<https://leetcode.com/problems/count-of-smaller-numbers-after-self/description/>

Given an integer array nums, return *an integer array* counts *where* counts[i] *is the number of smaller elements to the right of* nums[i].

**Example 1:**

Input: nums = [5,2,6,1]

Output: [2,1,1,0]

Explanation:

To the right of 5 there are 2 smaller elements (2 and 1).

To the right of 2 there is only 1 smaller element (1).

To the right of 6 there is 1 smaller element (1).

To the right of 1 there is 0 smaller element.

**Example 2:**

Input: nums = [-1]

Output: [0]

**Example 3:**

Input: nums = [-1,-1]

Output: [0,0]

**Constraints:**

1 <= nums.length <= 10^5

-10^4 <= nums[i] <= 10^4

**Attempt 1: 2023-09-29**

**Solution 1: Binary Indexed Tree + Binary Search (1200 min)**

class Solution {

    int[] copy;

    int[] bit;

    public List<Integer> countSmaller(int[] nums) {

        filterUniqueValueAndSort(nums);

        // Need a 'bit' array to record each number's frequency

        // in original input nums

        // Based on point 1: 'bit' is actually BIT structured array,

        // since BIT structured array is 1-indexed, follow standard

        // creation of BIT structured array, we add padding position

        // of index 0, hence length change to original array length + 1

        bit = new int[copy.length + 1];

        List<Integer> result = new ArrayList<>();

        // Scan original input nums from right to left

        for(int i = nums.length - 1; i >= 0; i--) {

            // Find which bucket(id) for nums[i] in sorted array, but no

            // need to '+ 1' since we will '+ 1' in both 'query' and 'update'

            // method later, the '+ 1' because binarySearch is 0-index based,

            // but BIT is 1-index based

            // Relation: bucket\_id\_before\_padding + 1 = bucket\_id\_after\_padding

            // e.g nums = [5,2,6,1], when we search for 5 in sorted array

            // [1,2,5,6] will get bucket\_id\_before\_padding = 2 and '+ 1' will

            // get bucket\_id\_after\_padding = 3, means the 3rd bucket

            // but we only need original bucket\_id\_before\_padding = 2 for 5 here,

            // and in 'query' and 'update' method we will make '+ 1' padding,

            // so no '+ 1' here

            //int bucket\_id\_after\_padding = Arrays.binarySearch(copy, nums[i]) + 1;

            //int bucket\_id\_before\_padding = Arrays.binarySearch(copy, nums[i]);

            // Use customized 'findNumIndexInSortedArray' method to replace build in one

            int bucket\_id\_before\_padding = findNumIndexInSortedArray(copy, nums[i]);

            // We need to query 'bucket\_id\_before\_padding - 1', the '- 1' means

            // presum range for buckets ids end at 'bucket\_id\_before\_padding - 1'

            // e.g

            // NOTE: Below based on bucket\_id\_after\_padding

            // For nums = [5,2,6,1] after mapping we skip non-exist buckets

            // as bucket 3 and 4, so 1 mapping to bucket 1(=bit[1]), 2 mapping

            // to bucket 2(=bit[2]), 5 mapping to bucket 3(=bit[3]), 6 mapping

            // to bucket 4(=bit[4]), bit[0] is the padding, when we query

            // all smaller number count on right of 5 equals to find all related

            // buckets under BIT structured count sum before 5's mapping bucket,

            // be careful, as 5 mapping to bucket 3, its not necessarily mean we

            // have to sum up both bucket 1 and bucket 2 count as a linear scan,

            // what we have to follow up is only based on BIT structured relation,

            // and consider each cell coverage definition in BIT we only add bucket

            // 2 count, because bucket 2 count already include bucket 1 count, and

            // the reason we come to BIT way is because BIT only spend O(logN) on

            // query and update, but linear scan will take O(N)

            result.add(query(bucket\_id\_before\_padding - 1));

            // We also need to update the count for current bucket

            // e.g nums = [5,2,6,1], when we encounter 5, we have to update count

            // of bucket 3 by increase 1

            update(bucket\_id\_before\_padding);

        }

        // Because we scan from right to left on original input nums,

        // the result always appending for the rightmost number first,

        // to display as left to right order in result, we have to

        // reverse the 'result' list

        Collections.reverse(result);

        return result;

    }

    // Given an index 'i', find all previous count sum

    private int query(int i) {

        int presum = 0;

        // Padding 1 to match BIT array indexes as [1, n] rather than [0, n - 1]

        // Passed in 'i' is bucket\_id\_before\_padding, after 'i++' is bucket\_id\_after\_padding

        i++;

        while(i > 0) {

            presum += bit[i];

            i -= i & -i;

        }

        return presum;

    }

    private void update(int i) {

        // Padding 1 to match BIT array indexes as [1, n] rather than [0, n - 1]

        // Passed in 'i' is bucket\_id\_before\_padding, after 'i++' is bucket\_id\_after\_padding

        i++;

        while(i < bit.length) {

            bit[i] += 1;

            i += i & -i;

        }

    }

    private void filterUniqueValueAndSort(int[] nums) {

        // Rebuild original input nums array with unique value only

        Set<Integer> numSet = new HashSet<>();

        for(int num : nums) {

            numSet.add(num);

        }

        copy = new int[numSet.size()];

        // Assign each number an index, like a bucket

        int index = 0;

        for(int num : numSet) {

            copy[index++] = num;

        }

        // Have to sort to quickly get each number's index

        Arrays.sort(copy);

    }

    // Find upper boundary

    private int findNumIndexInSortedArray(int[] arr, int val) {

        // Find largest element in arr which <= val

        int l = 0, r = arr.length - 1, m = 0;

        while(l <= r) {

            m = l + ((r - l) >> 1);

            if(arr[m] > val) {

                r = m - 1;

            } else {

                l = m + 1;

            }

        }

        // Corner case [-5, -5]

        if(l == 0 && arr[l] > val) {

            l--;

        }

        //return l + 1; -> this is wrong

        // The 'l - 1' (or 'r') is the actual upper boundary if we

        // consider condition 'if(arr[m] > val)', the upper boundary

        // means the last element index in arr that equal or less

        // than 'val'(<= val), the '+ 1' is the mandatory shift up

        // 1 position required for 1-based BIT array

        //return l - 1 + 1; // or 'r + 1' equally

        // Note: in this solution no need '+ 1' as we exactly only

        // want to find last element index which equal to given 'val'

        return l - 1;

    }

}

Time Complexity: O(NlogN)

Space Complexity: O(NlogN)

**Refer to**

**【树状数组】LeetCode 315 Count of Smaller Numbers After Self**

<https://www.youtube.com/watch?v=JqPA3Wgg0MY>

**Binary Indexed Tree (BIT) 不仅可以用来解决range sum的问题，也可以用来解决range count(frequency)的问题**

**问题： 为何本题与BIT有关？**

**这是比较难想的一部分,比如输入[5,2,6,1]，对于1，右边没有任何数，所以没有任何满足i > j and nums[i] < nums[j]的数，对于1计数为0，对于6，右边为1，满足坐标大于6，且数值小于6， 对于6计数为1，对于2，右边有6和1，但只有1满足要求，对于2计数为1，对于5，右边有2，6和1，其中2和1满足坐标大于5，且数值小于5，对于5计数为2，所以从右往左看更自然，然后我们会想到采用类似桶排序的方式，比如我们遇到1，就在标号为1的桶bucket里面计数count加一，遇到6，就在标号为6的桶bucket里面计数count加一。。。那么在遇到6的时候如果需要知道之前有多少个数比6小只需要知道从标号为1的桶bucket到标号为5的桶bucket中一共有多少个数即可，这个思路非常关键，下面我们用[5,2,6,1]这个例子继续说明**

e.g

Given an integer array nums, return an integer array counts where counts[i] is the number of smaller elements to the right of nums[i].

nums = [5,2,6,1]

result = [2,1,1,0]

prefix sum: how many items are less than self ?

right -> left, those items are smaller numbers after self

Presum on pass scanning demo:

Check from left to right for input nums = [5,2,6,1], the presum calculated from a plain relation in one pass scanning (for presum[i] scanning count[1, i - 1], sum all of them up in a single run) which spend O(N)

**本例只有1到6，我们就把桶bucket从1到6列出来，每个桶里面所对应的count初始都为0，我们是从右往左遍历的，而且我们在遍历的过程中强调一个“此时此刻”，因为每个桶的状态是在每一轮后实时变化的。我们第一轮遇到了1，所以桶1计数count变为1，1对应的presum则代表了我们想要知道“此时此刻”有多少个数在1右边比1小，因为桶1就是起点，没有桶0的存在，所以对于1的presum就是0，代表在1的右边只有0个数比1小。然后第二轮遇到6，我们想知道“此时此刻”桶1到桶5的情况，即“此时此刻”1到5号桶里count的总数，而对于此时的桶1到桶5，只有桶1里面count在第一轮中更新为1，其他都为0，所以6的presum是所有的count加起来只有桶1中的count即1，代表在6的右边只有1个数比6小，同样，我们需要把桶6的count更新为1。然后第三轮遇到2，我们需要知道桶1里“此时此刻”的count数，依然是1，所以2的presum就是1，代表在2的右边只有1个数比2小，同样，我们需要把桶2的count更新为1。最后第四轮遇到5，我们想知道“此时此刻”桶1到桶4的情况，即“此时此刻”1到4号桶里count的总数，而对于此时的桶1到桶4，我们看到桶1和桶2分别有count为1，总计2，所以5的presum就是2，代表在5的右边有2个数比5小，同样，我们需要把桶5的count更新为1。**

**Round 1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Current number is 1 |  |  |  |  |  |  |
| bucket | 1 | 2 | 3 | 4 | 5 | 6 |
| count | 1 |  |  |  |  |  |
| presum | 0 |  |  |  |  |  |

**Round 2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Current number is 6 |  |  |  |  |  |  |
| bucket | 1 | 2 | 3 | 4 | 5 | 6 |
| count | 1 |  |  |  |  | 1 |
| presum | 0 |  |  |  |  | 1 |

**Round 3**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Current number is 2 |  |  |  |  |  |  |
| bucket | 1 | 2 | 3 | 4 | 5 | 6 |
| count | 1 | 1 |  |  |  | 1 |
| presum | 0 | 1 |  |  |  | 1 |

**Round 4**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Current number is 5 |  |  |  |  |  |  |
| bucket | 1 | 2 | 3 | 4 | 5 | 6 |
| count | 1 | 1 |  |  | 1 | 1 |
| presum | 0 | 1 |  |  | 2 | 1 |

至此，我们有2个function需要实现：

给你一个i，如何快速的知道prefix sum从1到i - 1的和是多少

比如现在遇到一个i，如何快速的让其位置上所代表的桶nums[i]的count加1？

e.g 比如给定i是5，如何快速知道prefix sum从1到4的和是2？即桶1到桶4的count的和是2？

e.g 比如给定i是3，对于nums = [5,2,6,1]，如何快速让其代表的桶nums[3]即桶1的count加1？

用英文表达就是：

If current index is i, we need:

Function 1: get the prefix sum(i - 1)

Function 2: count(nums[i]) + 1

 其实我们早就解过类似的题目了，就是L307. Range Sum Query - Mutable，L315这里稍微有点不一样：

在L307中就是给你一个i和j，你想知道i到j之间的sum，然后就是我们想要做修改，我们想让坐标为i的位置的数有个delta为k的变化，用英文表达如下：

Binary indexed tree in L307:

Function 1: query(i, j): get the sum between (i, j)

Function 2: update(i, k): add k to nums[i]

实际上对应L315中的情况就是对于L307中的query(i, j)由于L315中i直接恒定为0，而j变为i - 1，即query(i, j)变为query(0, i - 1)，对于L307中的update(i, k)由于L307中k直接变为遇到一个数的时候增加的频率恒定为1，即update(i, k)变为update(count[i], 1)

L315和L307之间的关系如下：

Function 1 <- query(0, i - 1)

Function 2 <- update(count[i], 1)

 所以我们完全可以套用L307. Range Sum Query - Mutable 的方法来解L315

**当然，现在还有个悬而未决的问题，为何要用HashSet过滤原数组来获得桶？而不是直接在原数组index基础上一一对应设置桶？**

现在我们样例中的桶bucket只是从1开到了6，但是如果这个数非常的大，不可能开出一个巨大的用来存桶的所有bucket的存储空间，或者有负数，不能作为index，则需要做映射关系mapping

e.g nums = [-3,-2,-4] 全都是负数如何知道bucket对应为几号桶呢？

思路：首先要知道nums中有多少个独特的数，可以用hashset来实现这一部，然后我们需要排序sort，为什么要sort呢？我们query的时候需要这个数在原始数组中的index，我们update的时候也需要这个数在原始数组nums的index，那么我们怎么能快速得到某个数的index？首先想到的就是Binary Search，让时间复杂度降低到logN，而先sort原有数组是使用Binary Search的基础

First sort -> nums\_sorted = [-4,-3,-2] how to quickly get index of -3 ?

