<https://leetcode.com/problems/reverse-pairs/description/>

Given an integer array nums, return *the number of* ***reverse pairs*** *in the array*.

A **reverse pair** is a pair (i, j)where:

* 0 <= i < j  < nums.length and
* nums[i] > 2 \* nums[j].

**Example 1:**

Input: nums = [1,3,2,3,1]

Output: 2

Explanation: The reverse pairs are:

(1, 4) --> nums[1] = 3, nums[4] = 1, 3 > 2 \* 1

(3, 4) --> nums[3] = 3, nums[4] = 1, 3 > 2 \* 1

**Example 2:**

Input: nums = [2,4,3,5,1]

Output: 3

Explanation: The reverse pairs are:

(1, 4) --> nums[1] = 4, nums[4] = 1, 4 > 2 \* 1

(2, 4) --> nums[2] = 3, nums[4] = 1, 3 > 2 \* 1

(3, 4) --> nums[3] = 5, nums[4] = 1, 5 > 2 \* 1

**Constraints:**

* 1 <= nums.length <= 5 \* 104
* -231 <= nums[i] <= 231 - 1

**Attempt 1: 2023-09-20**

**Solution 1: Brute Force (10 min, TLE 32/140)**

class Solution {

public int reversePairs(int[] nums) {

int count = 0;

int n = nums.length;

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

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

// Have to transform into long type because number

// range is -2^31 <= nums[i] <= 2^31 - 1

// e.g [2147483647,2147483647,2147483647...etc]

if(Long.valueOf(nums[i]).compareTo(Long.valueOf(nums[j]) \* 2) > 0) {

count++;

}

}

}

return count;

}

}

Time Complexity: O(N^2)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/reverse-pairs/solutions/3185939/best-c-2-solution-divide-and-conquer-merge-sort-brute-force-optimize/>

/\*

Time Complexity : O(N^2), where N is the size of the array. As we check for possible pair which can satisfy

the given condition, and the total number of pairs are : N\*(N–1)/2.

Space complexity : O(1), Constant space.

Solved Using Array(Two Nested Loop). Brute Force Approach.

Note : This approach is Giving TLE.

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Approach 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Solution {

public:

int reversePairs(vector<int>& nums) {

int n = nums.size();

long long reversePairsCount = 0;

for(int i=0; i<n-1; i++){

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

if(nums[i] > 2\*(long long)nums[j]){

reversePairsCount++;

}

}

}

return reversePairsCount;

}

};

**Solution 2: Binary Indexed Tree + Binary Search (1200 min, refer L315.Count of Smaller Numbers After Self for detail, the difference is we pass in "1.0 \* num / 2" instead)**

**Wrong Solution (95/140, failed on [1,3,2,3,1]):**

class Solution {

public int reversePairs(int[] nums) {

int n = nums.length;

//int[] bit = new int[n + 1];

//int[] copy = nums.clone();

//Arrays.sort(copy);

int[] copy = filterUniqueValueAndSort(nums);

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

int result = 0;

for(int i = n - 1; i >= 0; i--) {

// Find the first position in the array, where copy[index] >= num / 2, then any rand before index will contribute to the result

int num = nums[i];

result += query(bit, findNumIndexInSortedArray(copy, num / 2) - 1);

update(bit, findNumIndexInSortedArray(copy, num));

}

return result;

}

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;

}

// 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;

}

}

**Correct Solution:**

**The only change is in findNumIndexInSortedArray() method, we MUST use double type to define input number => 1.0 \* num / 2 is mandatory, num / 2 won't work**

**Test out by nums = [1,3,2,3,1], expected result = 2 pairs, [3 at index = 1, 1 at index = 4] + [3 at index = 3, 1 at index = 4], but if only call findNumIndexInSortedArray() method with num / 2 the result = 0 pair, the reason is in Java, integer 3 / integer 2 will only be 1, 3 / 2 = 1 has no smaller choice in nums when we talking about two 3's, but double 3 / double 2 will be 1.5, 1.5 still have 1 as choice for both two 3's**

