  2

index of the presum   0  1  2

then sort based on presum value

             presum  -2  2  3

index of the presum   0  2  1

(2) nums = {-2, 5, -1}

original list = {{-2,0}, {3,1}, {2,2}}

^ ^ ^

(track of original indicies on list[i][1])

sorted list = {{-2,0}, {2,2}, {3,1}}

original presum value(list[i][0]) -> original presum index(list[i][1]) => sorted presum index(in for loop assign auto matically as ith iteration)

-2 -> 0 => 0

3 -> 1 => 2

2 -> 2 => 1

^ ^

map key map value

(track of original indicies on list[i][1]) (after sort indicies in list array)

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

Build Segment Tree:

    private List<Long> list; // a copy of the input 'tmp' list

    // The node tree[treeIndex] over the range [left, right] represents the number of valid tmp[k]'s in this range,

    // for left <= k <= right. tree[treeIndex] is just (right - left + 1) at the beginning.

    private List<Integer> tree;

    private int length; // 4\*N

    private int N; // size of the input 'tmp' list

    public SegmentTree(List<Long> tmp) {

        N = tmp.size();

        length = 4 \* N;

        tree = new ArrayList<>(Collections.nCopies(length, 0));

        list = new ArrayList<>(Collections.nCopies(N, 0L));

        // The build of Segment Tree based on given input array 'tmp' = {-2,2,3} will start with

        // treeIndex = 1, range [left, right](=[0, N - 1]=>[0, 3 - 1]=>[0, 2])

        // e.g

        //                               tree[1]=3

        //                            idx range=[0,2]

        //                      presum range=[tmp[0],tmp[2]] -> presum range can be derived by idx range

        //                      /                          \

        //                tree[2]=2                        tree[3]=1

        //             idx range=[0,1]                  idx range=[2,2]

        //          presum range=[tmp[0],tmp[1]]   presum range=[tmp[2],tmp[2]]

        //              /                    \

        //         tree[4]=1                tree[5]=1

        //      idx range=[0,0]           idx range=[1,1]

        // presum range=[tmp[0],tmp[0]] presum range=[tmp[1],tmp[1]]

        //

        // tree[4] as the leaf node over the given 'tmp' list index range [0, 0] represents the number of valid

        // tmp[k]'s in this range, for index 0 <= k <= 0. tree[4] is just (0 - 0 + 1 = 1) at the beginning, which

        // means we have tmp[0] = -2 this 1 valid tmp[k]'s in 'tmp' list index range [0, 0], so tree[4] will be 1

        // initially

        // tree[5] as the leaf node over the given 'tmp' list index range [1, 1] represents the number of valid

        // tmp[k]'s in this range, for index 1 <= k <= 1. tree[5] is just (1 - 1 + 1 = 1) at the beginning, which

        // means we have tmp[1] = 2 this 1 valid tmp[k]'s in 'tmp' list index range [1, 1], so tree[5] will be 1

        // initially

        // tree[2] as non-leaf node over the given 'tmp' list index range [0, 1] represents the number of valid

        // tmp[k]'s in this range, for index 0 <= k <= 1. tree[2] is just (1 - 0 + 1 = 2) at the beginning, which

        // means we have tmp[0] = -2, tmp[1] = 2 these 2 valid tmp[k]'s in 'tmp' list index range [0, 1], so tree[2]

        // will be 2 initially

        // tree[3] as the leaf node over the given 'tmp' list index range [2, 2] represents the number of valid

        // tmp[k]'s in this range, for index 2 <= k <= 2. tree[3] is just (2 - 2 + 1 = 1) at the beginning, which

        // means we have tmp[2] = 3 this 1 valid tmp[k]'s in 'tmp' list index range [2, 2], so tree[3] will be 1

        // initially

        // tree[1] as the root node over the given 'tmp' list index range [0, 2] represents the number of valid

        // tmp[k]'s in this range, for index 0 <= k <= 2. tree[1] is just (2 - 0 + 1 = 3) at the beginning, which

        // means we have tmp[0] = -2, tmp[1] = 2, tmp[2] = 3 these 3 valid tmp[k]'s in 'tmp' list index range

        // [0, 2], so tree[1] will be 3 initially

        build(tmp, 1, 0, N - 1);

        for (int i = 0; i < N; i++) {

            list.set(i, tmp.get(i));

        }

    }

    private void build(List<Long> tmp, int treeIndex, int left, int right) {

        if (left == right) {

            tree.set(treeIndex, 1);

            return;

        }

        int mid = (left + right) / 2;

        build(tmp, 2 \* treeIndex, left, mid);

        build(tmp, 2 \* treeIndex + 1, mid + 1, right);

        tree.set(treeIndex, tree.get(2 \* treeIndex) + tree.get(2 \* treeIndex + 1));

    }

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

    N = 3

    build(tmp, 1, 0, N - 1);

    tree = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] -> initial with size 3 \* 4 = 12

    tree = [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0] -> tree.set(4) = 1, tree.set(5) = 1

    tree = [0, 0, 2, 0, 1, 1, 0, 0, 0, 0, 0, 0] -> tree.set(2, tree.get(4) + tree.get(5)) = 2

    tree = [0, 0, 2, 1, 1, 1, 0, 0, 0, 0, 0, 0] -> tree.set(3) = 1

    tree = [0, 3, 2, 1, 1, 1, 0, 0, 0, 0, 0, 0] -> tree.set(1, tree.get(2) + tree.get(3)) = 3

    list = [-2, 2, 3] -> sorted presum of given input {-2, 5, -1}

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

Query:

    // Return the number of valid leaf nodes of which values are between Lower and Upper, both ends inclusive.

    // If the node tree[treeIndex] represents the range [left, right] on the input array 'tmp',

    // then the maximum value of 'tmp' over [left, right] = tmp[right]

    // and the minimum value of 'tmp' over [left, right] = tmp[left]

    // because 'tmp' is sorted increasingly.

    public int query(int treeIndex, int left, int right, long Lower, long Upper) {

        if (Lower <= list.get(left) && list.get(right) <= Upper) {

            return tree.get(treeIndex);

        }

        if (list.get(right) < Lower || list.get(left) > Upper) {

            return 0;

        }

        int mid = (left + right) / 2;

        return query(2 \* treeIndex, left, mid, Lower, Upper) + query(2 \* treeIndex + 1, mid + 1, right, Lower, Upper);

    }

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

Query Round 1:

    // e.g

    // The node tree[treeIndex] represents the index range [left, right] on the input array 'tmp',

    // which means tree[1] represents the index range as [0, 2] on the input array 'tmp',

    // the maximum value of 'tmp' over [left, right] = tmp[right] = tmp[2] = 3

    // the minimum value of 'tmp' over [left, right] = tmp[left] = tmp[0] = -2

    // so we want to compare maximum value with Upper and minimum value with Lower, it will

    // fall into 3 cases:

    // (1) If the node tree[treeIndex] fully inside the given [Lower, Upper] range,

    // return count stored in tree[treeIndex]

    // (2) Reversely if fully outside the given [Lower, Upper] range, return 0

    // (3) If intersect only, recursively check on child range till range fall into (1) or (2)

    // We found tree[1] represented index range [0, 2] and its equally presum value range [tmp[0], tmp[2]]

    // as [-2, 3] not fall into (1) or (2) directly, because [-2, 3] intersects with [Lower, Upper]

    // = [-2, 2], it fall into (3) first, then recursively check on child range represented by tree[2]

    // and tree[3], tree[2] represents index range [0, 1] and its equally presum value range [tmp[0], tmp[1]]

    // as [-2, 2] fall into (1) directly, because [-2, 2] fully inside given [Lower, Upper] = [-2, 2],

    // return count stored in tree[treeIndex] = tree[2] = 2, tree[3] represents index range [2, 2] and its

    // equally presum value range [tmp[2], tmp[2]] as [3, 3] fall into (2) directly, because [3, 3] fully

    // outside the given [Lower, Upper] = [-2, 2], return 0, so together return 2 + 0 = 2 for query on tree[1]

    count = 0

    Lower = -2

    Upper = 2

    count += segmentTree.query(1, 0, n - 1, Lower, Upper);

 => count += segmentTree.query(1, 0, 3 - 1, -2, 2);

    left = 0, list.get(0) = -2

    right = 2, list.get(2) = 3

    if (Lower <= list.get(left) && list.get(right) <= Upper)

 => -2 <= -2 && 3 <= 2 -> false

    if (list.get(right) < Lower || list.get(left) > Upper)

 => 3 < -2 || -2 > 2 -> false

    mid = 1

    return query(2 \* treeIndex, left, mid, Lower, Upper) + query(2 \* treeIndex + 1, mid + 1, right, Lower, Upper);

 => return query(2 \* 1, 0, 1, -2, 2) + query(2 \* 1 + 1, 1 + 1, 2, -2, 2);

    query(2 \* 1, 0, 1, -2, 2)

    left = 0, list.get(0) = -2

    right = 1, list.get(1) = 2

    if (Lower <= list.get(left) && list.get(right) <= Upper)

 => -2 <= -2 && 2 <= 2 -> true

    return tree.get(treeIndex) = tree.get(2) = 2

    query(2 \* 1 + 1, 1 + 1, 2, -2, 2)

    left = 2, list.get(2) = 3

    right = 2, list.get(2) = 3

    if (Lower <= list.get(left) && list.get(right) <= Upper)

 => -2 <= 3 && 3 <= 2 -> false

    if (list.get(right) < Lower || list.get(left) > Upper)

 => 3 < -2 || 3 > 2 -> true

    return 0

    return query(2 \* 1, 0, 1, -2, 2) + query(2 \* 1 + 1, 1 + 1, 2, -2, 2);

 => return 2 + 0 = 2

    count += 2 = 2

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

Query Round 2:

    count = 2

    Lower = -2 + nums[0] = -4

    Upper = 2 + nums[0] = 0

    count += segmentTree.query(1, 0, n - 1, Lower, Upper);

 => count += segmentTree.query(1, 0, 3 - 1, -4, 0);

    left = 0, list.get(0) = -2

    right = 2, list.get(2) = 3

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

Update:

// Call update method as update(int treeIndex, int left, int right, int index, int val)

// and since here you want to update the value at index with value val, and since we want

// tmp[index] as a removed entry, the val will always be 0, the udpate method signature can

// save 'val' parameter since always 0, update(int treeIndex, int left, int right, int index)

    // Update tmp[index] as a removed entry

    public void update(int treeIndex, int left, int right, int index) {

        if (left == right) {

            tree.set(treeIndex, 0);

            return;

        }

        int mid = (left + right) / 2;

        if (index <= mid) {

            update(2 \* treeIndex, left, mid, index);

        } else {

            update(2 \* treeIndex + 1, mid + 1, right, index);

        }

        tree.set(treeIndex, tree.get(2 \* treeIndex) + tree.get(2 \* treeIndex + 1));

    }

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

Update Round 1:

// Before we analysis how Update Round 1 works, we go through what happened before it,

// since we already call query() method once.

// Remember initially the [Lower, Upper] = [-2, 2], and we want to find number of presum[j]

// that satisfy below inequality by query() method:

// Lower <= presum[j] - presum[i] <= Upper

// => Lower + presum[i] <= presum[j] <= Upper + presum[i]

//

// Important definition of presum

// presum[k] is the sum of elements index range [0, k - 1] on input array 'tmp', and

// presum[k] serves as a base now.

//

// Before we call update() method as first time, presum[i] = 0, represents in initial iteration

// where query() method calling before first time call on update() method, presum[i] is the

// presum of NO elements on input array 'tmp', we want to find how many presum[j] satisfy

// 'Lower + presum[i] <= presum[j] <= Upper + presum[i]'

// => 'Lower + 0 <= presum[j] <= Upper + 0' -> presum[i] = base serves as 0

// => '-2 <= presum[j] <= 2'

//

// To find numbers of presum[j] that satisfy '-2 <= presum[j] <= 2' in initial iteration, it

// comes to how query() method works with the tree array which build up as Segment Tree, refer

// to Query Round 1 explain above, the query() method with given search index range as

// [0, 'tmp' array length - 1](=[0, 2]) to go through the Segment Tree from leaf node ->

// non-leaf node -> root node, which attempts on all potential index(or presum value)

// range combinations(treat as presum[j]) to dig out how many of them satisfy the inequality,

// e.g from above Query Round 1 explain, it go through one non-leaf node as tree[2] and one

// leaf node as tree[3], tree[2] which represents index range as [0, 1], the presum value

// range [tmp[0], tmp[1]] = [-2, 2] fully inside given query range [Lower, Upper] = [-2, 2],

// we return count 2 stored at tree[2], in another way to understand, this non-leaf node as

// tree[2] is the count sum up from its two children tree nodes as leaf nodes tree[4] and tree[5],

// tree[4] which represents index range as [0, 1], the presum value range [tmp[0], tmp[0]] = [-2, -2]

// fully inside given query range [Lower, Upper] = [-2, 2], and tree[4] = 1, same for tree[5]

// which represents index range as [1, 1], the presum value range [tmp[1], tmp[1]] = [2, 2]

// fully inside given query range [Lower, Upper] = [-2, 2], and tree[5] = 1, tree[2](=2) =

// tree[4](=1) + tree[5](=1), so we have just return at non-leaf node tree[2] will cover both

// leaf nodes tree[4] and tree[5], on the other side, tree[3] stored 1 but no inside [Lower, Upper]

// = [-2, 2], so we cannot add this 1 into count, we elaborate how the initial iteraion Query

// Round 1 works before we call update() method as first time.

//

// We found a phenomena that to query out all possibilities, we will keep 'shifting' nums[i]

// (where i is the ith iteration) from a contributor of 'presum[j]' into a contributor of 'presum[i]'

// for inequality 'Lower + presum[i] <= presum[j] <= Upper + presum[i]'

// e.g

// Initially presum[j] = nums[0] + nums[1] + nums[2] + ... + nums[j - 1], presum[i] = 0

// 1st time query:

// => presum[j] = nums[0] + nums[1] + nums[2] + ... + nums[j - 1]

// => presum[i] = 0, nums[0] is the contributor for presum[j], no contributor for presum[i]

// 2nd time query:

// => presum[j] = nums[1] + nums[2] + ... + nums[j - 1]

// => presum[i] = nums[0], nums[0] shift from contributor for presum[j] into contributor for presum[i]

// 3rd time query:

// => presum[j] = nums[2] + ... + nums[j - 1]

// => presum[i] = nums[0] + nums[1], after nums[0], nums[1] also shift from contributor for presum[j]

//    into contributor for presum[i]

//

// According to in for loop we keep 'shifting' nums[i] into presum[i] from presum[j] as part of the base,

// we have to remove all presum[i]'s contribution from Segment Tree before query happen in next iteration

// But how ? Theoretically this is the most tricky part:

// The Segment Tree build up based on "sorted" presum array 'tmp', NOT directly based on original presum

// of 'nums[i]', we cannot directly remove 'tmp[i]'s count contribution in Segment Tree, instead

// after we 'shift' nums[i] from presum[j] into presum[i] as part of the base, to reflect the 'shift',

// we have to remove natural ordered tmp[i]'s corresponding sorted tmp[index]'s count contribution in

// Segment Tree before query happen in next iteration, to find the 'tmp[i]'s' corresponding 'tmp[index]'

// is the critical part, and we use a map to deal with.