Arrays.binarySearch(nums\_sorted, -3) = 1 -> 得到-3在sort后的nums中的index为1

**接下来是整个流程到代码的过程：**

**首先我们要map一下，原因就是上面提到的超大规模输入或者负数导致无法建立1到n的数组，我们需要建立一个a数组int[] a，不过由于不知道数组开多大，所以我们需要用Set先过滤原有数组nums来获得所有unique元素的个数，设为a的大小，然后给每个数一个index**

class Solution {

    int[] a;

    public List<integer> countSmaller(int[] nums) {

        map(nums);

    }

    private void map(int[] nums) {

        // Rebuild original input nums array with unique value only

        Set<Integer> numSet = new HashSet<>();

        for(int num : nums) {

            numSet.add(num);

        }

        a = new int[numSet.size()];

        // Assign each number an index, like a bucket

        int index = 0;

        for(int num : numSet) {

            a[index++] = num;

        }

        // Have to sort to quickly get each number's index

        Arrays.sort(a);

    }

}

**接下来还需要设置一个数组count，因为我们必须知道每个数出现了多少次，count数组本质上存的是每个数在原数组中出现的频率，而且只有在知道了每个数的出现的频率后才能计算每个数的presum，然后我们就开始从右向左遍历原数组，并且找到原数组中每个数在BIT中对应的index(id)，当然也由于是从右向左遍历，原数组中最右边的数对应的右边比它小的数字个数会先添加到结果中，而结果应该遵循原输入数组的顺序从左向右，所以我们的结果必须要reverse一下**

class Solution {

    int[] a;

    int[] count;

    public List<integer> countSmaller(int[] nums) {

        map(nums);

        // Need a 'count' array to record each number's frequency

        // in original input nums

        count = new int[a.length];

        List<Integer> res = new ArrayList<>();

        // Scan original input nums from right to left

        for(int i = nums.length - 1; i >= 0; i--) {

            // To find index(id) of nums[i] in sorted array(a)

            // Need to '+ 1' because binarySearch is 0-index

            // based but BIT is 1-index based

            int id = Arrays.binarySearch(a, nums[i]) + 1;

        }

        // Because we scan from right to left on original input nums,

        // the result always appending for the rightmost number first,

        // to display as left to right order in result, we have to

        // reverse the 'res' list

        Collections.reverse(res);

        return res;

    }

    private void map(int[] nums) {

        // Rebuild original input nums array with unique value only

        Set<Integer> numSet = new HashSet<>();

        for(int num : nums) {

            numSet.add(num);

        }

        a = new int[numSet.size()];

        // Assign each number an index, like a bucket

        int index = 0;

        for(int num : numSet) {

            a[index++] = num;

        }

        // Have to sort to quickly get each number's index

        Arrays.sort(a);

    }

}

**接下来具体实现了query和update还有lowBit三个方法**

// Below code even accept by OJ, it still need a bit revise (point 1 & 2) to improve the table view

// 1.'count' array is 'BIT' array, we may need to reformat 'count' initialize same as 'BIT' way

// by adding 1 padding slot ahead to make it 1-index based

// 2. L307 take care of 1 more slot and handle i == n case in while loop condition as while(i <= n),

// but for L315 since only while(i < count.length) and int[] count = new int[a.length], not include i == count.length case, it will encounter issue on table view when i >= count.length

// 3.'res' in L315 query() equal to 'sum' in L307 getSum(), also equal to 'presum' in table view

// 4. 1 in L315 update equal to 'val' in L307 init()

/\*\*

L307

    public NumArray(int[] nums) {

        this.nums = nums;

        n = nums.length;

        BIT = new int[n + 1];

        for (int i = 0; i < n; i++)

            init(i, nums[i]);

    }

    public void init(int i, int val) {

        i++;

        while (i <= n) {

            BIT[i] += val;

            i += (i & -i);

        }

    }

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

L315

    private void update(int i) {

        while(i < count.length) {

            count[i] += 1;

            i += lowBit(i);

        }

    }

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

L307

    private int getSum(int i) {

        int sum = 0;

        // Shift 1 for match BIT array indexes as [1, n] rather than [0, n - 1]

        i++;

        while(i > 0) {

            sum += BIT[i];

            // Just reverse the logic how BIT build up to find path from given

            // node and trace back to top node

            i -= (i & -i);

        }

        return sum;

    }

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

L315

    private int query(int i) {

        int res = 0;

        while(i > 0) {

            res += count[i];

            i -= lowBit(i);

        }

        return res;

    }

\*/

class Solution {

    int[] a;

    int[] count;

    public List<Integer> countSmaller(int[] nums) {

        map(nums);

        // Need a 'count' array to record each number's frequency

        // in original input nums

        count = new int[a.length];

        List<Integer> res = new ArrayList<>();

        // Scan original input nums from right to left

        for(int i = nums.length - 1; i >= 0; i--) {

            // Find which bucket(id) for nums[i] in sorted

            // array(a), need to '+ 1' because binarySearch

            // is 0-index based but BIT is 1-index based

            // e.g nums = [5,2,6,1], when we search for 5

            // in sorted array [1,2,5,6] and + 1 will get

            // id = 3, means the 3rd bucket

            int id = Arrays.binarySearch(a, nums[i]) + 1;

            // We need to query 'id - 1', the '- 1' means

            // presum range for bucket(=id) end at 'id - 1'

            // e.g nums = [5,2,6,1], when we query for 5, we

            // want to know the count sum of bucket 1 to 4,

            // the ending bucket 4 is coming from (5 - 1)

            // Note: not require all buckets exist on a range

            res.add(query(id - 1));

            // We also need to update the count for current bucket

            // e.g nums = [5,2,6,1], when we encounter 5, we

            // have to update count of bucket 3 by increase 1

            update(id);

        }

        // Because we scan from right to left on original input nums,

        // the result always appending for the rightmost number first,

        // to display as left to right order in result, we have to

        // reverse the 'res' list

        Collections.reverse(res);

        return res;

    }

    private int lowBit(int x) {

        return x & (-x);

    }

    // Given an index 'i', find all previous count sum

    private int query(int i) {

        int res = 0;

        while(i > 0) {

            res += count[i];

            i -= lowBit(i);

        }

        return res;

    }

    private void update(int i) {

        while(i < count.length) {

            count[i] += 1;

            i += lowBit(i);

        }

    }

    private void map(int[] nums) {

        // Rebuild original input nums array with unique value only

        Set<Integer> numSet = new HashSet<>();

        for(int num : nums) {

            numSet.add(num);

        }

        a = new int[numSet.size()];

        // Assign each number an index, like a bucket

        int index = 0;

        for(int num : numSet) {

            a[index++] = num;

        }

        // Have to sort to quickly get each number's index

        Arrays.sort(a);

    }

}

**现在再来跑一遍nums = [5,2,6,1]，与之前理论上构建的bucket从1到6都有不同，在实际解法的map(nums)思路中，我们会抛弃中间不存在的数所对应的bucket，比如nums = [5,2,6,1]这个输入中没有3和4，所以这两个bucket我们并不需要，bucket编号对应上面代码中的id**

**Round 1:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Current number is 1 |  |  |  |  |
| bucket\_id(num) | 1(num=1) | 2(num=2) | 3(num=5) | 4(num=6) |
| count | 1 | 1 |  |  |
| presum | 0 |  |  |  |

Step 1: Find bucket id

i = nums.length - 1 = 4 - 1 = 3

int id = Arrays.binarySearch(a, nums[i]) + 1

-> Arrays.binarySearch(a, nums[3]) + 1

-> Arrays.binarySearch(a, 1) + 1

-> id = 1

So we are dealing with 1st bucket (id = 1)

Step 2: Query presum of bucket at id, which is the count sum up on range bucket[1, id - 1], and the actual BIT structure impacted bucket ids used for bucket[id]’s presum must follow i -= lowBit(i) => i -= i & -i to retrieve, it will skip some ids between [1, id - 1]

res.add(query(id - 1))

-> res.add(query(0))

-> given bucket(id) = 1, find presum of bucket 1, previous bucket range not exist

-> presum = 0

-> res = {0}

So the 1st bucket (id = 1) have presum = 0

Step 3: Update count of bucket at id and its BIT structure impacted bucket ids must follow i += lowBit(i) => i += i & -i to retrieve, it means not only the given id bucket will update count, may also update other ids’ bucket count by increase 1 (frequency)

update(id)

-> update(1)

-> count[1] += 1 -> {0,1,0,0}

-> i += i & -i = 1 + 1 & -1 = 2

-> count[2] += 1 -> {0,1,1,0}

-> i += i & -i = 2 + 2 & -2 = 4 break out while loop

**Round 2:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Current number is 6 |  |  |  |  |
| bucket\_id(num) | 1(num=1) | 2(num=2) | 3(num=5) | 4(num=6) |
| count | 1 | 1 |  |  |
| presum | 0 |  |  | 1 |

Step 1: Find bucket id

i = Round 1: i (= 3) - 1 = 2

int id = Arrays.binarySearch(a, nums[i]) + 1

-> Arrays.binarySearch(a, nums[2]) + 1

-> Arrays.binarySearch(a, 6) + 1

-> id = 4

So we are dealing with 4th bucket (id = 4)

Step 2: Query presum of bucket at id, which is the count sum up on range bucket[1, id - 1], and the actual BIT structure impacted bucket ids used for bucket[id]’s presum must follow i -= lowBit(i) => i -= i & -i to retrieve, it will skip some ids between [1, id - 1]

res.add(query(id - 1))

-> res.add(query(3))

-> given bucket(id) = 4, find presum of bucket 4, previous bucket range [1, 3], in BIT structure, it means count[3] (cover bucket[3]) + count[2] (cover bucket[1, 2]) to cover the range [1, 3], here is how presum calculated from BIT range coverage, not a plain relation in previous one pass scanning demo (for presum[i] scanning count[1, i - 1], sum all of them up in a single run) which spend O(N)