**Style 1: Use HashSet to filter unique element and sort them**

class Solution {

public int reversePairs(int[] nums) {

int n = nums.length;

//int[] bit = new int[n + 1];

//int[] copy = nums.clone();

//Arrays.sort(copy);

int[] copy = filterUniqueValueAndSort(nums);

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

int result = 0;

for(int i = n - 1; i >= 0; i--) {

// Find the first position in the array, where copy[index] >= num / 2, then any rand before index will contribute to the result

int num = nums[i];

result += query(bit, findNumIndexInSortedArray(copy, 1.0 \* num / 2) - 1);

update(bit, findNumIndexInSortedArray(copy, num));

}

return result;

}

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;

}

// 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, double 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;

}

}

**Style 2: Not use HashSet to filter unique element, keep the same input and just sort, it will take longer time since the bucket length based on original input array length will be no less (and always larger) than bucket length based on HashSet size**

class Solution {

public int reversePairs(int[] nums) {

int n = nums.length;

int[] bit = new int[n + 1];

int[] copy = nums.clone();

Arrays.sort(copy);

//int[] copy = filterUniqueValueAndSort(nums);

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

int result = 0;

for(int i = n - 1; i >= 0; i--) {

// Find the first position in the array, where copy[index] >= num / 2, then any rand before index will contribute to the result

int num = nums[i];

result += query(bit, findNumIndexInSortedArray(copy, 1.0 \* num / 2) - 1);

update(bit, findNumIndexInSortedArray(copy, num));

}

return result;

}

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;

}

// 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, double 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;

}

}

**Style 3 ???: 这个方法在应用到L315的时候被证明是有问题的，失败在[5,2,6,1]这个输入，期待是[2,1,1,0]，实际输出[3,0,1,0]，分析原理其实是result += i - query(bit, findNumIndexInSortedArray(copy, val))这个理论有问题，为了追求logN的时间复杂度，我们不得不采用边更新bucket（index是当前有序数组中第一个>=val的数位置，使用update方法更新BIT[1, index]中所有相关buckets）边计算有多少右边的数比当前数小（index是当前有序数组中第一个>=val的数位置，使用query方法获取BIT[1,index]中所有相关buckets中存储的count / frequency）的方式，而这种边更新边计算的模式决定了要在当前数右边的所有数中找有多少个比当前数小的数[L315]或当前数1/2小的数[L493]都【必须】从右往左计算，这样才能满足每轮循环query正确读取上一轮update的结果（debug一下就能看出），而从左往右是无法做到的，下面的解法能过有巧合成分**

**(1) Scan from left to right as normal use HashSet to filter unique element to build bucket (2) Not use HashSet to filter unique element, keep the same input and just sort, it will take longer time since the bucket length based on original input array length will be no less (and always larger) than bucket length based on HashSet size**

**The logic is natural, we will try to find the bucket count sum < given value at index 'i' when scan from left to right, then use 'i - bucket count sum' to get the target value which is the count >= given value at index 'i'**

class Solution {

public int reversePairs(int[] nums) {

int n = nums.length;

//int[] bit = new int[n + 1];

//int[] copy = nums.clone();

//Arrays.sort(copy);

int[] copy = filterUniqueValueAndSort(nums);

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

int result = 0;

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

// Find the last position in the array, where 2 \* copy[i] <= num,

// then any index after this index will contribute to the result,

// we can use current index 'i' minus the query out index

int num = nums[i];

// Must use long type to multiple: num \* 2 -> num \* 2L

// Test case: expect = 15, actual = 0

// [2147483647,2147483647,2147483647,2147483647,2147483647,2147483647]

result += i - query(bit, findNumIndexInSortedArray(copy, num \* 2L));

update(bit, findNumIndexInSortedArray(copy, num));

}

return result;

}

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;

}

// 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 upper boundary

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

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

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

// Find the last element in arr which <= val (upper 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 - 1; // OR return r

