# Merge Sort

Merge sort is one of the most important sort algorithms due to two reasons:

1. The worst time and average time complex in merge sort are same O(nlogn).
2. Merge sort is a form of divide and conquer solution.

## 88. Merge Sorted Array

Easy

14443266FavoriteShare

Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.

**Note:**

* The number of elements initialized in *nums1* and *nums2* are *m* and *n* respectively.
* You may assume that *nums1* has enough space (size that is greater or equal to *m* + *n*) to hold additional elements from *nums2*.

**Example:**

**Input:**

nums1 = [1,2,3,0,0,0], m = 3

nums2 = [2,5,6], n = 3

**Output:** [1,2,2,3,5,6]

### Analysis:

The key part in merge sort is the merge, the key part in merge is select which buffer to place the result. So the above algorithm is to use the buffer at the end of nums1, to do so you must merge reversely from last element to the first.

/// <summary>

/// Leet code #88. Merge Sorted Array

///

/// Given two sorted integer arrays nums1 and nums2, merge nums2 into

/// nums1 as one sorted array.

///

/// Note:

///

/// The number of elements initialized in nums1 and nums2 are m and n

/// respectively.

/// You may assume that nums1 has enough space (size that is greater

/// or equal to m + n) to hold additional elements from nums2.

///

/// Example:

/// Input:

/// nums1 = [1,2,3,0,0,0], m = 3

/// nums2 = [2,5,6], n = 3

///

/// Output: [1,2,2,3,5,6]

/// </summary>

void LeetCodeSort::merge(vector<int>& nums1, int m, vector<int>& nums2, int n)

{

int index = m + n - 1;

int pos1 = m - 1, pos2 = n - 1;

while (index >= 0)

{

// no more elements in nums1, just copy remaining in nums2

if (pos1 < 0)

{

nums1[index] = nums2[pos2];

pos2--;

}

// no more elements in nums2, assume all sorted.

else if (pos2 < 0)

{

break;

}

else if (nums1[pos1] > nums2[pos2])

{

nums1[index] = nums1[pos1];

pos1--;

}

else

{

nums1[index] = nums2[pos2];

pos2--;

}

index--;

}

}

## 315. Count of Smaller Numbers After Self

Hard

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:**

**Input:** [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.

### Analysis:

This is one example that merge sort is applied in the area of divide and conquer, during the merges sort we process the group of 2, 4, 8, until half N. This is a O(log(N)) complexity, during the merge process, we compare every number in left group to the right group to see how many numbers in the right group are smaller than this number. The process within the group itself is already processed in the previous batch. This is a O(N) complexity, this is because if a particular number in the right group is smaller that a specific number in the left group, it must be smaller than the numbers in the left group after that number. We do not need to re-compare.

The whole solution ends up with O(nLog(n)).

/// <summary>

/// Leet code #315. Count of Smaller Numbers After Self

/// Note: end is out of the boundary of the last element, e.g. vector.size(),

/// not the index of last element.

/// </summary>

void LeetCode::mergeSortCountSmaller(vector<pair<int, int>>& nums, int begin, int end, vector<int>&result)

{

if (begin + 1 >= end) return;

int mid = begin + (end - begin) / 2;

mergeSortCountSmaller(nums, begin, mid, result);

mergeSortCountSmaller(nums, mid, end, result);

int left = begin, right = mid;

int count = 0;

// we may have right pointer stay at end position for a

// while for the left elements to catch up

while (left < mid && right <= end)

{

while ((right < end) && (nums[left].first > nums[right].first)) right++;

count = right - mid;

result[nums[left].second] += count;

left++;

}

sort(nums.begin() + begin, nums.begin() + end);

}

/// <summary>

/// Leet code #315. Count of Smaller Numbers After Self

///

/// 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:

///

/// Input: [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.

/// Return the array [2, 1, 1, 0].

/// </summary>

vector<int> LeetCodeSort::countSmaller(vector<int>& nums)

{

vector<pair<int, int>> num\_pairs;

vector<int> result(nums.size());

for (size\_t i = 0; i < nums.size(); i++)

{

num\_pairs.push\_back(make\_pair(nums[i], i));

}

mergeSortCountSmaller(num\_pairs, 0, nums.size(), result);

return result;

}

## 327. Count of Range Sum

Hard

Given an integer array nums, return the number of range sums that lie in [lower, upper] inclusive.  
Range sum S(i, j) is defined as the sum of the elements in nums between indices i and j (i ≤ j), inclusive.

