Count Reverse Pairs

Mark as Completed
185

Problem Statement: Given an array of numbers, you need to return the count of reverse pairs. **Reverse Pairs** are those pairs where i<j and arr[i]>2*arr[j].

Examples

Brute Force Approach
Algorithm / Intuition

Solution:

This question is slightly different from the question count inversion where the condition was a[i] > a[j] but here in this question, the condition is a[i] > 2*a[j]. In both questions, the index i < j.

Naive Approach (Brute-force):

Approach:

The steps are as follows:

1. First, we will run a loop(say i) from 0 to N-1 to select the a[i].
2. As index j should be greater than index i, inside loop i, we will run another loop i.e. j from i+1 to N-1, and select the element a[j].
3. Inside this second loop, we will check if a[i] > 2*a[j] i.e. if a[i] and a[j] can be a pair. If they satisfy the condition, we will increase the count by 1.
4. Finally, we will return the count i.e. the number of such pairs.

Intuition:

The naive approach is pretty straightforward. We will use nested loops to generate all possible pairs. We know index i must be smaller than index j. So, we will fix i at one index at a time through a loop, and with another loop, we will check(*the condition a[i] > 2*a[j]*) the elements from index i+1 to N-1 if they can form a pair with a[i].

Note: *For a better understanding of intuition, please watch the video at the bottom of the page.*

Code

```
import java.util.*;

public class Main {

    public static int countPairs(int[] a, int n) {

        // Count the number of pairs:
        int cnt = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > 2 * a[j])
                    cnt++;
            }
        }
        return cnt;
    }

    public static int team(int[] skill, int n) {
        return countPairs(skill, n);
    }

    public static void main(String[] args) {
        int[] a = {4, 1, 2, 3, 1};
        int n = 5;
        int cnt = team(a, n);
        System.out.println("The number of reverse pair is: " + cnt);
    }
}
```

Output: The number of reverse pair is: 3

Complexity Analysis

Time Complexity: O(N2), where N = size of the given array.
Reason: We are using nested loops here and those two loops roughly run for N times.

Space Complexity: O(1) as we are not using any extra space to solve this problem.

Optimal Approach
Algorithm / Intuition

Optimal Approach:

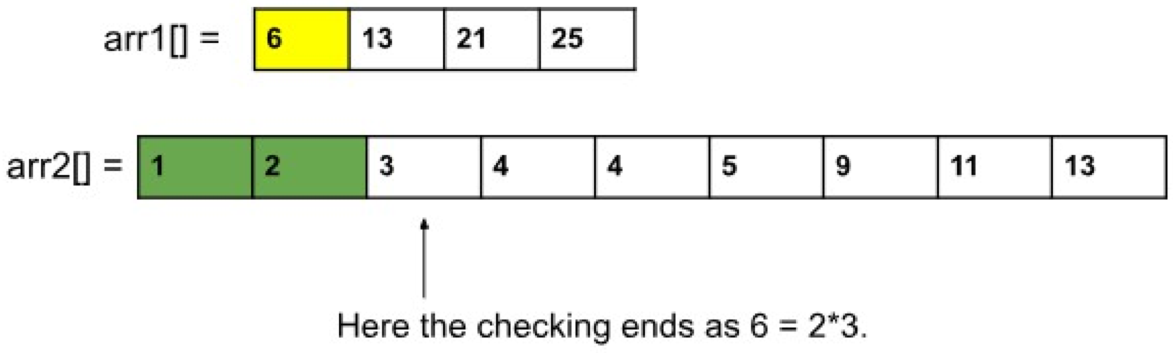
Intuition:

In order to solve this problem we will use the merge sort algorithm like we used in the problem count inversion with a slight modification of the merge() function. But in this case, the same logic will not work. In order to understand this, we need to deep dive into the merge() function.

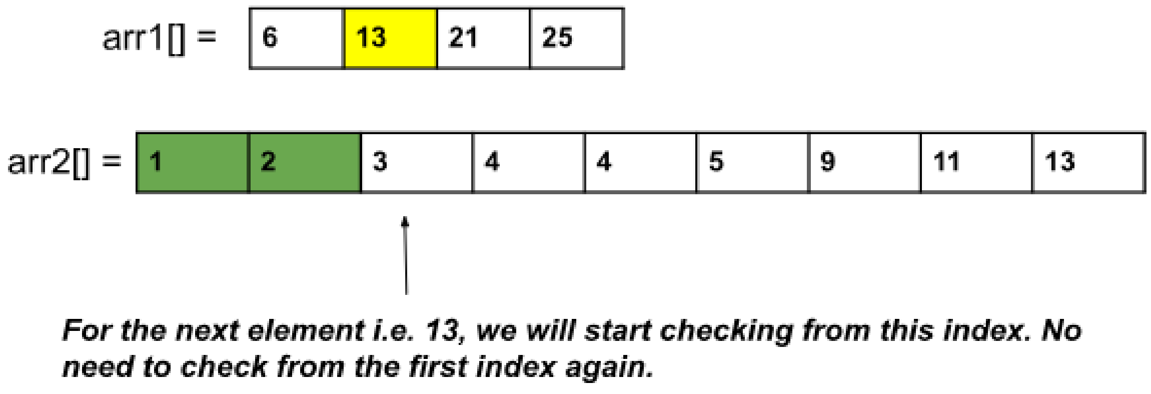
Why the same logic of count inversion will not work?

