C++ Please

Inversion Count shows how far (or close) the array is from being sorted. If the array is already sorted then the inversion count is 0. If the array is sorted in reverse order, that inversion count is maximum.

Meaning: two elements a[i] and a[j] form an inversion IF a[i] > a[j] and i < j

For example:

Sequence 2, 4, 1, 3 ,5 has three inversions (2,1) (4,1) (4,3). What would the complexity of brute force approach be?

Code an O(nlog(n)) algorithm to count the number of inversions. Hint - piggy back on the merging step of the mergesort algorithm.

--------------------------------------------------------------------------------------------------------------------------

Complete the code with the functions indicated to count inversions:

// Based off of mergesort

#include <vector>

#include <algorithm> // For copy

using namespace std;

int mergeInv(vector<int>& nums, vector<int>& left, vector<int>& right) {

// You will need this helper function that calculates the inversion while merging

// Your code here

}

int countInv(vector<int>&nums) {

// Your code here

}

--------------------------------------------------------------------------------------------------------------------------

Use this code as test code to test the functions:

/\* Count the number of inversions in O(n log n) time \*/

#include <iostream>

#include <vector>

using namespace std;

int countInv(vector<int>& numvec);

int main()

{

int n;

vector<int> numvec{4, 5, 6, 1, 2, 3};

n = countInv(numvec);

cout << "Number of inversions " << n << endl; // Should be 9

numvec = {1, 2, 3, 4, 5, 6};

n = countInv(numvec);

cout << "Number of inversions " << n << endl; // Should be 0

numvec = {6, 5, 4, 3, 2, 1};

n = countInv(numvec);

cout << "Number of inversions " << n << endl; // Should be 15

numvec = {0, 0, 0, 0, 0, 0};

n = countInv(numvec);

cout << "Number of inversions " << n << endl;; // Should be 0

}

--------------------------------------------------------------------------------------------------------------------------

mergesort algorithm.

#include <iostream>

#include <vector>

#include <algorithm> // For copy

#include <random> // For random number generators

#include <chrono> // For timinig measurements

using namespace std;

void print(vector<int>& vec) { //\*\*\* helper function which will print the every element in the vector\*\*\*

for (auto ele : vec)

cout << ele << " ";

cout << endl;

}

void merge(vector<int>& nums, vector<int>& left, vector<int>& right) {

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

while ((j < left.size()) && (k < right.size())) {

if (left[j] <= right[k]) {

nums[i] = left[j];

i++;

j++;

} else {

nums[i] = right[k];

i++;

k++;

}

}

while(j < left.size()) { // \*\*\*Remaining elements of left\*\*\*

nums[i] = left[j];

i++;

j++;

}

while(k < right.size()) {

nums[i] = right[k];

i++;

k++;

}

}

void mergesort(vector<int>&nums) { //RETURN from here

// Base case - one element, nothing to sort

if (nums.size() > 1) {

int mid = nums.size()/2;

vector<int> left(mid);

vector<int> right(nums.size()-mid);

copy(nums.begin(), nums.begin() + mid, left.begin());

copy(nums.begin() + mid, nums.end(), right.begin());

mergesort(left);

mergesort(right);

merge(nums, left, right);

}

}

Expert Answerinformation icon

Anonymous's Avatar

Anonymous answered thisWas this answer helpful?

Thumbs up inactive0Thumbs down inactive0

59 answers

I have used the code of merge provided by you and made some changes to count the inversion

I have introduced a variable named inversion count and count the no of inversion in every recurrsion call and by adding all that you can get no of inversion.

inv\_count += (left.size() - j);

I have used the above code in mergeInv function,

In merge process, let j is used for indexing left sub-array and k for right sub-array. At any step in merge(), if left[j] is greater than right[k], then there are (left.size() â€“ j) inversions.

#include <iostream>

#include <vector>

#include <algorithm> // For copy

#include <random> // For random number generators

#include <chrono> // For timinig measurements

using namespace std;

void print(vector<int>& vec) { //\*\*\* helper function which will print the every element in the vector\*\*\*

for (auto ele : vec)

cout << ele << " ";

cout << endl;

}

int mergeInv(vector<int>& nums, vector<int>& left, vector<int>& right, int mid) {

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

int inv\_count=0;

while ((j < left.size()) && (k < right.size())) {

if (left[j] <= right[k]) {

nums[i] = left[j];

i++;

j++;

} else {

inv\_count += (left.size() - j);

nums[i] = right[k];

i++;

k++;

}

}

while(j < left.size()) { // \*\*\*Remaining elements of left\*\*\*

nums[i] = left[j];

i++;

j++;

}

while(k < right.size()) {

nums[i] = right[k];

i++;

k++;

}

return inv\_count;

}

int countInv(vector<int>& nums){

int inv\_count=0;

if (nums.size() > 1) {

int mid = nums.size()/2;

vector<int> left(mid);

vector<int> right(nums.size()-mid);

copy(nums.begin(), nums.begin() + mid, left.begin());

copy(nums.begin() + mid, nums.end(), right.begin());

inv\_count+=countInv(left);

// cout<<inv\_count;

inv\_count+=countInv(right);

// cout<<inv\_count;

inv\_count+=mergeInv(nums, left, right, mid);

// cout<<inv\_count;

if(inv\_count==14)

print(nums);

}

return inv\_count;

}

int main()

{

int n;

vector<int> numvec{4, 5, 6, 1, 2, 3};

n = countInv(numvec);

cout << "Number of inversions " << n << endl; // Should be 9

// print(numvec);

numvec = {1, 2, 3, 4, 5, 6};

n = countInv(numvec);

cout << "Number of inversions " << n << endl; // Should be 0

// print(numvec);

numvec = {6, 5, 4, 3, 2, 1};

n = countInv(numvec);

cout << "Number of inversions " << n << endl; // Should be 15

// print(numvec);

numvec = {0, 0, 0, 0, 0, 0};

n = countInv(numvec);

cout << "Number of inversions " << n << endl;; // Should be 0

}

///////////////////////////////////////////////////7OTRA SOLUCION /////////////////////////////////////////////////////////////

2. Let A[1 Â· Â· Â· n] be an array of n distinct numbers (i.e., no two numbers are equal). If i < j and A[i] > A[j], then the pair (A[i], A[j]) is called an inversion of A.

