Count inversions in an array

Problem Statement: Given an array of N integers, count the inversion of the array (using merge-sort).

What is an inversion of an array? Definition: for all i & j < size of array, if i < j then you have to find pair (A[i],A[j]) such that A[j] < A[i].

Examples

```
Example 1:
Input Format: N = 5, array[] = {1,2,3,4,5}
Result: 0
Explanation: we have a sorted array and the sorted array has 0 inversions as for i < j you will
never find a pair such that A[j] < A[i]. More clear example: 2 has index 1 and 5 has index 4 now 1 <
5 but 2 < 5 so this is not an inversion.

Example 2:
Input Format: N = 5, array[] = {5,4,3,2,1}
Result: 10
Explanation: we have a reverse sorted array and we will get the maximum inversions as for i < j we
will always find a pair such that A[j] < A[i]. Example: 5 has index 0 and 3 has index 2 now (5,3)
pair is inversion as 0 < 2 and 5 > 3 which will satisfy out conditions and for reverse sorted array
we will get maximum inversions and that is (n)*(n-1) / 2.For above given array there is 4 + 3 + 2 +
1 = 10 inversions.

Example 3:
Input Format: N = 5, array[] = {5,3,2,1,4}
Result: 7
Explanation: There are 7 pairs (5,1), (5,3), (5,2), (5,4),(3,2), (3,1), (2,1) and we have left 2
pairs (2,4) and (1,4) as both are not satisfy our condition.
```

Brute Force Approach
Algorithm / Intuition

Solution:

Let's understand the Question more deeply. We are required to give the total number of inversions and the inversions are: For i & j < size of an array if i < j then you have to find pair (a[i], a[j]) such that a[i] > a[j].

For example, for the given array: [5,3,2,1,4], (5, 3) will be a valid pair as 5 > 3 and
index 0 < index 1. But (1, 4) cannot be valid pair.

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 first element in the pair.
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.
3. Inside this second loop, we will check if a[i] > 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 solve this problem. 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] > a[j]*) the elements from index i+1 to N-1 if they can form a pair with a[i]. This is the first naive approach we can think of.

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

Output: The number of inversions is: 10

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

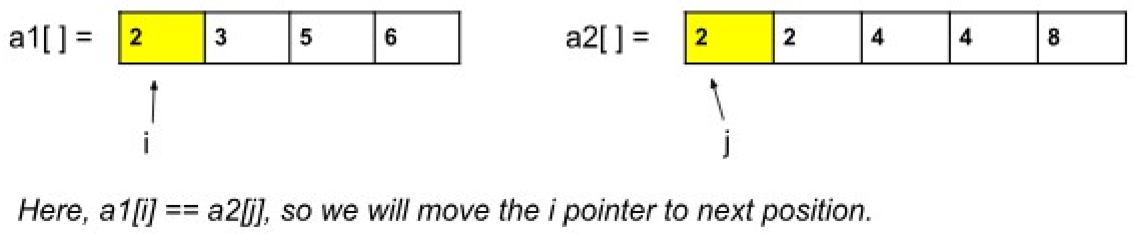
Observation:

Let’s build the intuition for this approach using a modified version of the given question.

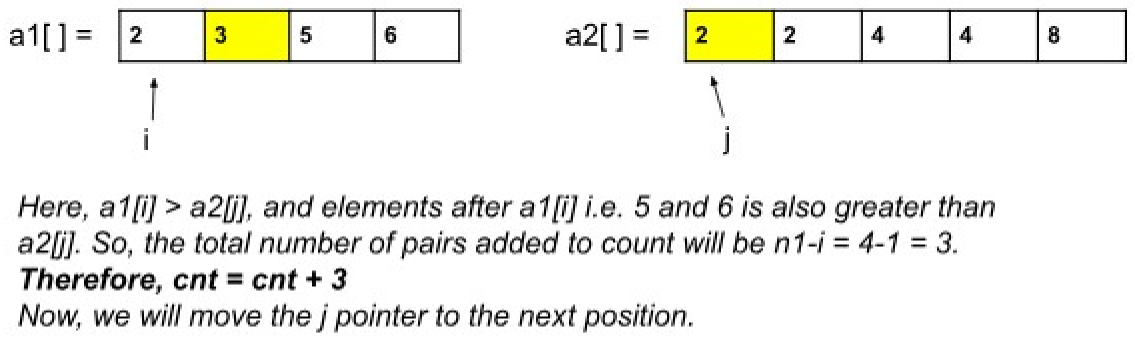
Assume two sorted arrays are given i.e. a1[] = {2, 3, 5, 6} and a2[] = {2, 2, 4, 4, 8}. Now, we have to count the pairs i.e. a1[i] and a2[j] such that a1[i] > a2[j].