}

}

**Refer to**

**Implementation with regular BIT**

<https://leetcode.com/problems/reverse-pairs/solutions/97268/general-principles-behind-problems-similar-to-reverse-pairs/comments/474820>

Implementation with regular BIT

public class Solution {

// T(k) = T(k - 1) + number of IRPs with i in left(0 ~ k - 1), j as nums[k]

public int reversePairs(int[] nums) {

int res = 0;

int[] copy = Arrays.copyOf(nums, nums.length);

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

Arrays.sort(copy);

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

int ele = nums[i];

int ceilingIndex = index(copy, 2L \* ele);

//res += ceilingIndex < bit.length ? i - getSum(bit, ceilingIndex) : 0;

res += i - getSum(bit, ceilingIndex);

// In sorted copy of original input array, find the largest element

// which <= ele, shift up position in BIT array by 1 as 1-based

int updateIndex = index(copy, ele);

update(bit, updateIndex);

}

return res;

}

private int getSum(int[] bit, int index) {

int sum = 0;

while (index > 0) {

sum += bit[index];

index -= index & -index;

}

return sum;

}

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

while (index < bit.length) {

bit[index] += 1;

index += index & -index;

}

}

// Find upper boundary template

private int index(int[] arr, long val) {

// find largest ele in arr which <= val

// shift up by 1

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

}

public static void main(String[] args) {

int[] nums = new int[]{2,4,3,5,1};

Solution so = new Solution();

int result = so.reversePairs(nums);

System.out.println(result);

}

}

**Solution 3: Binary Search Tree (60 min, TLE 36/140)**

**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 int reversePairs(int[] nums) {

Node root = null;

int[] cnt = new int[1];

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

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

search(cnt, root, nums[i] / 2.0);

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

}

return cnt[0];

}

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, Node node, double target) {

if(node == null) {

return;

}

if(target == node.val) {

cnt[0] += node.smaller;

} else if(target < node.val) {

search(cnt, node.left, target);

} else {

cnt[0] += node.smaller + node.same;

search(cnt, node.right, target);

}

}

}

**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/>. 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 int reversePairs(int[] nums) {

Node root = null;

int[] cnt = new int[1];

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

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

search(cnt, root, nums[i] / 2.0); // search and count the partially built tree

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

}

return cnt[0];

}

// 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, Node node, double target){

if(node == null) {

return;

}

if(target == node.val) {

cnt[0] += node.less;

} else if(target < node.val) {

search(cnt, node.left, target);

} else {

cnt[0] += node.less + node.same;

search(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;

}

}

**Solution 4: Merge Sort (180 min)**

**Style 1: void return, counting method inside merge method**

class Solution {

int count = 0;

public int reversePairs(int[] nums) {

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

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

return count;

}

private void mergeSort(int[] nums, int left, int right, int[] 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(nums, left, mid, temp);

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

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

}

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

// Reverse pair counting logic

int l = left;

int r = mid + 1;

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

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

count += (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++];

}

}

}

**Step by step example**

**Every time when hit the reverse pair counting logic, the left and right subarray in the 'merge' logic are both sorted [l:sorted | r:sorted], that's why we can use 'count += (mid - l + 1)', because if 'l' satisfy "arr[l] > 2 \* arr[r]", all indexes between [l, mid] will satisfy because the subarray already sorted in ascending order**

Every time when hit the reverse pair counting logic, the left and right subarray in the 'merge' logic are both sorted [l:sorted | r:sorted], that's why we can use 'count += (mid - l + 1)', because if 'l' satisfy "arr[l] > 2 \* arr[r]", all indexes between [l, mid] will satisfy because the subarray already sorted in ascending order

e.g nums = [4,2,8,7,3,1]