(a) List all inversions of the array {4, 2, 9, 1, 7}. (5 points)

(b) What array with elements from the set {1, 2, . . . , n} has the most inversions? How many inversions does it have? (5 points)

(c) Give a divide-and-conquer algorithm that computes the number of inversions in array A in O(n log n) time. (Hint: Modify merge sort.) (20 points)

Expert Answerinformation icon

Vivek Kashyap's Avatar

Vivek Kashyap answered thisWas this answer helpful?

Thumbs up inactive0Thumbs down inactive1

935 answers

4,2,9,1,7

0 1 2 3 4

0 < 1 and 4 > 2 Yes so --> 1

0 < 2 and 4 > 9 No so --> 0

0 < 3 and 4 > 1 Yes so --> 1

0 < 4 and 4 > 7 No so --> 0

1 < 2 and 2 > 9 No so --> 0

1 < 3 and 2 > 1 Yes so --> 1

1 < 4 and 2 > 7 Yes so --> 0

2 < 3 and 9 > 1 Yes so --> 1

2 < 4 and 9 > 7 Yes so --> 1

3 < 4 and 1 > 2 No so --> 0

total = 5

b)

1,2,3.....n

for n=1:

if you see 1 is smaller than all the other items.

for n=2:

if you see 2 is smaller than all the other items.

You see all the other element same as the above.

there is no inversion.

so number of inversion = 0

c)

#include<iostream>