-> presum += count[3] -> presum = 0

-> i -= i & -i = 3 - 3 & -3 = 2

-> presum += count[2] -> presum = 1

-> i -= i & -i = 2 - 2 & -2 = 0 break out while loop

-> res = {0,1}

So the 4th bucket (id = 4) have presum = 1

Explain the coverage of BIT structure:

e.g, the node that represents position 4 will contain information from index 0 to index 3 => [0,3] in the original array, so to cover original array range [0, 2] or bit range [1, 3], just need bit[2] + bit[3]

                                      root

                                      0000(0)

            /                |                        |                \

          0001(1)            0010(2)                  0100(4)            1000(8)

          [0,0]              [0,1]                      [0,3]              [0,7]

                                |                        |      \

                              0011(3)                  0101(5) 0110(6)

                              [2,2]                    [4,4]  [4,5]

                                                          |

                                                      0111(7)

                                                        [6,6]

Coverage relation

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

              \_\_\_\_\_\_\*

              \_\_\*    \_\_\*

              \*  \*  \*  \*

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

arr indexes:  0 1 2 3 4 5 6 7

Step 3: Update count of bucket at id and its BIT structure impacted bucket ids must follow i += lowBit(i) => i += i & -i to retrieve, it means not only the given id bucket will update count, may also update other ids’ bucket count by increase 1 (frequency)

update(id)

-> update(4)

-> i = 4, length exceed

-> count keep as {0,1,1,0}

**Round 3:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Current number is 2 |  |  |  |  |
| bucket\_id(num) | 1(num=1) | 2(num=2) | 3(num=5) | 4(num=6) |
| count | 1 | 2 |  |  |
| presum | 0 | 1 |  | 1 |

Step 1: Find bucket id

i = Round 2: i (= 2) - 1 = 1

int id = Arrays.binarySearch(a, nums[i]) + 1

-> Arrays.binarySearch(a, nums[1]) + 1

-> Arrays.binarySearch(a, 2) + 1

-> id = 2

So we are dealing with 2nd bucket (id = 2)

Step 2: Query presum of bucket at id, which is the count sum up on range bucket[1, id - 1], and the actual BIT structure impacted bucket ids used for bucket[id]’s presum must follow i -= lowBit(i) => i -= i & -i to retrieve, it will skip some ids between [1, id - 1]

res.add(query(id - 1))

-> res.add(query(1))

-> given bucket(id) = 2, find presum of bucket 2, previous bucket range [1]

-> presum += count[1] -> presum = 1

-> i -= i & -i = 1 - 1 & -1 = 0 break out while loop

-> res = {0,1,1}

So the 2nd bucket (id = 2) have presum = 1

Step 3: Update count of bucket at id and its BIT structure impacted bucket ids must follow i += lowBit(i) => i += i & -i to retrieve, it means not only the given id bucket will update count, may also update other ids’ bucket count by increase 1 (frequency)

update(id)

-> update(2)

-> count[2] += 1 -> {0,1,2,0}

-> i += i & -i = 2 + 2 & -2 = 4 break out while loop

**Round 4:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Current number is 5 |  |  |  |  |
| bucket\_id(num) | 1(num=1) | 2(num=2) | 3(num=5) | 4(num=6) |
| count | 1 | 2 | 1 |  |
| presum | 0 | 1 | 2 | 1 |

Step 1: Find bucket id

i = Round 3: i (= 1) - 1 = 0

int id = Arrays.binarySearch(a, nums[i]) + 1

-> Arrays.binarySearch(a, nums[0]) + 1

-> Arrays.binarySearch(a, 5) + 1

-> id = 3

So we are dealing with 3rd bucket (id = 3)

Step 2: Query presum of bucket at id, which is the count sum up on range bucket[1, id - 1], and the actual BIT structure impacted bucket ids used for bucket[id]’s presum must follow i -= lowBit(i) => i -= i & -i to retrieve, it will skip some ids between [1, id - 1]

res.add(query(id - 1))

-> res.add(query(2))

-> given bucket(id) = 3, find presum of bucket 3, previous bucket range [1, 2]

-> presum += count[2] -> presum = 2

-> i -= i & -i = 2 - 2 & -2 = 0 break out while loop

-> res = {0,1,1,2}

So the 3rd bucket (id = 3) have presum = 2

Step 3: Update count of bucket at id and its BIT structure impacted bucket ids must follow i += lowBit(i) => i += i & -i to retrieve, it means not only the given id bucket will update count, may also update other ids’ bucket count by increase 1 (frequency)

update(id)

-> update(3)

-> count[3] += 1 -> {0,1,2,1}

-> i += i & -i = 3 + 3 & -3 = 4 break out while loop

Finally we reverse the res to get the answer:

Collections.reverse(res) = {2,1,1,0}, so for original input nums = [5,2,6,1] the result is [2,1,1,0]

**但是上面的推导过程不是特别严谨，尤其是count的处理上在table view里面存在明显的缺陷，而且整体代码风格不是很符合L307中定义好的BIT构建模版，下面是我们对各方面做了改进以后获得的完美符合BIT构建模版的代码：**

// Below code even accept by OJ, it still need a bit revise (point 1 & 2) to improve the table view

// 1.'count' array is 'BIT' array, we may need to reformat 'count' initialize same as 'BIT' way

// by adding 1 padding slot ahead to make it 1-index based

// 2. L307 take care of 1 more slot and handle i == n case in while loop condition as while(i <= n),

// but for L315 since only while(i < count.length) and int[] count = new int[a.length], not include i == count.length case, it will encounter issue on table view when i >= count.length

// 3.'res' in L315 query() equal to 'sum' in L307 getSum(), also equal to 'presum' in table view

// 4. 1 in L315 update equal to 'val' in L307 init()

/\*\*

L307

    public NumArray(int[] nums) {

        this.nums = nums;

        n = nums.length;

        BIT = new int[n + 1];

        for (int i = 0; i < n; i++)

            init(i, nums[i]);

    }

    public void init(int i, int val) {

        i++;

        while (i <= n) {

            BIT[i] += val;

            i += (i & -i);

        }

    }

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

L315

    private void update(int i) {

        while(i < count.length) {

            count[i] += 1;

            i += lowBit(i);

        }

    }

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

L307

    private int getSum(int i) {

        int sum = 0;

        // Shift 1 for match BIT array indexes as [1, n] rather than [0, n - 1]

        i++;

        while(i > 0) {

            sum += BIT[i];

            // Just reverse the logic how BIT build up to find path from given

            // node and trace back to top node

            i -= (i & -i);

        }

        return sum;

    }

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

L315

    private int query(int i) {

        int res = 0;

        while(i > 0) {

            res += count[i];

            i -= lowBit(i);

        }

        return res;

    }

\*/

class Solution {

    int[] a;

    int[] count;

    public List<Integer> countSmaller(int[] nums) {

        map(nums);

        // Need a 'count' array to record each number's frequency

        // in original input nums

        //count = new int[a.length];

        // Based on point 1: 'count' is actually BIT structured array,

        // since BIT structured array is 1-indexed, follow standard

        // creation of BIT structured array, we add padding position

        // of index 0, hence length change to original array length + 1

        count = new int[a.length + 1];

        List<Integer> res = new ArrayList<>();

        // Scan original input nums from right to left

        for(int i = nums.length - 1; i >= 0; i--) {

            // Find which bucket(id) for nums[i] in sorted array(a), but no

            // need to '+ 1' since we will '+ 1' in both 'query' and 'update'

            // method later, the '+ 1' because binarySearch is 0-index based,

            // but BIT is 1-index based

            // e.g nums = [5,2,6,1], when we search for 5 in sorted array

            // [1,2,5,6] and '+ 1' will get bucket id = 3, means the 3rd bucket

            // but we only need original index = 2 for 5 here, and in 'query'

            // and 'update' method we will make '+ 1' padding, so no '+ 1' here

            //int id = Arrays.binarySearch(a, nums[i]) + 1;

            int id = Arrays.binarySearch(a, nums[i]);

            // We need to query 'id - 1', the '- 1' means presum range for

            // bucket(=id) end at 'id - 1'

            // e.g For nums = [5,2,6,1] after mapping we skip non-exist bucket

            // as bucket 3 and 4, so 1 mapping to bucket 1(=count[1]), 2 mapping

            // to bucket 2(=count[2]), 5 mapping to bucket 3(=count[3]), 6 mapping

            // to bucket 4(=count[3]), count[0] is the padding, when we query

            // all smaller number count on right of 5 equals to find all related

            // buckets under BIT structured count sum before 5's mapping bucket,

            // be careful, as 5 mapping to bucket 3, its not necessarily mean we

            // have to sum up both bucket 1 and bucket 2 count as a linear scan,

            // what we have to follow up is only based on BIT structured relation,

            // and consider each cell coverage definition in BIT we only add bucket

            // 2 count, because bucket 2 count already include bucket 1 count, and

            // the reason we come to BIT way is because BIT only spend O(logN) on

            // query and update, but linear scan will take O(N)

            res.add(query(id - 1));

            // We also need to update the count for current bucket

            // e.g nums = [5,2,6,1], when we encounter 5, we have to update count

            // of bucket 3 by increase 1

            update(id);

        }

        // Because we scan from right to left on original input nums,

        // the result always appending for the rightmost number first,

        // to display as left to right order in result, we have to

        // reverse the 'res' list

        Collections.reverse(res);

        return res;

    }

    private int lowBit(int x) {

        return x & (-x);

    }

    // Given an index 'i', find all previous count sum

    private int query(int i) {

        int res = 0;

        // Padding 1 to match BIT array indexes as [1, n] rather than [0, n - 1]

        i++;

        while(i > 0) {

            res += count[i];

            i -= lowBit(i);

        }

        return res;

    }

    private void update(int i) {

        // Padding 1 to match BIT array indexes as [1, n] rather than [0, n - 1]

        i++;

        while(i < count.length) {

            count[i] += 1;

            i += lowBit(i);

        }

    }

    private void map(int[] nums) {

        // Rebuild original input nums array with unique value only

        Set<Integer> numSet = new HashSet<>();

        for(int num : nums) {

            numSet.add(num);

        }

        a = new int[numSet.size()];

        // Assign each number an index, like a bucket

        int index = 0;

        for(int num : numSet) {

            a[index++] = num;

        }

        // Have to sort to quickly get each number's index

        Arrays.sort(a);

    }

}

**接下来是做了更好的命名改进和自我实现的binary search方法替换后得到的代码：**

class Solution {

    int[] copy;

    int[] bit;

    public List<Integer> countSmaller(int[] nums) {

        filterUniqueValueAndSort(nums);

        // Need a 'bit' array to record each number's frequency

        // in original input nums

        // Based on point 1: 'bit' is actually BIT structured array,

        // since BIT structured array is 1-indexed, follow standard

        // creation of BIT structured array, we add padding position

        // of index 0, hence length change to original array length + 1

        bit = new int[copy.length + 1];

        List<Integer> result = new ArrayList<>();

        // Scan original input nums from right to left

        for(int i = nums.length - 1; i >= 0; i--) {

            // Find which bucket(id) for nums[i] in sorted array, but no

            // need to '+ 1' since we will '+ 1' in both 'query' and 'update'