mergeSort -> 4 2 8 7 3 1

mergeSort -> 4 2 8

mergeSort -> 4 2

mergeSort -> 4

=> return 4

mergeSort -> 2

=> return 2

1st merge -> 4 2

=> hit reverse pair counting logic with [l:4 |r:2] => count = 0

=> do the original merge job => 2 4 8 7 3 1

mergeSort -> 8

=> return 8

2nd merge -> 2 4 8

=> hit reverse pair counting logic with [l:2 4 |r:8] => count = 0

=> do the original merge job => 2 4 8 7 3 1

mergeSort -> 7 3 1

mergeSort -> 7 3

mergeSort -> 7

=> return 7

mergeSort -> 3

=> return 3

3rd merge -> 7 3

=> hit reverse pair counting logic with [l:7 | r:3] => count + 1 = 1

=> do the original merge job => 2 4 8 3 7 1

mergeSort -> 1

=> return 1

4th merge -> 3 7 1

=> hit reverse pair counting logic with [l:3 7 | r:1] => count + 2 = 3

=> do the original merge job => 2 4 8 1 3 7

5th merge -> 2 4 8 1 3 7

=> hit reverse pair counting logic with [l:2 4 8 | r:1 3 7] => count + 3 = 6

\*the detail for 3 more reverse pair here:

when l = 1, r = 0 we pick up 4 from left subarray and 1 from right subarray, then we find 4 > 1 \* 2, then count += (mid - l + 1) = 3 + (2 - 1 + 1) = 5, then when l = 2, r = 0 we pick up 8 from left subarray and 1 from right subarray, and 8 > 1 \* 2, then count += (mid - l + 1) = 5 + (2 - 2 + 1) = 6

=> do the original merge job => 1 2 3 4 7 8

**Refer to**

<https://leetcode.com/problems/reverse-pairs/solutions/97319/c-solution-using-merge-sort/comments/1204446>

class Solution {

private:

int count;

void checkCount(vector<int>& nums, int start, int mid, int end) {

int l = start, r = mid + 1;

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

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

count += (mid - l + 1);

r++;

}else {

l++;

}

}

int a[end - start + 1], t = 0;

l = start, r = mid + 1;

while (l <= mid and r <= end) {

if (l <= mid and nums[l] < nums[r]) {

a[t++] = nums[l++];

} else {

a[t++] = nums[r++];

}

}

while (l <= mid) {

a[t++] = nums[l++];

}

while (r <= end) {

a[t++] = nums[r++];

}

for (int i = 0; i < end - start + 1; i++) {

nums[start + i] = a[i];

}

return;

}

void mergeSort(vector<int>& nums, int start, int end){

if(start == end) return;

int mid = (start + end)/2;

mergeSort(nums, start, mid);

mergeSort(nums, mid + 1,end);

checkCount(nums, start, mid, end);

return;

}

public:

int reversePairs(vector<int>& nums) {

if (!nums.size()) return 0;

count = 0;

mergeSort(nums, 0, nums.size()-1);

return count;

}

};

**Refer to: modified based on original void return Merge Sort template**

class Solution {

int count = 0;

public int reversePairs(int[] nums) {

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

mergeSort(nums, 0, nums.length - 1, temporaryArray);

return count;

}

// Recursive function to sort an array using merge sort

private void mergeSort(int[] arr, int left, int right, int[] tempArr) {

if (left >= right) {

return;

}

int mid = (left + right) / 2;

// Sort first and second halves recursively.

mergeSort(arr, left, mid, tempArr);

mergeSort(arr, mid + 1, right, tempArr);

// Merge the sorted halves.

merge(arr, left, mid, right, tempArr);

}

private void merge(int[] arr, int left, int mid, int right, int[] tempArr) {

// Calculate the start and sizes of two halves.

int start1 = left;

int start2 = mid + 1;

int n1 = mid - left + 1;

int n2 = right - mid;

int l = left, r = mid + 1;

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

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

count += (mid - l + 1);

r++;

} else {

l++;

}

}

// Copy elements of both halves into a temporary array.

for(int i = 0; i < n1; i++) {

tempArr[start1 + i] = arr[start1 + i];

}

for (int i = 0; i < n2; i++) {

tempArr[start2 + i] = arr[start2 + i];

}

// Merge the sub-arrays 'in tempArray' back into the original array 'arr' in sorted order.

