Count Inversions in Array

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in reverse order, the inversion count is the maximum.

Given an array **a[]**. The task is to find the inversion count of **a[]**. Where two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

Examples:

> ***Input:*** *arr[] = {8, 4, 2, 1}*
> ***Output:*** *6*
> ***Explanation:*** *Given array has six inversions: (8, 4), (4, 2), (8, 2), (8, 1), (4, 1), (2, 1).*
>
> ***Input:*** *arr[] = {1, 20, 6, 4, 5}*
> ***Output:*** *5*
> ***Explanation:*** *Given array has five inversions: (20, 6), (20, 4), (20, 5), (6, 4), (6, 5).*

Naive Approach:

> *Traverse through the array, and for every index, find the number of smaller elements on its right side of the array. This can be done using a nested loop. Sum up the counts for all indices in the array and print the sum.*

Follow the below steps to Implement the idea:

- Traverse through the array from start to end
- For every element, find the count of elements smaller than the current number up to that index using another loop.
- Sum up the count of inversion for every index.
- Print the count of inversions.

Below is the Implementation of the above approach:

C++

```
// C++ program to Count Inversions
// in an array
#include <bits/stdc++.h>
using namespace std;

int getInvCount(int arr[], int n)
{
    int inv_count = 0;
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            if (arr[i] > arr[j])
                inv_count++;

    return inv_count;
}

// Driver Code
int main()
```

```
{
    int arr[] = { 1, 20, 6, 4, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << " Number of inversions are "
        << getInvCount(arr, n);
    return 0;
}

// This code is contributed
// by Akanksha Rai
```


Output

```
Number of inversions are 5
```

Time Complexity: O(N^2), Two nested loops are needed to traverse the array from start to end.
Auxiliary Space: O(1), No extra space is required.

Count Inversions in an array using Merge Sort:

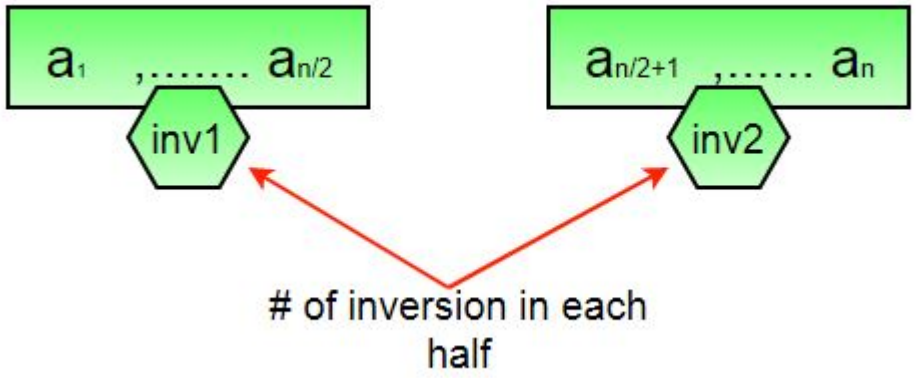
Below is the idea to solve the problem:

> *Use **Merge sort** with modification that every time an unsorted pair is found increment **count** by one and return count at the end.*

Illustration:

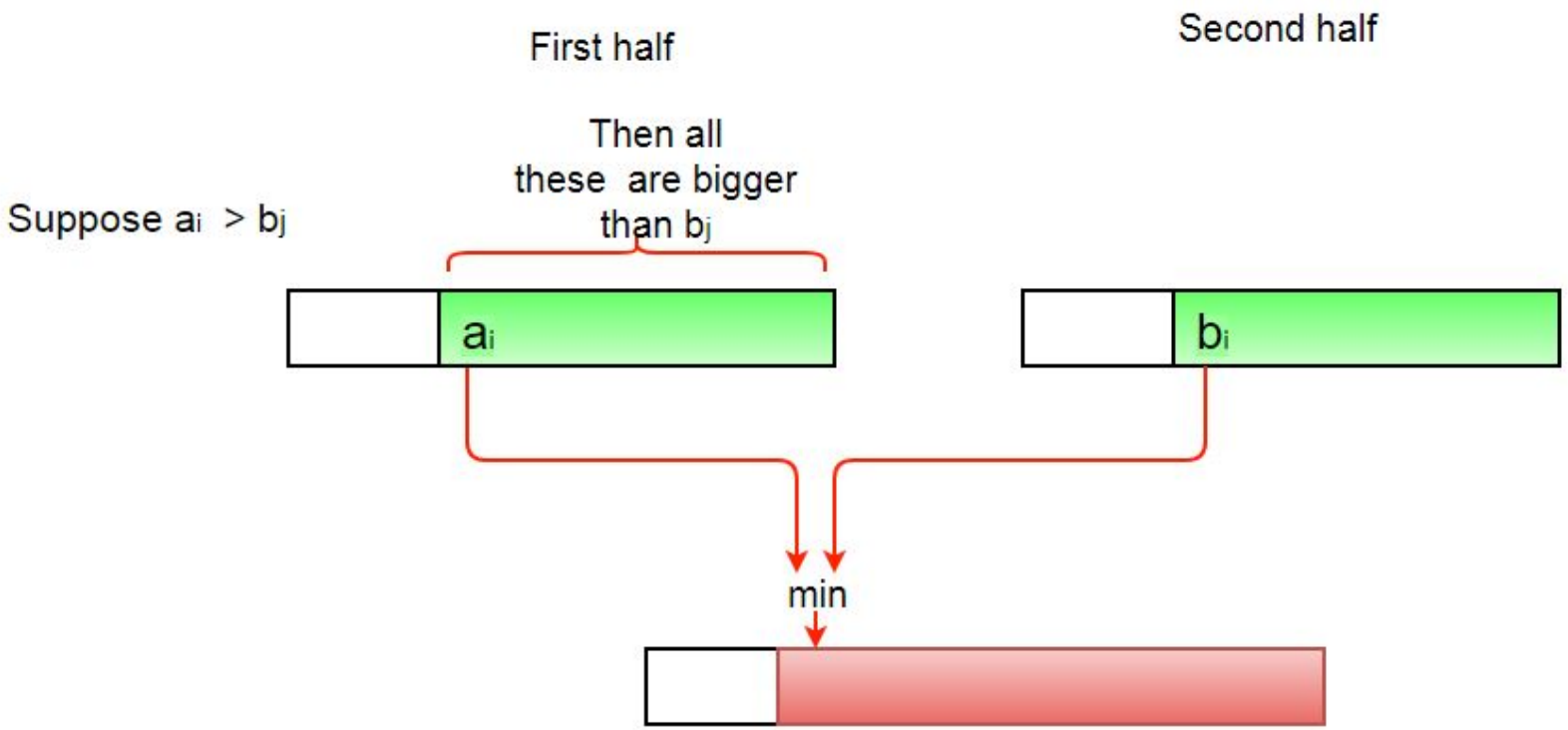
Dash
All
Articles
Videos
Problems
Quiz

Suppose the number of inversions in the left half and right half of the array (let be inv1 and inv2); what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is – the inversions that need to be counted during the merge step. Therefore, to get the total number of inversions that needs to be added are the number of inversions in the left subarray, right subarray, and merge().