            // method later, the '+ 1' because binarySearch is 0-index based,

            // but BIT is 1-index based

            // Relation: bucket\_id\_before\_padding + 1 = bucket\_id\_after\_padding

            // e.g nums = [5,2,6,1], when we search for 5 in sorted array

            // [1,2,5,6] will get bucket\_id\_before\_padding = 2 and '+ 1' will

            // get bucket\_id\_after\_padding = 3, means the 3rd bucket

            // but we only need original bucket\_id\_before\_padding = 2 for 5 here,

            // and in 'query' and 'update' method we will make '+ 1' padding,

            // so no '+ 1' here

            //int bucket\_id\_after\_padding = Arrays.binarySearch(copy, nums[i]) + 1;

            //int bucket\_id\_before\_padding = Arrays.binarySearch(copy, nums[i]);

            // Use customized 'findNumIndexInSortedArray' method to replace build in one

            int bucket\_id\_before\_padding = findNumIndexInSortedArray(copy, nums[i]);

            // We need to query 'bucket\_id\_before\_padding - 1', the '- 1' means

            // presum range for buckets ids end at 'bucket\_id\_before\_padding - 1'

            // e.g

            // NOTE: Below based on bucket\_id\_after\_padding

            // For nums = [5,2,6,1] after mapping we skip non-exist buckets

            // as bucket 3 and 4, so 1 mapping to bucket 1(=bit[1]), 2 mapping

            // to bucket 2(=bit[2]), 5 mapping to bucket 3(=bit[3]), 6 mapping

            // to bucket 4(=bit[4]), bit[0] is the padding, when we query

            // all smaller number count on right of 5 equals to find all related

            // buckets under BIT structured count sum before 5's mapping bucket,

            // be careful, as 5 mapping to bucket 3, its not necessarily mean we

            // have to sum up both bucket 1 and bucket 2 count as a linear scan,

            // what we have to follow up is only based on BIT structured relation,

            // and consider each cell coverage definition in BIT we only add bucket

            // 2 count, because bucket 2 count already include bucket 1 count, and

            // the reason we come to BIT way is because BIT only spend O(logN) on

            // query and update, but linear scan will take O(N)

            result.add(query(bucket\_id\_before\_padding - 1));

            // We also need to update the count for current bucket

            // e.g nums = [5,2,6,1], when we encounter 5, we have to update count

            // of bucket 3 by increase 1

            update(bucket\_id\_before\_padding);

        }

        // Because we scan from right to left on original input nums,

        // the result always appending for the rightmost number first,

        // to display as left to right order in result, we have to

        // reverse the 'result' list

        Collections.reverse(result);

        return result;

    }

    // Given an index 'i', find all previous count sum

    private int query(int i) {

        int presum = 0;

        // Padding 1 to match BIT array indexes as [1, n] rather than [0, n - 1]

        // Passed in 'i' is bucket\_id\_before\_padding, after 'i++' is bucket\_id\_after\_padding

        i++;

        while(i > 0) {

            presum += bit[i];

            i -= i & -i;

        }

        return presum;

    }

    private void update(int i) {

        // Padding 1 to match BIT array indexes as [1, n] rather than [0, n - 1]

        // Passed in 'i' is bucket\_id\_before\_padding, after 'i++' is bucket\_id\_after\_padding

        i++;

        while(i < bit.length) {

            bit[i] += 1;

            i += i & -i;

        }

    }

    private void filterUniqueValueAndSort(int[] nums) {

        // Rebuild original input nums array with unique value only

        Set<Integer> numSet = new HashSet<>();

        for(int num : nums) {

            numSet.add(num);

        }

        copy = new int[numSet.size()];

        // Assign each number an index, like a bucket

        int index = 0;

        for(int num : numSet) {

            copy[index++] = num;

        }

        // Have to sort to quickly get each number's index

        Arrays.sort(copy);

    }

    // Find upper boundary

    private int findNumIndexInSortedArray(int[] arr, int val) {

        // Find largest element in arr which <= val

        int l = 0, r = arr.length - 1, m = 0;

        while(l <= r) {

            m = l + ((r - l) >> 1);

            if(arr[m] > val) {

                r = m - 1;

            } else {

                l = m + 1;

            }

        }

        // Corner case [-5, -5]

        if(l == 0 && arr[l] > val) {

            l--;

        }

        //return l + 1; -> this is wrong

        // The 'l - 1' (or 'r') is the actual upper boundary if we

        // consider condition 'if(arr[m] > val)', the upper boundary

        // means the last element index in arr that equal or less

        // than 'val'(<= val), the '+ 1' is the mandatory shift up

        // 1 position required for 1-based BIT array

        //return l - 1 + 1; // or 'r + 1' equally

        // Note: in this solution no need '+ 1' as we exactly only

        // want to find last element index which equal to given 'val'

        return l - 1;

    }

}

**现在再来跑一遍nums = [5,2,6,1]，我们会发现与之前的table view有明显区别**

**Round 1:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Current number is 1 |  |  |  |  |  |
| bucket id after padding(num) | 0(padding) | 1(num=1) | 2(num=2) | 3(num=5) | 4(num=6) |
| bit | 0(padding) | 1 | 1 | 0 | 1 |
| presum | 0(padding) | 0 |  |  |  |

Step 1: Find bucket id before padding (and + 1 in later query and update method will be bucket id after padding)

i = nums.length - 1 = 4 - 1 = 3

int bucket\_id\_before\_padding = findNumIndexInSortedArray(copy, nums[i])

-> findNumIndexInSortedArray(copy, nums[3])

-> findNumIndexInSortedArray(copy, 1)

-> bucket\_id\_before\_padding = 0

So we will dealing with bucket 1 (bucket\_id\_after\_padding) in later query and update method

Step 2: Query presum of bucket at bucket\_id\_after\_padding, which is the count sum up on range bucket[1, bucket\_id\_after\_padding - 1], and the actual BIT structure impacted bucket ids used for bucket[bucket\_id\_after\_padding]’s presum must follow i -= lowBit(i) => i -= i & -i to retrieve, it will skip some ids between [1, bucket\_id\_after\_padding - 1]

result.add(query(bucket\_id\_before\_padding - 1))

-> result.add(query(0))

-> bucket\_id\_before\_padding(0) + 1 = bucket\_id\_after\_padding (1)

-> given bucket(bucket\_id\_after\_padding) = 1, find presum of bucket 1, previous bucket range not exists as bucket 0 is used for padding bit array into 1-indexed

-> presum = 0

-> result = {0}

So the bucket 1 have presum = 0

Step 3: Update count of bucket at id and its BIT structure impacted bucket ids must follow i += lowBit(i) => i += i & -i to retrieve, it means not only the given id bucket will update count, may also update other ids’ bucket count by increase 1 (frequency)

update(bucket\_id\_before\_padding)

-> update(0)

-> bucket\_id\_before\_padding(0) + 1 = bucket\_id\_after\_padding (1)

-> bit[1] += 1 -> {0,1,0,0,0}

-> i += i & -i = 1 + 1 & -1 = 2

-> bit[2] += 1 -> {0,1,1,0,0}

-> i += i & -i = 2 + 2 & -2 = 4

-> bit[4] += 1 -> {0,1,1,0,1}

-> i += i & -i = 4 + 4 & -4 = 8 break out while loop

**Round 2:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Current number is 6 |  |  |  |  |  |
| bucket id after padding(num) | 0(padding) | 1(num=1) | 2(num=2) | 3(num=5) | 4(num=6) |
| bit | 0(padding) | 1 | 1 | 0 | 2 |
| presum | 0(padding) | 0 |  |  | 1 |

Step 1: Find bucket id before padding (and + 1 in later query and update method will be bucket id after padding)

i = Round 1: i (= 3) - 1 = 2

int bucket\_id\_before\_padding = findNumIndexInSortedArray(copy, nums[i])

-> findNumIndexInSortedArray(copy, nums[2])

-> findNumIndexInSortedArray(copy, 6)

-> bucket\_id\_before\_padding = 3

So we will dealing with bucket 4 (bucket\_id\_after\_padding) in later query and update method

Step 2: Query presum of bucket at bucket\_id\_after\_padding, which is the count sum up on range bucket[1, bucket\_id\_after\_padding - 1], and the actual BIT structure impacted bucket ids used for bucket[bucket\_id\_after\_padding]’s presum must follow i -= lowBit(i) => i -= i & -i to retrieve, it will skip some ids between [1, bucket\_id\_after\_padding - 1]

result.add(query(bucket\_id\_before\_padding - 1))

-> result.add(query(3 - 1)) = result.add(query(2))

-> bucket\_id\_before\_padding(2) + 1 = bucket\_id\_after\_padding (3)

-> given bucket(bucket\_id\_after\_padding) = 4, find presum of bucket 4, previous bucket range [1, 3], in BIT structure, it means bit[3] (cover bucket[3]) + bit[2] (cover bucket[1, 2]) to cover the range [1, 3], here is how presum calculated from BIT range coverage, not a plain relation in previous one pass scanning demo (for presum[i] scanning count[1, bucket\_id\_after\_padding - 1], sum all of them up in a single run) which spend O(N)

-> presum += bit[3] -> presum = 0

-> i -= i & -i = 3 - 3 & -3 = 2

-> presum += bit[2] -> presum = 1

-> i -= i & -i = 2 - 2 & -2 = 0 break out while loop

-> result = {0,1}

So the bucket 4 we have presum = 1

Step 3: Update count of bucket at id and its BIT structure impacted bucket ids must follow i += lowBit(i) => i += i & -i to retrieve, it means not only the given id bucket will update count, may also update other ids’ bucket count by increase 1 (frequency)

update(bucket\_id\_before\_padding)

-> update(3)

-> bucket\_id\_before\_padding(3) + 1 = bucket\_id\_after\_padding (4)

-> bit[4] += 1 -> {0,1,1,0,2}

-> i += i & -i = 4 + 4 & -4 = 8 break out while loop

**Round 3:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Current number is 2 |  |  |  |  |  |
| bucket id after padding(num) | 0(padding) | 1(num=1) | 2(num=2) | 3(num=5) | 4(num=6) |
| bit | 0(padding) | 1 | 2 | 0 | 3 |
| presum | 0(padding) | 0 | 1 |  | 1 |

Step 1: Find bucket id before padding (and + 1 in later query and update method will be bucket id after padding)

i = Round 2: i (= 2) - 1 = 1

int bucket\_id\_before\_padding = findNumIndexInSortedArray(copy, nums[i])

-> findNumIndexInSortedArray(copy, nums[1])

-> findNumIndexInSortedArray(copy, 2)

-> bucket\_id\_before\_padding = 1

So we will dealing with bucket 2 (bucket\_id\_after\_padding) in later query and update method

Step 2: Query presum of bucket at bucket\_id\_after\_padding, which is the count sum up on range bucket[1, bucket\_id\_after\_padding - 1], and the actual BIT structure impacted bucket ids used for bucket[bucket\_id\_after\_padding]’s presum must follow i -= lowBit(i) => i -= i & -i to retrieve, it will skip some ids between [1, bucket\_id\_after\_padding - 1]

result.add(query(bucket\_id\_before\_padding - 1))

-> result.add(query(1 - 1)) = result.add(query(0))

-> bucket\_id\_before\_padding(1) + 1 = bucket\_id\_after\_padding (2)

-> given bucket(bucket\_id\_after\_padding) = 2, find presum of bucket 2, previous bucket range [1], in BIT structure, it means bit[1] (cover bucket[1]) to cover the range [1], here is how presum calculated from BIT range coverage, not a plain relation in previous one pass scanning demo (for presum[i] scanning count[1, bucket\_id\_after\_padding - 1], sum all of them up in a single run) which spend O(N)