// --------------------------------------------------------------------------------------

// In our example nums = {-2, 5, -1}, Lower = -2, Upper = 2, we get sorted presum array

// 'tmp' = {-2, 2, 3}, map = {{0, 0}, {1, 2}, {2, 1}}

// When for loop build as below:

//      for (int i = 0; i < n; i++) {

//          count += segmentTree.query(1, 0, n - 1, Lower, Upper);

//          Lower += nums[i];

//          Upper += nums[i];

//          segmentTree.update(1, 0, n - 1, map.get(i));

//      }

// --------------------------------------------------------------------------------------

// The tree initial status: tree = [0, 3, 2, 1, 1, 1, 0, 0, 0, 0, 0, 0]

//

//                               tree[1]=3

//                            idx range=[0,2]

//                      presum range=[tmp[0],tmp[2]] -> presum range can be derived by idx range

//                      /                          \

//                tree[2]=2                        tree[3]=1

//             idx range=[0,1]                  idx range=[2,2]

//          presum range=[tmp[0],tmp[1]]   presum range=[tmp[2],tmp[2]]

//              /                    \

//         tree[4]=1                tree[5]=1

//      idx range=[0,0]           idx range=[1,1]

// presum range=[tmp[0],tmp[0]] presum range=[tmp[1],tmp[1]]

// --------------------------------------------------------------------------------------

// i = 0

// 1st query:

// 'Lower + presum[i] <= presum[j] <= Upper + presum[i]', presum[i] = 0

// => '-2 + 0 <= presum[j] <= 2 + 0' => '-2 <= presum[j] <= 2'

// Lower = -2

// Upper = 2

// we query count of presum[j] between [-2, 2] on initial Segment Tree

//

// 1st update:

// 'Lower + presum[i] <= presum[j] <= Upper + presum[i]', presum[i] = nums[0]

// => '-2 + (-2) <= presum[j] <= 2 + (-2)' => '-4 <= presum[j] <= 0'

// Lower = -2 + nums[0] = -2 + (-2) = -4

// Upper = 2 + nums[0] = -2 + 2 = 0

// map.get(0) = 0 -> update(... tmp[i]'s natural/before sort index = 0)

// => For inequality 'Lower + presum[i] <= presum[j] <= Upper + presum[i]', initially it has

//    presum[i] = 0, presum[j] = nums[0] + nums[1] + ... + nums[j - 1], which is

//    'Lower + 0 <= nums[0] + nums[1] + ... + nums[j - 1] <= Upper + 0'

//    and the Segment Tree is build on presum[j]

//    Now we shift nums[0] from presum[j] into presum[i], it change to

//    presum[i] = nums[0], presum[j] = nums[1] + ... + nums[j - 1], which is

//    'Lower + nums[0] <= nums[1] + ... + nums[j - 1] <= Upper + nums[0]'

// => nums[0] serves as presum[i] as base, the tmp[i]'s natural index is 0(in tmp = {-2, 3, 2}, we

//    have -2 as natural index at 0), we can get its after sort index in sorted presum array also = 0(in

//    sorted tmp = {-2, 2, 3}, we have -2 as sorted index at 0), the tmp[i]'s sorted index = 0 is

//    what we intrested in, will remove tmp[sorted index] = tmp[0]'s all count contribution in

//    Segment Tree

// --------------------------------------------------------------------------------------

// The tree status after first update: tree = [0, 2, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0]

//

//                               tree[1]=3 --> 2

//                            idx range=[0,2]

//                      presum range=[tmp[0],tmp[2]] -> presum range can be derived by idx range

//                      /                          \

//                tree[2]=2 --> 1                  tree[3]=1

//             idx range=[0,1]                  idx range=[2,2]

//          presum range=[tmp[0],tmp[1]]   presum range=[tmp[2],tmp[2]]

//              /                    \

//         tree[4]=1 --> 0           tree[5]=1

//      idx range=[0,0]           idx range=[1,1]

// presum range=[tmp[0],tmp[0]] presum range=[tmp[1],tmp[1]]

// --------------------------------------------------------------------------------------

    Lower = -2 + nums[0] = -4

    Upper = 2 + nums[0] = 0

    segmentTree.update(1, 0, n - 1, map.get(i));

 => segmentTree.update(1, 0, 3 - 1, map.get(0)(= 0));

    left = 0, right = 2, mid = (left + right) / 2 = 1

    index = 0 <= mid = 1

    update(2 \* treeIndex, left, mid, index);

 => update(2 \* 1, 0, 1, 0);

    left = 0, right = 1, mid = (left + right) / 2 = 0

    index = 0 <= mid = 0

    update(2 \* treeIndex, left, mid, index);

 => update(2 \* 2, 0, 0, 0);

    left = right = 0

    tree.set(treeIndex, 0);

 => tree.set(4, 0);

    tree = [0, 3, 2, 1, 0, 1, 0, 0, 0, 0, 0, 0] -> update tree[4] from 1 to 0

    tree.set(treeIndex, tree.get(2 \* treeIndex) + tree.get(2 \* treeIndex + 1));

 => tree.set(2, tree.get(2 \* 2) + tree.get(2 \* 2 + 1));

    tree.set(2, tree.get(4) + tree.get(5));

    tree.set(2, 1);

    tree = [0, 3, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0] -> update tree[2] from 2 to 1

    tree.set(treeIndex, tree.get(2 \* treeIndex) + tree.get(2 \* treeIndex + 1));

 => tree.set(1, tree.get(2 \* 1) + tree.get(2 \* 1 + 1));

    tree.set(1, tree.get(2) + tree.get(3));

    tree.set(1, 2);

    tree = [0, 2, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0] -> update tree[1] from 3 to 2

Remember when we build the Segment Tree, tmp[0] contribute on tree[4] -> tree[2] -> tree[1]

now after query count contribute by tmp[0], before we query count countribute by tmp[1] and

tmp[2], we have to remove all contributes from tmp[0] from the Segment Tree, in Update

Round 1 we fully remove the tmp[0] contributes from the Segment Tree.

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

Update Round 2:

// i = 1

// 2nd query:

// We query count of presum[j] between [-4, 0] on Segment Tree that removing nums[0]

// count contribution

//

// 2nd update:

// Lower += (long)nums[1] -> -4 + 5 = 1

// Upper += (long)nums[1] -> 0 + 5 = 5

// map.get(1) = 2 -> update(... tmp[i]'s natural/before sort index = 1)

// => For inequality 'Lower + presum[i] <= presum[j] <= Upper + presum[i]', after 1st round update it has

//    presum[i] = nums[0], presum[j] = nums[1] + nums[2] + ... + nums[j - 1], which is

//    'Lower + nums[0] <= nums[1] + ... + nums[j - 1] <= Upper + nums[0]'

//    and the Segment Tree is build on presum[j]

//    Now we shift nums[1] from presum[j] into presum[i], it change to

//    presum[i] = nums[0] + nums[1], presum[j] = nums[2] + ... + nums[j - 1], which is

//    'Lower + nums[0] + nums[1] <= nums[2] + ... + nums[j - 1] <= Upper + nums[0] + nums[1]'

// => nums[0] + nums[1] serves as presum[i] as base, the tmp[i]'s natural index is 1(in tmp = {-2, 3, 2},

//    we have 3 as natural index at 1), we can get its after sort index in sorted tmp array = 2(in

//    sorted tmp = {-2, 2, 3}, we have 3 as sorted index at 2), the tmp[i]'s sorted index = 2 is

//    what we intrested in, will remove tmp[sorted index] = tmp[2]'s all count contribution in

//    Segment Tree

// --------------------------------------------------------------------------------------

// The tree status after second update: tree = [0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0]

//

//                               tree[1]=2 --> 1

//                            idx range=[0,2]

//                      presum range=[tmp[0],tmp[2]] -> presum range can be derived by idx range

//                      /                          \

//                tree[2]=1                        tree[3]=1 --> 0

//             idx range=[0,1]                  idx range=[2,2]

//          presum range=[tmp[0],tmp[1]]   presum range=[tmp[2],tmp[2]]

//              /                    \

//         tree[4]=0                 tree[5]=1

//      idx range=[0,0]           idx range=[1,1]

// presum range=[tmp[0],tmp[0]] presum range=[tmp[1],tmp[1]]

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

Update Round 3:

// i = 2

// 3rd query:

// We query count of presum[j] between [1, 5] on Segment Tree that removing both nums[0] and nums[1]

// count contribution

//

// 3rd update:

// Lower += (long)nums[2] -> 1 + (-1) = 0

// Upper += (long)nums[2] -> 5 + (-1) = 4

// map.get(2) = 1 -> update(... tmp[i]'s natural/before sort index = 2)

// => For inequality 'Lower + presum[i] <= presum[j] <= Upper + presum[i]', after 2nd round update it has

//    presum[i] = nums[0] + nums[1], presum[j] = nums[2] + ... + nums[j - 1], which is

//    'Lower + nums[0] + nums[1] <= nums[2] + ... + nums[j - 1] <= Upper + nums[0] + nums[1]'

//    and the Segment Tree is build on presum[j]

//    Now we shift nums[2] from presum[j] into presum[i], it change to

//    presum[i] = nums[0] + nums[1] + nums[2], presum[j] = nums[3] + ... + nums[j - 1], which is

//    'Lower + nums[0] + nums[1] + nums[2] <= nums[3] + ... + nums[j - 1] <= Upper + nums[0] + nums[1] + nums[2]'

// => nums[0] + nums[1] + nums[2] serves as presum[i] as base, the tmp[i]'s natural index is 2

//    (in tmp = {-2, 3, 2}, we have 2 as natural index at 2), we can get its after sort index in sorted

//    tmp array = 1(in sorted tmp = {-2, 2, 3}, we have 2 as sorted index at 1), the tmp[i]'s

//    sorted index = 1 is what we intrested in, will remove tmp[sorted index] = tmp[1]'s all count

//    contribution in Segment Tree

// --------------------------------------------------------------------------------------

// The tree status after third update: tree = [0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0]

//

//                               tree[1]=1 --> 0

//                            idx range=[0,2]

//                      presum range=[tmp[0],tmp[2]] -> presum range can be derived by idx range

//                      /                          \

//                tree[2]=1 --> 0                  tree[3]=1 --> 0

//             idx range=[0,1]                  idx range=[2,2]

//          presum range=[tmp[0],tmp[1]]   presum range=[tmp[2],tmp[2]]

//              /                    \

//         tree[4]=0                 tree[5]=1 --> 0

//      idx range=[0,0]           idx range=[1,1]

// presum range=[tmp[0],tmp[0]] presum range=[tmp[1],tmp[1]]

**Style 1-2: ArrayList implementation without Discretization and build / update / query Segment Tree start with treeIndex = 0, refer L307**

class Solution {

    public int countRangeSum(int[] nums, int lower, int upper) {

        int n = nums.length;

        //List<Pair<Long, Integer>> v = new ArrayList<>();

        List<Long[]> list = new ArrayList<>();

        // v.add(new Pair<>((long) nums[0], 0));

        list.add(new Long[]{(long)nums[0], (long)0});

        for (int i = 1; i < n; i++) {

            //v.add(new Pair<>(v.get(i - 1).getKey() + nums[i], i));

            list.add(new Long[]{list.get(i - 1)[0] + nums[i], (long)i});

        }

        //v.sort(Comparator.comparingLong(Pair::getKey));

        Collections.sort(list, (a, b) -> a[0].compareTo(b[0]));

        //List<Long> u = new ArrayList<>(Collections.nCopies(n, 0L));

        List<Long> tmp = new ArrayList<>(Collections.nCopies(n, 0L));

        //Map<Integer, Integer> uMap = new HashMap<>();

        Map<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < n; i++) {

            //u.set(i, v.get(i).getKey());

            tmp.set(i, list.get(i)[0]);

            //uMap.put(v.get(i).getValue(), i);

            map.put(list.get(i)[1].intValue(), i);

        }

        long Lower = lower;

        long Upper = upper;

        //SegmentTree sgTree = new SegmentTree(u);

        SegmentTree segmentTree = new SegmentTree(tmp);

        int count = 0;

        for (int i = 0; i < n; i++) {

            //count += sgTree.query(1, 0, n - 1, Lower, Upper);

            // Query method change to treeIndex start with 0 style

            //count += segmentTree.query(1, 0, n - 1, Lower, Upper);

            count += segmentTree.query(0, 0, n - 1, Lower, Upper);

            Lower += nums[i];

            Upper += nums[i];

            // Update method change to treeIndex start with 0 style

            //sgTree.update(1, 0, n - 1, uMap.get(i));

            segmentTree.update(0, 0, n - 1, map.get(i));

        }

        return count;

    }

}

// The Segment Tree build / update / query method start with treeIndex = 1,

// not treeIndex = 0, then child = 2 \* i & 2 \* i + 1, if start with

// treeIndex = 0, then child = 2 \* i + 1 & 2 \* i + 2, refer L307

class SegmentTree {

    private List<Long> list; // a copy of the input vector

    // The node tree[treeIndex] over the range [left, right] represents the number of valid tmp[k]'s in this range,

    // for left <= k <= right. tree[treeIndex] is just (right - left + 1) at the beginning.

    private List<Integer> tree;

    private int length; // 4\*N

    private int N; // size of the input vector

    public SegmentTree(List<Long> tmp) {

        N = tmp.size();

        length = 4 \* N;

        tree = new ArrayList<>(Collections.nCopies(length, 0));

        list = new ArrayList<>(Collections.nCopies(N, 0L));

        // Change treeIndex start with 0

        //build(tmp, 1, 0, N - 1);

        build(tmp, 0, 0, N - 1);

        for (int i = 0; i < N; i++) {

            list.set(i, tmp.get(i));

        }

    }

    private void build(List<Long> tmp, int treeIndex, int left, int right) {

        if (left == right) {

            tree.set(treeIndex, 1);

            return;

        }

        int mid = (left + right) / 2;

        // Corresponding to build tree start with treeIndex = 0, change child

        // node build with '2 \* treeIndex + 1' and '2 \* treeIndex + 2'

        //build(tmp, 2 \* treeIndex, left, mid);

        //build(tmp, 2 \* treeIndex + 1, mid + 1, right);

        //tree.set(treeIndex, tree.get(2 \* treeIndex) + tree.get(2 \* treeIndex + 1));

        build(tmp, 2 \* treeIndex + 1, left, mid);

        build(tmp, 2 \* treeIndex + 2, mid + 1, right);

        tree.set(treeIndex, tree.get(2 \* treeIndex + 1) + tree.get(2 \* treeIndex + 2));

    }

    // Update tmp[index] as a removed entry

    public void update(int treeIndex, int left, int right, int index) {

        if (left == right) {

            tree.set(treeIndex, 0);

            return;

        }

        int mid = (left + right) / 2;

        // Corresponding to update tree start with treeIndex = 0, change child

        // node update with '2 \* treeIndex + 1' and '2 \* treeIndex + 2'

        if (index <= mid) {

            //update(2 \* treeIndex, left, mid, index);

            update(2 \* treeIndex + 1, left, mid, index);

        } else {

            //update(2 \* treeIndex + 1, mid + 1, right, index);

            update(2 \* treeIndex + 2, mid + 1, right, index);

        }

        //tree.set(treeIndex, tree.get(2 \* treeIndex) + tree.get(2 \* treeIndex + 1));

        tree.set(treeIndex, tree.get(2 \* treeIndex + 1) + tree.get(2 \* treeIndex + 2));

    }

    // Return the number of valid leaf nodes of which values are between Lower and Upper, both ends inclusive.