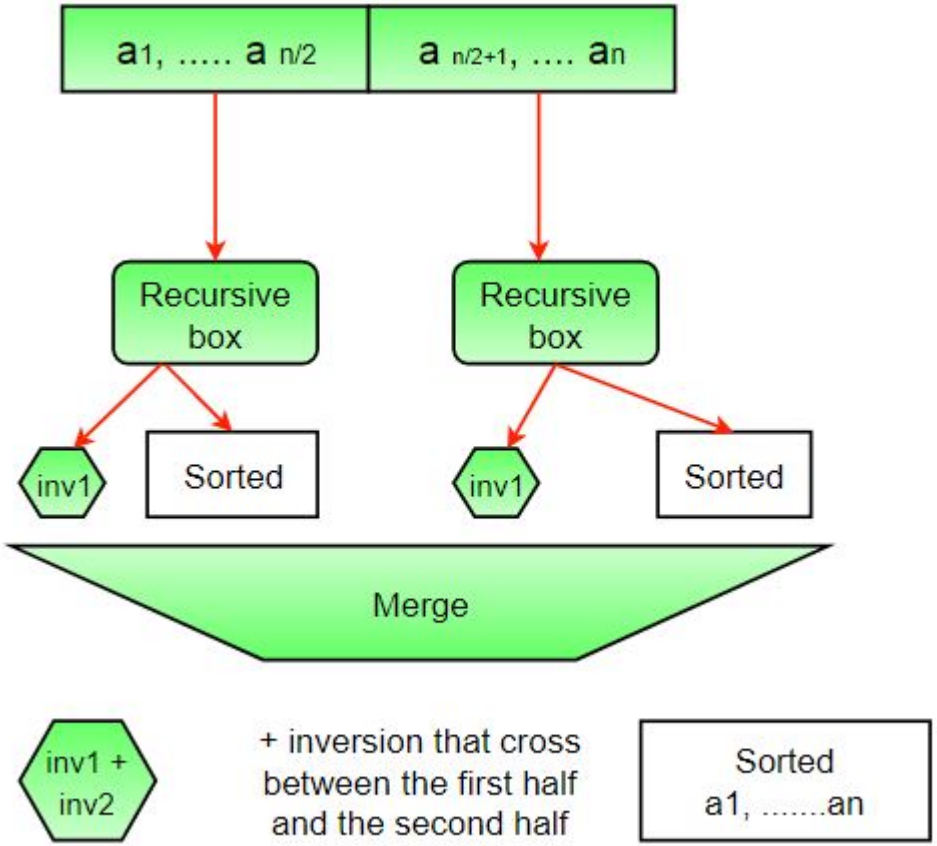


How to get** the **number of inversions in merge()?

In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in merge(), if a[i] is greater than a[j], then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] ... a[mid]) will be greater than a[j]



The complete picture:



Follow the below steps to Implement the idea

- The idea is similar to merge sort, divide the array into two equal or almost equal halves in each step until the base case is reached.
- Create a function merge that counts the number of inversions when two halves of the array are merged,
  - Create two indices i and j, i is the index for the first half, and j is an index of the second half.

- If a[i] is greater than a[j], then there are (mid – i) inversions because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j].

- Create a recursive function to divide the array into halves and find the answer by summing the number of inversions in the first half, the number of inversions in the second half and the number of inversions by merging the two.
  - The base case of recursion is when there is only one element in the given half.
- Print the answer.

Below is the Implementation of the above approach:

C++

```
// C++ program to Count
// Inversions in an array
// using Merge Sort
#include <bits/stdc++.h>
using namespace std;

int _mergeSort(int arr[], int temp[], int left, int right);
int merge(int arr[], int temp[], int left, int mid,
          int right);

// This function sorts the
// input array and returns the
// number of inversions in the array
int mergeSort(int arr[], int array_size)
{
    int temp[array_size];
    return _mergeSort(arr, temp, 0, array_size - 1);
```

```
}

// An auxiliary recursive function
// that sorts the input array and
// returns the number of inversions in the array.
int _mergeSort(int arr[], int temp[], int left, int right)
{
    int mid, inv_count = 0;
    if (right > left) {
        // Divide the array into two parts and
        // call _mergeSortAndCountInv()
        // for each of the parts
        mid = (right + left) / 2;

        // Inversion count will be sum of
        // inversions in left-part, right-part
        // and number of inversions in merging
        inv_count += _mergeSort(arr, temp, left, mid);
        inv_count += _mergeSort(arr, temp, mid + 1, right);

        // Merge the two parts
        inv_count += merge(arr, temp, left, mid + 1, right);
    }
    return inv_count;
}

// This function merges two sorted arrays
// and returns inversion count in the arrays.
int merge(int arr[], int temp[], int left, int mid,
```

```
        int right)
{
    int i, j, k;
    int inv_count = 0;

    i = left;
    j = mid;
    if (arr[i] <= arr[j]) {
        temp[k++] = arr[i++];
    }
    else {
        temp[k++] = arr[j++];

        // this is tricky -- see above
        // explanation/diagram for merge()
        inv_count = inv_count + (mid - i);
    }
}

// Copy the remaining elements of left subarray
// (if there are any) to temp
while (i <= mid - 1)
    temp[k++] = arr[i++];

// Copy the remaining elements of right subarray
// (if there are any) to temp
while (j <= right)
```

```
        temp[k++] = arr[j++];

    // Copy back the merged elements to original array
    for (i = left; i <= right; i++)
        arr[i] = temp[i];

    return inv_count;
}

// Driver code
int main()
{
    int arr[] = { 1, 20, 6, 4, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int ans = mergeSort(arr, n);
    cout << " Number of inversions are " << ans;
    return 0;
}

// This is code is contributed by rathbhupendra
```

Output

```
Number of inversions are 5
```

Time Complexity: O(n * log n), The algorithm used is divide and conquer i.e. merge sort whose complexity is O(n log n).
Auxiliary Space: O(n), Temporary array.

Note: The above code modifies (or sorts) the input array. If we want to count only inversions, we need to create a copy of the original array and call mergeSort() on the copy to preserve the original array's order.