#include<bits/stdc++.h>

using namespace std;

//int inversion=0;

int Merge(int A[],int B[],int start,int mid,int end)

{

int inversion=0;

int k=start;

int oldstart=start;

int mid1=mid;

mid++;

while(start<=mid1 &&mid<=end ){

if(A[start]<=A[mid]){

B[k]=A[start];

k++;

start++;

}

else{

B[k]=A[mid];

inversion+=mid-k;

k++;

mid++;

}

}

while(start<=mid1){

B[k]=A[start];

k++;

start++;

}

while(mid<=end){

B[k]=A[mid];

k++;

mid++;

}

start=oldstart;

while(start<=end){

A[start]=B[start];

start++;

}

return inversion;

}

int Mergesort(int A[],int B[],int start,int end)

{ int inversion=0;

if(start<end)

{

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

inversion+=Mergesort(A,B,start,mid);

inversion+=Mergesort(A,B,mid+1,end);

inversion+=Merge(A,B,start,mid,end);

}

return inversion;

}

int main(){

int A[5]={4, 2, 9, 1, 7};

int B[5]={0};

int inversion=Mergesort(A,B,0,3);

for(int i=0;i<=3;i++){

cout<<A[i]<<" ";

}

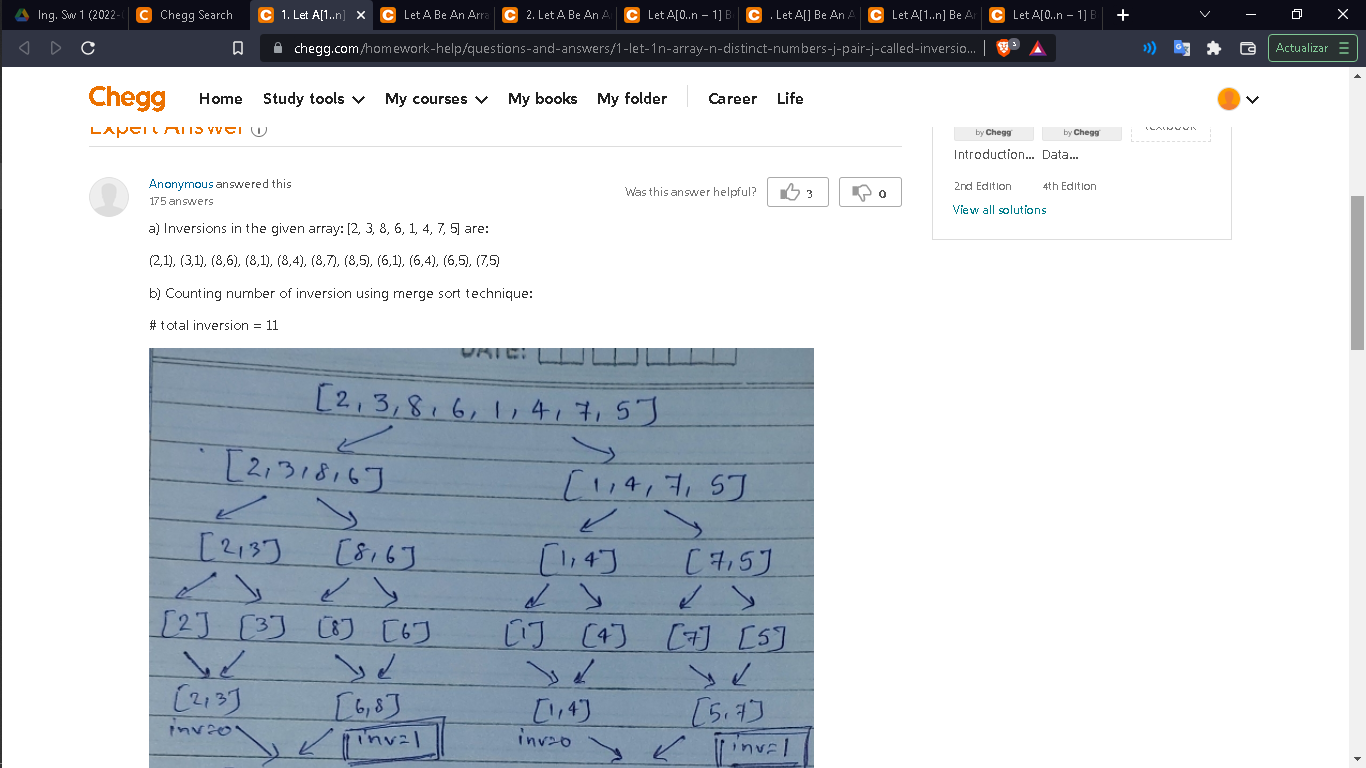
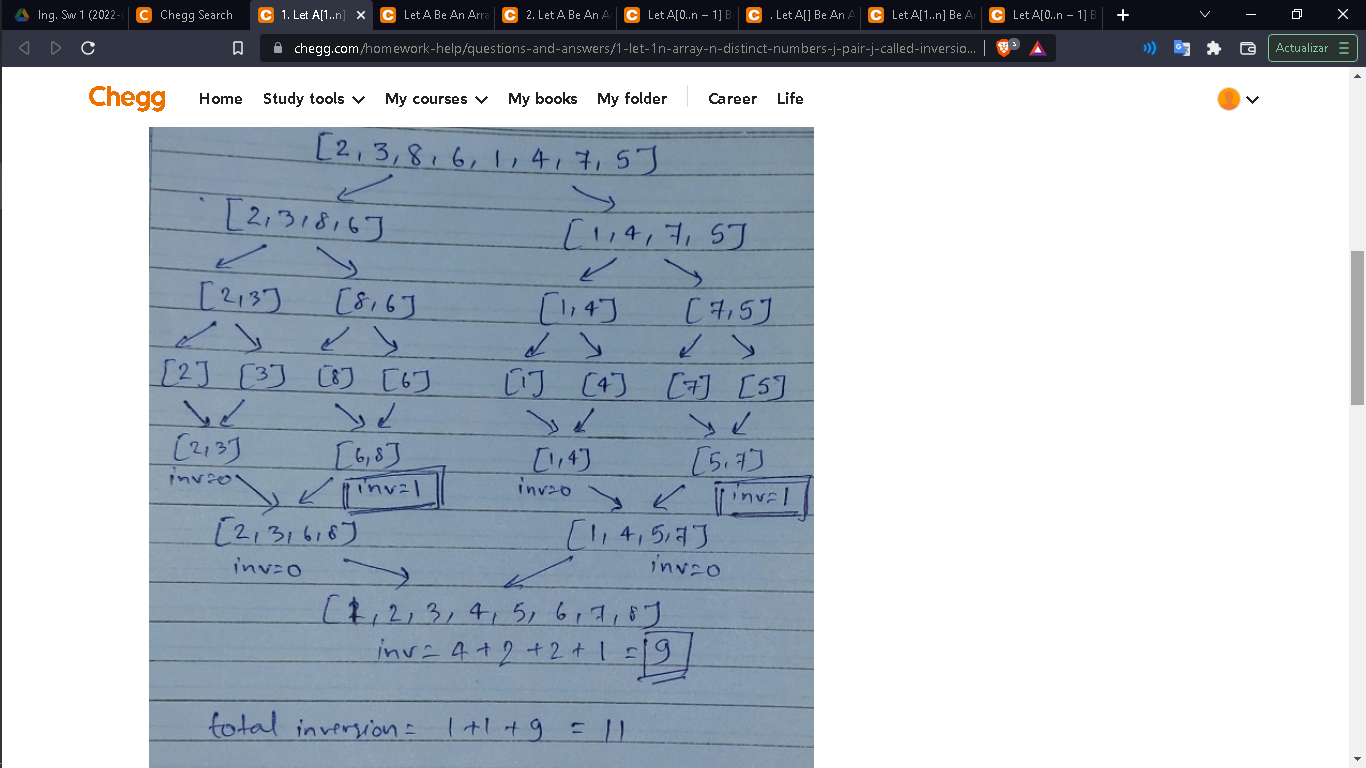
cout<<"\nnumberOfInversion: "<<inversion;