    // If the node tree[treeIndex] represents the range [left, right] on the input array 'tmp',

    // then the maximum value of 'tmp' over [left, right] = tmp[right]

    // and the minimum value of 'tmp' over [left, right] = tmp[left]

    // because 'tmp' is sorted increasingly.

    public int query(int treeIndex, int left, int right, long Lower, long Upper) {

        if (Lower <= list.get(left) && list.get(right) <= Upper) {

            return tree.get(treeIndex);

        }

        if (list.get(right) < Lower || list.get(left) > Upper) {

            return 0;

        }

        int mid = (left + right) / 2;

        // Corresponding to query tree start with treeIndex = 0, change child

        // node query with '2 \* treeIndex + 1' and '2 \* treeIndex + 2'

        return query(2 \* treeIndex + 1, left, mid, Lower, Upper) + query(2 \* treeIndex + 2, mid + 1, right, Lower, Upper);

    }

}

**Refer to**

<https://leetcode.com/problems/count-of-range-sum/solutions/2208532/c-simple-solution-by-sorting-and-segment-tree/>

struct segmentTree{

std::vector<long long int> u; //a copy of the input vector

//the node tree[x] over the range [left, right] represents the number of valid v[k]'s in this range,

// for left<=k<=right. tree[x] is just (right-left+1) at the beginning.

std::vector<int> tree;

int length; //4\*N

int N; //size of input vector

segmentTree(std::vector<long long int>& v){

N = v.size();

length = 4\*N;

tree.resize(length, 0);

u.resize(N, 0);

build(v, 1, 0, N-1);

for (int i=0; i<N; i++) u[i] = v[i];

}

void build(std::vector<long long int>& v, int x, int left, int right){

if (left==right){

tree[x] = 1;

return;

}

int mid = (left+right)/2;

build(v, 2\*x, left, mid);

build(v, 2\*x+1, mid+1, right);

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

return;

}

//update v[index] as a removed entry

void update(int x, int left, int right, int index){

if (left==right){

tree[x] = 0;

return;

}

int mid = (left+right)/2;

if (index<=mid){

update(2\*x, left, mid, index);

}else{

update(2\*x+1, mid+1, right, index);

}

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

return;

}

//return the number of valid leaf nodes of which values are between Lower and Upper, both ends inclusive.

//If the node tree[x] represents the range [left, right] on the input array v,

// then maximum value of v over [left, right] = v[right]

// and minimum value of v over [left, right] = v[left]

// because v is sorted increasingly.

int query(int x, int left, int right, long long int Lower, long long int Upper){

if (Lower<=u[left] && u[right]<=Upper){

return tree[x];

}

if (u[right]<Lower || u[left]>Upper){

return 0;

}

int mid = (left+right)/2;

return query(2\*x, left, mid, Lower, Upper) + query(2\*x+1, mid+1, right, Lower, Upper);

}

};

/\*exmple: nums={1,2,3,4,5,6}, lower=10, upper=15 (10<=y<=15)

(i=0) we find the prefix sum of nums S0 = {1(i=0), 3, 6, 10, 15, 21}.

we sort S0 ascendingly if necessary while keeping track of original indicies of numbers in S0.

create a simple segment tree of 6 leaf nodes, counting number of leaf nodes in a range:

(6)

/ \

(3) (3)

/ \ / \

(2) (1) (2) (1)

/ \ / \

(1) (1) (1) (1)

check: there are two numbers of S0 satisfying 10<=y<=15. They are 10, 15 => count = 2

The first node of S0 (i=0) is the first node after sorting, so we remove the first leaf node:

(5)

/ \

(2) (3)

/ \ / \

(1) (1) (2) (1)

/ \ / \

(0) (1) (1) (1)

(i=1) now, S0 = {1(removed), 3(i=1), 6, 10, 15}

for any number y=nums[0]+nums[1]+...+nums[k] in S0

10<=nums[1]+...+nums[k]<=15 <=> 10+nums[0]<=y<=15+nums[0]

except when y has been removed. Therefore, we repeat the step at (i=0) after putting

lower = lower + nums[0]

upper = upper + nums[0]

...

\*/

class Solution {

public:

int countRangeSum(vector<int>& nums, int lower, int upper) {

int n = nums.size();

std::vector<std::pair<long long int, int>> v;

v.push\_back(std::make\_pair(nums[0], 0));

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

v.push\_back(std::make\_pair(v.back().first+nums[i], i));

}

std::sort(v.begin(), v.end());

std::vector<long long int> u(n, 0);

std::unordered\_map<int, int> uMap;

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

u[i] = v[i].first;

uMap[v[i].second] = i;

}

long long int Lower = lower;

long long int Upper = upper;

segmentTree sgTree = segmentTree(u);

int count = 0;

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

count += sgTree.query(1, 0, n-1, Lower, Upper);

Lower += nums[i];

Upper += nums[i];

sgTree.update(1, 0, n-1, uMap[i]);

}

return count;

}

};

**Java version (convert by chatGPT)**

import java.util.\*;

class SegmentTree {

    private List<Long> u; // a copy of the input vector

    // the node tree[x] over the range [left, right] represents the number of valid v[k]'s in this range,

    // for left<=k<=right. tree[x] is just (right-left+1) at the beginning.

    private List<Integer> tree;

    private int length; // 4\*N

    private int N; // size of the input vector

    public SegmentTree(List<Long> v) {

        N = v.size();

        length = 4 \* N;

        tree = new ArrayList<>(Collections.nCopies(length, 0));

        u = new ArrayList<>(Collections.nCopies(N, 0L));

        build(v, 1, 0, N - 1);

        for (int i = 0; i < N; i++) {

            u.set(i, v.get(i));

        }

    }

    private void build(List<Long> v, int x, int left, int right) {

        if (left == right) {

            tree.set(x, 1);

            return;

        }

        int mid = (left + right) / 2;

        build(v, 2 \* x, left, mid);

        build(v, 2 \* x + 1, mid + 1, right);

        tree.set(x, tree.get(2 \* x) + tree.get(2 \* x + 1));

    }

    // update v[index] as a removed entry

    public void update(int x, int left, int right, int index) {

        if (left == right) {

            tree.set(x, 0);

            return;

        }

        int mid = (left + right) / 2;

        if (index <= mid) {

            update(2 \* x, left, mid, index);

        } else {

            update(2 \* x + 1, mid + 1, right, index);

        }

        tree.set(x, tree.get(2 \* x) + tree.get(2 \* x + 1));

    }

    // return the number of valid leaf nodes of which values are between Lower and Upper, both ends inclusive.

    // If the node tree[x] represents the range [left, right] on the input array v,

    // then the maximum value of v over [left, right] = v[right]

    // and the minimum value of v over [left, right] = v[left]

    // because v is sorted increasingly.

    public int query(int x, int left, int right, long Lower, long Upper) {

        if (Lower <= u.get(left) && u.get(right) <= Upper) {

            return tree.get(x);

        }

        if (u.get(right) < Lower || u.get(left) > Upper) {

            return 0;

        }

        int mid = (left + right) / 2;

        return query(2 \* x, left, mid, Lower, Upper) + query(2 \* x + 1, mid + 1, right, Lower, Upper);

    }

}

public class Solution {

    public int countRangeSum(int[] nums, int lower, int upper) {

        int n = nums.length;

        List<Pair<Long, Integer>> v = new ArrayList<>();

        v.add(new Pair<>((long) nums[0], 0));

        for (int i = 1; i < n; i++) {

            v.add(new Pair<>(v.get(i - 1).getKey() + nums[i], i));

        }

        v.sort(Comparator.comparingLong(Pair::getKey));

        List<Long> u = new ArrayList<>(Collections.nCopies(n, 0L));

        Map<Integer, Integer> uMap = new HashMap<>();

        for (int i = 0; i < n; i++) {

            u.set(i, v.get(i).getKey());

            uMap.put(v.get(i).getValue(), i);

        }

        long Lower = lower;

        long Upper = upper;

        SegmentTree sgTree = new SegmentTree(u);

        int count = 0;

        for (int i = 0; i < n; i++) {

            count += sgTree.query(1, 0, n - 1, Lower, Upper);

            Lower += nums[i];

            Upper += nums[i];

            sgTree.update(1, 0, n - 1, uMap.get(i));

        }

        return count;

    }

}

**Style 2: Array Segment Tree**

**Style 2-1: Array implementation without Discretization and build / update / query Segment Tree start with treeIndex = 1**

class Solution {

    public int countRangeSum(int[] nums, int lower, int upper) {

        int n = nums.length;

        Long[][] presum = new Long[n][2];

        // presum element -> {presum value, presum index before sort}

        presum[0] = new Long[]{(long)nums[0], (long)0};

        for(int i = 1; i < n; i++) {

            presum[i] = new Long[]{presum[i - 1][0] + (long)nums[i], (long)i};

        }

        Arrays.sort(presum, (a, b) -> a[0].compareTo(b[0]));

        // map -> {key = original indices on presum, value = after sort indices on presum}

        Map<Integer, Integer> map = new HashMap<>();

        for(int i = 0; i < n; i++) {

            map.put(presum[i][1].intValue(), i);

        }

        int[] segmentTree = buildSegmentTree(presum);

        int result = 0;

        long Lower = (long)lower;

        long Upper = (long)upper;

        for(int i = 0; i < n; i++) {

            result += querySegmentTree(segmentTree, presum, 0, n - 1, Lower, Upper);

            Lower += (long)nums[i];

            Upper += (long)nums[i];

            // "map.get(i)" -> Use presum[i]'s natural index get its sorted index

            updateSegmentTree(segmentTree, 0, n - 1, map.get(i));

        }

        return result;

    }

    private int[] buildSegmentTree(Long[][] presum) {

        int n = presum.length;

        int[] tree = new int[n \* 4];

        build(tree, 1, 0, n - 1);

        return tree;

    }

    // The node tree[treeIndex] over the range [left, right] represents the number of

    // valid presum[k]'s in this range, for left <= k <= right. tree[treeIndex] is

    // just (right - left + 1) at the beginning

    private void build(int[] tree, int treeIndex, int lo, int hi) {

        if(lo == hi) {

            tree[treeIndex] = 1;

            return;

        }

        int mid = lo + (hi - lo) / 2;

        build(tree, 2 \* treeIndex, lo, mid);

        build(tree, 2 \* treeIndex + 1, mid + 1, hi);

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

    }

    // Query range start with [0, presum.length(=n) - 1] to cover whole range of sorted input

    // array 'presum', its not [0, tree.length(=4\*n) - 1] because we only deal with original

    // input 'presum', the 'tree' is only a Segment Tree structure we build to accelerate

    // the query speed, and node as tree[treeIndex] stores count only

    private int querySegmentTree(int[] tree, Long[][] presum, int left, int right, long Lower, long Upper) {

        return query(tree, 1, presum, left, right, Lower, Upper);

    }

    private int query(int[] tree, int treeIndex, Long[][] presum, int left, int right, long Lower, long Upper) {

        // tree node fully inside query range

        if(Lower <= presum[left][0] && presum[right][0] <= Upper) {

            return tree[treeIndex];

        }

        // tree node fully outside query range

        if(presum[left][0] > Upper || presum[right][0] < Lower) {

            return 0;

        }

        int mid = left + (right - left) / 2;

        return query(tree, 2 \* treeIndex, presum, left, mid, Lower, Upper) + query(tree, 2 \* treeIndex + 1, presum, mid + 1, right, Lower, Upper);

    }

    private void updateSegmentTree(int[] tree, int left, int right, int index) {

        update(tree, 1, left, right, index);

    }

// Refer to below detail explain on how update method works and why we need a map

// to track presum index before and after sort

    private void update(int[] tree, int treeIndex, int left, int right, int index) {

        if(left == right) {

            tree[treeIndex] = 0;

            return;

        }

        int mid = left + (right - left) / 2;

        if(index <= mid) {

            update(tree, 2 \* treeIndex, left, mid, index);

        } else {

            update(tree, 2 \* treeIndex + 1, mid + 1, right, index);

        }

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

    }

}

**Step by step to explain the update method and why we need a map to track presum index before and after sort**

nums = {-2, 5, -1}

original presum = {{-2,0}, {3,1}, {2,2}}

^ ^ ^

(track of original indicies on presum[i][1])

sorted presum = {{-2,0}, {2,2}, {3,1}}

original presum value(presum[i][0]) -> original presum index(presum[i][1]) => sorted presum index(in for loop assign auto matically as ith iteration)

-2 -> 0 => 0

3 -> 1 => 2

2 -> 2 => 1

^ ^

map key map value

(track of original indicies on presum[i][1]) (after sort indicies in presum array)

Why we need mapping as {key = original indices on presum, value = after sort indices on presum} ?

Because based on inequality 'Lower + presum[i] <= presum[j] <= Upper + presum[i]', in below for loop, we keep

shifting nums[i] from presum[j] to presum[i], the presum[i] serves as base will looply growing up, such as

0 -> nums[0] -> nums[0] + nums[1] -> nums[0] + nums[1] + nums[2] -> ..., reversely, the presum[j] will looply

shrinking, such as nums[0] + nums[1] + nums[2] + ... + nums[j - 1] -> nums[1] + nums[2] + ... + nums[j - 1]

-> nums[2] + ... + nums[j - 1] -> ..., and the Segment Tree is built upon initial "sorted" presum[j], each time

it shrinks in for loop, we have to remove the count contribution of presum[i] on Segment Tree in that iteration,

but there is a conflict happens, in each iteration when we shift nums[i] from presum[j] to presum[i], the presum[i]

growing up as natural order, which means not get sorted yet, but the Segment Tree is built upon initial "sorted"

presum[j], so to get the correct presum[i]'s count contribution removed from Segment Tree, we have to get the

correct mapping relation between current presum[i]'s natural index and sorted index in presum array, and look

at below for loop, we will see the presum[i]'s natural index pre-stored in 'presum[i][1]' as map's key, the

presum[i]'s sorted index pre-stored as 'ith' iteration time itself, in each time calling update method, we will

use presum[i]'s natural index to get its sorted index, and use that sorted index to find correct presum[sorted index]

to get its count contribution removed from Segment Tree which build on "sorted" presum.