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

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

arr[k] = tempArr[start1 + i];

i += 1;

} else {

arr[k] = tempArr[start2 + j];

j += 1;

}

k += 1;

}

// Copy remaining elements

while (i < n1) {

arr[k] = tempArr[start1 + i];

i += 1;

k += 1;

}

while (j < n2) {

arr[k] = tempArr[start2 + j];

j += 1;

k += 1;

}

}

}

**Style 2: int return, counting method inside merge method**

class Solution {

public int reversePairs(int[] nums) {

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

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

}

private int mergeSort(int[] nums, int left, int right, int[] 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 0;

}

int count = 0;

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

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

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

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

return count;

}

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

int count = 0;

// Reverse pair counting logic

int l = left;

int r = mid + 1;

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

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

count += (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++];

}

return count;

}

}

**Refer to**

<https://www.jiuzhang.com/solutions/reverse-pairs/>

<https://medium.com/@sheefanaaz6417/493-reverse-pairs-leetcode-step-by-step-approach-3264f3f11e1b>

public class Solution {

/\*\*

\* @param A an array

\* @return total of reverse pairs

\*/

public long reversePairs(int[] A) {

return mergeSort(A, 0, A.length - 1);

}

private int mergeSort(int[] A, int start, int end) {

if (start >= end) {

return 0;

}

int mid = (start + end) / 2;

int sum = 0;

sum += mergeSort(A, start, mid);

sum += mergeSort(A, mid+1, end);

sum += merge(A, start, mid, end);

return sum;

}

private int merge(int[] A, int start, int mid, int end) {

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

int leftIndex = start;

int rightIndex = mid + 1;

int index = start;

int sum = 0;

while (leftIndex <= mid && rightIndex <= end) {

if (A[leftIndex] <= A[rightIndex]) {

temp[index++] = A[leftIndex++];

} else {

temp[index++] = A[rightIndex++];

sum += mid - leftIndex + 1;

}

}

while (leftIndex <= mid) {

temp[index++] = A[leftIndex++];

}

while (rightIndex <= end) {

temp[index++] = A[rightIndex++];

}

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

A[i] = temp[i];

}

return sum;

}

}

**Why we can use Merge Sort ?**

**Refer to**

<https://grandyang.com/leetcode/493/>

fun4LeetCode 大神归纳的第二种方法叫做分割重现关系 (Partition Recurrence Relation)，用式子表示是 T(i, j) = T(i, m) + T(m+1, j) + C。这里的C就是处理合并两个部分的子问题，那么用文字来描述就是“已知翻转对的两个数字分别在子数组 nums[i, m] 和 nums[m+1, j] 之中，求满足要求的翻转对的个数”，这里翻转对的两个条件中的顺序条件已经满足，就只需要找到满足大小关系的的数对即可。这里两个数字都是不确定的，如果用暴力搜索肯定会被 OJ 唾弃。但是如果两个子数组是有序的，那么我们可以用双指针的方法在线性时间内就可以统计出符合题意的翻转对的个数。要想办法产生有序的子数组，那么这就和 MergeSort 的核心思想完美匹配了。我们知道混合排序就是不断的将数组对半拆分成子数组，拆到最小的数组后开始排序，然后一层一层的返回，最后原数组也是有序的了。这里我们在混合排序的递归函数中，对有序的两个子数组进行统计翻转对的个数，区间 [left, mid] 和 [mid+1, right] 内的翻转对儿个数就被分别统计出来了，此时还要统计翻转对儿的两个数字分别在两个区间中的情况，那么i遍历 [left, mid] 区间所有的数字，j则从 mid+1 开始检测，假如 nums[i] 大于 nums[j] 的二倍，则这两个数字就是翻转对，此时j再自增1，直到不满足这个条件停止，则j增加的个数就是符合题意的翻转对的个数，所以用当前的j减去其初始值 mid+1 即为所求，然后再逐层返回，这就完美的实现了上述的分割重现关系的思想