-> presum += bit[1] -> presum = 1

-> i -= i & -i = 1 - 1 & -1 = 0 break out while loop

-> result = {0,1,1}

So the bucket 2 we have presum = 1

Step 3: Update count of bucket at id and its BIT structure impacted bucket ids must follow i += lowBit(i) => i += i & -i to retrieve, it means not only the given id bucket will update count, may also update other ids’ bucket count by increase 1 (frequency)

update(bucket\_id\_before\_padding)

-> update(1)

-> bucket\_id\_before\_padding(1) + 1 = bucket\_id\_after\_padding (2)

-> bit[2] += 1 -> {0,1,2,0,2}

-> i += i & -i = 2 + 2 & -2 = 4

-> bit[4] += 1 -> {0,1,2,0,3}

-> i += i & -i = 4 + 4 & -4 = 8 break out while loop

**Round 4:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Current number is 5 |  |  |  |  |  |
| bucket id after padding(num) | 0(padding) | 1(num=1) | 2(num=2) | 3(num=5) | 4(num=6) |
| bit | 0(padding) | 1 | 2 | 1 | 4 |
| presum | 0(padding) | 0 | 1 | 2 | 1 |

Step 1: Find bucket id before padding (and + 1 in later query and update method will be bucket id after padding)

i = Round 3: i (= 1) - 1 = 0

int bucket\_id\_before\_padding = findNumIndexInSortedArray(copy, nums[i])

-> findNumIndexInSortedArray(copy, nums[0])

-> findNumIndexInSortedArray(copy, 5)

-> bucket\_id\_before\_padding = 2

So we will dealing with bucket 3 (bucket\_id\_after\_padding) in later query and update method

Step 2: Query presum of bucket at bucket\_id\_after\_padding, which is the count sum up on range bucket[1, bucket\_id\_after\_padding - 1], and the actual BIT structure impacted bucket ids used for bucket[bucket\_id\_after\_padding]’s presum must follow i -= lowBit(i) => i -= i & -i to retrieve, it will skip some ids between [1, bucket\_id\_after\_padding - 1]

result.add(query(bucket\_id\_before\_padding - 1))

-> result.add(query(2 - 1)) = result.add(query(1))

-> bucket\_id\_before\_padding(2) + 1 = bucket\_id\_after\_padding (3)

-> given bucket(bucket\_id\_after\_padding) = 3, find presum of bucket 3, previous bucket range [1,2], in BIT structure, it means bit[2] (cover bucket[1] and bucket[2]) to cover the range [1,2], here is how presum calculated from BIT range coverage, not a plain relation in previous one pass scanning demo (for presum[i] scanning count[1, bucket\_id\_after\_padding - 1], sum all of them up in a single run) which spend O(N)

-> presum += bit[2] -> presum = 2

-> i -= i & -i = 2 - 2 & -2 = 0 break out while loop

-> result = {0,1,1,2}

So the bucket 3 we have presum = 2

Step 3: Update count of bucket at id and its BIT structure impacted bucket ids must follow i += lowBit(i) => i += i & -i to retrieve, it means not only the given id bucket will update count, may also update other ids’ bucket count by increase 1 (frequency)

update(bucket\_id\_before\_padding)

-> update(2)

-> bucket\_id\_before\_padding(2) + 1 = bucket\_id\_after\_padding (3)

-> bit[3] += 1 -> {0,1,2,1,2}

-> i += i & -i = 3 + 3 & -3 = 4

-> bit[4] += 1 -> {0,1,2,1,4}

-> i += i & -i = 4 + 4 & -4 = 8 break out while loop

So finally we get the final presum array for scanning input {5,2,6,1} from right to left as result = {0,1,1,2}, then we revert the order to get scanning input from left ot right result as {2,1,1,0}

进一步问题和改进：

没有必要用hashset找独立元素来构建BIT(bucket)，但是这个并不是优化，虽然不影响最终结果，但是导致了BIT(bucket)的构建含有了重复元素导致的无效位置，在遍历长度上增长导致了更大的时间消耗

不要用global variable

Binary Search到底是找最后一个 <= val的还是找第一个 >= val的？ 用[5,6,2,6,1]或者[1,3,2,3,1]验证？根据L493的测试结果，其实前面的推导中一直使用找最后一个 <= val的做法是错误的，必须是找第一个 >= val，所以Binary Search 必须按照Find Lower Boundary的模版来重构，所以L315我们也改用Find Lower Boundary

猜想：在L493中如果也使用hashset应该不会遇到Binary Search返回index在找最后一个和在找第一个的时候不一致的问题，因为每个值在基于hashset建立的数组中也就是bucket中只会有一个，不会有多个bucket对应同一个值导致寻找某值第一个和最后一个的位置结果不同

综上：去掉global variable和Binary Search改用Find Lower Boundary：

class Solution {

    public List<Integer> countSmaller(int[] nums) {

        int[] copy = filterUniqueValueAndSort(nums);

        // Need a 'bit' array to record each number's frequency

        // in original input nums

        // Based on point 1: 'bit' is actually BIT structured array,

        // since BIT structured array is 1-indexed, follow standard

        // creation of BIT structured array, we add padding position

        // of index 0, hence length change to original array length + 1

        int[] bit = new int[copy.length + 1];

        List<Integer> result = new ArrayList<>();

        // Scan original input nums from right to left

        for (int i = nums.length - 1; i >= 0; i--) {

            // Find which bucket(id) for nums[i] in sorted array, but no

            // need to '+ 1' since we will '+ 1' in both 'query' and 'update'

            // method later, the '+ 1' because binarySearch is 0-index based,

            // but BIT is 1-index based

            // Relation: bucket\_id\_before\_padding + 1 = bucket\_id\_after\_padding

            // e.g nums = [5,2,6,1], when we search for 5 in sorted array

            // [1,2,5,6] will get bucket\_id\_before\_padding = 2 and '+ 1' will

            // get bucket\_id\_after\_padding = 3, means the 3rd bucket

            // but we only need original bucket\_id\_before\_padding = 2 for 5 here,

            // and in 'query' and 'update' method we will make '+ 1' padding,

            // so no '+ 1' here

            //int bucket\_id\_after\_padding = Arrays.binarySearch(copy, nums[i]) + 1;

            //int bucket\_id\_before\_padding = Arrays.binarySearch(copy, nums[i]);

            // Use customized 'findNumIndexInSortedArray' method to replace build in one

            int bucket\_id\_before\_padding = findNumIndexInSortedArray(copy, nums[i]);

            // We need to query 'bucket\_id\_before\_padding - 1', the '- 1' means

            // presum range for buckets ids end at 'bucket\_id\_before\_padding - 1'

            // e.g

            // NOTE: Below based on bucket\_id\_after\_padding

            // For nums = [5,2,6,1] after mapping we skip non-exist buckets

            // as bucket 3 and 4, so 1 mapping to bucket 1(=bit[1]), 2 mapping

            // to bucket 2(=bit[2]), 5 mapping to bucket 3(=bit[3]), 6 mapping

            // to bucket 4(=bit[4]), bit[0] is the padding, when we query

            // all smaller number count on right of 5 equals to find all related

            // buckets under BIT structured count sum before 5's mapping bucket,

            // be careful, as 5 mapping to bucket 3, its not necessarily mean we

            // have to sum up both bucket 1 and bucket 2 count as a linear scan,

            // what we have to follow up is only based on BIT structured relation,

            // and consider each cell coverage definition in BIT we only add bucket

            // 2 count, because bucket 2 count already include bucket 1 count, and

            // the reason we come to BIT way is because BIT only spend O(logN) on

            // query and update, but linear scan will take O(N)

            result.add(query(bit, bucket\_id\_before\_padding - 1));

            // We also need to update the count for current bucket

            // e.g nums = [5,2,6,1], when we encounter 5, we have to update count

            // of bucket 3 by increase 1

            update(bit, bucket\_id\_before\_padding);

        }

        // Because we scan from right to left on original input nums,

        // the result always appending for the rightmost number first,

        // to display as left to right order in result, we have to

        // reverse the 'result' list

        Collections.reverse(result);

        return result;

    }

    // Given an index 'i', find all previous count sum

    private int query(int[] bit, int i) {

        int presum = 0;

        // Padding 1 to match BIT array indexes as [1, n] rather than [0, n - 1]

        // Passed in 'i' is bucket\_id\_before\_padding, after 'i++' is bucket\_id\_after\_padding

        i++;

        while (i > 0) {

            presum += bit[i];

            i -= i & -i;

        }

        return presum;

    }

    private void update(int[] bit, int i) {

        // Padding 1 to match BIT array indexes as [1, n] rather than [0, n - 1]

        // Passed in 'i' is bucket\_id\_before\_padding, after 'i++' is bucket\_id\_after\_padding

        i++;

        while (i < bit.length) {

            bit[i] += 1;

            i += i & -i;

        }

    }

    private int[] filterUniqueValueAndSort(int[] nums) {

        // Rebuild original input nums array with unique value only

        Set<Integer> numSet = new HashSet<>();

        for(int num : nums) {

            numSet.add(num);

        }

        int[] tmp = new int[numSet.size()];

        // Assign each number an index, like a bucket

        int index = 0;

        for(int num : numSet) {

            tmp[index++] = num;

        }

        // Have to sort to quickly get each number's index

        Arrays.sort(tmp);

        return tmp;

    }

    // Find Lower boundary

    // Find the first position 'index' in the bit array, where bit[index] >= val, then any

    // element(count) in bit array before this index will contribute to the result count

    private int findNumIndexInSortedArray(int[] arr, int val) {

        // Find the first element in arr which >= val (lower boundary)

        int l = 0, r = arr.length - 1, m = 0;

        while(l <= r) {

            m = l + ((r - l) >> 1);

            if (arr[m] >= val) {

                r = m - 1;

            } else {

                l = m + 1;

            }

        }

        return l;

    }

}

另外根据第一点重构一种不用harshset过滤的写法：

class Solution {

    public List<Integer> countSmaller(int[] nums) {

        int[] copy = nums.clone();

        Arrays.sort(copy);

        // Need a 'bit' array to record each number's frequency

        // in original input nums

        // Based on point 1: 'bit' is actually BIT structured array,

        // since BIT structured array is 1-indexed, follow standard

        // creation of BIT structured array, we add padding position

        // of index 0, hence length change to original array length + 1

        int[] bit = new int[copy.length + 1];

        List<Integer> result = new ArrayList<>();

        // Scan original input nums from right to left

        for (int i = nums.length - 1; i >= 0; i--) {

            // Find which bucket(id) for nums[i] in sorted array, but no

            // need to '+ 1' since we will '+ 1' in both 'query' and 'update'

            // method later, the '+ 1' because binarySearch is 0-index based,

            // but BIT is 1-index based

            // Relation: bucket\_id\_before\_padding + 1 = bucket\_id\_after\_padding

            // e.g nums = [5,2,6,1], when we search for 5 in sorted array

            // [1,2,5,6] will get bucket\_id\_before\_padding = 2 and '+ 1' will

            // get bucket\_id\_after\_padding = 3, means the 3rd bucket

            // but we only need original bucket\_id\_before\_padding = 2 for 5 here,

            // and in 'query' and 'update' method we will make '+ 1' padding,

            // so no '+ 1' here

            //int bucket\_id\_after\_padding = Arrays.binarySearch(copy, nums[i]) + 1;