return 1;

}

In the merge sort we handle the case so merge sort time complexity is O(nlogn).

so this code time complexity is (nlogn).

c) A reverse sorted array will have max number of inversion.

Proof:

Let given a reverse sorted array: [n, n-1, ..., 2, 1]

#inversion for 1 = n-1

#inversion for 2 = n-2

...

#inversion for n-1 = n - (n-1) = 1

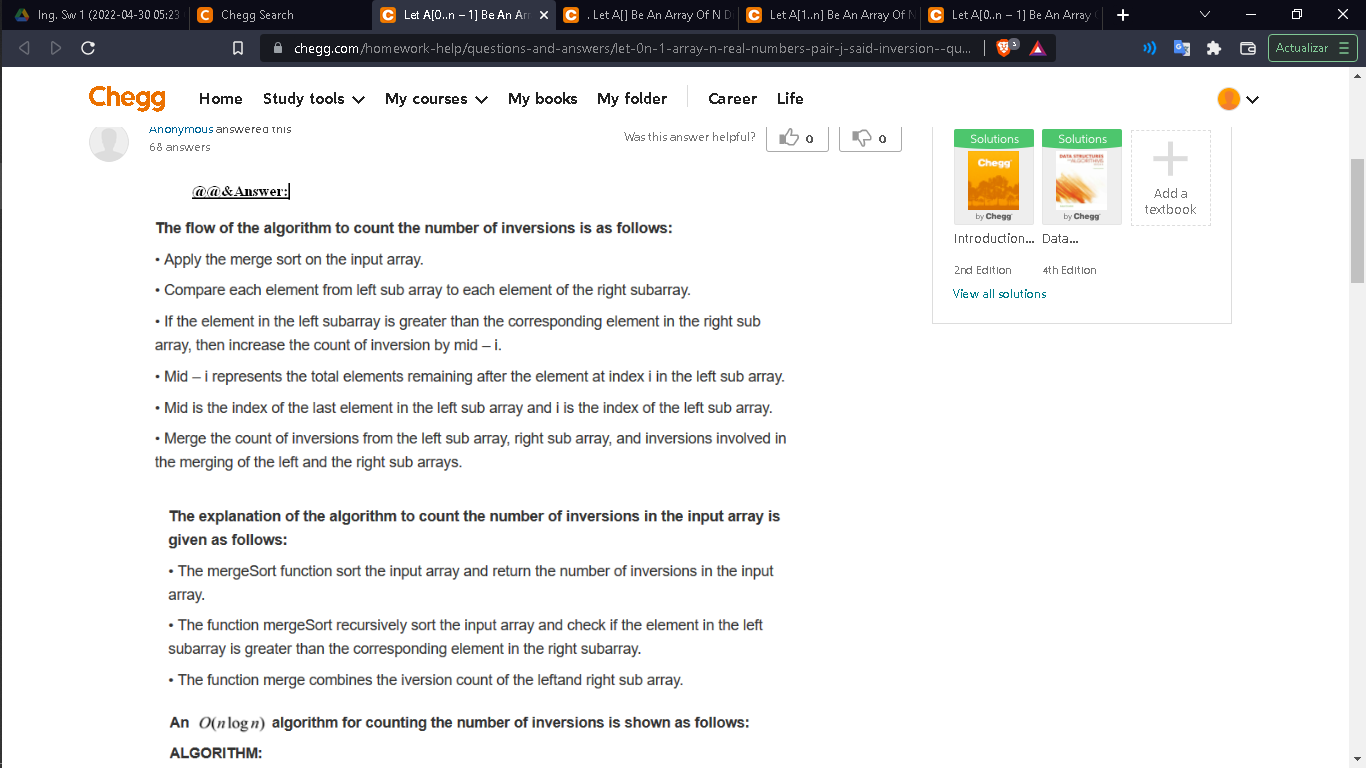
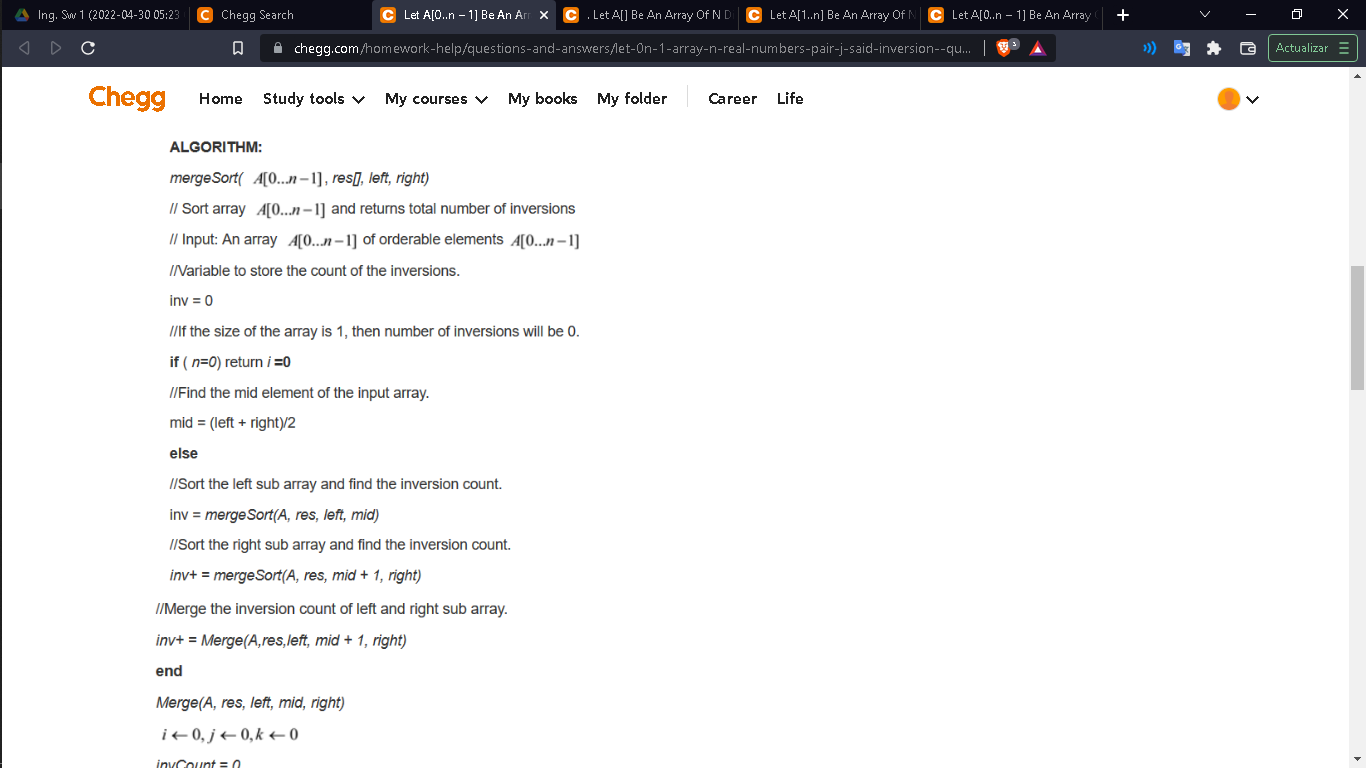
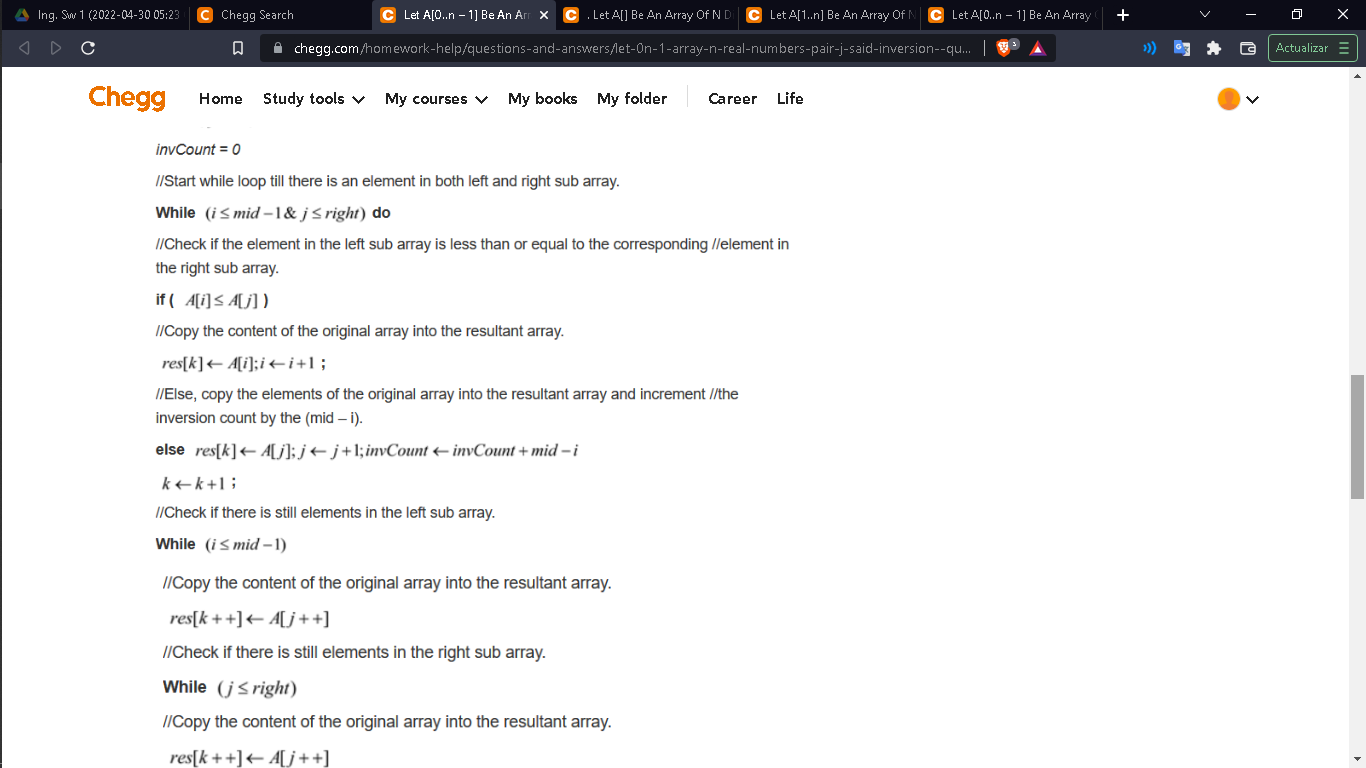
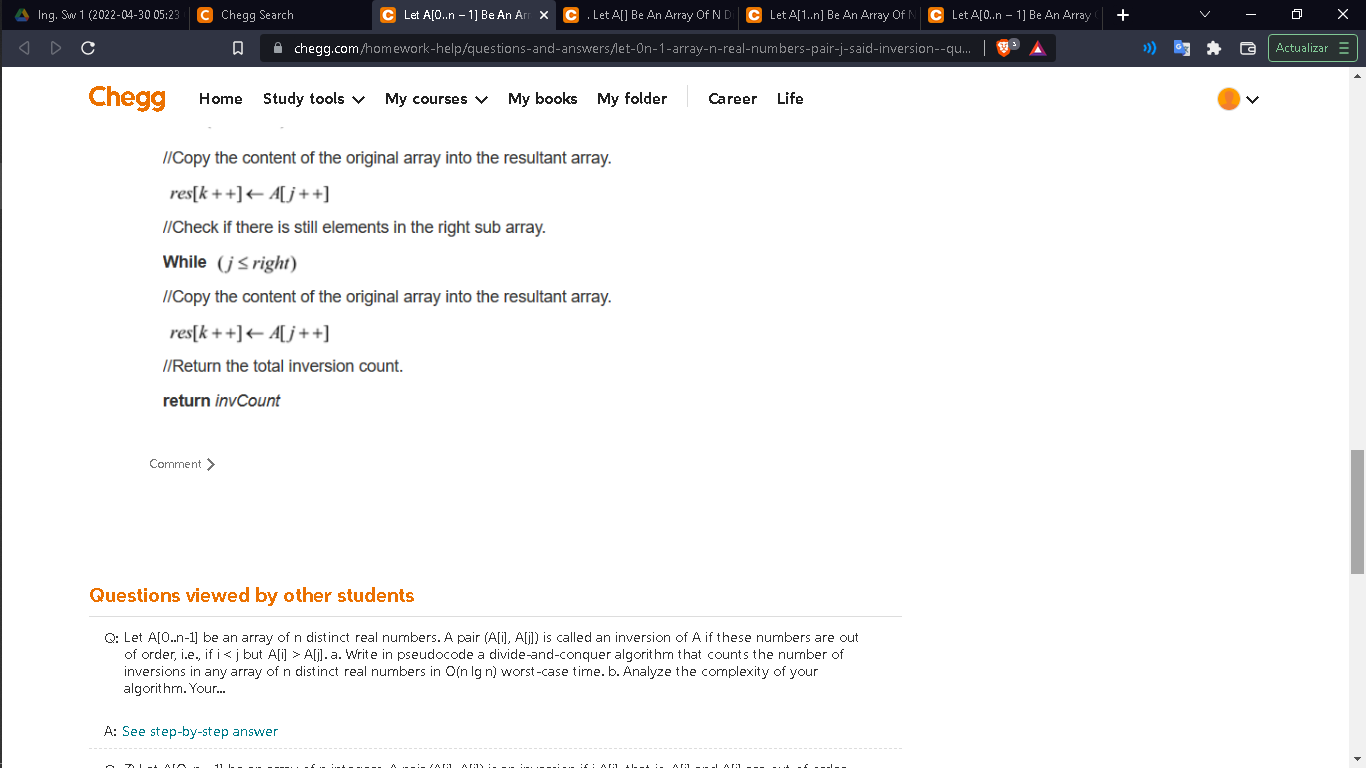
#inversion for n = n - n = 0