Long[][] presum = new Long[n][2];

// presum element -> {presum value, presum index before sort}

presum[0] = new Long[]{(long)nums[0], (long)0};

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

presum[i] = new Long[]{presum[i - 1][0] + (long)nums[i], (long)i};

}

Arrays.sort(presum, (a, b) -> a[0].compareTo(b[0]));

Map<Integer, Integer> map = new HashMap<>();

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

map.put(presum[i][1].intValue(), i);

}

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

result += querySegmentTree(segmentTree, presum, 0, n - 1, Lower, Upper);

Lower += (long)nums[i];

Upper += (long)nums[i];

updateSegmentTree(segmentTree, 0, n - 1, map.get(i));

}

In for loop:

initial status

Lower = -2

Upper = 2

// The tree initial status: tree = [0, 3, 2, 1, 1, 1, 0, 0, 0, 0, 0, 0]

//

// tree[1]=3

// idx range=[0,2]

// presum range=[presum[0],presum[2]] -> presum range can be derived by idx range

// / \

// tree[2]=2 tree[3]=1

// idx range=[0,1] idx range=[2,2]

// presum range=[presum[0],presum[1]] presum range=[presum[2],presum[2]]

// / \

// tree[4]=1 tree[5]=1

// idx range=[0,0] idx range=[1,1]

// presum range=[presum[0],presum[0]] presum range=[presum[1],presum[1]]

1st round update:

i = 0

Lower += (long)nums[0] -> -2 + (-2) = -4

Upper += (long)nums[0] -> 2 + (-2) = 0

map.get(0) = 0 -> update(... presum[i]'s natural/before sort index = 0)

=> For inequality 'Lower + presum[i] <= presum[j] <= Upper + presum[i]', initially it has

presum[i] = 0, presum[j] = nums[0] + nums[1] + ... + nums[j - 1], which is

'Lower + 0 <= nums[0] + nums[1] + ... + nums[j - 1] <= Upper + 0'

and the Segment Tree is build on presum[j]

Now we shift nums[0] from presum[j] into presum[i], it change to

presum[i] = nums[0], presum[j] = nums[1] + ... + nums[j - 1], which is

'Lower + nums[0] <= nums[1] + ... + nums[j - 1] <= Upper + nums[0]'

=> nums[0] serves as presum[i] as base, the presum[i]'s natural index is 0(in presum = {-2, 3, 2}, we

have -2 as natural index at 0), we can get its after sort index in sorted presum array also = 0(in

sorted presum = {-2, 2, 3}, we have -2 as sorted index at 0), the presum[i]'s sorted index = 0 is

what we intrested in, will remove presum[sorted index] = presum[0]'s all count contribution in

Segment Tree

// presum[0] map to tree[4] as leaf node

// so we update from bottom to top: leaf node tree[4] = 1 - 1 = 0 -> non-leaf node tree[2] =

// 2 - 1 = 1 -> root node tree[1] = 3 - 1 = 2

// --------------------------------------------------------------------------------------

// The tree status after first update: tree = [0, 2, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0]

//

// tree[1]=3 --> 2

// idx range=[0,2]

// presum range=[presum[0],presum[2]] -> presum range can be derived by idx range

// / \

// tree[2]=2 --> 1 tree[3]=1

// idx range=[0,1] idx range=[2,2]

// presum range=[presum[0],presum[1]] presum range=[presum[2],presum[2]]

// / \

// tree[4]=1 --> 0 tree[5]=1

// idx range=[0,0] idx range=[1,1]

// presum range=[presum[0],presum[0]] presum range=[presum[1],presum[1]]

// --------------------------------------------------------------------------------------

Lower = -2 + nums[0] = -4

Upper = 2 + nums[0] = 0

segmentTree.update(1, 0, n - 1, map.get(i));

=> segmentTree.update(1, 0, 3 - 1, map.get(0)(= 0));

left = 0, right = 2, mid = (left + right) / 2 = 1

index = 0 <= mid = 1

update(2 \* treeIndex, left, mid, index);

=> update(2 \* 1, 0, 1, 0);

left = 0, right = 1, mid = (left + right) / 2 = 0

index = 0 <= mid = 0

update(2 \* treeIndex, left, mid, index);

=> update(2 \* 2, 0, 0, 0);

left = right = 0

tree.set(treeIndex, 0);

=> tree.set(4, 0);

tree = [0, 3, 2, 1, 0, 1, 0, 0, 0, 0, 0, 0] -> update tree[4] from 1 to 0

tree.set(treeIndex, tree.get(2 \* treeIndex) + tree.get(2 \* treeIndex + 1));

=> tree.set(2, tree.get(2 \* 2) + tree.get(2 \* 2 + 1));

tree.set(2, tree.get(4) + tree.get(5));

tree.set(2, 1);

tree = [0, 3, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0] -> update tree[2] from 2 to 1

tree.set(treeIndex, tree.get(2 \* treeIndex) + tree.get(2 \* treeIndex + 1));

=> tree.set(1, tree.get(2 \* 1) + tree.get(2 \* 1 + 1));

tree.set(1, tree.get(2) + tree.get(3));

tree.set(1, 2);

tree = [0, 2, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0] -> update tree[1] from 3 to 2

Remember when we build the Segment Tree, presum[0] contribute on tree[4] -> tree[2] -> tree[1]

now after query count contribute by presum[0], before we query count countribute by presum[1] and

presum[2], we have to remove all contributes from presum[0] from the Segment Tree, in Update

Round 1 we fully remove the presum[0] contributes from the Segment Tree.

2nd round update:

i = 1

Lower += (long)nums[1] -> -4 + 5 = 1

Upper += (long)nums[1] -> 0 + 5 = 5

map.get(1) = 2 -> update(... presum[i]'s natural/before sort index = 1)

=> For inequality 'Lower + presum[i] <= presum[j] <= Upper + presum[i]', after 1st round update it has

presum[i] = nums[0], presum[j] = nums[1] + nums[2] + ... + nums[j - 1], which is

'Lower + nums[0] <= nums[1] + ... + nums[j - 1] <= Upper + nums[0]'

and the Segment Tree is build on presum[j]

Now we shift nums[1] from presum[j] into presum[i], it change to

presum[i] = nums[0] + nums[1], presum[j] = nums[2] + ... + nums[j - 1], which is

'Lower + nums[0] + nums[1] <= nums[2] + ... + nums[j - 1] <= Upper + nums[0] + nums[1]'

=> nums[0] + nums[1] serves as presum[i] as base, the presum[i]'s natural index is 1(in presum = {-2, 3, 2},

we have 3 as natural index at 1), we can get its after sort index in sorted presum array = 2(in

sorted presum = {-2, 2, 3}, we have 3 as sorted index at 2), the presum[i]'s sorted index = 2 is

what we intrested in, will remove presum[sorted index] = presum[2]'s all count contribution in

Segment Tree

// presum[2] map to tree[3] as leaf node so we update from bottom to top: leaf node tree[3] = 1 - 1

// = 0 -> root node tree[1] = 2 - 1 = 1

// The tree status after second update: tree = [0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0]

//

// tree[1]=2 --> 1

// idx range=[0,2]

// presum range=[presum[0],presum[2]] -> presum range can be derived by idx range

// / \

// tree[2]=1 tree[3]=1 --> 0

// idx range=[0,1] idx range=[2,2]

// presum range=[presum[0],presum[1]] presum range=[presum[2],presum[2]]

// / \

// tree[4]=0 tree[5]=1

// idx range=[0,0] idx range=[1,1]

// presum range=[presum[0],presum[0]] presum range=[presum[1],presum[1]]

// --------------------------------------------------------------------------------------

3rd round update:

i = 2

Lower += (long)nums[2] -> 1 + (-1) = 0

Upper += (long)nums[2] -> 5 + (-1) = 4

map.get(2) = 1 -> update(... presum[i]'s natural/before sort index = 2)

=> For inequality 'Lower + presum[i] <= presum[j] <= Upper + presum[i]', after 2nd round update it has

presum[i] = nums[0] + nums[1], presum[j] = nums[2] + ... + nums[j - 1], which is

'Lower + nums[0] + nums[1] <= nums[2] + ... + nums[j - 1] <= Upper + nums[0] + nums[1]'

and the Segment Tree is build on presum[j]

Now we shift nums[2] from presum[j] into presum[i], it change to

presum[i] = nums[0] + nums[1] + nums[2], presum[j] = nums[3] + ... + nums[j - 1], which is

'Lower + nums[0] + nums[1] + nums[2] <= nums[3] + ... + nums[j - 1] <= Upper + nums[0] + nums[1] + nums[2]'

=> nums[0] + nums[1] + nums[2] serves as presum[i] as base, the presum[i]'s natural index is 2

(in presum = {-2, 3, 2}, we have 2 as natural index at 2), we can get its after sort index in sorted

presum array = 1(in sorted presum = {-2, 2, 3}, we have 2 as sorted index at 1), the presum[i]'s

sorted index = 1 is what we intrested in, will remove presum[sorted index] = presum[1]'s all count

contribution in Segment Tree

// presum[1] map to tree[5] as leaf node so we update from bottom to top: leaf node tree[5] = 1 - 1 = 0 -> non-leaf node

// tree[2] = 1 - 1 = 0 -> root node tree[1] = 1 - 1 = 0

// The tree status after third update: tree = [0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0]

//

// tree[1]=1 --> 0

// idx range=[0,2]

// presum range=[presum[0],presum[2]] -> presum range can be derived by idx range

// / \

// tree[2]=1 --> 0 tree[3]=0

// idx range=[0,1] idx range=[2,2]

// presum range=[presum[0],presum[1]] presum range=[presum[2],presum[2]]

// / \

// tree[4]=0 tree[5]=1 --> 0

// idx range=[0,0] idx range=[1,1]

// presum range=[presum[0],presum[0]] presum range=[presum[1],presum[1]]

// --------------------------------------------------------------------------------------

**Style 2-2: Array implementation without Discretization and build / update / query Segment Tree start with treeIndex = 0, refer L307**

class Solution {

    public int countRangeSum(int[] nums, int lower, int upper) {

        int n = nums.length;

        Long[][] presum = new Long[n][2];

        // presum element -> {presum value, presum index before sort}

        presum[0] = new Long[]{(long)nums[0], (long)0};

        for(int i = 1; i < n; i++) {

            presum[i] = new Long[]{presum[i - 1][0] + (long)nums[i], (long)i};

        }

        Arrays.sort(presum, (a, b) -> a[0].compareTo(b[0]));

        // map -> {key = original indices on presum, value = after sort indices on presum}

        Map<Integer, Integer> map = new HashMap<>();

        for(int i = 0; i < n; i++) {

            map.put(presum[i][1].intValue(), i);

        }

        int[] segmentTree = buildSegmentTree(presum);

        int result = 0;

        long Lower = (long)lower;

        long Upper = (long)upper;

        for(int i = 0; i < n; i++) {

            result += querySegmentTree(segmentTree, presum, 0, n - 1, Lower, Upper);

            Lower += (long)nums[i];

            Upper += (long)nums[i];

            // "map.get(i)" -> Use presum[i]'s natural index get its sorted index

            updateSegmentTree(segmentTree, 0, n - 1, map.get(i));

        }

        return result;

    }

    private int[] buildSegmentTree(Long[][] presum) {

        int n = presum.length;

        int[] tree = new int[n \* 4];

        build(tree, 0, 0, n - 1);

        return tree;

    }

    // The node tree[treeIndex] over the range [left, right] represents the number of

    // valid presum[k]'s in this range, for left <= k <= right. tree[treeIndex] is

    // just (right - left + 1) at the beginning

    private void build(int[] tree, int treeIndex, int lo, int hi) {

        if(lo == hi) {

            tree[treeIndex] = 1;

            return;

        }

        int mid = lo + (hi - lo) / 2;

        build(tree, 2 \* treeIndex + 1, lo, mid);

        build(tree, 2 \* treeIndex + 2, mid + 1, hi);

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

    }

    // Query range start with [0, presum.length(=n) - 1] to cover whole range of sorted input

    // array 'presum', its not [0, tree.length(=4\*n) - 1] because we only deal with original

    // input 'presum', the 'tree' is only a Segment Tree structure we build to accelerate

    // the query speed, and node as tree[treeIndex] stores count only

    private int querySegmentTree(int[] tree, Long[][] presum, int left, int right, long Lower, long Upper) {

        return query(tree, 0, presum, left, right, Lower, Upper);

    }

    private int query(int[] tree, int treeIndex, Long[][] presum, int left, int right, long Lower, long Upper) {

        // tree node fully inside query range

        if(Lower <= presum[left][0] && presum[right][0] <= Upper) {

            return tree[treeIndex];

        }

        // tree node fully outside query range

        if(presum[left][0] > Upper || presum[right][0] < Lower) {

            return 0;

        }

        int mid = left + (right - left) / 2;

        return query(tree, 2 \* treeIndex + 1, presum, left, mid, Lower, Upper) + query(tree, 2 \* treeIndex + 2, presum, mid + 1, right, Lower, Upper);

    }

    private void updateSegmentTree(int[] tree, int left, int right, int index) {

        update(tree, 0, left, right, index);

    }

    private void update(int[] tree, int treeIndex, int left, int right, int index) {

        if(left == right) {

            tree[treeIndex] = 0;

            return;

        }

        int mid = left + (right - left) / 2;

        if(index <= mid) {

            update(tree, 2 \* treeIndex + 1, left, mid, index);

        } else {

            update(tree, 2 \* treeIndex + 2, mid + 1, right, index);

        }

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

    }

}

**Style 3: TreeNode implementation**

**Style 3-1: Segment Tree update method include both build Segment Tree and update Segment Tree functions (TLE 66/67)**

**Note: The query() method below strictly follow compare by absolute value 'mid' (e.g long mid = arr[lo] + (arr[hi] - arr[lo]) / 2), not compare by index 'mid' (e.g int mid = lo + (hi - lo) / 2)**

class Solution {

    public int countRangeSum(int[] nums, int lower, int upper) {

        int n = nums.length;

        // Find presum based on 'nums', remove all duplicates and sort by TreeSet

        // Why we have to sort ?

