Count of Range Sum

Given an integer array `nums`, 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` (`i` ≤ `j`), inclusive.

Note:
A naive algorithm of $O(n^2)$ is trivial. You MUST do better than that.

Example:
Given *nums* = `[-2, 5, -1]`, *lower* = `-2`, *upper* = `2`,
Return `3`.
The three ranges are : `[0, 0]`, `[2, 2]`, `[0, 2]` and their respective sums are: `-2, -1, 2`.

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

See here for better view

First of all, let's look at the naive solution. Preprocess to calculate the prefix sums `S[i]` = `S(0, i)`, then `S(i, j)` = `S[j] - S[i]`. Note that here we define `S(i, j)` as the sum of range `[i, j)` where `j` exclusive and `j` > `i`. With these prefix sums, it is trivial to see that with `O(n^2)` time we can find all `S(i, j)` in the range `[lower, upper]`

Java - Naive Solution

```
public int countRangeSum(int[] nums, int lower, int upper) {
    int n = nums.length;
    long[] sums = new long[n + 1];
    for (int i = 0; i < n; ++i)
        sums[i + 1] = sums[i] + nums[i];
    int ans = 0;
    for (int i = 0; i < n; ++i)
        for (int j = i + 1; j <= n; ++j)
            if (sums[j] - sums[i] >= lower && sums[j] - sums[i] <= upper)
                ans++;
    return ans;
}
```

However the naive solution is set to TLE intentionally

Now let's do better than this.

Recall count smaller number after self where we encountered the problem

- `count[i]` = count of `nums[j]` - `nums[i]` < `0` with `j` > `i`

Here, after we did the preprocess, we need to solve the problem

- `count[i]` = count of `a` <= `S[j]` - `S[i]` <= `b` with `j` > `i`
- `ans` = sum(count[:])

Therefore the two problems are almost the same. We can use the same technique used in that problem to solve this problem. One solution is merge sort based; another one is Balanced BST based. The time complexity are both `O(n log n)`.

The merge sort based solution counts the answer while doing the merge. During the merge stage, we have already sorted the left half `[start, mid)` and right half `[mid, end)`. We then iterate through the left half with index `i`. For each `i`, we need to find two indices `k` and `j` in the right half where

- `j` is the first index satisfy `sums[j] - sums[i] > upper` and
- `k` is the first index satisfy `sums[k] - sums[i] >= lower`.

Then the number of sums in `[lower, upper]` is `j-k`. We also use another index `t` to copy the elements satisfy `sums[t] < sums[i]` to a cache in order to complete the merge sort.

Despite the nested loops, the time complexity of the "merge & count" stage is still linear. Because the indices `k`, `j`, `t` will only increase but not decrease, each of them will only traversal the right half once at most. The total time complexity of this divide and conquer solution is then `O(n log n)`.

One other concern is that the `sums` may overflow integer. So we use long instead.

Java - Merge Sort Solution

```java
public int countRangeSum(int[] nums, int lower, int upper) {
    int n = nums.length;
    long[] sums = new long[n + 1];
    for (int i = 0; i < n; ++i)
        sums[i + 1] = sums[i] + nums[i];
    return countWhileMergeSort(sums, 0, n + 1, lower, upper);
}

private int countWhileMergeSort(long[] sums, int start, int end, int lower, int up
per) {
    if (end - start <= 1) return 0;
    int mid = (start + end) / 2;
    int count = countWhileMergeSort(sums, start, mid, lower, upper)
              + countWhileMergeSort(sums, mid, end, lower, upper);
    int j = mid, k = mid, t = mid;
    long[] cache = new long[end - start];
    for (int i = start, r = 0; i < mid; ++i, ++r) {
        while (k < end && sums[k] - sums[i] < lower) k++;
        while (j < end && sums[j] - sums[i] <= upper) j++;
        while (t < end && sums[t] < sums[i]) cache[r++] = sums[t++];
        cache[r] = sums[i];
        count += j - k;
    }
    System.arraycopy(cache, 0, sums, start, t - start);
    return count;
}
```

written by dietpepsi original link here

Solution 2

Thanks for those contributing excellent ideas to this problem. Here is a quick summary of solutions based on either divide and conquer or binary indexed tree.

To start, we already know there is a straightforward solution by computing each range sum and checking whether it lies in [lower, upper] or not. If the number of elements is n, we have n*(n+1)/2 such range sums so the naive solution will end up with O(n^2) time complexity. Now we are asked to do better than that. So what are the targeted time complexities in your mind? When I first looked at the problem, my instinct is that O(n) solution is too ambitious, so I will target at linearithmic-like (O(n*(logn)^b)) solutions. To get the logarithmic part, it's natural to think of breaking down the original array, and that's where the divide-and-conquer idea comes from.

For this problem, we need some array to apply our divide and conquer algorithm. Without much thinking, we can do that directly with the input array (nums) itself. Since our problem also involves range sums and I believe you have the experience of computing range sums from prefix array of the input array, we might as well apply divide and conquer ideas on the prefix array. So I will give both the input-array based and prefix-array based divide&conquer solutions.