**General principles behind problems similar to "Reverse Pairs"**

**Refer to**

<https://leetcode.com/problems/reverse-pairs/solutions/97268/general-principles-behind-problems-similar-to-reverse-pairs/>

**【注意，这里总结的是思路性的问题，所列举的3个样例方法的代码几乎全部没法很容易的使用和理解，代码本身还是参考前面所列举的具体样例，但是思路还是很值得一看】**

It looks like a host of solutions are out there (BST-based, BIT-based, Merge-sort-based). Here I'd like to focus on the general principles behind these solutions and its possible application to a number of similar problems.

The fundamental idea is very simple: **break down the array and solve for the subproblems**.

A breakdown of an array naturally reminds us of subarrays. To smoothen our following discussion, let's assume the input array is nums, with a total of n elements. Let nums[i, j] denote the subarray starting from index i to index j (both inclusive), T(i, j) as the same problem applied to this subarray (for example, for **[Reverse Pairs](https://leetcode.com/problems/reverse-pairs/?tab=Description)**, T(i, j) will represent the total number of important reverse pairs for subarray nums[i, j]).

With the definition above, it's straightforward to identify our original problem as T(0, n - 1). Now the key point is how to construct solutions to the original problem from its subproblems. This is essentially equivalent to building recurrence relations for T(i, j). Since if we can find solutions to T(i, j) from its subproblems, we surely can build solutions to larger subarrays until eventually the whole array is spanned.

While there may be many ways for establishing recurrence relations for T(i, j), here I will only introduce the following two common ones:

* T(i, j) = T(i, j - 1) + C, i.e., elements will be processed sequentially and C denotes the subproblem for processing the last element of subarray nums[i, j]. We will call this sequential recurrence relation.
* T(i, j) = T(i, m) + T(m + 1, j) + C where m = (i+j)/2, i.e., subarray nums[i, j] will be further partitioned into two parts and C denotes the subproblem for combining the two parts. We will call this partition recurrence relation.

For either case, the nature of the subproblem C will depend on the problem under consideration, and it will determine the overall time complexity of the original problem. So usually it's crucial to find efficient algorithm for solving this subproblem in order to have better time performance. Also pay attention to possibilities of overlapping subproblems, in which case a dynamic programming (DP) approach would be preferred.

Next, I will apply these two recurrence relations to this problem "Reverse Pairs" and list some solutions for your reference.

**I -- Sequential recurrence relation**

Again we assume the input array is nums with n elements and T(i, j) denotes the total number of important reverse pairs for subarray nums[i, j]. For sequential recurrence relation, we can set i = 0, i.e., the subarray always starts from the beginning. Therefore we end up with:

T(0, j) = T(0, j - 1) + C

where the subproblem C now becomes **"find the number of important reverse pairs with the first element of the pair coming from subarray nums[0, j - 1] while the second element of the pair being nums[j]"**.

Note that for a pair (p, q) to be an important reverse pair, it has to satisfy the following two conditions:

* p < q: the first element must come before the second element;
* nums[p] > 2 \* nums[q]: the first element has to be greater than twice of the second element.

For subproblem C, the first condition is met automatically; so we only need to consider the second condition, which is equivalent to searching for all elements within subarray nums[0, j - 1] that are greater than twice of nums[j].

The straightforward way of searching would be a linear scan of the subarray, which runs at the order of O(j). From the sequential recurrence relation, this leads to the naive O(n^2) solution.

To improve the searching efficiency, a key observation is that the order of elements in the subarray does not matter, since we are only interested in the total number of important reverse pairs. This suggests we may sort those elements and do a binary search instead of a plain linear scan.

If the searching space (formed by elements over which the search will be done) is "static" (it does not vary from run to run), placing the elements into an array would be perfect for us to do the binary search. However, this is not the case here. After the j-th element is processed, we need to add it to the searching space so that it becomes searchable for later elements, which renders the searching space expanding as more and more elements are processed.