            //int bucket\_id\_before\_padding = Arrays.binarySearch(copy, nums[i]);

            // Use customized 'findNumIndexInSortedArray' method to replace build in one

            int bucket\_id\_before\_padding = findNumIndexInSortedArray(copy, nums[i]);

            // We need to query 'bucket\_id\_before\_padding - 1', the '- 1' means

            // presum range for buckets ids end at 'bucket\_id\_before\_padding - 1'

            // e.g

            // NOTE: Below based on bucket\_id\_after\_padding

            // For nums = [5,2,6,1] after mapping we skip non-exist buckets

            // as bucket 3 and 4, so 1 mapping to bucket 1(=bit[1]), 2 mapping

            // to bucket 2(=bit[2]), 5 mapping to bucket 3(=bit[3]), 6 mapping

            // to bucket 4(=bit[4]), bit[0] is the padding, when we query

            // all smaller number count on right of 5 equals to find all related

            // buckets under BIT structured count sum before 5's mapping bucket,

            // be careful, as 5 mapping to bucket 3, its not necessarily mean we

            // have to sum up both bucket 1 and bucket 2 count as a linear scan,

            // what we have to follow up is only based on BIT structured relation,

            // and consider each cell coverage definition in BIT we only add bucket

            // 2 count, because bucket 2 count already include bucket 1 count, and

            // the reason we come to BIT way is because BIT only spend O(logN) on

            // query and update, but linear scan will take O(N)

            result.add(query(bit, bucket\_id\_before\_padding - 1));

            // We also need to update the count for current bucket

            // e.g nums = [5,2,6,1], when we encounter 5, we have to update count

            // of bucket 3 by increase 1

            update(bit, bucket\_id\_before\_padding);

        }

        // Because we scan from right to left on original input nums,

        // the result always appending for the rightmost number first,

        // to display as left to right order in result, we have to

        // reverse the 'result' list

        Collections.reverse(result);

        return result;

    }

    // Given an index 'i', find all previous count sum

    private int query(int[] bit, int i) {

        int presum = 0;

        // Padding 1 to match BIT array indexes as [1, n] rather than [0, n - 1]

        // Passed in 'i' is bucket\_id\_before\_padding, after 'i++' is bucket\_id\_after\_padding

        i++;

        while (i > 0) {

            presum += bit[i];

            i -= i & -i;

        }

        return presum;

    }

    private void update(int[] bit, int i) {

        // Padding 1 to match BIT array indexes as [1, n] rather than [0, n - 1]

        // Passed in 'i' is bucket\_id\_before\_padding, after 'i++' is bucket\_id\_after\_padding

        i++;

        while (i < bit.length) {

            bit[i] += 1;

            i += i & -i;

        }

    }

    // Find Lower boundary

    // Find the first position 'index' in the bit array, where bit[index] >= val, then any

    // element(count) in bit array before this index will contribute to the result count

    private int findNumIndexInSortedArray(int[] arr, int val) {

        // Find the first element in arr which >= val (lower boundary)

        int l = 0, r = arr.length - 1, m = 0;

        while(l <= r) {

            m = l + ((r - l) >> 1);

            if (arr[m] >= val) {

                r = m - 1;

            } else {

                l = m + 1;

            }

        }

        return l;

    }

}

**Solution 2: Binary Search Tree (120 min, TLE 62/66)**

**Style 1: Separate search() and build() method**

class Solution {

    class Node {

        int val;

        // The number of nodes that equal to this node.val

        int same = 1;

        // The number of nodes that less than this node.val

        int smaller = 0;

        Node left, right;

        public Node(int val) {

            this.val = val;

        }

    }

    public List<Integer> countSmaller(int[] nums) {

        int[] cnt = new int[nums.length];

        Node root = null;

        // Same as BIT solution, traverse from right to left

        for(int i = nums.length - 1; i >= 0; i--) {

            search(cnt, i, root, nums[i]);

            root = build(nums[i], root);

        }

        List<Integer> result = new ArrayList<>();

        for(int num : cnt) {

            result.add(num);

        }

        return result;

    }

    private Node build(int val, Node node) {

        if(node == null) {

            return new Node(val);

        }

        if(node.val == val) {

            node.same++;

        } else if(node.val > val) {

            node.smaller++;

            node.left = build(val, node.left);

        } else {

            node.right = build(val, node.right);

        }

        return node;

    }

    // We need an object 'int[] cnt' to record through the recursion to sum up all

    // if only 'int cnt' won't keep previous recursion value

    private void search(int[] cnt, int index, Node node, int val) {

        if(node == null) {

            return;

        }

        if(node.val == val) {

            cnt[index] += node.smaller;

            return;

        } else if(node.val > val) {

            search(cnt, index, node.left, val);

        } else {

            cnt[index] += node.same + node.smaller;

            search(cnt, index, node.right, val);

        }

    }

}

**模拟一下输入 [7, 5, 2, 6, 1]后搜索和构建的过程，值得注意的一点是，L315和L493都是边搜索边构建BST的，并不是完全构建好了BST再搜索的，BST本身的节点数值是动态变化的，所以如果按照先完全构建好BST再来搜索找到每个节点的值的思路是完全错误的，那样得到的数值也和边搜索边构建得到的有着巨大偏差**

[7,5,2,6,1]

Round 1:

search(cnt, null, 1) -> cnt[0] = 0

based on tree => null

build(1, null) ->

        tree = node 1(same = 1, smaller = 0)

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

Round 2:

search(cnt, node 1, 6) -> node 1 < target = 6 -> cnt[0] += node 1.smaller + node 1.same = 0 + 1 = 1

based on tree => node 1(same = 1, smaller = 0)

build(6, node 1) ->

        tree = node 1(same = 1, smaller = 0)

                            \

                        node 6(same = 1, smaller = 0)

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

Round 3:

search(cnt, node 1, 2) -> node 1 < target = 2 -> cnt[0] += node 1.smaller + node 1.same = 0 + 1 = 1 -> node 6 > target = 2 -> cnt[0] no change keep as 1

based on tree => node 1(same = 1, smaller = 0)

                            \

                        node 6(same = 1, smaller = 0)

build(2, node 1) ->

        tree = node 1(same = 1, smaller = 0)

                            \

                        node 6(same = 1, smaller = 1)

                          /

          node 2(same = 1, smaller = 0)

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

Round 4:

search(cnt, node 1, 5) -> node 1 < target = 5 -> cnt[0] += node 1.smaller + node 1.same = 0 + 1 = 1 -> node 6 > target = 5 -> cnt[0] no change keep as 1 -> node 2 < target = 5 -> cnt[0] += node 2.smaller + node 2.same = 1 + (0 + 1) = 2

based on tree = node 1(same = 1, smaller = 0)

                            \

                        node 6(same = 1, smaller = 1)

                          /

          node 2(same = 1, smaller = 0)

build(5, node 1) ->

        tree = node 1(same = 1, smaller = 0)

                            \

                        node 6(same = 1, smaller = 2)

                          /

          node 2(same = 1, smaller = 0)

                            \

                        node 5(same = 1, smaller = 0)

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

Round 5:

search(cnt, node 1, 7) -> node 1 < target = 7 -> cnt[0] += node 1.smaller + node 1.same = 0 + 1 = 1 -> node 6 < target = 7 -> cnt[0] += node 6.smaller + node 6.same = 1 + (2 + 1) = 4

based on tree = node 1(same = 1, smaller = 0)

                            \

                        node 6(same = 1, smaller = 2)

                          /

          node 2(same = 1, smaller = 0)

                            \

                        node 5(same = 1, smaller = 0)

build(7, node 1) ->

        tree = node 1(same = 1, smaller = 0)

                            \

                        node 6(same = 1, smaller = 2)

                          /                          \

          node 2(same = 1, smaller = 0)      node 7(same = 1, smaller = 4)

                            \

                        node 5(same = 1, smaller = 0)

**Refer to L493.Reverse Pairs Separate search() and build() method to create above can be applied on both L493 and L315**

<https://leetcode.com/problems/reverse-pairs/solutions/97280/very-short-and-clear-mergesort-bst-java-solutions/>

**BST**

BST solution is no longer acceptable, because it's performance can be very bad, O(n^2) actually, for extreme cases like [1,2,3,4......49999], due to the its unbalance, but I am still providing it below just FYI.

We build the Binary Search Tree from right to left, and at the same time, search the partially built tree with nums[i]/2.0. The code below should be clear enough. **Similar to this** [**https://leetcode.com/problems/count-of-smaller-numbers-after-self/.**](https://leetcode.com/problems/count-of-smaller-numbers-after-self/.) **But the main difference is: here, the number to add and the number to search are different (add nums[i], but search nums[i]/2.0), so not a good idea to combine build and search together.**

针对本题返回对应每个nums中的数值所构成的ArrayList结果List<Integer>而不是L493原装的总和int，做了如下改进

class Solution {

    // We separate 'cnt' as the total number of elements in the subtree rooted at current

    // node that are greater than or equal to val into two different variable 'less' and 'same'

    class Node{

        int val;

        // less: number of nodes that less than this node.val

        int less = 0;

        // same: number of nodes that equal to this node.val

        int same = 1;

        Node left, right;

        public Node(int val){

            this.val = val;

        }

    }

    public List<Integer> countSmaller(int[] nums) {

        Node root = null;

        //int[] cnt = new int[1]; -> -> This is original L493 statement

        int[] cnt = new int[nums.length];

        // Same as BIT solution, traverse from right to left

        for(int i = nums.length - 1; i >= 0; i--){

            search(i, cnt, root, nums[i]); // search and count the partially built tree

            //search(cnt, root, nums[i]/2.0); -> This is original L493 statement

            root = build(nums[i], root); // add nums[i] to BST

        }

        List<Integer> result = new ArrayList<>();

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

            result.add(cnt[i]);

        }

        return result;

    }

    // We need an object 'int[] cnt' to record through the recursion to sum up all

    // if only 'int cnt' won't keep previous recursion value

    private void search(int index, int[] cnt, Node node, double target){

        if(node == null) {

            return;

        }

        if(target == node.val) {

            cnt[index] += node.less;

        } else if(target < node.val) {

            search(index, cnt, node.left, target);

        } else {

            cnt[index] += node.less + node.same;

            search(index, cnt, node.right, target);

        }

    }

    private Node build(int val, Node node){

        if(node == null) {

            return new Node(val);

        } else if(val == node.val) {

            node.same += 1;

        } else if(val > node.val) {

            node.right = build(val, node.right);

        } else {

            node.less += 1;

            node.left = build(val, node.left);

        }

        return node;

    }

}

**Style 2: Merge search() into build() method**

class Solution {

    class Node {

        int val;

        // The number of nodes that equal to this node.val

        int same = 1;

        // The number of nodes that less than this node.val

        int smaller = 0;

        Node left, right;

        public Node(int val) {

            this.val = val;

        }

    }

    public List<Integer> countSmaller(int[] nums) {

        int[] cnt = new int[nums.length];

        Node root = null;

        // Same as BIT solution, traverse from right to left

        for(int i = nums.length - 1; i >= 0; i--) {

            //search(cnt, i, root, nums[i]);

            root = build(nums[i], root, cnt, i);

        }

        List<Integer> result = new ArrayList<>();

        for(int num : cnt) {

            result.add(num);

        }

        return result;

    }

    // Merge search() method into build() method

    private Node build(int val, Node node, int[] cnt, int index) {

        if(node == null) {

            return new Node(val);

