You are given an integer array nums and you have to return a new counts array. The counts array has the property 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 <= 105
* -104 <= nums[i] <= 104

Solution

Overview

The problem is straightforward. For each num in nums, we need to obtain the number of smaller elements after num.

A straightforward approach is to use brute force with two for-loops. The first loop iterates over all num in nums, and the second loop iterates over all elements after num. However, this approach costs O(N^2)*O*(*N*2) and yields *Time Limit Exceed*, given that N*N* is the length of nums.

Luckily, there are two helpful data structures: [segment tree](https://en.wikipedia.org/wiki/Segment_tree) and [binary indexed tree](https://en.wikipedia.org/wiki/Fenwick_tree), which are able to do the range query in logarithmic time.

Also, a solution based on [Merge Sort](https://en.wikipedia.org/wiki/Merge_sort) is available.

Below, we will discuss each of the three approaches: *Segment Tree*, *Binary Indexed Tree*, and *Merge Sort*.

After you finish, you can practice by solving some similar questions:

* [Reverse Pairs](https://leetcode.com/problems/reverse-pairs/solution/)
* [Create Sorted Array through Instructions](https://leetcode.com/problems/create-sorted-array-through-instructions/)

Approach 1: Segment Tree

**Intuition**

**Prerequisite: segment tree**

If you are not familiar with segment trees, you should check out our [Recursive Approach to segment trees](https://leetcode.com/articles/a-recursive-approach-to-segment-trees-range-sum-queries-lazy-propagation/) tutorial before continuing.

Also, here are some relevant applications for segment trees that you can practice on:

* [Range Sum Query - Mutable](https://leetcode.com/problems/range-sum-query-mutable/)
* [Count of Range Sum](https://leetcode.com/problems/count-of-range-sum/)

For a full list, check out the [segment tree Tag](https://leetcode.com/tag/segment-tree/).

For a particular element in nums, located at index i, we want to count how many of the numbers on the right side of index i are smaller than nums[i]. Notice that the value of the smaller numbers must be in the range (-\infty, \text{nums[i]}-1](−∞,nums[i]−1].

Hence, if we can find the count of **each number** in the range (-\infty, \text{nums[i]}-1](−∞,nums[i]−1] on the right side of index i, then the answer will be the sum of those counts.

Therefore, for each index i, we need a query to find the sum of those counts. Recall that the segment tree and the binary indexed tree are two data structures that are generally helpful when solving range query problems.

Since we need counts of values, we can use an approach similar to [bucket sort](https://en.wikipedia.org/wiki/Bucket_sort), where we have buckets of values and buckets[value] stores the count of value. For each value, we increment buckets[value] by 1. With this approach, the number of elements smaller than nums[i] is the range sum of (-\infty, \text{num}-1](−∞,num−1] in buckets.

With the help of a segment tree or binary indexed tree, we can perform the range sum query in logarithmic time.

With the given constraint -10^4 <= nums[i] <= 10^4, we can initialize buckets from -10^4 to 10^4.

Wait, there is a problem: Usually, we store buckets in an array, so the indices of buckets are non-negative. However, here we need to store some **negative** values. How can we resolve this problem?

There are two solutions:

1. Use a map rather than an array.
2. Shift all numbers to non-negative.

Both solutions work, and here we have chosen the second one since it is easier to implement. Interested readers are welcome to try the first one on their own.

To shift all numbers to non-negative, we simply add a constant. Here we chose the constant offset = 10^4 and increase each number by offset:

nums[i] = nums[i] + offset

The smallest number -10^4 becomes 0 under this shift.

Note that while querying a particular index, we only need to consider elements that are on the right side of the index. Therefore we need to make sure that when we query an index, say i, only elements from index i+1 to the end of the array are present in the buckets.

To achieve this, we need to traverse nums from **right to left**, while performing range sum queries and updating the counts.

Similarly, with the help of a segment tree or binary indexed tree, we can perform the updates in logarithmic time.

(For convenience, the offset is not included in the above picture.)

**Algorithm**

* Implement the segment tree. Since the tree is initialized with all zeros, only update and query need to be implemented. Set offset = 10^4.
* Iterate over each num in nums in reverse. For each num:
  + Shift num to num + offset.
  + Query the number of elements in the segment tree smaller than num.
  + Update the count of num in the segment tree.
* Return the result.