        // Because when we query(root, min, max) we compare each TreeNode.min

        // with min and TreeNode.max with max, when build the Segment Tree,

        // we have to maintain each TreeNode's interval from min to max, cannot

        // be scrambled, so the basement array used to build Segment Tree must

        // be sorted first

        TreeSet<Long> set = new TreeSet<>();

        long presum = 0;

        for(int i = 0; i < n; i++) {

            presum += nums[i];

            set.add(presum);

        }

        // Convert TreeSet to array for 'get' method O(1) time complexity

        // access at any index

        Long[] list = set.toArray(new Long[0]);

        int result = 0;

// Build Segment Tree based on absolute value (e.g 'mid' will be

// 'arr[lo] + (arr[hi] - arr[lo]) / 2')

        SegmentTree segmentTree = new SegmentTree(list[0], list[list.length - 1]);

        for(int i = nums.length - 1; i >= 0; i--) {

            segmentTree.update(presum);

            presum -= (long)nums[i];

            // lower <= sums[i] - sums[j] <= upper (sums[k] is the sum of nums[0...k]),

            // which is lower + sums[j] <= sums[i] <= upper + sums[j]

            // so for each sum[j]('presum'), we want to find out the # of

            // all sums[i] which satisfy the condition ----- give a range,

            // and find the # of values in the range, which is what segment

            // tree does

            result += segmentTree.query((long)lower + presum, (long)upper + presum);

        }

        return result;

    }

}

class SegmentTree {

    TreeNode root;

    public SegmentTree(long min, long max) {

        this.root = new TreeNode(min, max);

    }

    public void update(long presum) {

        update(root, presum);

    }

    // The 'udpate' method coding format here merge both

    // 'build Segment Tree' and 'update Segment Tree'

    // into one method, similar to L715, L307

    private void update(TreeNode root, long presum) {

        // Base condition for 'build Segment Tree'

        // When lower boundary larger than upper boundary,

        // means no further traversal, already pass leaf

        // node, no 'count' need to add on any node now

        // directly return

        if(root.min > root.max) {

            return;

        }

        // Base condition for 'update Segment Tree'

        if(root.min > presum || root.max < presum) {

            return;

        }

        // The if condition can be removed since above if

        // condition 'root.min > presum || root.max < presum'

        // check equal to below if condition, below if

        // condition just for keep similar coding format

        // convention than L715, L307

        //if(root.min <= presum && root.max >= presum) {

            root.count++;

            // No return here since we have to update

            // count on children nodes)

            //return;

        //}

        // Base condition for 'build Segment Tree'

        // If no below condition will throw stackover flow

        // exception since no close condition for tree build

        // Test out by:

        // Input: nums = [-2,5,-1], lower = -2, upper = 2

        // Also we can merge 'root.min > root.max' to

        // 'root.min == root.max', but we have to make sure

        // 'return' because of 'root.min == root.max'

        // must happen after 'root.count++'

        // Test out by:

        // Input: nums = [0], lower = 0, upper = 0

        // Output = 0, Expected = 1

        if(root.min == root.max) {

            return;

        }

        long mid = root.min + (root.max - root.min) / 2;

        if(root.left == null) {

            root.left = new TreeNode(root.min, mid);

        }

        if(root.right == null) {

            root.right = new TreeNode(mid + 1, root.max);

        }

        if(presum <= mid) {

            update(root.left, presum);

        } else if(presum > mid) {

            update(root.right, presum);

        }

    }

    public int query(long min, long max) {

        return query(root, min, max);

    }

// The query() method below strictly follow compare by absolute

// value 'mid' (e.g long mid = arr[lo] + (arr[hi] - arr[lo]) / 2),

// not compare by index 'mid' (e.g int mid = lo + (hi - lo) / 2)

    private int query(TreeNode root, long min, long max) {

        if(root == null) {

            return 0;

        }

        if(root.max < min || root.min > max) {

            return 0;

        }

        // The node completely inside query range

        if(root.min >= min && root.max <= max) {

            return root.count;

        }

        long mid = root.min + (root.max - root.min) / 2;

        if(max <= mid) {

            return query(root.left, min, max);

        } else if(min > mid) {

            return query(root.right, min, max);

        } else {

            return query(root.left, min, max) + query(root.right, min, max);

        }

    }

}

class TreeNode {

    long min;

    long max;

    int count;

    TreeNode left;

    TreeNode right;

    public TreeNode(long min, long max) {

        this.min = min;

        this.max = max;

    }

}

**Style 3-2: Segment Tree update method only include both update Segment Tree functions, build Segment Tree separately finished before update**

**第一个问题：当Segment Tree构建方法为基于绝对数值二分时，会出现MLE (猜测Style 1中Segment Tree虽然也基于绝对数值二分构建但没有MLE是因为只在有需要的时候生成子树，这一点基于update方法融合了build tree，只在需要update的时候构建，而当build tree方法独立出来以后我们就不仅仅是只在需要update的时候构建，而是一开始就构建成完整的Segment Tree，如果此时基于绝对值二分构建，那当上下边界值差距很大时就会导致Segment Tree层数和TreeNode会非常多，导致MLE)**

**Wrong Solution (MLE 4/67)**

**Test out by**

Last Executed

Input: nums = [2147483647,-2147483648,-1,0], lower =-1, upper = 0

**MLE 的原因是因为buildSegmentTree() 方法基于数值本身的二分生成的TreeNode过多吗？**

**因为如果是基于数值本身的二分，会出现log2(2147483647) 深度的二叉树，而TreeNode则更多**

class Solution {

    public int countRangeSum(int[] nums, int lower, int upper) {

        int n = nums.length;

        // Find presum based on 'nums', remove all duplicates and sort by TreeSet

        // Why we have to sort ?

        // Because when we query(root, min, max) we compare each TreeNode.min

        // with min and TreeNode.max with max, when build the Segment Tree,

        // we have to maintain each TreeNode's interval from min to max, cannot

        // be scrambled, so the basement array used to build Segment Tree must

        // be sorted first

        TreeSet<Long> set = new TreeSet<>();

        long presum = 0;

        for(int i = 0; i < n; i++) {

            presum += nums[i];

            set.add(presum);

        }

        // Convert TreeSet to array for 'get' method O(1) time complexity

        // access at any index

        Long[] list = set.toArray(new Long[0]);

        int result = 0;

        //SegmentTree segmentTree = new SegmentTree(list[0], list[list.length - 1]);

        TreeNode root = buildSegmentTree(list[0], list[list.length - 1]);

        SegmentTree segmentTree = new SegmentTree(root);

        for(int i = nums.length - 1; i >= 0; i--) {

            segmentTree.update(presum);

            presum -= (long)nums[i];

            // lower <= sums[i] - sums[j] <= upper (sums[k] is the sum of nums[0...k]),

            // which is lower + sums[j] <= sums[i] <= upper + sums[j]

            // so for each sum[j]('presum'), we want to find out the # of

            // all sums[i] which satisfy the condition ----- give a range,

            // and find the # of values in the range, which is what segment

            // tree does

            result += segmentTree.query((long)lower + presum, (long)upper + presum);

        }

        return result;

    }

    private TreeNode buildSegmentTree(long min, long max) {

        if(min > max) {

            return null;

        }

        TreeNode root = new TreeNode(min, max);

        if(min == max) {

            return root;

        }

        long mid = min + (max - min) / 2;

        root.left = buildSegmentTree(min, mid);

        root.right = buildSegmentTree(mid + 1, max);

        return root;

    }

}

class SegmentTree {

    TreeNode root;

    public SegmentTree(TreeNode root) {

        //this.root = new TreeNode(min, max);

        this.root = root;

    }

    public void update(long presum) {

        update(root, presum);

    }

    // The 'udpate' method coding format only contains

    // 'update Segment Tree' related, no 'build Segment Tree'

    private void update(TreeNode root, long presum) {

        // Base condition for 'update Segment Tree'

        if(root == null) {

            return;

        }

        if(root.min > presum || root.max < presum) {

            return;

        }

        // The if condition can be removed since above if

        // condition 'root.min > presum || root.max < presum'

        // check equal to below if condition, below if

        // condition just for keep similar coding format

        // convention than L715, L307

        if(root.min <= presum && root.max >= presum) {

            root.count++;

            update(root.left, presum);

            update(root.right, presum);

        }

    }

    public int query(long min, long max) {

        return query(root, min, max);

    }

    private int query(TreeNode root, long min, long max) {

        if(root == null) {

            return 0;

        }

        if(root.max < min || root.min > max) {

            return 0;

        }

        // The node completely inside query range

        if(root.min >= min && root.max <= max) {

            return root.count;

        }

        long mid = root.min + (root.max - root.min) / 2;

        if(max <= mid) {

            return query(root.left, min, max);

        } else if(min > mid) {

            return query(root.right, min, max);

        } else {

            return query(root.left, min, max) + query(root.right, min, max);

        }

    }

}

class TreeNode {

    long min;

    long max;

    int count;

    TreeNode left;

    TreeNode right;

    public TreeNode(long min, long max) {

        this.min = min;

        this.max = max;

    }

}

**第二个问题：当改变Segment Tree构建方法从基于绝对数值二分变成基于坐标二分时，query()方法也要做相应改变**

**And when try to turn mid absolute value based Segment Tree (e.g long mid = arr[lo] + (arr[hi] - arr[lo]) / 2) into mid index based Segment Tree (e.g int mid = lo + (hi - lo) / 2), we ecnounter problem on query() method, the potential issue may happen on Segment Tree node {min, max} interval build based on array index (e.g int mid = lo + (hi - lo) / 2), not on absolute value (e.g long mid = arr[lo] + (arr[hi] - arr[lo]) / 2)? Though in below query() method we can see it kept calculating 'mid' based on absolute value (long mid = root.min + (root.max - root.min) / 2), but the Segment Tree build turn into index based**

private int query(TreeNode root, long min, long max) {

if(root == null) {

return 0;

}

if(root.max < min || root.min > max) {

return 0;

}

// The node completely inside query range

if(root.min >= min && root.max <= max) {

return root.count;

}

// Not like L715, L307 query() method, compare based on absolute value 'mid',

// L327 query() method compare absed on index 'mid'

// The 'mid' which build on absolute value calculation is the root cause,

// because here we build the Segment Tree left, right subtree based on

// 'mid' of index range on sorted array, not the absolute value

// when we want to go into left or right subtree, the if condition should

// also based on index range divide (e.g array total length = 4, if based

// on index range divide, int mid = lo(=0) + (hi(=4) - lo(=0)) / 2 = 2,

// when we want to go into left subtree, we should compare index range as

// 'max index <= 2', when we want to go into right subtree, we should

// compare index range as 'min > 2'), but that's not the case we use here

long mid = root.min + (root.max - root.min) / 2; // need remove

// Should remove first and second condition and only keep the third

if(max <= mid) {

return query(root.left, min, max); // need remove

} else if(min > mid) {

return query(root.right, min, max); // need remove

} else {

return query(root.left, min, max) + query(root.right, min, max); // The only remain condition

}

}

**Test out by below, expect = 3, output = 2**

import java.util.\*;

public class Solution {

class SegmentTree {

TreeNode root;

public SegmentTree(TreeNode root) {

//this.root = new TreeNode(min, max);

this.root = root;

}

public void update(long presum) {

update(root, presum);

}

// The 'udpate' method coding format only contains

// 'update Segment Tree' related, no 'build Segment Tree'

private void update(TreeNode root, long presum) {

// Base condition for 'update Segment Tree'

if(root == null) {

return;

}

if(root.min > presum || root.max < presum) {

return;

}

// The if condition can be removed since above if

// condition 'root.min > presum || root.max < presum'

// check equal to below if condition, below if

// condition just for keep similar coding format

// convention than L715, L307

if(root.min <= presum && root.max >= presum) {

root.count++;

update(root.left, presum);

update(root.right, presum);

}

}

public int query(long min, long max) {

return query(root, min, max);

}

private int query(TreeNode root, long min, long max) {

if(root == null) {

return 0;

}

if(root.max < min || root.min > max) {

return 0;

}

// The node completely inside query range

if(root.min >= min && root.max <= max) {

return root.count;

}

long mid = root.min + (root.max - root.min) / 2;

if(max <= mid) {

return query(root.left, min, max);

} else if(min > mid) {

return query(root.right, min, max);

} else {

return query(root.left, min, max) + query(root.right, min, max);

}

}

}

class TreeNode {

long min;

long max;

int count;

TreeNode left;

TreeNode right;

public TreeNode(long min, long max) {

this.min = min;

this.max = max;

}

}

public int countRangeSum(int[] nums, int lower, int upper) {

int n = nums.length;

// Find presum based on 'nums', remove all duplicates and sort by TreeSet

// Why we have to sort ?

// Because when we query(root, min, max) we compare each TreeNode.min

// with min and TreeNode.max with max, when build the Segment Tree,

// we have to maintain each TreeNode's interval from min to max, cannot

// be scrambled, so the basement array used to build Segment Tree must

// be sorted first

TreeSet<Long> set = new TreeSet<>();

long presum = 0;

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

presum += nums[i];

set.add(presum);

}

// Convert TreeSet to array for 'get' method O(1) time complexity

// access at any index

Long[] list = set.toArray(new Long[0]);

int result = 0;

//SegmentTree segmentTree = new SegmentTree(list[0], list[list.length - 1]);

// Here we change Segment Tree build way from based on absolute value

// (e.g 'mid' will be 'arr[lo] + (arr[hi] - arr[lo]) / 2') to based on

// index (e.g 'mid' will be 'lo + (hi - lo) / 2')

TreeNode root = buildSegmentTree(list, 0, list.length - 1);

SegmentTree segmentTree = new SegmentTree(root);

for(int i = nums.length - 1; i >= 0; i--) {

segmentTree.update(presum);

presum -= (long)nums[i];

// lower <= sums[i] - sums[j] <= upper (sums[k] is the sum of nums[0...k]),

// which is lower + sums[j] <= sums[i] <= upper + sums[j]

// so for each sum[j]('presum'), we want to find out the # of

// all sums[i] which satisfy the condition ----- give a range,

// and find the # of values in the range, which is what segment

// tree does

result += segmentTree.query((long)lower + presum, (long)upper + presum);

}

return result;

}

private TreeNode buildSegmentTree(Long[] list, int lo, int hi) {

if(lo > hi) {

return null;

}

TreeNode root = new TreeNode(list[lo], list[hi]);

if(lo == hi) {

return root;

}

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

root.left = buildSegmentTree(list, lo, mid);

root.right = buildSegmentTree(list, mid + 1, hi);

return root;

}

public static void main(String[] args) {

int[] nums = new int[]{-2,5,-1};

//int[] nums = new int[]{2147483647,-2147483648,-1,0};

int lower = -2;

//int lower = -1;

int upper = 2;

//int upper = 0;

Solution so = new Solution();

int result = so.countRangeSum(nums, lower, upper);

System.out.println(result);

}

}

**下面是错误的query()方法的debug过程如下：**

**1.第一次query(1, 5) 逻辑会进入如下分支，因为** if(root.min(=3) >= min(=1) && root.max(=3) <= max(=5)) {return root.count;}**返回结果0**

long mid = root.min + (root.max - root.min) / 2; ==> mid = 0

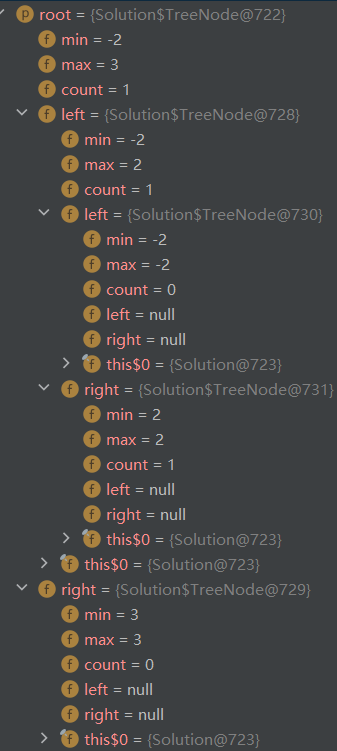
...

else if(min(=1) > mid(=0)) {

return query(root.right, min, max);

...