        }

        if(node.val == val) {

            node.same++;

            cnt[index] += node.smaller;

        } else if(node.val > val) {

            node.smaller++;

            node.left = build(val, node.left, cnt, index);

        } else {

            cnt[index] += node.same + node.smaller;

            node.right = build(val, node.right, cnt, index);

        }

        return node;

    }

}

**Refer to**

<https://leetcode.com/problems/count-of-smaller-numbers-after-self/solutions/76580/9ms-short-java-bst-solution-get-answer-when-building-bst/>

public class Solution {

    class Node {

        Node left, right;

        int val, sum, dup = 1;

        public Node(int v, int s) {

            val = v;

            sum = s;

        }

    }

    public List<Integer> countSmaller(int[] nums) {

        Integer[] ans = new Integer[nums.length];

        Node root = null;

        for (int i = nums.length - 1; i >= 0; i--) {

            root = insert(nums[i], root, ans, i, 0);

        }

        return Arrays.asList(ans);

    }

    private Node insert(int num, Node node, Integer[] ans, int i, int preSum) {

        if (node == null) {

            node = new Node(num, 0);

            ans[i] = preSum;

        } else if (node.val == num) {

            node.dup++;

            ans[i] = preSum + node.sum;

        } else if (node.val > num) {

            node.sum++;

            node.left = insert(num, node.left, ans, i, preSum);

        } else {

            node.right = insert(num, node.right, ans, i, preSum + node.dup + node.sum);

        }

        return node;

    }

}

**Solution 3: Merge Sort (??? min)**

**在Merge Sort的写法上，L315借鉴于L493，但是难度更大，归并排序后，虽然数组有序的，但是原始顺序变化了，计算每个元素数量需要找到他们的位置，因此需要记录每个元素的index。**

**The basic idea is to do merge sort to nums[]. To record the result, we need to keep the index of each number in the original array. So instead of sort the number in nums, we sort the indexes of each number. Example: nums = [5,2,6,1], indexes = [0,1,2,3] After sort: indexes = [3,1,0,2]**

**Wrong Solution:**

**Because Map<Integer, Integer> not able to handle duplicate value scenario, as different indexes may have same value but we should not store into same slot of map**

class Solution {

    public List<Integer> countSmaller(int[] nums) {

        List<Integer> result = new ArrayList<>();

        int[] temp = new int[nums.length];

        // <k, v> -> <num, num's mapping reverse pair count>

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

        mergeSort(nums, 0, nums.length - 1, temp, map);

        for(int num : nums) {

            result.add(map.getOrDefault(num, 0));

        }

        return result;

    }

    private void mergeSort(int[] nums, int left, int right, int[] temp, Map<Integer, Integer> map) {

        // Why return at 'left == right'?

        // Easy to get when the deepest level recursion happen only

        // remain one number, nothing to sort and directly return

        if(left >= right) {

            return;

        }

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

        mergeSort(nums, left, mid, temp, map);

        mergeSort(nums, mid + 1, right, temp, map);

        merge(nums, left, mid, right, temp, map);

    }

    private void merge(int[] nums, int left, int mid, int right, int[] temp, Map<Integer, Integer> map) {

        // Reverse pair counting logic

        int l = left;

        int r = mid + 1;

        while(l <= mid && r <= right) {

            if((long) nums[l] > (long) nums[r] \* 2) {

                map.put(nums[l], map.getOrDefault(nums[l], 0) + (mid - l) + 1);

                r++;

            } else {

                l++;

            }

        }

        // Sort logic

        int start1 = left;

        int start2 = mid + 1;

        int n1 = mid - left + 1;

        int n2 = right - (mid + 1) + 1;

        // Copy elements of both halves into a temporary array

        for(int i = 0; i < n1; i++) {

            temp[start1 + i] = nums[start1 + i];

        }

        for(int i = 0; i < n2; i++) {

            temp[start2 + i] = nums[start2 + i];

        }

        // Merge the sub-arrays 'in temp' back into the original array 'nums' in sorted order.

        int i = 0;

        int j = 0;

        int k = left;

        while(i < n1 && j < n2) {

            if(temp[start1 + i] <= temp[start2 + j]) {

                nums[k++] = temp[start1 + i++];

            } else {

                nums[k++] = temp[start2 + j++];

            }

        }

        // Copy remaining elements

        while(i < n1) {

            nums[k++] = temp[start1 + i++];

        }

        while(j < n2) {

            nums[k++] = temp[start2 + j++];

        }

    }

}

**目前最接近成功的写法？？？**

import java.util.\*;

public class Solution {

    class Node {

        int val;

        int originalIndex;

        public Node(int val, int originalIndex) {

            this.val = val;

            this.originalIndex = originalIndex;

        }

    }

    public List<Integer> countSmaller(int[] nums) {

        //int[] temp = new int[nums.length];

        int n = nums.length;

        int[] count = new int[n];

        Node[] temp = new Node[n];

        Node[] numsAndIndexes = new Node[n];

        for(int i = 0; i < n; i++) {

            numsAndIndexes[i] = new Node(nums[i], i);

        }

        mergeSort(count, numsAndIndexes, 0, n - 1, temp);

        List<Integer> result = new ArrayList<>();

        for(int i = 0; i < n; i++) {

            result.add(count[i]);

        }

        return result;

    }

    private void mergeSort(int[] count, Node[] nums, int left, int right, Node[] temp) {

        // Why return at 'left == right'?

        // Easy to get when the deepest level recursion happen only

        // remain one number, nothing to sort and directly return

        if(left >= right) {

            return;

        }

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

        mergeSort(count, nums, left, mid, temp);

        mergeSort(count, nums, mid + 1, right, temp);

        merge(count, nums, left, mid, right, temp);

    }

    private void merge(int[] count, Node[] nums, int left, int mid, int right, Node[] temp) {

        // Reverse pair counting logic

        int l = left;

        int r = mid + 1;

        int cnt = 0;

        while(l <= mid && r <= right) {

            // Store initial value of r

            int int\_r = r;

            if(nums[l].val > nums[r].val) {

                //int oriIdx = nums[l].originalIndex;

                //count[oriIdx] += (mid - l) + 1;

                cnt++;

                r++;

            } else {

                int oriIdx = nums[l].originalIndex;

                count[oriIdx] += cnt;

                l++;

                //r = int\_r;

            }

        }

        // Sort logic

        int start1 = left;

        int start2 = mid + 1;

        int n1 = mid - left + 1;

        int n2 = right - (mid + 1) + 1;

        // Copy elements of both halves into a temporary array

        for(int i = 0; i < n1; i++) {

            temp[start1 + i] = nums[start1 + i];

        }

        for(int i = 0; i < n2; i++) {

            temp[start2 + i] = nums[start2 + i];

        }

        // Merge the sub-arrays 'in temp' back into the original array 'nums' in sorted order.

        int i = 0;

        int j = 0;

        int k = left;

        while(i < n1 && j < n2) {

            if(temp[start1 + i].val <= temp[start2 + j].val) {

                nums[k++] = temp[start1 + i++];

            } else {

                nums[k++] = temp[start2 + j++];

            }

        }

        // Copy remaining elements

        while(i < n1) {

            nums[k++] = temp[start1 + i++];

        }

        while(j < n2) {

            nums[k++] = temp[start2 + j++];

        }

    }

    public static void main(String[] args) {

        int[] nums = new int[]{5,2,6,1};

        Solution so = new Solution();

        List<Integer> result = so.countSmaller(nums);

        System.out.println(result);

    }

}

**Refer to**

<https://leetcode.com/problems/count-of-smaller-numbers-after-self/solutions/76583/11ms-java-solution-using-merge-sort-with-explanation/>

class Solution {

    int[] count;

    public List<Integer> countSmaller(int[] nums) {

        List<Integer> res = new ArrayList<Integer>();

        count = new int[nums.length];

        int[] indexes = new int[nums.length];

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

            indexes[i] = i;

        }

        mergesort(nums, indexes, 0, nums.length - 1);

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

            res.add(count[i]);

        }

        return res;

    }

    private void mergesort(int[] nums, int[] indexes, int start, int end){

        if(end <= start){

            return;

        }

        int mid = (start + end) / 2;

        mergesort(nums, indexes, start, mid);

        mergesort(nums, indexes, mid + 1, end);

        merge(nums, indexes, start, end);

    }

    private void merge(int[] nums, int[] indexes, int start, int end){

        int mid = (start + end) / 2;

        int left\_index = start;

        int right\_index = mid+1;

        int rightcount = 0;

        int[] new\_indexes = new int[end - start + 1];

        int sort\_index = 0;

        while(left\_index <= mid && right\_index <= end){

            if(nums[indexes[right\_index]] < nums[indexes[left\_index]]){

                new\_indexes[sort\_index] = indexes[right\_index];

                rightcount++;

                right\_index++;

            }else{

                new\_indexes[sort\_index] = indexes[left\_index];

                count[indexes[left\_index]] += rightcount;

                left\_index++;

            }

            sort\_index++;

        }

        while(left\_index <= mid){

            new\_indexes[sort\_index] = indexes[left\_index];

            count[indexes[left\_index]] += rightcount;

            left\_index++;

            sort\_index++;

        }

        while(right\_index <= end){

            new\_indexes[sort\_index++] = indexes[right\_index++];

        }

        for(int i = start; i <= end; i++){

            indexes[i] = new\_indexes[i - start];

        }

    }

}

**Refer to**

<https://leetcode.com/problems/count-of-smaller-numbers-after-self/solutions/445769/merge-sort-clear-simple-explanation-with-examples-o-n-lg-n/>

// Wrapper class for each and every value of the input array,

    // to store the original index position of each value, before we merge sort the array

    private class ArrayValWithOrigIdx {

        int val;

        int originalIdx;

        public ArrayValWithOrigIdx(int val, int originalIdx) {

            this.val = val;

            this.originalIdx = originalIdx;

        }

    }

    public List<Integer> countSmaller(int[] nums) {

        if (nums == null || nums.length == 0) return new LinkedList<Integer>();

        int n = nums.length;

        int[] result = new int[n];

        ArrayValWithOrigIdx[] newNums = new ArrayValWithOrigIdx[n];

        for (int i = 0; i < n; ++i) newNums[i] = new ArrayValWithOrigIdx(nums[i], i);

        mergeSortAndCount(newNums, 0, n - 1, result);

        // notice we don't care about the sorted array after merge sort finishes.

        // we only wanted the result counts, generated by running merge sort

        List<Integer> resultList = new LinkedList<Integer>();

        for (int i : result) resultList.add(i);

        return resultList;

    }

    private void mergeSortAndCount(ArrayValWithOrigIdx[] nums, int start, int end, int[] result) {

        if (start >= end) return;

        int mid = (start + end) / 2;

        mergeSortAndCount(nums, start, mid, result);

        mergeSortAndCount(nums, mid + 1, end, result);

        // left subarray start...mid

        // right subarray mid+1...end

        int leftPos = start;

        int rightPos = mid + 1;

        LinkedList<ArrayValWithOrigIdx> merged = new LinkedList<ArrayValWithOrigIdx>();

        int numElemsRightArrayLessThanLeftArray = 0;

        while (leftPos < mid + 1 && rightPos <= end) {

            if (nums[leftPos].val > nums[rightPos].val) {

                // this code block is exactly what the problem is asking us for:

                // a number from the right side of the original input array, is smaller

                // than a number from the left side

                //

                // within this code block,

                // nums[rightPos] is smaller than the start of the left sub-array.