Let's first look at input-array based divide&conquer solution. Our original problem is like this: given an input array nums with length n and a range [lower, upper], find the total number of range sums that lie in the given range. Note the range [lower, upper] and the input array are both fixed. Therefore each range sum can be characterized by two indices i1 and i2 (i1 <= i2), such that range sum S(i1, i2) is the summation of input elements with indices going from i1 up to i2 (both inclusive). Then our problem can be redefined in terms of the value ranges of i1 and i2. For example our original problem can be restated as finding the total number of range sums lying in the given range with 0 <= i1 <= i2 <= n - 1, or in a symbolic way T(0, n-1).

Now if we break our original input array into two subarrays, [0, m] and [m+1, n-1] with m = (n-1)/2, our original problem can be divided into three parts, depending on the values of i1 and i2. If i1 and i2 are both from the first subarray [0, m], we have a subproblem T(0, m); if i1 and i2 are both from the second subarray, we have a subproblem T(m+1, n-1); if i1 is from the first subarray and i2 from the second (note we assume i1 <= i2, therefore we don't have the other case with i2 from first subarray and i1 from second), then we have a new problem which I define as C. In summary we should have:

T(0, n-1) = T(0, m) + T(m+1, n-1) + C

Now from the master theorem, the time complexity of the new problem C should be better than O(n^2), otherwise we make no improvement by applying this divide&conquer idea. So again, I will aim at linearithmic-like solutions for the new problem C: find the total number of range sums lying in the given range with each range sum starting from the first subarray and ending at the second subarray.

First let's try to compute all such range sums. The way I did it was first computing the prefix array of the second subarray and the suffix array (or "backward" prefix array if you like) of the first subarray. Then I can naively add each element in the suffix array to all elements in the prefix array to obtain all the possible range sums. Of course you end up with O(n^2) solution, as expected. So how can we approach it with better time complexity?

Here are the facts I observed: for each element e in the suffix array, we need to add it to all elements in the prefix array. But the order in which we add it doesn't matter. This implies that we can sort our prefix array. This can be done in O(nlogn) time. Now we have a sorted prefix array, do we still need to add the element e to all elements in the prefix array? The answer is no. Because our final goal is to compare the resulted range sums with the given range bounds lower and upper. It is equivalent to modifying the range bounds so we have new bounds (lower - e) and (upper - e) and leave the prefix array unchanged. Now we can compare these new bounds with the sorted prefix array, and I'm sure you can write your own binary search algorithm to do that. So for each element e in the suffix array, we can compute the modified range bounds and get the number of range sums in this new range in logn time. Therefore the total time will be O(nlogn). So in summary, our new problem C can be solved in O(nlogn) time and according to the master theorem, our original problem can be solved in O(n(logn)^2) time. The following is the complete java program:

```java
public int countRangeSum(int[] nums, int lower, int upper) {
    if (nums == null || nums.length == 0 || lower > upper) {
        return 0;
    }

    return countRangeSumSub(nums, 0, nums.length - 1, lower, upper);
}

private int countRangeSumSub(int[] nums, int l, int r, int lower, int upper) {
    if (l == r) {
        return nums[l] >= lower && nums[r] <= upper ? 1 : 0;  // base case
    }

    int m = l + (r - l) / 2;
    long[] arr = new long[r - m];  // prefix array for the second subarray
    long sum = 0;
    int count = 0;

    for (int i = m + 1; i <= r; i++) {
        sum += nums[i];
        arr[i - (m + 1)] = sum; // compute the prefix array
    }

    Arrays.sort(arr);  // sort the prefix array

    // Here we can compute the suffix array element by element.
    // For each element in the suffix array, we compute the corresponding
    // "insertion" indices of the modified bounds in the sorted prefix array
    // then the number of valid ranges sums will be given by the indices differen
ce.
    // I modified the bounds to be "double" to avoid duplicate elements.
    sum = 0;
    for (int i = m; i >= l; i--) {
        sum += nums[i];
        count += findIndex(arr, upper - sum + 0.5) - findIndex(arr, lower - sum -
0.5);
    }

    return countRangeSumSub(nums, l, m, lower, upper) + countRangeSumSub(nums, m
+ 1, r, lower, upper) + count;
}

// binary search function
private int findIndex(long[] arr, double val) {
    int l = 0, r = arr.length - 1, m = 0;

    while (l <= r) {
        m = l + (r - l) / 2;

        if (arr[m] <= val) {
            l = m + 1;
        } else {
            r = m - 1;
        }
    }

    return l;
}
```

(Next two parts will come in answers due to the limitation of maximum length of characters)

written by fun4LeetCode original link here

Solution 3

Understand my segmentTree implementation is not optimized. Please feel free to give me suggestions.

```java
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;
    }

}
```

written by fangyang original link here