**Implementation**

class Solution {

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

int offset = 10000; // offset negative to non-negative

int size = 2 \* 10000 + 1; // total possible values in nums

int[] tree = new int[size \* 2];

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

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

int smaller\_count = query(0, nums[i] + offset, tree, size);

result.add(smaller\_count);

update(nums[i] + offset, 1, tree, size);

}

Collections.reverse(result);

return result;

}

// implement segment tree

private void update(int index, int value, int[] tree, int size) {

index += size; // shift the index to the leaf

// update from leaf to root

tree[index] += value;

while (index > 1) {

index /= 2;

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

}

}

private int query(int left, int right, int[] tree, int size) {

// return sum of [left, right)

int result = 0;

left += size; // shift the index to the leaf

right += size;

while (left < right) {

// if left is a right node

// bring the value and move to parent's right node

if (left % 2 == 1) {

result += tree[left];

left++;

}

// else directly move to parent

left /= 2;

// if right is a right node

// bring the value of the left node and move to parent

if (right % 2 == 1) {

right--;

result += tree[right];

}

// else directly move to parent

right /= 2;

}

return result;

}

}

**Complexity Analysis**

Let N*N* be the length of nums and M*M* be the difference between the maximum and minimum values in nums.

Note that for convenience, we fix M=2\*10^4*M*=2∗104 in the above implementations.

* Time Complexity: O(N\log(M))*O*(*N*log(*M*)).  
  We need to iterate over nums. For each element, we spend O(\log(M))*O*(log(*M*)) to find the number of smaller elements after it, and spend O(\log(M))*O*(log(*M*)) time to update the counts. In total, we need O(N \cdot \log(M)) = O(N\log(M))*O*(*N*⋅log(*M*))=*O*(*N*log(*M*)) time.
* Space Complexity: O(M)*O*(*M*), since we need, at most, an array of size 2M+22*M*+2 to store the segment tree.  
  We need at most M+1*M*+1 buckets, where the extra 11 is for the value 00. For the segment tree, we need twice the number of buckets, which is (M+1)\times 2 = 2M+2(*M*+1)×2=2*M*+2.

Approach 2: Binary Indexed Tree (Fenwick Tree)

**Intuition**

**Prerequisite: binary indexed tree**

If you are not familiar with binary indexed tree (BIT), you should check relevant tutorials, such as [Range Sum Query 2D - Mutable](https://leetcode.com/problems/range-sum-query-2d-mutable/solution/) before continuing.

Also, here are some relevant applications for binary indexed trees that you can practice on:

* [Range Sum Query - Mutable](https://leetcode.com/problems/range-sum-query-mutable/)
* [Count of Range Sum](https://leetcode.com/problems/count-of-range-sum/)

(Yes, many problems which can be solved by segment tree can also be solved by binary indexed tree.)

For a full list, you can check the [binary indexed tree Tag](https://leetcode.com/tag/binary-indexed-tree/).

Binary indexed tree is similar to segment tree. It allows us to perform a prefix query, such as prefix sum, in \loglog time. Can we transform this problem into a **prefix sum** problem?

Yes, using the same tricks that we used in approach 1, buckets and shift, we can transform the number of smaller elements into a prefix sum for the range [0, \text{num}+\text{offset}-1][0,num+offset−1], where \text{offset}=10^4offset=104.

Similarly, when querying, we need to traverse nums from right to left in order to ensure that only the elements to the right are in the buckets.

**Algorithm**

* Implement the binary indexed tree. Since the tree is initialized with all zeros, only update and query need to be implemented. Set offset = 10^4.
* Iterate over each num in nums in reverse. For each num:
  + Shift num to num + offset.
  + Query the number of elements in the BIT that are smaller than num.
  + Update the count of num in the BIT.
* Return the result.

**Implementation**

class Solution {

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

int offset = 10000; // offset negative to non-negative

int size = 2 \* 10000 + 2; // total possible values in nums plus one dummy

int[] tree = new int[size];

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

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

int smaller\_count = query(nums[i] + offset, tree);

result.add(smaller\_count);

update(nums[i] + offset, 1, tree, size);

}

Collections.reverse(result);

return result;

}

// implement Binary Index Tree

private void update(int index, int value, int[] tree, int size) {

index++; // index in BIT is 1 more than the original index

while (index < size) {

tree[index] += value;

index += index & -index;

}

}

private int query(int index, int[] tree) {

// return sum of [0, index)

int result = 0;

while (index >= 1) {

result += tree[index];

index -= index & -index;

}

return result;

}

}

**Complexity Analysis**

Let N*N* be the length of nums and M*M* be the difference between the maximum and minimum values in nums.