Therefore we'd like to strike a balance between searching and insertion operations. This is where data structures like binary search tree (BST) or binary indexed tree (BIT) prevail, which offers relatively fast performance for both operations.

**1. BST-based solution**

we will define the tree node as follows, where val is the node value and cnt is the total number of elements in the subtree rooted at current node that are greater than or equal to val:

class Node {

int val, cnt;

Node left, right;

Node(int val) {

this.val = val;

this.cnt = 1;

}

}

The searching and insertion operations can be done as follows:

private int search(Node root, long val) {

if (root == null) {

return 0;

} else if (val == root.val) {

return root.cnt;

} else if (val < root.val) {

return root.cnt + search(root.left, val);

} else {

return search(root.right, val);

}

}

private Node insert(Node root, int val) {

if (root == null) {

root = new Node(val);

} else if (val == root.val) {

root.cnt++;

} else if (val < root.val) {

root.left = insert(root.left, val);

} else {

root.cnt++;

root.right = insert(root.right, val);

}

return root;

}

And finally the main program, in which we will search for all elements no less than twice of current element plus 1 (converted to long type to avoid overflow) while insert the element itself into the BST.

**Note:** this homemade BST is not self-balanced and the time complexity can go as bad as O(n^2) (in fact you will get TLE if you copy and paste the solution here). To guarantee O(nlogn) performance, use one of the self-balanced BST's (e.g. Red-black tree, AVL tree, etc.).

public int reversePairs(int[] nums) {

int res = 0;

Node root = null;

for (int ele : nums) {

res += search(root, 2L \* ele + 1);

root = insert(root, ele);

}

return res;

}

**2. BIT-based solution**

For BIT, the searching and insertion operations are:

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

int sum = 0;

while (i < bit.length) {

sum += bit[i];

i += i & -i;

}

return sum;

}

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

while (i > 0) {

bit[i] += 1;

i -= i & -i;

}

}

And the main program, where again we will search for all elements greater than twice of current element while insert the element itself into the BIT. For each element, the "index" function will return its index in the BIT. Unlike the BST-based solution, this is guaranteed to run at O(nlogn).

public int reversePairs(int[] nums) {

int res = 0;

int[] copy = Arrays.copyOf(nums, nums.length);

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

Arrays.sort(copy);

for (int ele : nums) {

res += search(bit, index(copy, 2L \* ele + 1));

insert(bit, index(copy, ele));

}

return res;

}

private int index(int[] arr, long 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;

}

}

return l + 1;

}

More explanation for the BIT-based solution:

1. We want the elements to be sorted so there is a sorted version of the input array which is copy.
2. The bit is built upon this sorted array. Its length is one greater than that of the copy array to account for the root.
3. Initially the bit is empty and we start doing a sequential scan of the input array. For each element being scanned, we first search the bit to find all elements greater than twice of it and add the result to res. We then insert the element itself into the bit for future search.
4. Note that conventionally searching of the bit involves traversing towards the root from some index of the bit, which will yield a predefined running total of the copy array up to the corresponding index. For insertion, the traversing direction will be opposite and go from some index towards the end of the bit array.
5. For each scanned element of the input array, its searching index will be given by the index of the first element in the copy array that is greater than twice of it (shifted up by 1 to account for the root), while its insertion index will be the index of the first element in the copy array that is no less than itself (again shifted up by 1). This is what the index function is for.
6. For our case, the running total is simply the number of elements encountered during the traversal process. If we stick to the convention above, the running total will be the number of elements smaller than the one at the given index, since the copy array is sorted in ascending order. However, we'd actually like to find the number of elements greater than some value (i.e., twice of the element being scanned), therefore we need to flip the convention. This is what you see inside the search and insert functions: the former traversing towards the end of the bit while the latter towards the root.