summing all of the above, we get total inversion count

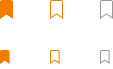
= (n-1) + (n-2) + ... + 1 + 0

= (n-1)(n-1+1)/2 ( sum of natural numbers formula)

= n(n-1)/2

*Question:*. Let A[] be an array of n distinct numbers. If index i A[j], then the pair (i, j) is called an i...



Flag

. Let A[] be an array of n distinct numbers. If index i A[j], then the pair (i, j) is called an inversion of A. In other words, two elements of the array are considered to form an inversion if they are

Best Answer

* Harsh Tibrewal's Avatar

Harsh Tibrewal answered this

Was this answer helpful?

2

0

41 answers

a . The five inversions

<2,1>, <3,1>, <8,6>, <8,1> and <6,1>.

b. Array with most inversions

It is the reversed array, that is <n,n?1,

[View comments (2)](https://www.chegg.com/homework-help/questions-and-answers/-let-array-n-distinct-numbers-index-j-pair-j-called-inversion--words-two-elements-array-co-q6466529)

**More Answers**

* Akshay Verma's Avatar

Akshay Verma answered this

Was this answer helpful?

3

0

First answer!

Array a[] = {2,3,8,6,1}

(a). Inversions :- with 2 as first element of pair = 1, with 3 = 1, with 8 = 2, with 6 = 1, with 1 = 0;

so total = 1+1+2+1 = 5

(b). the array with most inversions is array sorted in reverse order i.e {n,n-1,n-2,......1}

since in this case ith element has i-1 inversions, so there are 1+2+3+.....(n-1), total inversions, which is equal to number of ordered pair of array, so there can't be more inversions.

(c). They both are O(n^2). in worst case.

(d). Modified merge sort, counts number of inversions in O(nlogn) steps.

3. This can be solved in O(max(S)) or O(nlogn), complexities.

for order max(S), maintain a hash of size max(S),

with ith entry 0 if its not present, and equal to its count if its present.

Now for all a in S, check if x-a is present, i.e its hash value is non zero.

for order O(nlog n):-

Sort S, take two pointers one pointing to smallest (first)element ptr1, second pointing to last element ptr2.

loop

take sum = \*ptr1 + \*ptr2, if sum=x: then we are good

if sum>x

move ptr2 left,

if sum<x

move ptr1 right

if ptr1>=ptr2 report does not exist, break

repeat loop

14 a. Prove it inductively,

prove that an^i + bn^(i-1) belongs to O(n^i)

induct from i=1 to n,

i.e O(n^2) = O(n) + O(n^2) + O(1)

O(n^3) = O(n^3) + O(n^2) + O(n) + O(1) and so on till k

Remaining questions are available here http://www.csee.umbc.edu/~kalpakis/Courses/441-sp03/hws/hw2-sol.pdf , they are based on masters theorem, look up masters theorem. Its difficult to draw a recursion tree here, cheers!

2. Inversions. Let A[1..n] be an array of n distinct numbers. If i < j and A[i] > A[j], then the pair (i, j) is called an inversion of A.

(a) List the five inversions of the array < 2, 3, 8, 6, 1 >.

(b) What array with elements from the set {1, 2, . . . , n} has the most number of inversions? How many does it have?

(c) What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

(d) Give an algorithm that determines the number of inversions in any permutation of n elements in O(n log n) worst-case running time. (Hint: Modify Merge-sort.)

**Expert Answer**

* Violet Taylor's Avatar

Violet Taylor Chegg expert answered this

Was this answer helpful?

3

0

1,010 answers

**Solution:**

**a)**

The definition of an inversion is as follows: Let A[1..n] be an array of n distinct numbers. If i < j and A[i] > A[j], then the pair (i, j) is called an inversion of A.

The five inversions of the array < 2, 3, 8, 6, 1 > are as follows:

(2, 1), (3,1), (8, 6), (8, 1) and (6, 1)

**b)**

The n-element array with most inversions is <n, n-1,..2,1> because it will be the reverse sorted array. The number of inversions are as follows: (n-1) + (n-2) + (n-3) +…+2+1 = n(n-1)/2.

**c)**

The code for the insertion sort is as follows:

for j= 2 to length(A).

do key = A[j]

       i = j-1

       while i>0 and A[i]>A[i+1]

          do A[i+1] = A[i]

               i = i-1

        A[i+1] = key

Here, the invariant is maintained such that the first k elements of the array A are already sorted and then element A[k+1] is inserted into the correct place. Thus, the running time of the insertion sort is a constant time the number of inversions. Let the inversion be denoted by X(i) such that A[j]>A[i], then the number of inversions in the array A will be summation of all inversions. If the array is sorted, there will be no inversions, so the running time of insertion sort will be proportional to the number of inversions in the array A.

d)

An algorithm that determines the number of inversions in any permutation on n elements in O(nlogn) worst case time is as follows:

//initialise inversions to 0.

int inversions = 0

//The mergeSort algorithm.

       int[] mergeSort (int[] a)

       //if the length of the array is less than 1, return the array.

       if a.length <= 1

              return a

              //take a variable mid, store the half of the array in it.

              int mid = a.length / 2

              int[] left = new int[mid]

              //Now, store the elements from first position till mid, the left

              //array.

              for (int i=0;i<mid;i++)

                     left[i] = a[i]

         //Store the rest of the elements in the right array.

                     int[] right <-new int[a.length - mid]

                     for (int i = mid; i < a.length; i++)

                           right[i - mid] = a[i]

                           left = mergeSort(left)

                           right = mergeSort(right)

                           return merge(left, right)

   //The merge algorithm.

              int[] merge(int[] l, int[] r)

              int[] a = new int[l.length + r.length]

              int lIndex = 0

              int rIndex = 0

              int i = 0

                     while lIndex < l.length && rIndex < r.length

                           do

                           if (l[lIndex] <= r[rIndex]) then

                                  a[i++] = l[lIndex++]

                           else

                                  inversions <-inversions + l.length - lIndex

                                  a[i++] = r[rIndex++]

              while lIndex < l.length

                           do

                              a[i++] = l[lIndex++]