Note that for convenience, we fix M=2\*10^4*M*=2∗104 in the above implementations.

* Time Complexity: O(N\log(M))*O*(*N*log(*M*)).  
  We need to iterate over nums. For each element, we spend O(\log(M))*O*(log(*M*)) to find the number of smaller elements after it, and spend O(\log(M))*O*(log(*M*)) time to update the counts. In total, we need O(N \cdot \log(M)) = O(N\log(M))*O*(*N*⋅log(*M*))=*O*(*N*log(*M*)) time.
* Space Complexity: O(M)*O*(*M*), since we need, at most, an array of size M+2*M*+2 to store the BIT.  
  We need at most M+1*M*+1 buckets, where the extra 11 is for the value 00. The BIT requires an extra dummy node, so the size is (M+1)+1 = M+2(*M*+1)+1=*M*+2.

Approach 3: Merge Sort

**Intuition**

**Prerequisite: Merge Sort**

If you are not familiar with Merge Sort, you should check relevant tutorials before continuing.

Also, here is a basic application of Merge Sort that you can practice on:

* [Sort an Array](https://leetcode.com/problems/sort-an-array/)

To apply merge sort, one key observation is that:

The smaller elements on the right of a number will **jump from its right to its left** during the sorting process.

If we can record the numbers of those elements during sorting, then the problem is solved.

Can we modify the merge sort a little to meet our needs?

Consider when merging two sorted list

Yes! When we select an element i on the left array, we know that elements selected previously from the right array **jump** from i's right to i's left.

By adding the counts of those elements in every merge step, we get the total number of elements that jumped from i's right to i's left.

**Algorithm**

* Implement a merge sort function.
  + For each element i, the function records the number of elements jumping from i's right to i's left during the merge sort.
* Merge sort nums, store the number of elements jumping from right to left in result.
  + Alternatively, one can sort the *indices* with corresponding values in nums. That is to say, we are going to sort list [0, 1, ..., n-1] according to the comparator nums[i]. This helps to track the indices and update result. You can find additional details in the implementations below.
* Return result.

**Implementation**

class Solution {

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

int n = nums.length;

int[] result = new int[n];

int[] indices = new int[n]; // record the index. we are going to sort this array

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

indices[i] = i;

}

// sort indices with their corresponding values in nums, i.e., nums[indices[i]]

mergeSort(indices, 0, n, result, nums);

// change int[] to List to return

List<Integer> resultToReturn = new ArrayList<Integer>(n);

for (int i : result) {

resultToReturn.add(i);

}

return resultToReturn;

}

private void mergeSort(int[] indices, int left, int right, int[] result, int[] nums) {

if (right - left <= 1) {

return;

}

int mid = (left + right) / 2;

mergeSort(indices, left, mid, result, nums);

mergeSort(indices, mid, right, result, nums);

merge(indices, left, right, mid, result, nums);

}

private void merge(int[] indices, int left, int right, int mid, int[] result, int[] nums) {

// merge [left, mid) and [mid, right)

int i = left; // current index for the left array

int j = mid; // current index for the right array

// use temp to temporarily store sorted array

List<Integer> temp = new ArrayList<Integer>(right - left);

while (i < mid && j < right) {

if (nums[indices[i]] <= nums[indices[j]]) {

// j - mid numbers jump to the left side of indices[i]

result[indices[i]] += j - mid;

temp.add(indices[i]);

i++;

} else {

temp.add(indices[j]);

j++;

}

}

// when one of the subarrays is empty

while (i < mid) {

// j - mid numbers jump to the left side of indices[i]

result[indices[i]] += j - mid;

temp.add(indices[i]);

i++;

}

while (j < right) {

temp.add(indices[j]);

j++;

}

// restore from temp

for (int k = left; k < right; k++) {

indices[k] = temp.get(k - left);

}

}

}

**Complexity Analysis**

Let N*N* be the length of nums.

* Time Complexity: O(N\log(N))*O*(*N*log(*N*)). We need to perform a merge sort which takes O(N\log(N))*O*(*N*log(*N*)) time. All other operations take at most O(N)*O*(*N*) time.
* Space Complexity: O(N)*O*(*N*), since we need a constant number of arrays of size O(N)*O*(*N*).