                // Since left sub-array is already sorted,

                // nums[rightPos] must also be smaller than the entire remaining left sub-array

                ++numElemsRightArrayLessThanLeftArray;

                // continue with normal merge sort, merge

                merged.add(nums[rightPos]);

                ++rightPos;

            } else {

                // a number from left side of array, is smaller than a number from

                // right side of array

                result[nums[leftPos].originalIdx] += numElemsRightArrayLessThanLeftArray;

                // Continue with normal merge sort

                merged.add(nums[leftPos]);

                ++leftPos;

            }

        }

        // part of normal merge sort, if either left or right sub-array is not empty,

        // move all remaining elements into merged result

        while (leftPos < mid + 1) {

            result[nums[leftPos].originalIdx] += numElemsRightArrayLessThanLeftArray;

            merged.add(nums[leftPos]);

            ++leftPos;

        }

        while (rightPos <= end) {

            merged.add(nums[rightPos]);

            ++rightPos;

        }

        // part of normal merge sort

        // copy back merged result into array

        int pos = start;

        for (ArrayValWithOrigIdx m : merged) {

            nums[pos] = m;

            ++pos;

        }

    }

**本质上，结合L327.Count of Range Sum所提出的思路，在使用BIT计算频率的时候，使用了Discretization(离散化)的技术，而L315中也是如此，下面是关于Discretization(离散化)在L315中的讲解：**

**Refer to**

<https://algo.monster/liteproblems/315>

**Problem Description**

The problem at hand is a classic algorithmic challenge where we are given an array of integers, which we will refer to as nums. The goal is to find, for each element in the array, how many numbers to its right are smaller than it. To clarify, for every index i in the array, we want to calculate the count of elements that are located at any index greater than i and have a value less than nums[i]. The output should be an array of integers where each element counts[i] represents the number of smaller elements to the right of nums[i].

For example, if the input array nums is [5, 2, 6, 1], the output array should be [2, 1, 1, 0]. For the first element 5, there are two elements (2 and 1) to the right that are smaller. For the second element 2, there is only one element (1) to the right that is smaller, and so on.

**Intuition**

The straightforward brute force solution would be to go through each element in the array and count the number of smaller elements to its right with nested loops. However, this method would result in a time complexity of O(n^2), which is not efficient for large arrays.

To optimize this, we can use a data structure called a Binary Indexed Tree (BIT), also known as a Fenwick Tree, or alternatively, a Segment Tree. The Binary Indexed Tree is a data structure that allows us to update values and calculate prefix sums in a logarithmic time complexity, which is much faster than the brute force approach.

The intuition behind using BIT in this solution is as follows:

First, we need to discretize the values in the array to make sure they can fit into our Binary Indexed Tree without taking too much space. Discretization involves mapping each value to its corresponding rank in the sorted list of unique values.

We process the elements of the original array in reverse order. This means we start with the rightmost element and move to the left. This allows us to use the tree to keep track of the number of elements that have already been seen and are less than the current element.

For each element, we find its index in the discretized array, which represents its value in the BIT.

Before we add the current element to the tree, we query the tree to find out how many elements less than the current one have already been processed (those to the right in the original array). This query gives us the count of smaller numbers to the right.

Update the Binary Indexed Tree to indicate that we have now seen an instance of the current element.

We store the query result in the result list, which we then reverse at the end since we processed the elements from right to left.

Using the BIT, both the query and update operations are performed in O(log n) time, which significantly improves the efficiency for large datasets.

**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.

**Solution Approach**

The solution relies on two key components: the discretization of array values and a Binary Indexed Tree (BIT). Let's walk through the implementation step by step, highlighting how the algorithms, data structures, and patterns are used:

**Discretization of Values:** Before we can work with the BIT, we need to make sure array indices won't be too large. This is achieved by discretizing the array values. In Python, this can be done by sorting the unique values and then creating a mapping from the original values to their indices in the sorted array, starting with 1 (since BIT works with 1-based indexing). This process can be seen in the code with:

alls = sorted(set(nums))

2m = {v: i for i, v in enumerate(alls, 1)}

Here, alls is the array of unique, sorted values and m maps each original value to its rank in the sorted array.

**Implementation of BIT**: A BIT is a data structure that supports updating individual elements and calculating prefix sums efficiently. The Python class BinaryIndexedTree includes methods for updating the tree (update) and querying prefix sums (query), utilizing a helper method lowbit to isolate the rightmost bit of a binary representation.

class BinaryIndexedTree:

    def \_\_init\_\_(self, n):

        self.n = n

        self.c = [0] \* (n + 1)

    @staticmethod

    def lowbit(x):

        return x & -x

    def update(self, x, delta):

        while x <= self.n:

            self.c[x] += delta

            x += BinaryIndexedTree.lowbit(x)

    def query(self, x):

        s = 0

        while x > 0:

            s += self.c[x]

            x -= BinaryIndexedTree.lowbit(x)

        return s

self.c is the internal array of the BIT, initialized with zeros.

lowbit method isolates the lowest one bit of x, which is essential for navigating the tree structure.

update method adds delta to the value at index x, affecting all subsequent sums that include this element.

query method sums all the elements up to index x (exclusive), allowing us to get the count of smaller elements seen so far.

**Processing Elements in Reverse**: We initialize a BIT with the size equal to the number of unique elements in nums. Then, we traverse the nums array in reverse, translating each element to its discretized form and using the update and query operations of BIT to build the counts array.

tree = BinaryIndexedTree(len(m))

ans = []

for v in nums[::-1]:

    x = m[v]

    tree.update(x, 1)

    ans.append(tree.query(x - 1))

return ans[::-1]

For each element v encountered, we look up its discretized index x.

Before updating x in BIT, we perform a query to get the count of elements smaller than x.

The result of each query is added to ans.

Since we've traversed the array in reverse, we need to reverse ans before returning it to get the correct order.

This implementation ensures that each update and query call is handled in O(log n) time, leading to an overall time complexity of O(n log n) due to processing each of the n elements once. It's a significant improvement over the brute force method and is well-suited for handling large input arrays.

**Example Walkthrough**

Let's consider a small example with the input array nums = [3, 4, 2, 7, 5]. Using the solution approach outlined, we want to find the number of elements to the right that are smaller than each element in nums.

Discretization of Values: First, we discretize the array. The sorted unique values will be [2, 3, 4, 5, 7]. Our mapping m will be {2: 1, 3: 2, 4: 3, 5: 4, 7: 5}.

Implementation of BIT: We create a Binary Indexed Tree for the five unique values.

tree = BinaryIndexedTree(5)

Processing Elements in Reverse: We process the elements from right to left, updating the tree and querying for counts:

Starting with the last element 5, its discretized value is 4. Before we update, there are no smaller elements, so ans[] starts with [0].

Next is 7, with discretized value 5. The query(4) will find 0 elements smaller, so ans[] becomes [0, 0].

Now 2 is encountered with a discretized value of 1. Previous smaller elements don't exist (query(0)), thus ans[] is [0, 0, 0].

The fourth element is 4, corresponding to discretized 3. The query(2) shows there is one smaller (2), so ans[] is [0, 0, 1, 0].

The first element in reverse is 3, with discretized value 2. A query(1) will find there is one element smaller (2), adding up to ans[] as [0, 1, 1, 0, 0].

After processing, we have ans[] = [0, 1, 1, 0, 0], and we reverse it to get the final output: [0, 0, 1, 1, 0].

This corresponds to:

3 (no smaller elements to the right),

4 (no smaller elements to the right),

2 (one smaller element, 5, to the right),

7 (one smaller element, 5, to the right),

5 (no smaller elements to the right).

Final Output: [0, 0, 1, 1, 0].

The Binary Indexed Tree speeds up the process where each update and query happens in logarithmic time relative to the number of unique elements. Overall, the time complexity of processing the array in this manner is O(n log n).

**Java Solution**

import java.util.\*;

class Solution {

    public List<Integer> countSmaller(int[] nums) {

        // Create a set to hold unique values from the input array

        Set<Integer> uniqueValues = new HashSet<>();

        for (int value : nums) {

            uniqueValues.add(value);

        }

        // Create a sorted list of unique values

        List<Integer> sortedUniqueValues = new ArrayList<>(uniqueValues);

        Collections.sort(sortedUniqueValues);

        // Map each unique number to its index in the sorted array

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

        for (int i = 0; i < sortedUniqueValues.size(); ++i) {

            valueToIndex.put(sortedUniqueValues.get(i), i + 1);

        }

        // Initialize a Fenwick Tree (Binary Indexed Tree)

        FenwickTree fenwickTree = new FenwickTree(sortedUniqueValues.size());

        // Use a LinkedList to store the results as we need to add elements to the beginning

        LinkedList<Integer> result = new LinkedList<>();

        // Iterate over the input array in reverse order

        for (int i = nums.length - 1; i >= 0; --i) {

            int index = valueToIndex.get(nums[i]); // Get the index in the Fenwick Tree

            fenwickTree.update(index, 1);          // Update the tree with the current number

            result.addFirst(fenwickTree.query(index - 1)); // Query the count of numbers smaller than the current number

        }

        return result;

    }

}

class FenwickTree {

    private int size;

    private int[] tree;

    // Constructor for Fenwick Tree with a given size

    public FenwickTree(int size) {

        this.size = size;

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

    }

    // Update the Fenwick Tree with a given value at a specified index

    public void update(int index, int delta) {

        while (index <= size) {

            tree[index] += delta;

            index += lowBit(index);

        }

    }

    // Query the cumulative frequency from index 1 to the given index

    public int query(int index) {

        int sum = 0;

        while (index > 0) {

            sum += tree[index];

            index -= lowBit(index);

        }

        return sum;

    }

    // Method to get the lowest one bit value of a given number

    public static int lowBit(int x) {

        return x & (-x);

    }

}

**Time and Space Complexity**

The given Python code implements a Binary Indexed Tree (also known as a Fenwick Tree) to solve the problem of finding the count of smaller numbers after each element for a given list of integers.

**Time Complexity:**

Sorting unique values (sorted(set(nums))):

This operation involves extracting unique elements from nums and then sorting them, which has a time complexity of O(k log k) where k is the number of unique elements.

Building a map for value to index (m = {v: i for i, v in enumerate(alls, 1)}):

This takes O(k) time, as it is iterating over all unique elements once.

Binary Indexed Tree operations within the main loop (for v in nums[::-1]: ...):

For each v in the list nums, we perform an update and a query operation on the tree. Since each update and query takes O(log n) time and we are doing it for all n elements, this part takes a total of O(n log n) time.

Reversing the answer list (return ans[::-1]):

This operation takes O(n) time for a list of n elements.

Combining these parts, the overall time complexity is O(n log n) since the sorting can be considered a preprocessing step with a negligible O(k log k) complexity compared to the O(n log n) complexity of the main operations with the Binary Indexed Tree.

**Space Complexity:**

Storage for the Binary Indexed Tree (self.c = [0] \* (n + 1)):

The space used by the tree is O(n) where n is the number of elements in the input nums.

Space for the sorted unique values and map (alls and m):

The sorted list of unique values alls and the map m both take O(k) space where k is the number of unique elements.

Space for the answer (ans):

Since we store the count of smaller elements for each input element, this also takes O(n) space.

The dominant space complexity for the entire algorithm is O(n), which takes into account the space for the Binary Indexed Tree and the results list.

In conclusion, the code has a time complexity of O(n log n) and a space complexity of O(n).