**第一次query时Segment Tree的状态，**



**2. 第二次query(-4, 0)逻辑会进入如下分支，因为** if(root.min(=-2) >= min(=-4) && root.max(=-2) <= max(=0)) {return root.count;}**返回结果0**

long mid = root.min + (root.max - root.min) / 2; ==> mid = 0

...

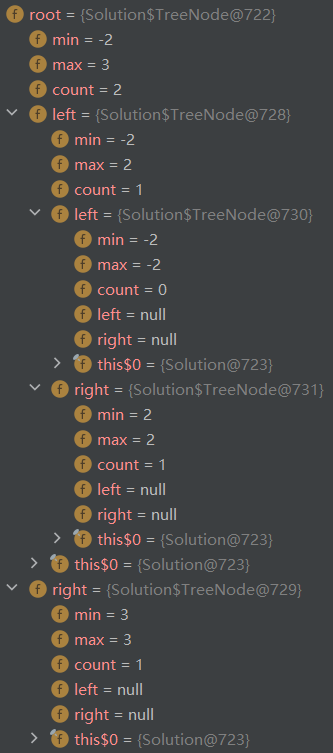
if(max <= mid) {

return query(root.left, min, max);

}

...

**第二次query时Segment Tree更新后的状态**



**3. 第三次query(-2, 2)逻辑会进入如下分支，返回结果2** (query(root.left, min, max) = 2**, 问题出在**query(root.right, min, max) = 0**, 这是因为**if(root.max(=3) < min(=-2) false || root.min(=3) > max(=2) true) {return 0}

long mid = root.min + (root.max - root.min) / 2; ==> mid = 0

...

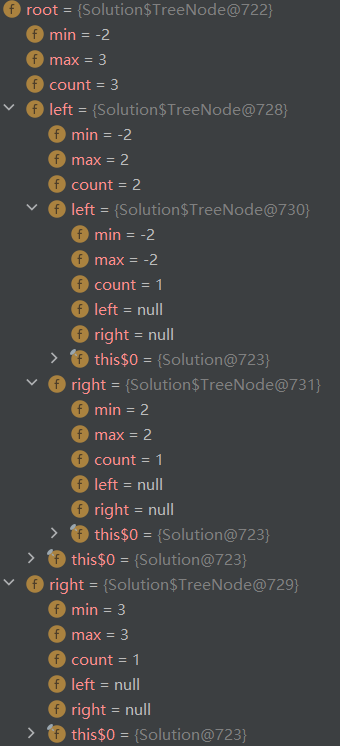
else {

return query(root.left, min, max) + query(root.right, min, max);

}

...

**第三次query时Segment Tree更新后的状态**



**实际上应该基于数组坐标的二分生成TreeNode，正确的query()方法如下：**

private int query(TreeNode root, long min, long max) {

if(root == null) {

return 0;

}

if(root.max < min || root.min > max) {

return 0;

}

// The node completely inside query range

if(root.min >= min && root.max <= max) {

return root.count;

}

//long mid = root.min + (root.max - root.min) / 2;

//if(max <= mid) {

// return query(root.left, min, max);

//} else if(min > mid) {

// return query(root.right, min, max);

//} else {

return query(root.left, min, max) + query(root.right, min, max);

//}

}

**下面是正确的query()方法对应的debug过程：**

**1.第一次query(1, 5) 逻辑会进入如下分支，因为** if(root.min(=2) >= min(=1) && root.max(=2) <= max(=5)) {return root.count;}**返回结果1**

**这里返回的1也是在错误的query()方法中由于进入了下面的分支而仅仅query了root.right分支返回结果0而没有query出root.left分支而错过的部分**

long mid = root.min + (root.max - root.min) / 2; ==> mid = 0

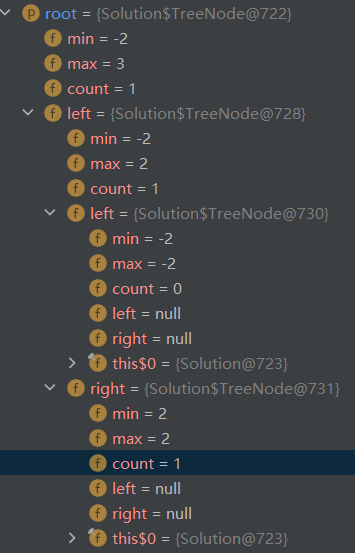
...

else if(min(=1) > mid(=0)) {

// Only return 0 when query on root.right, missing query on root.left which suppose return 1

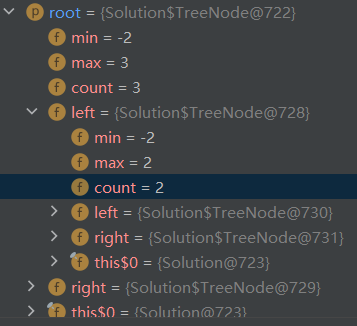
return query(root.right, min, max);

...



**2. 第二次query(-4, 0)逻辑，返回结果0**

**3. 第三次query(-2, 2)逻辑，** if(root.min(=-2) >= min(=-2) && root.max(=2) <= max(=2)) {return root.count;}**返回结果2**



**最后三轮加起来的总和是1 + 0 + 2 = 3**

**Style 3-2 Correct Solution**

class Solution {

    public int countRangeSum(int[] nums, int lower, int upper) {

        int n = nums.length;

        // Find presum based on 'nums', remove all duplicates and sort by TreeSet

        // Why we have to sort ?

        // Because when we query(root, min, max) we compare each TreeNode.min

        // with min and TreeNode.max with max, when build the Segment Tree,

        // we have to maintain each TreeNode's interval from min to max, cannot

        // be scrambled, so the basement array used to build Segment Tree must

        // be sorted first

        TreeSet<Long> set = new TreeSet<>();

        long presum = 0;

        for(int i = 0; i < n; i++) {

            presum += nums[i];

            set.add(presum);

        }

        // Convert TreeSet to array for 'get' method O(1) time complexity

        // access at any index

        Long[] list = set.toArray(new Long[0]);

        int result = 0;

        //SegmentTree segmentTree = new SegmentTree(list[0], list[list.length - 1]);

        TreeNode root = buildSegmentTree(list, 0, list.length - 1);

        SegmentTree segmentTree = new SegmentTree(root);

        for(int i = nums.length - 1; i >= 0; i--) {

            segmentTree.update(presum);

            presum -= (long)nums[i];

            // lower <= sums[i] - sums[j] <= upper (sums[k] is the sum of nums[0...k]),

            // which is lower + sums[j] <= sums[i] <= upper + sums[j]

            // so for each sum[j]('presum'), we want to find out the # of

            // all sums[i] which satisfy the condition ----- give a range,

            // and find the # of values in the range, which is what segment

            // tree does

            result += segmentTree.query((long)lower + presum, (long)upper + presum);

        }

        return result;

    }

    private TreeNode buildSegmentTree(Long[] list, int lo, int hi) {

        if(lo > hi) {

            return null;

        }

        TreeNode root = new TreeNode(list[lo], list[hi]);

        if(lo == hi) {

            return root;

        }

        int mid = lo + (hi - lo) / 2;

        root.left = buildSegmentTree(list, lo, mid);

        root.right = buildSegmentTree(list, mid + 1, hi);

        return root;

    }

}

class SegmentTree {

    TreeNode root;

    public SegmentTree(TreeNode root) {

        //this.root = new TreeNode(min, max);

        this.root = root;

    }

    public void update(long presum) {

        update(root, presum);

    }

    // The 'udpate' method coding format only contains

    // 'update Segment Tree' related, no 'build Segment Tree'

    private void update(TreeNode root, long presum) {

        // Base condition for 'update Segment Tree'

        if(root == null) {

            return;

        }

        if(root.min > presum || root.max < presum) {

            return;

        }

        // The if condition can be removed since above if

        // condition 'root.min > presum || root.max < presum'

        // check equal to below if condition, below if

        // condition just for keep similar coding format

        // convention than L715, L307

        if(root.min <= presum && root.max >= presum) {

            root.count++;

            update(root.left, presum);

            update(root.right, presum);

        }

    }

    public int query(long min, long max) {

        return query(root, min, max);

    }

    private int query(TreeNode root, long min, long max) {

        if(root == null) {

            return 0;

        }

        if(root.max < min || root.min > max) {

            return 0;

        }

        // The node completely inside query range

        if(root.min >= min && root.max <= max) {

            return root.count;

        }

        // Query always on both left and right subtree since

        // we convert Segment Tree build by absolute 'mid' value

        // binary divide way to index 'mid' binary divide way,

        // in absolute 'mid' value binary divide way (e.g L715,

        // L307) we can calculate 'mid' and only go into one

        // subtree branch if condition match, but if we calculate

        // 'mid' by index here, the 'mid' index corresponding value

        // in input array is 'arr[mid]', may not be the 1/2 value

        // of 'arr[lo] + arr[hi]', its not the absolute 'mid' value

        // we used like L715, L307 style to promote speed by divide

        // logic into only go left or right branch

        //long mid = root.min + (root.max - root.min) / 2;

        //if(max <= mid) {

        //    return query(root.left, min, max);

        //} else if(min > mid) {

        //    return query(root.right, min, max);

        //} else {

            return query(root.left, min, max) + query(root.right, min, max);

        //}

    }

}

class TreeNode {

    long min;

    long max;

    int count;

    TreeNode left;

    TreeNode right;

    public TreeNode(long min, long max) {

        this.min = min;

        this.max = max;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**最后回头对 Segment Tree Solution Style 3-1 TLE 66/67的基于absolute value 'mid' based Segment Tree做改进, 换成index 'mid' based Segment Tree以后(主要是对TreeNode的构造和融合了build + update的Style 3-1的update方法改造)，我们获得了如下通过全部67个测试的版本**

class Solution {

    public int countRangeSum(int[] nums, int lower, int upper) {

        int n = nums.length;

        // Find presum based on 'nums', remove all duplicates and sort by TreeSet

        // Why we have to sort ?

        // Because when we query(root, min, max) we compare each TreeNode.min

        // with min and TreeNode.max with max, when build the Segment Tree,

        // we have to maintain each TreeNode's interval from min to max, cannot

        // be scrambled, so the basement array used to build Segment Tree must

        // be sorted first

        TreeSet<Long> set = new TreeSet<>();

        long presum = 0;

        for(int i = 0; i < n; i++) {

            presum += nums[i];

            set.add(presum);

        }

        // Convert TreeSet to array for 'get' method O(1) time complexity

        // access at any index

        Long[] list = set.toArray(new Long[0]);

        int result = 0;

        // Build Segment Tree based on absolute value (e.g 'mid' will be

        // 'arr[lo] + (arr[hi] - arr[lo]) / 2')

        //SegmentTree segmentTree = new SegmentTree(list[0], list[list.length - 1]);

// Build Segment Tree changed to 'index' based (e.g 'mid' will be

        // 'lo + (hi - lo) / 2')

        SegmentTree segmentTree = new SegmentTree(list, 0, list.length - 1);

        for(int i = nums.length - 1; i >= 0; i--) {

            segmentTree.update(presum);

            presum -= (long)nums[i];

            // lower <= sums[i] - sums[j] <= upper (sums[k] is the sum of nums[0...k]),

            // which is lower + sums[j] <= sums[i] <= upper + sums[j]

            // so for each sum[j]('presum'), we want to find out the # of

            // all sums[i] which satisfy the condition ----- give a range,

            // and find the # of values in the range, which is what segment

            // tree does

            result += segmentTree.query((long)lower + presum, (long)upper + presum);

        }

        return result;

    }

}

class SegmentTree {

    TreeNode root;

    //public SegmentTree(long min, long max) {

    //    this.root = new TreeNode(min, max);

    //}

    public SegmentTree(Long[] list, int lo, int hi) {

        this.root = new TreeNode(list, lo, hi);

    }

    public void update(long presum) {

        update(root, presum);

    }

    // The 'udpate' method coding format here merge both

    // 'build Segment Tree' and 'update Segment Tree'

    // into one method, similar to L715, L307

    private void update(TreeNode root, long presum) {

        // Base condition for 'build Segment Tree'

        // When lower boundary larger than upper boundary,

        // means no further traversal, already pass leaf

        // node, no 'count' need to add on any node now

        // directly return

        //if(root.min > root.max) {

        //    return;

        //}

        // Base condition changed to 'index' based

        if(root.lo > root.hi) {

            return;

        }

        // Base condition for 'update Segment Tree'

        if(root.min > presum || root.max < presum) {

            return;

        }

        // The if condition can be removed since above if

        // condition 'root.min > presum || root.max < presum'

        // check equal to below if condition, below if

        // condition just for keep similar coding format

        // convention than L715, L307

        //if(root.min <= presum && root.max >= presum) {

            root.count++;

            // No return here since we have to update

            // count on children nodes)

            //return;

        //}

        // Base condition for 'build Segment Tree'

        // If no below condition will throw stackover flow

        // exception since no close condition for tree build

        // Test out by:

        // Input: nums = [-2,5,-1], lower = -2, upper = 2

        // Also we can merge 'root.min > root.max' to

        // 'root.min == root.max', but we have to make sure

        // 'return' because of 'root.min == root.max'

        // must happen after 'root.count++'

        // Test out by:

        // Input: nums = [0], lower = 0, upper = 0

        // Output = 0, Expected = 1

        //if(root.min == root.max) {

        //    return;

        //}

        // Base condition changed to 'index' based

        if(root.lo == root.hi) {

            return;