**II -- Partition recurrence relation**

For partition recurrence relation, setting i = 0, j = n - 1, m = (n-1)/2, we have:

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

where the subproblem C now reads **"find the number of important reverse pairs with the first element of the pair coming from the left subarray nums[0, m] while the second element of the pair coming from the right subarray nums[m + 1, n - 1]"**.

Again for this subproblem, the first of the two aforementioned conditions is met automatically. As for the second condition, we have as usual this plain linear scan algorithm, applied for each element in the left (or right) subarray. This, to no surprise, leads to the O(n^2) naive solution.

Fortunately the observation holds true here that the order of elements in the left or right subarray does not matter, which prompts sorting of elements in both subarrays. With both subarrays sorted, the number of important reverse pairs can be found in linear time by employing the so-called two-pointer technique: one pointing to elements in the left subarray while the other to those in the right subarray and both pointers will go only in one direction due to the ordering of the elements.

The last question is which algorithm is best here to sort the subarrays. Since we need to partition the array into halves anyway, it is most natural to adapt it into a Merge-sort. Another point in favor of Merge-sort is that the searching process above can be embedded seamlessly into its merging stage.

So here is the Merge-sort-based solution, where the function "reversePairsSub" will return the total number of important reverse pairs within subarray nums[l, r]. The two-pointer searching process is represented by the nested while loop involving variable p, while the rest is the standard merging algorithm.

public int reversePairs(int[] nums) {

return reversePairsSub(nums, 0, nums.length - 1);

}

private int reversePairsSub(int[] nums, int l, int r) {

if (l >= r) return 0;

int m = l + ((r - l) >> 1);

int res = reversePairsSub(nums, l, m) + reversePairsSub(nums, m + 1, r);

int i = l, j = m + 1, k = 0, p = m + 1;

int[] merge = new int[r - l + 1];

while (i <= m) {

while (p <= r && nums[i] > 2 L \* nums[p]) p++;

res += p - (m + 1);

while (j <= r && nums[i] >= nums[j]) merge[k++] = nums[j++];

merge[k++] = nums[i++];

}

while (j <= r) merge[k++] = nums[j++];

System.arraycopy(merge, 0, nums, l, merge.length);

return res;

}

**III -- Summary**

Many problems involving arrays can be solved by breaking down the problem into subproblems applied on subarrays and then link the solution to the original problem with those of the subproblems, to which we have sequential recurrence relation and partition recurrence relation. For either case, it's crucial to identify the subproblem C and find efficient algorithm for approaching it.

If the subproblem C involves searching on "dynamic searching space", try to consider data structures that support relatively fast operations on both searching and updating (such as self-balanced BST, BIT, Segment tree, ...).

If the subproblem C of partition recurrence relation involves sorting, Merge-sort would be a nice sorting algorithm to use. Also, the code could be made more elegant if the solution to the subproblem can be embedded into the merging process.

If there are overlapping among the subproblems T(i, j), it's preferable to cache the intermediate results for future lookup.

Lastly let me name a few leetcode problems that fall into the patterns described above and thus can be solved with similar ideas.

[315. Count of Smaller Numbers After Self](https://leetcode.com/problems/count-of-smaller-numbers-after-self/)

[327. Count of Range Sum](https://leetcode.com/problems/count-of-range-sum/)

For leetcode 315, applying the sequential recurrence relation (with j fixed), the subproblem C reads: **find the number of elements out of visited ones that are smaller than current element**, which involves searching on "dynamic searching space"; applying the partition recurrence relation, we have a subproblem C: **for each element in the left half, find the number of elements in the right half that are smaller than it**, which can be embedded into the merging process by noting that these elements are exactly those swapped to its left during the merging process.

For leetcode 327, applying the sequential recurrence relation (with j fixed) on the pre-sum array, the subproblem C reads: **find the number of elements out of visited ones that are within the given range**, which again involves searching on "dynamic searching space"; applying the partition recurrence relation, we have a subproblem C: **for each element in the left half, find the number of elements in the right half that are within the given range**, which can be embedded into the merging process using the two-pointer technique.