- The merge function works by comparing two elements from two halves i.e. arr[left] and arr[right]. Now, the condition in the question was arr[i] > arr[j]. That is why we merged the logic. While comparing the elements, we counted the number of pairs.
- But in this case, the condition is arr[i] > 2*arr[j]. And, we cannot change the condition of comparing the elements in the merge() function. If we change the condition, the merge() function will fail to merge the elements. So, we need to check this condition and count the number of pairs separately.

Here, our approach will be to check, for every element in the sorted left half(*sorted*), how many elements in the right half(*also sorted*) can make a pair. Let's try to understand, using the following example:



For the first element of the left half i.e. 6, we will start checking from index 0 of the right half i.e. arr2[]. Now, we can clearly see that the first two elements of arr2[] can make a pair with arr1[0] i.e. 6.


arr1[] =
6
13
21
25
arr2[] =
1
2
3
4
4
5
9
11
13
For the next element i.e. 13, we will start checking from this index. No need to check from the first index again.

For the next element i.e. arr1[1], we will start checking from index 2(*0-based indexing*) i.e. where we stopped for the previous element.

Note: *This process will work because arr1[1] will always be greater than arr1[0] which concludes if arr2[0] and arr2[1] are making a pair with arr1[0], they will obviously make pairs with a number greater than arr1[0] i.e. arr1[1].*

Thus before the merge step in the merge sort algorithm, we will calculate the total number of pairs each time.

Approach:

The steps are basically the same as they are in the case of the merge sort algorithm. The change will be just in the mergeSort() function:

- *In order to count the number of pairs, we will keep a count variable, cnt, initialized to 0 beforehand inside the mergeSort().*
- *We will add the numbers returned by the previous mergeSort() calls.*

- *Before the merge step, we will count the number of pairs using a function, named countPairs().*
- *We need to remember that the left half starts from low and ends at mid, and the right half starts from mid+1 and ends at high.*

The steps of the countPairs() function will be as follows:

1. We will declare a variable, cnt, initialized with 0.
2. We will run a loop from low to mid, to select an element at a time from the left half.
3. Inside that loop, we will use another loop to check how many elements from the right half can make a pair.
4. Lastly, we will add the total number of elements i.e. (right-(mid+1)) (where *right = current index)*, to the cnt and return it.

Note: *For a better understanding of intuition, please watch the video at the bottom of the page.*

Code

```
import java.util.*;

public class Main {

    private static void merge(int[] arr, int low, int mid, int high) {
        ArrayList<Integer> temp = new ArrayList<>(); // temporary array
        int left = low;      // starting index of left half of arr
        int right = mid + 1;   // starting index of right half of arr

        //storing elements in the temporary array in a sorted manner//

        while (left <= mid && right <= high) {
            if (arr[left] <= arr[right]) {
                temp.add(arr[left]);
                left++;
            } else {
                temp.add(arr[right]);
                right++;
            }
        }

        // if elements on the left half are still left //

        while (left <= mid) {
            temp.add(arr[left]);
            left++;
        }

        //  if elements on the right half are still left //
        while (right <= high) {
            temp.add(arr[right]);
            right++;
        }

        // transfering all elements from temporary to arr //
        for (int i = low; i <= high; i++) {
            arr[i] = temp.get(i - low);
        }
    }

    public static int countPairs(int[] arr, int low, int mid, int high) {
        int right = mid + 1;
        int cnt = 0;
        for (int i = low; i <= mid; i++) {
            while (right <= high && arr[i] > 2 * arr[right]) right++;
            cnt += (right - (mid + 1));
        }
        return cnt;
    }

    public static int mergeSort(int[] arr, int low, int high) {
        int cnt = 0;
        if (low >= high) return cnt;
        int mid = (low + high) / 2 ;
        cnt += mergeSort(arr, low, mid);  // left half
        cnt += mergeSort(arr, mid + 1, high); // right half
        cnt += countPairs(arr, low, mid, high); //Modification
        merge(arr, low, mid, high);  // merging sorted halves
        return cnt;
    }

    public static int team(int[] skill, int n) {
        return mergeSort(skill, 0, n - 1);
    }
    public static void main(String[] args) {
        int[] a = {4, 1, 2, 3, 1};
        int n = 5;
        int cnt = team(a, n);
        System.out.println("The number of reverse pair is: " + cnt);
    }
}
```

Output: The number of reverse pair is: 3

Complexity Analysis

Time Complexity: O(2N*logN), where N = size of the given array.
Reason: Inside the mergeSort() we call merge() and countPairs() except mergeSort() itself. Now, inside the function countPairs(), though we are running a nested loop, we are actually iterating the left half once and the right half once in total. That is why, the time complexity is O(N). And the merge() function also takes O(N). The mergeSort() takes O(logN) time complexity. Therefore, the overall time complexity will be O(logN * (N+N)) = O(2N*logN).

Space Complexity: O(N), as in the merge sort We use a temporary array to store elements in sorted order.