        }

        //long mid = root.min + (root.max - root.min) / 2;

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

        if(root.left == null) {

            //root.left = new TreeNode(root.min, mid);

            // Create TreeNode change to 'index' based

            root.left = new TreeNode(root.list, root.lo, mid);

        }

        if(root.right == null) {

            //root.right = new TreeNode(mid + 1, root.max);

            // Create TreeNode change to 'index' based

            root.right = new TreeNode(root.list, mid + 1, root.hi);

        }

        //if(presum <= mid) {

        //    update(root.left, presum);

        //} else if(presum > mid) {

        //    update(root.right, presum);

        //}

        // Since 'mid' is index now, comparison inequality modified

        if(presum <= root.list[mid]) {

            update(root.left, presum);

        } else if(presum > root.list[mid]) {

            update(root.right, presum);

        }

    }

    public int query(long min, long max) {

        return query(root, min, max);

    }

    // The query() method below strictly follow compare by absolute

    // value 'mid' (e.g long mid = arr[lo] + (arr[hi] - arr[lo]) / 2),

    // not compare by index 'mid' (e.g int mid = lo + (hi - lo) / 2)

    private int query(TreeNode root, long min, long max) {

        if(root == null) {

            return 0;

        }

        if(root.max < min || root.min > max) {

            return 0;

        }

        // The node completely inside query range

        if(root.min >= min && root.max <= max) {

            return root.count;

        }

        //long mid = root.min + (root.max - root.min) / 2;

        //if(max <= mid) {

        //    return query(root.left, min, max);

        //} else if(min > mid) {

        //    return query(root.right, min, max);

        //} else {

            return query(root.left, min, max) + query(root.right, min, max);

        //}

    }

}

class TreeNode {

    int lo;

    int hi;

    long min;

    long max;

    int count;

    TreeNode left;

    TreeNode right;

    Long[] list;

    // Improve to pass in original list, index 'lo' and 'hi'

    // the 'min' and 'max' can be retrieved by 'list[index]'

    public TreeNode(Long[] list, int lo, int hi) {

        this.list = list;

        this.min = list[lo];

        this.max = list[hi];

        this.lo = lo;

        this.hi = hi;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/count-of-range-sum/solutions/77987/java-segmenttree-solution-36ms/>

public class Solution {

class SegmentTreeNode {

SegmentTreeNode left;

SegmentTreeNode right;

int count;

long min;

long max;

public SegmentTreeNode(long min, long max) {

this.min = min;

this.max = max;

}

}

private SegmentTreeNode buildSegmentTree(Long[] valArr, int low, int high) {

if(low > high) return null;

SegmentTreeNode stn = new SegmentTreeNode(valArr[low], valArr[high]);

if(low == high) return stn;

int mid = (low + high)/2;

stn.left = buildSegmentTree(valArr, low, mid);

stn.right = buildSegmentTree(valArr, mid+1, high);

return stn;

}

private void updateSegmentTree(SegmentTreeNode stn, Long val) {

if(stn == null) return;

if(val >= stn.min && val <= stn.max) {

stn.count++;

updateSegmentTree(stn.left, val);

updateSegmentTree(stn.right, val);

}

}

private int getCount(SegmentTreeNode stn, long min, long max) {

if(stn == null) return 0;

if(min > stn.max || max < stn.min) return 0;

if(min <= stn.min && max >= stn.max) return stn.count;

return getCount(stn.left, min, max) + getCount(stn.right, min, max);

}

public int countRangeSum(int[] nums, int lower, int upper) {

if(nums == null || nums.length == 0) return 0;

int ans = 0;

Set<Long> valSet = new HashSet<Long>();

long sum = 0;

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

sum += (long) nums[i];

valSet.add(sum);

}

Long[] valArr = valSet.toArray(new Long[0]);

Arrays.sort(valArr);

SegmentTreeNode root = buildSegmentTree(valArr, 0, valArr.length-1);

for(int i = nums.length-1; i >=0; i--) {

updateSegmentTree(root, sum);

sum -= (long) nums[i];

ans += getCount(root, (long)lower+sum, (long)upper+sum);

}

return ans;

}

}

<https://leetcode.com/problems/count-of-range-sum/solutions/77987/java-segmenttree-solution-36ms/comments/82533>

This is really a great method of solving this problem. However, without any comment, this code really makes make me spend a decade to figure all this out.

Let's start from the beginning of solving this problem to understand this code step by step.

This problem requires us to return the number of sum(i, j) which result is between [lower, upper]

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

for (int j = i; j < length; j++) {

.....

}

}

this above method can solve this problem easily with a TLE because of its O(n2) time complexity;

To avoid this situation, the method uses Segment Tree data structure to prevent unnecessary checking which make it into O(n \* log n). How n2 ==> n\*log n? Because the data structure looks like a binary tree, which make the checking from n to log n.

Before going any further, you must know what is segment tree. This is a brilliant data structure which is been used for searching the minimum value within a certain range with O(log n) time.

It is hard to explain using several lines. So check this [link](https://www.youtube.com/watch?v=ZBHKZF5w4YU) out, believe me, it is really fun to watch and the instructor in this video is excellent in explaining things you do not know.

Ok, let start from the main function

public int countRangeSum(int[] nums, int lower, int upper) {

if(nums == null || nums.length == 0) return 0;

int ans = 0;

// Questions 1: why are we using set here?

Set<Long> valSet = new HashSet<Long>();

/\*\*

\* Because in this method, what really matters is the range of sum. So duplicates has no use at all.

\* You will know it after goint through the whole process.

\*/

long sum = 0;

// Use long to prevent overflow.

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

sum += (long) nums[i];

// (long) is a must, you can delete it and you will get a wrong answer

valSet.add(sum);

}

//valSet now contains all sum of range(i, j) where i = 0 and j from 0 to nums.length - 1

Long[] valArr = valSet.toArray(new Long[0]);

// Do not use primitive here, "long" does not work;

Arrays.sort(valArr);

// You must sort here. Because we are going to extract the range of sum. Or, you will Orz

SegmentTreeNode root = buildSegmentTree(valArr, 0, valArr.length-1);

/\*\*

\* Before diving into "buildSegmentTree" function, you can imagine the tree looks like this:

\* This is a binary tree, each node contains a range formed by "min" and "max".

\* the "min" of a parent node is determined by the minimum lower boundary of all its children

\* the "max" is determined by the maximum upper boundary of all its children.

\* And remember, the boundary value must be a sum of a certain range(i, j). And values between

\* min and max may not corresponding to a valid sum;

\* This node also contains a "Count" property which marks how many sub ranges under this node.

\*/

for(int i = nums.length-1; i >=0; i--) {

updateSegmentTree(root, sum);

/\*\*

\* Core part 1 : "updateTree" function will update nodes cnt value by plusing 1 if this node cotains range [sum(0, i)].

\* How?

\* Each leafe of the segment tree contains range [sum[0, i], sum[0,i]] where i starts from 1 to nums.length

\* so, we will definitely find the leafe if we search from the root of the tree;

\* And during the process of finding this leafe, update every node's count value by 1

\* because it must contains the leafe's range by definition.

\*/

sum -= (long) nums[i];

/\*\*

\* Core part 2 : why subtract nums[i] here ?

\* because of its usage in the next part;

\*/

ans += getCount(root, (long)lower+sum, (long)upper+sum);

/\*\*

\* Core part 3 :

\* why sum + lower and sum + upper

\* In core part 2, sum is now the sum of range (0, i - 1), and it serves as a base now.

\* What base?

\* getCount method is trying to return how many valid subranges under [sum + lower, sum + upper]

\* we plus "sum" to range[lower, upper] is because we want it to search the ranges formed by all

\* ranges which starts from i - 1;

\* why ?

\* To understand this, let's imagine sum is 0, and it will be getCount(root, 0 + lower, 0 + upper)

\* this will return number of valid ranges formed by sum(0, j)

\* Oh yeah. Hope you accept this.

\* but we still need the number of valid of ranges formed by sum(i, j) where i is not 0

\* that is what "base" is doing now

\* sum serves as a base here which makes ranges must start from sum(0, i - 1)

\* really hard to explain...... Sorry

\*/

}

return ans;

}

You will understand the following two functions if you understand the implementation of Segment tree.

private SegmentTreeNode buildSegmentTree(Long[] valArr, int low, int high) {

}

private int getCount(SegmentTreeNode stn, long min, long max) {

}

The following function is used to update "Count" variable

No tricky part

private void updateSegmentTree(SegmentTreeNode stn, Long val) {

}

}

**Segment Tree解法比较好的解释**

**Refer to**

<https://leetcode.com/problems/count-of-range-sum/solutions/78052/segment-tree-solution-c-with-brief-explain/>

This is inspired by <https://leetcode.com/discuss/79073/java-segmenttree-solution-36ms>, but the author didn't give any explain. After coding by myself, it becomes pretty clear:

we have the formula: lower<= sums[i] - sums[j] <= upper (sums[k] is the sum of nums[0...k]), which is

lower+sums[j] <= sums[i] <= upper+sums[j], so for each sum[j], we want to find out the # of all sums[i] which satisfy the condition ----- give a range, and find the # of values in the range, which is what segment tree does

Since we are looking for different "sum", not the number itself, so we first need to get all sums from nums, which is:

unordered\_set<long> values;

long sum = 0;

for(int i=0;i<nums.size();++i) { // get all sums

sum+=nums[i];

values.insert(sum);

}

vector<long> sums; // convert to array

for(auto& v : values) {

sums.push\_back(v);

}

sort(sums.begin(), sums.end()); // sort it in order for building segment tree

Now build segment tree (my code is not very clean, but nothing special here)

from step 2, we can see "sum" is the sum of total numbers, then we can reduce nums[i] from sum one by one, to get all different sum values, and update the segment tree (initially the segment tree's counter is 0). During the process, each time we get an updated sum, we can create the range [lower+sum, upper+sum] and lookup in the segment tree

int count = 0;

for(int i=nums.size()-1;i>=0;i--) {

tree->add(sum);

sum-=nums[i];

// lower<= sum[i]-sum[j]<=upper

count += tree->getCount(lower+sum, upper+sum);

}

return count;

It's better to code urself and debug, then it will become clear.

class Solution {

struct SegmentTree {

SegmentTree\* left;

SegmentTree\* right;

int count;

long min;

long max;

// nums is sorted

SegmentTree(vector<long>& nums, int l, int r):left(NULL),right(NULL) {

if (l>r) {

return;

}

min = nums[l];

max = nums[r];

count=0;

if (l==r) {

return;

}

int m = (l+r)/2;

if (l<=m) {

left = new SegmentTree(nums, l,m);

} else {

left=NULL;

}

if (m+1<=r) {

right = new SegmentTree(nums, m+1, r);

} else {

right = NULL;

}

}

void add(long value) {

if (value<min || value>max) {

return;

}

if (value==0) {

// cout<<"add value"<<endl;

}

count++;

if (left && value<=left->max) {

left->add(value);

} else if (right && value>=right->min) {

right->add(value);

}

}

int getCount(long minValue, long maxValue) {

if (minValue<=min && max<=maxValue) {

return count;

}

if (maxValue<min || minValue>max) {

return 0;

}

int leftCount = left ? left->getCount(minValue, maxValue) : 0;

int rightCount = right ? right->getCount(minValue, maxValue) : 0;

return leftCount+rightCount;

}

};

public:

int countRangeSum(vector<int>& nums, int lower, int upper) {

if (nums.size()==0) {

return 0;

}

unordered\_set<long> values;

long sum = 0;

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

sum+=nums[i];

values.insert(sum);

}

vector<long> sums;

for(auto& v : values) {

sums.push\_back(v);

}

sort(sums.begin(), sums.end());

SegmentTree\* tree = new SegmentTree(sums, 0, sums.size()-1);

int count = 0;

for(int i=nums.size()-1;i>=0;i--) {

tree->add(sum);

sum-=nums[i];

// lower<= sum[i]-sum[j]<=upper

count += tree->getCount(lower+sum, upper+sum);

}

return count;

}

};

**Java version (convert from ChatGPT)**

Note: The Java code uses **List** and **Set** from **java.util** to replace C++ vectors and unordered\_set, respectively. The constructor names are updated according to Java conventions, and the **System.out.println** statements are commented out. Additionally, array indexing in Java starts from 0, so adjustments are made accordingly.

import java.util.\*;

class Solution {

static class SegmentTree {

SegmentTree left;

SegmentTree right;

int count;

long min;

long max;

// nums is sorted

SegmentTree(List<Long> nums, int l, int r) {

if (l > r) {

return;

}

min = nums.get(l);

max = nums.get(r);

count = 0;

if (l == r) {

return;

}

int m = (l + r) / 2;

if (l <= m) {

left = new SegmentTree(nums, l, m);

} else {

left = null;

}

if (m + 1 <= r) {

right = new SegmentTree(nums, m + 1, r);

} else {

right = null;

}

}

void add(long value) {

if (value < min || value > max) {

return;

}

if (value == 0) {

// System.out.println("add value");

}

count++;

if (left != null && value <= left.max) {

left.add(value);

} else if (right != null && value >= right.min) {

right.add(value);

}

}

int getCount(long minValue, long maxValue) {

if (minValue <= min && max <= maxValue) {

// System.out.println(minValue + "," + maxValue);

return count;

}

if (maxValue < min || minValue > max) {

return 0;

}

int leftCount = left != null ? left.getCount(minValue, maxValue) : 0;

int rightCount = right != null ? right.getCount(minValue, maxValue) : 0;

return leftCount + rightCount;

}

}

public int countRangeSum(int[] nums, int lower, int upper) {

if (nums.length == 0) {

return 0;

}

Set<Long> values = new HashSet<>();

long sum = 0;

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

sum += nums[i];

values.add(sum);

}

List<Long> sums = new ArrayList<>(values);

sums.sort(Comparator.naturalOrder());

SegmentTree tree = new SegmentTree(sums, 0, sums.size() - 1);

int count = 0;

for (int i = nums.length - 1; i >= 0; i--) {

tree.add(sum);

sum -= nums[i];

// lower <= sum[i] - sum[j] <= upper

count += tree.getCount(lower + sum, upper + sum);

}

return count;

}

}

**看了这个解释以后的几点想法：**

Since we are looking for different "sum", not the number itself, so we first need to get all 'sums' from nums, which is:

     unordered\_set<long> values;

     long sum = 0;

     for(int i=0;i<nums.size();++i) {   // get all sums

         sum+=nums[i];

         values.insert(sum);

     }

     vector<long> sums;     // convert to array

     for(auto& v : values) {

         sums.push\_back(v);

     }

     sort(sums.begin(), sums.end());  // sort it in order for building segment tree

and all these 'sums' need to calculate from index = 0, which means we need all 'sums' like 'nums[0]', 'nums[0] + nums[1]',

'nums[0] + nums[1] + nums[2]' .... etc, 'nums[0] + nums[2]' or 'nums[1] + nums[2]' are not the 'sums' we need here, we

only need all 'sums' accumulate from index = 0 and not skip indexes in between, this is based on what we actually need

from below analysis: critical => sums[k] is the sum of nums[0...k], we want all sums[i] calculated from nums[0...i]

we have the formula: lower <= sums[i] - sums[j] <= upper (sums[k] is the sum of nums[0...k]), which is

lower + sums[j] <= sums[i] <= upper + sums[j], so for each sum[j], we want to find out the # of all sums[i]

which satisfy the condition ----- give a range, and find the # of values in the range, which is what segment tree does

Why we have to sort the input array ?