In order to solve this, we will keep two pointers i and j, where i will point to the first index of a1[] and j will point to the first index of a2[]. Now in each iteration, we will do the following:

- **If a1[i] <= a2[j]:** These two elements cannot be a pair and so we will move the pointer i to the next position. This case is illustrated below:


a1[ ] =
2
3
5
6
a2[ ] =
2
2
4
4
8
i
j
Here, a1[i] == a2[j], so we will move the i pointer to next position.

- **Why we moved the i pointer:** We know, that the given arrays are sorted. So, all the elements after the pointer j, should be greater than a2[j]. Now, as a1[i] is smaller or equal to a2[j], it is obvious that a1[i] will be smaller or equal to all the elements after a2[j]. We need a bigger value of a1[i] to make a pair and so we move the i pointer to the next position i.e. next bigger value.
- **If a1[i] > a2[j]:** These two elements can be a pair and so we will update the count of pairs. Now, here, we should observe that as a1[i] is greater than a2[j], all the elements after a1[i] will also be greater than a2[j] and so, those elements will also make pair with a2[j]. So, the number of pairs added will be n1-i (*where n1 = size of a1[ ]*). Now, we will move the j pointer to the next position. This case is also illustrated below:


a1[ ] =
2
3
5
6
a2[ ] =
2
2
4
4
8
i
j
Here, a1[i] > a2[j], and elements after a1[i] i.e. 5 and 6 is also greater than a2[j]. So, the total number of pairs added to count will be n1-i = 4-1 = 3.
Therefore, cnt = cnt + 3
Now, we will move the j pointer to the next position.

The above process will continue until at least one of the pointers reaches the end.

Until now, we have figured out how to count the number of pairs in one go if two sorted arrays are given. But in our actual question, only a single unsorted array is given. So, how to break it into two sorted halves so that we can apply the above observation?

We can think of the merge sort algorithm that works in a similar way we want. In the merge sort algorithm, at every step, we divide the given array into two halves and then sort them, and while doing that we can actually count the number of pairs.

Basically, we will use the merge sort algorithm to use the observation in the correct way.

Approach:

The steps are basically the same as they are in the case of the merge sort algorithm. The change will be just a one-line addition inside the **merge()** function. Inside the merge(), we need to add the number of pairs to the count when a[left] > a[right].

The steps of the merge() function were the following:

1. In the merge function, we will use a temp array to store the elements of the two sorted arrays after merging. Here, the range of the left array is low to mid and the range for the right half is mid+1 to high.
2. Now we will take two pointers left and right, where left starts from low and right starts from mid+1.
3. Using a while loop(while(left <= mid && right <= high)), we will select two elements, one from each half, and will consider the smallest one among the two. Then, we will insert the smallest element in the temp array.
4. After that, the left-out elements in both halves will be copied as it is into the temp array.
5. Now, we will just transfer the elements of the temp array to the range low to high in the original array.

Modifications in merge() and mergeSort():

- *In order to count the number of pairs, we will keep a count variable, cnt, initialized to 0 beforehand inside the merge().*
- *While comparing a[left] and a[right] in the 3rd step of merge(), if a[left] > a[right], we will simply add this line:*
  cnt += mid-left+1 (mid+1 = size of the left half)
- Now, we will return this cnt from merge() to mergeSort().
- Inside mergeSort(), we will keep another counter variable that will store the final answer. With this cnt, we will add the answer returned from mergeSort() of the left half, mergeSort() of the right half, and merge().
- Finally, we will return this cnt, as our answer from mergeSort().

Note: The code implementation will further clarify the modifications.

Dry Run: Please refer to the video for a better understanding of the dry run.

Code
C++JavaPythonJavaScript

```
import java.util.*;

public class Main {

    private static int 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

        //Modification 1: cnt variable to count the pairs:
        int cnt = 0;

        //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]);
                cnt += (mid - left + 1); //Modification 2
                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);
        }
        return cnt; // Modification 3
    }

    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 += merge(arr, low, mid, high);  // merging sorted halves
        return cnt;
    }

    public static int numberOfInversions(int[] a, int n) {
        // Count the number of pairs:
        return mergeSort(a, 0, n - 1);
    }


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

Output: The number of inversions is: 10

Complexity Analysis

Time Complexity: O(N*logN), where N = size of the given array.
Reason: We are not changing the merge sort algorithm except by adding a variable to it. So, the time complexity is as same as the merge sort.

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