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

**Example:**

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

**Output:** 3

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

### Analysis:

This is also a problem we can do in divide and conquer by using merge sort, first you calculate the sum at every index, then, during the merge for every sum in left group, calculate how many sums in the right group having difference within the range.

/// <summary>

/// Leet code #327. Count of Range Sum

/// </summary>

int LeetCodeSort::mergeCountRangeSum(vector<long long>& sums, int begin, int end, int lower, int upper)

{

if (begin + 1 >= end) return 0;

int middle = begin + (end - begin) / 2;

int count = 0;

count = mergeCountRangeSum(sums, begin, middle, lower, upper) +

mergeCountRangeSum(sums, middle, end, lower, upper);

int m = middle, n = middle;

for (int i = begin; i < middle; i++)

{

while ((m < end) && (sums[m] - sums[i] < lower)) m++;

while ((n < end) && (sums[n] - sums[i] <= upper)) n++;

count += n - m;

}

inplace\_merge(sums.begin() + begin, sums.begin() + middle, sums.begin() + end);

return count;

}

/// <summary>

/// Leet code #327. Count of Range Sum

///

/// Given an integer array nums, return the number of range sums that lie in

/// [lower, upper] inclusive.

/// Range sum S(i, j) is defined as the sum of the elements in nums between

/// indices i and j (i ≤ j), inclusive.

///

/// Note:

/// A naive algorithm of O(n2) is trivial. You MUST do better than that.

/// Example:

/// Given nums = [-2, 5, -1], lower = -2, upper = 2,

/// Return 3.

/// The three ranges are : [0, 0], [2, 2], [0, 2] and their respective sums

/// are: -2, -1, 2.

/// </summary>

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

{

vector<long long> sums;

long long sum = 0;

sums.push\_back(0);

for (size\_t i = 0; i < nums.size(); i++)

{

sum += nums[i];

sums.push\_back(sum);

}

return mergeCountRangeSum(sums, 0, sums.size(), lower, upper);

}

## 493. Reverse Pairs

Hard

Given an array nums, we call (i, j) an ***important reverse pair*** if i < j and nums[i] > 2\*nums[j].

You need to return the number of important reverse pairs in the given array.

**Example1:**

**Input**: [1,3,2,3,1]

**Output**: 2

**Example2:**

**Input**: [2,4,3,5,1]

**Output**: 3

**Note:**

1. The length of the given array will not exceed 50,000.
2. All the numbers in the input array are in the range of 32-bit integer.

### Analysis:

This is also a problem we can do in divide and conquer by using merge sort, for every number in left group you may need to calculate how many number in right group are in in reverse.

/// <summary>

/// Leet code #493. Reverse Pairs

/// </summary>

int LeetCodeSort::mergeSortReversePairs(vector<int>& nums, int begin, int end)

{

if (begin + 1 >= end) return 0;

int mid = begin + (end - begin) / 2;

int count = mergeSortReversePairs(nums, begin, mid) + mergeSortReversePairs(nums, mid, end);

int first = begin, next = mid;

while (first < mid && next <= end)

{

while ((next < end) && (nums[first] / 2.0 > nums[next])) next++;

count += next - mid;

first++;

}

sort(nums.begin() + begin, nums.begin() + end);

return count;

}

/// <summary>

/// Leet code #493. Reverse Pairs

///

/// Given an array nums, we call (i, j) an important reverse pair if

/// i < j and nums[i] > 2\*nums[j].

/// You need to return the number of important reverse pairs in the given array.

///

/// Example1:

/// Input: [1,3,2,3,1]

/// Output: 2

///

/// Example2:

/// Input: [2,4,3,5,1]

/// Output: 3

///

/// Note:

/// 1.The length of the given array will not exceed 50,000.

/// 2.All the numbers in the input array are in the range of 32-bit integer.

/// </summary>

int LeetCodeSort::reversePairs(vector<int>& nums)

{

return mergeSortReversePairs(nums, 0, nums.size());

}