Because when we query(root, min, max) we compare each TreeNode.min with min and TreeNode.max with max, when build the

Segment Tree, we have to maintain each TreeNode's interval from min to max, cannot be scrambled, so

the basement array used to build Segment Tree must be sorted first

Test out by:

Input

nums =[2147483647,-2147483648,-1,0]

lower = -1

upper = 0

Output = 0, Expected = 4

Full code for test:

import java.util.\*;

public class Solution {

class SegmentTreeNode {

SegmentTreeNode left;

SegmentTreeNode right;

int count;

long min;

long max;

public SegmentTreeNode(long min, long max) {

this.min = min;

this.max = max;

}

}

private SegmentTreeNode buildSegmentTree(Long[] valArr, int low, int high) {

if(low > high) return null;

SegmentTreeNode stn = new SegmentTreeNode(valArr[low], valArr[high]);

if(low == high) return stn;

int mid = (low + high)/2;

stn.left = buildSegmentTree(valArr, low, mid);

stn.right = buildSegmentTree(valArr, mid+1, high);

return stn;

}

private void updateSegmentTree(SegmentTreeNode stn, Long val) {

if(stn == null) return;

if(val >= stn.min && val <= stn.max) {

stn.count++;

updateSegmentTree(stn.left, val);

updateSegmentTree(stn.right, val);

}

}

private int getCount(SegmentTreeNode stn, long min, long max) {

if(stn == null) return 0;

if(min > stn.max || max < stn.min) return 0;

if(min <= stn.min && max >= stn.max) return stn.count;

return getCount(stn.left, min, max) + getCount(stn.right, min, max);

}

public int countRangeSum(int[] nums, int lower, int upper) {

if(nums == null || nums.length == 0) return 0;

int ans = 0;

Set<Long> valSet = new HashSet<Long>();

long sum = 0;

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

sum += (long) nums[i];

valSet.add(sum);

}

Long[] valArr = valSet.toArray(new Long[0]);

Arrays.sort(valArr);

SegmentTreeNode root = buildSegmentTree(valArr, 0, valArr.length-1);

for(int i = nums.length-1; i >=0; i--) {

updateSegmentTree(root, sum);

sum -= (long) nums[i];

ans += getCount(root, (long)lower+sum, (long)upper+sum);

}

return ans;

}

public static void main(String[] args) {

//int[] nums = new int[]{-2,5,-1};

int[] nums = new int[]{2147483647,-2147483648,-1,0};

//int lower = -2;

int lower = -1;

//int upper = 2;

int upper = 0;

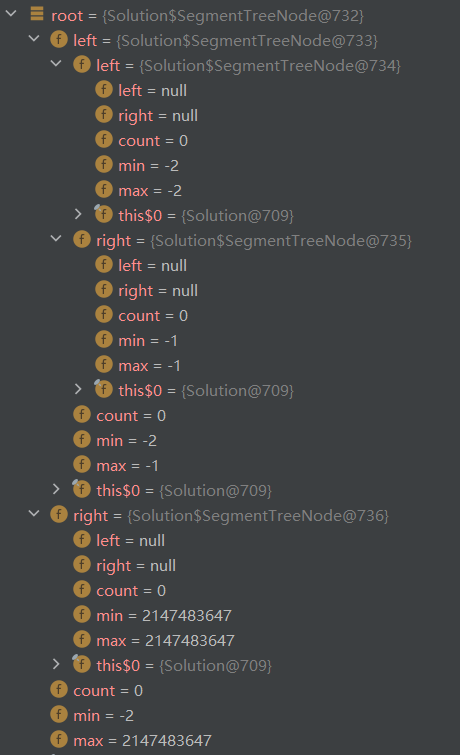
Solution so = new Solution();

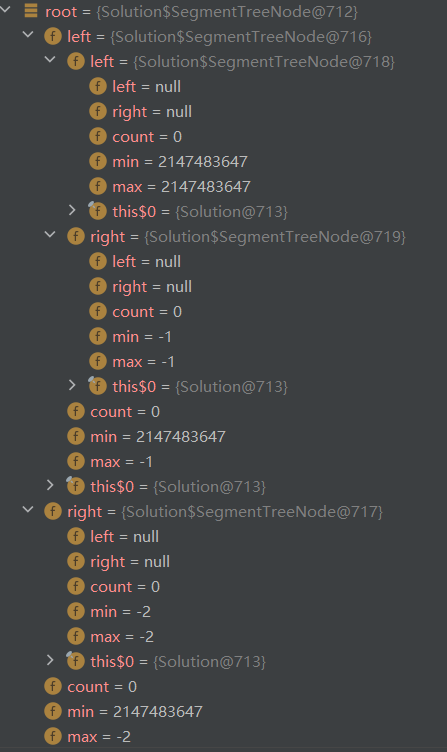
int result = so.countRangeSum(nums, lower, upper);

System.out.println(result);

}

}





**Segment Tree with Discretization**

**Refer to**

<https://leetcode.com/problems/count-of-range-sum/solutions/1674377/java-segment-tree-with-explanation/>

Let's first look at Brute Force solution:

class Solution {

public int countRangeSum(int[] nums, int lower, int upper) {

int n = nums.length;

long[] presum = new long[n + 1];

int res = 0;

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

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

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

if (lower <= presum[i] - presum[j] && presum[i] - presum[j] <= upper) {

res++;

}

}

}

return res;

}

}

T: O(n^2) S: O(n)

the second **for** loop is used to find the number of j that satisify:

lower <= presum[i] - presum[j] && presum[i] - presum[j] <= upper ==> presum[i] - upper <= presum[j] <= presum[i] - lower

We need to find out how many j satisifies this condition.

If we store the presum[] sorted, we can easily find the position of presum[i] - upper and presum[i] - lower.

So the number of satisified j will be count(pos2 - pos1)

We could get the API from the code above:

add(value): add a value into a data structure

query(): query how many numbers of values in this range

This is very close to Segment Tree!

class Solution {

class Node {

int cnt;

}

Node[] tree;

int N;

Map<Long, Integer> rank;

public int countRangeSum(int[] nums, int lower, int upper) {

if (nums == null || nums.length == 0) return 0;

int n = nums.length;

long[] preSum = new long[n + 1];

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

preSum[i + 1] = preSum[i] + (long)nums[i];

}

List<Long> list = new ArrayList<>();

for (int i = 0; i <= n; i++) { // we need add preSum[0]

list.add(preSum[i]);

list.add(preSum[i] - (long)upper); // discretize all possible values we will use

list.add(preSum[i] - (long)lower);

}

list = new ArrayList<>(new HashSet<>(list));

Collections.sort(list);

N = list.size();

rank = new HashMap<>();

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

rank.put(list.get(i), i);

}

tree = new Node[N << 2];

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

tree[i] = new Node();

}

int res = 0;

for (int i = 0; i <= n; i++) { // we need add preSum[0]

int val = rank.get(preSum[i]);

int low = rank.get(preSum[i] - (long)upper);

int high = rank.get(preSum[i] - (long)lower);

res += query(low, high, 0, N - 1, 0);

add(val, 0, N - 1, 0);

}

return res;

}

public void add(int value, int l, int r, int n) {

if (l == r) {

tree[n].cnt++;

return;

}

int mid = l + (r - l) / 2;

if (value <= mid) {

add(value, l, mid, n \* 2 + 1);

} else {

add(value, mid + 1, r, n \* 2 + 2);

}

tree[n].cnt = tree[n \* 2 + 1].cnt + tree[n \* 2 + 2].cnt;

}

public int query(int L, int R, int l, int r, int n) {

if (L <= l && r <= R) {

return tree[n].cnt;

}

int mid = l + (r - l) / 2;

int res = 0;

if (mid >= L) res += query(L, R, l, mid, n \* 2 + 1);

if (mid < R) res += query(L, R, mid + 1, r, n \* 2 + 2);

return res;

}

}

**样例代码中存在一个新的技术离散化(Discretization)，在构建Segment Tree的过程中使用到了用于Binary Indexed Tree构建的Discretization步骤，甚至这个包含Discretization步骤的模版也同时作用于其他类似的题目，比如** <https://leetcode.com/problems/count-of-smaller-numbers-after-self/solutions/705701/java-standard-segment-tree-template-with-discretization-very-easy/>**，Segment Tree的Array实现中最难的部分在于Segment Tree的构造部分，如何定义leaf node是个比较复杂的问题，最简单的一种是直接使用原始输入，例如L307中的Segment Tree构造中我们直接使用了给定的nums中原始数值，并没有改造(e.g 排序)或者进行Discretization这样比较tricky的部分，如下：**

int[] tree;

int n;

public NumArray(int[] nums) {

n = nums.length;

// Why not n \* 2 but n \* 4 ?

// https://stackoverflow.com/questions/28470692/how-is-the-memory-of-the-array-of-segment-tree-2-2-ceillogn-1

tree = new int[4 \* n];

// Call this method as buildTree(nums, 0, 0, n-1),

// Here nums[] is input array and n is its size

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

}

// The build tree process is quite different than iterative segment tree, which

// directly assign original array into tree as leaves, indexes between [n, 2n - 1],

// in recursive solution, the leaves node not directly locate between [n, 2n - 1],

// instead follow 2 \* treeIndex + 1 or 2 \* treeIndex + 2 rule level by level

// until the leaves level reached, that's also why instead only requires 2 \* n

// length for iterative segement tree, in recursive segement tree requires 4 \* n

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;

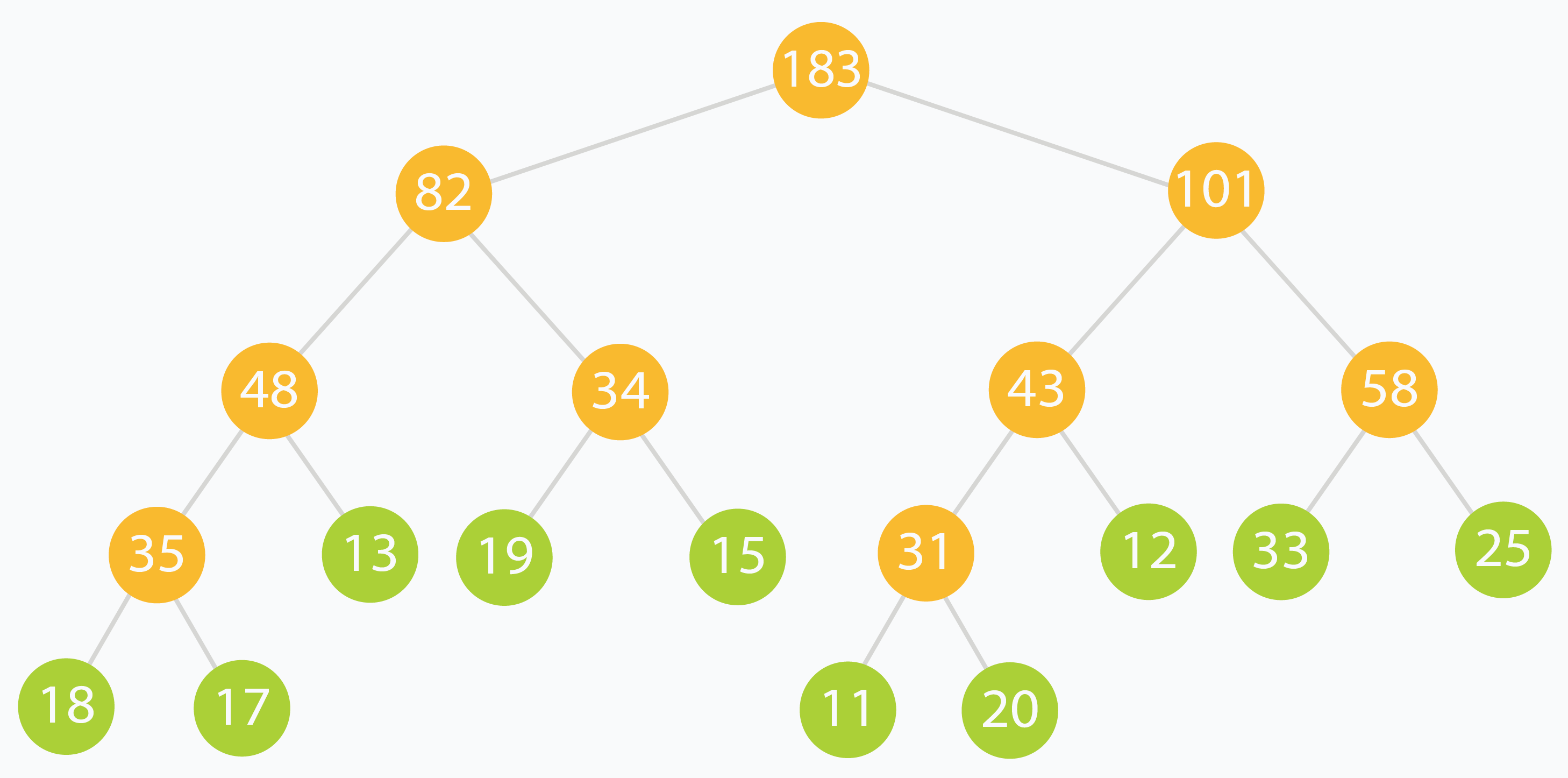
buildTree(nums, 2 \* treeIndex + 1, lo, mid);

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

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

}

**根据L307样例中给出的具体例子，直接使用原始输入nums = {18,17,13,19,15,11,20,12,33,25} 赋值给对应的leaf node后，我们构建的Segment Tree就会如下图所示：**



We will use the above tree as a practical example of what a Range Sum Query segment tree looks and behaves like.

**How do we make one?**

Let our data be in an array arr[] of size *n*.

The root of our segment tree typically represents the entire interval of data we are interested in. This would be arr[0:n-1].

Each leaf of the tree represents a range comprising of just a single element. Thus the leaves represent arr[0], arr[1]and so on till arr[n-1].

The internal nodes of the tree would represent the

**merged** or **union** result of their children nodes.

Each of the children nodes could represent approximately half of the range represented by their parent.

**但是本题L327中显然没法直接使用原始输入的nums给Segment Tree的leaf node赋值6**