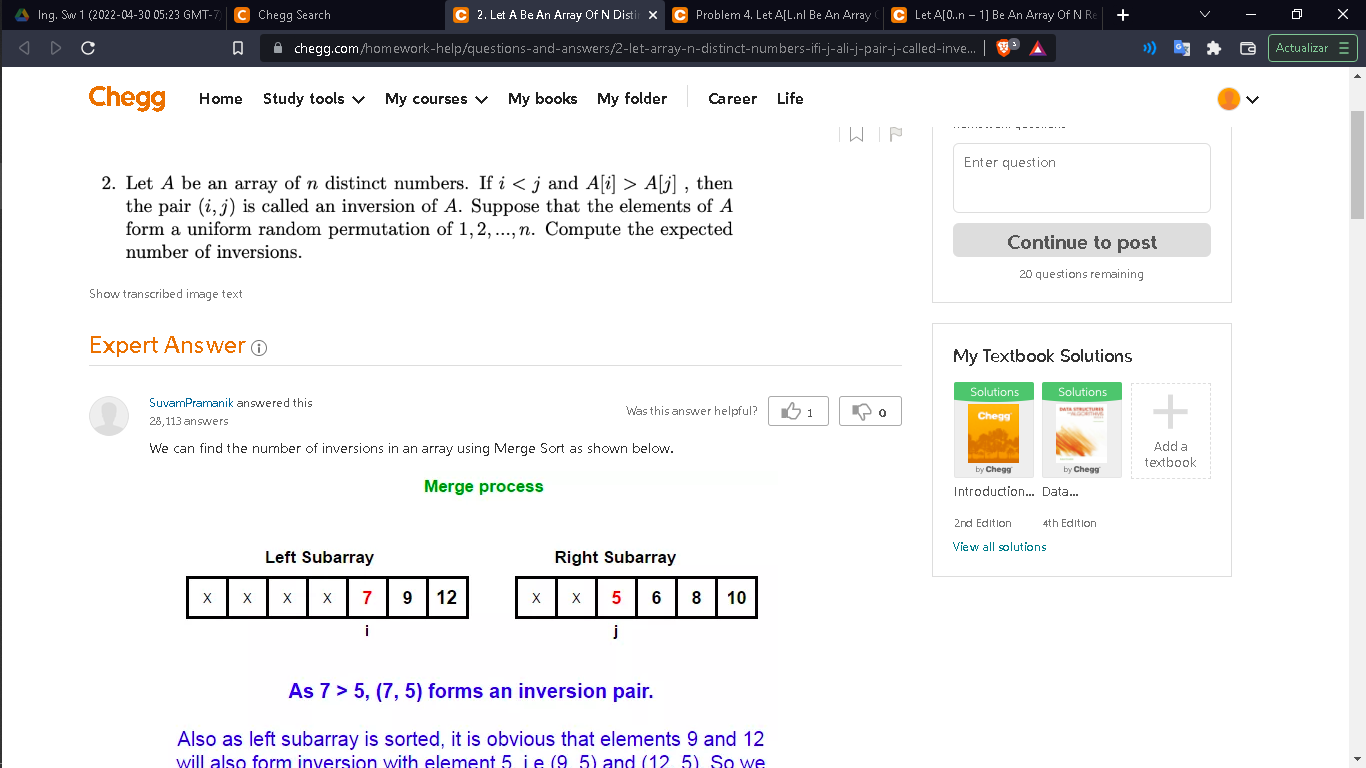
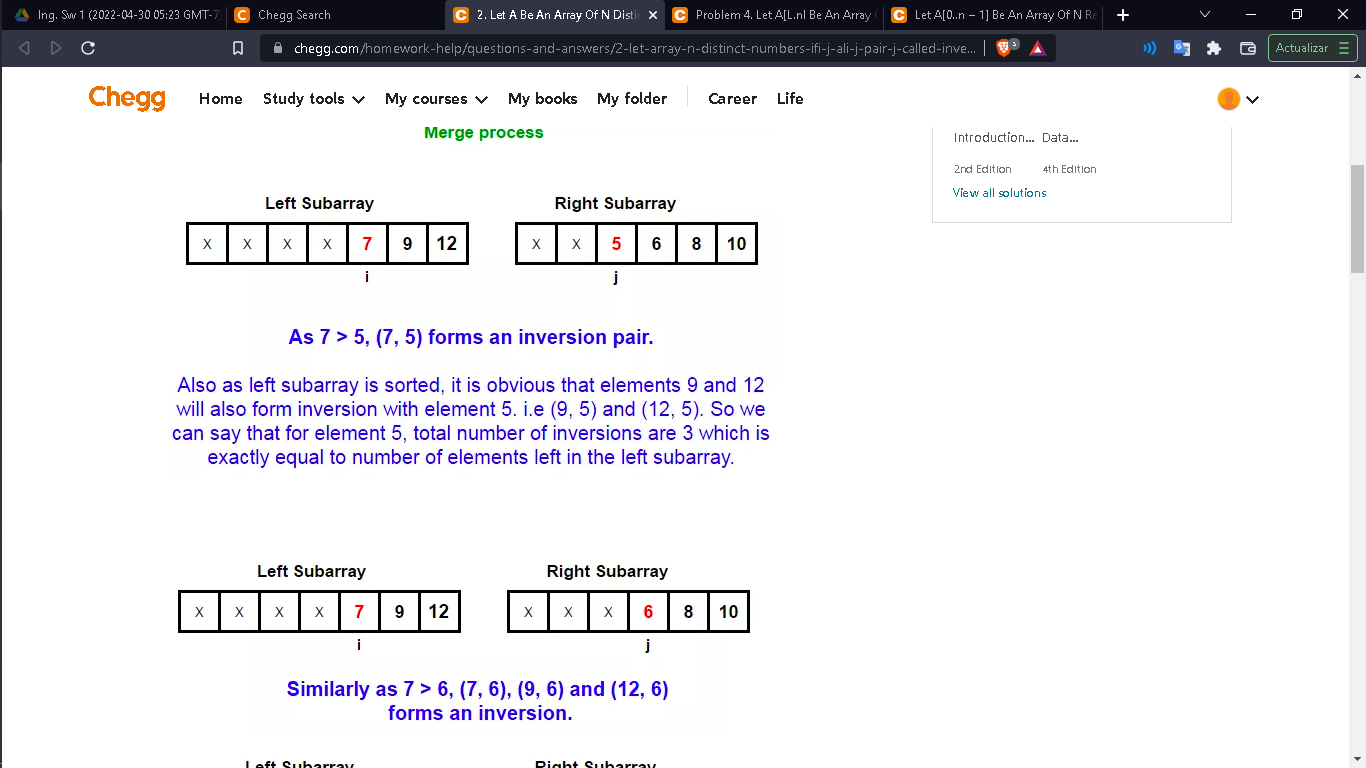
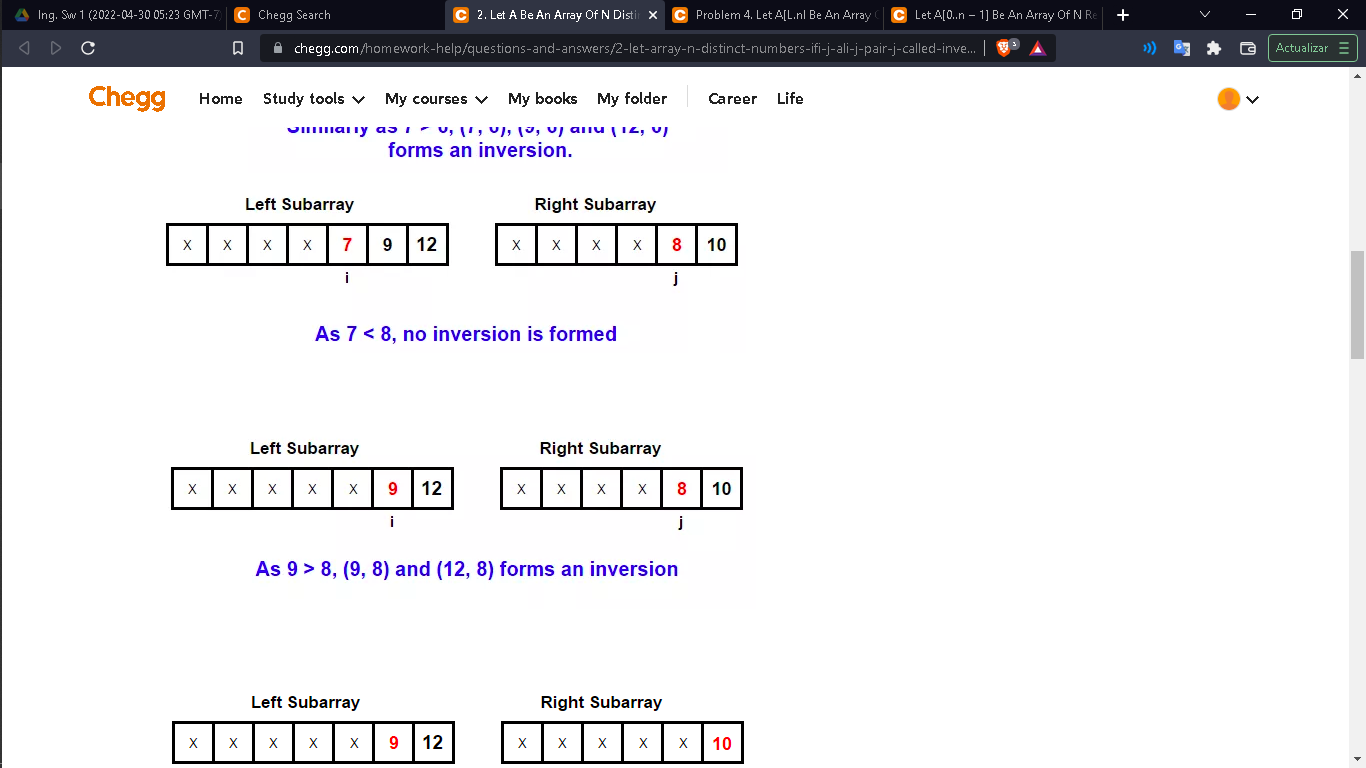
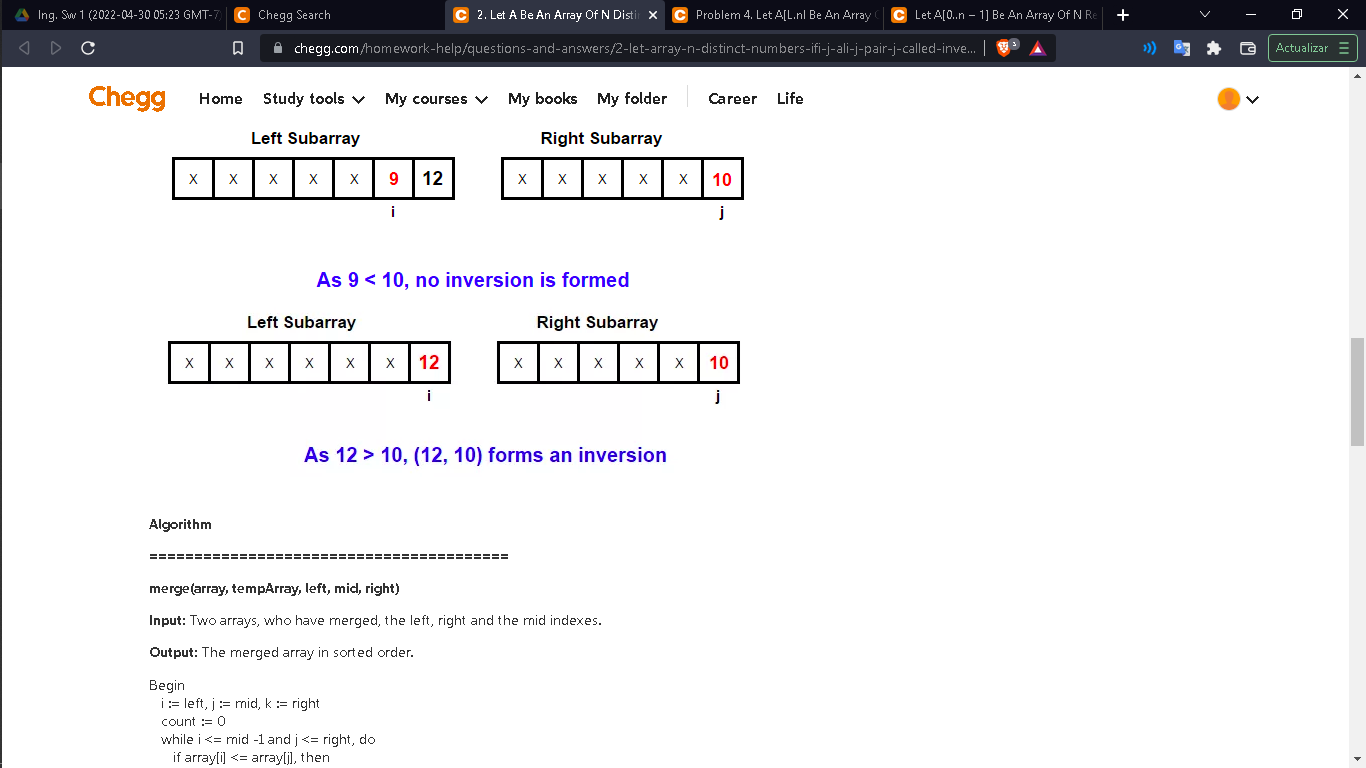
                 while rIndex < r.length

                           do

                                  a[i++] = r[rIndex++]

       return a

This algorithm will determine the number of inversions in any permutation of n elements in O (n logn) worst case.

**Algorithm**

**========================================**

**merge(array, tempArray, left, mid, right)**

**Input:** Two arrays, who have merged, the left, right and the mid indexes.

**Output:** The merged array in sorted order.

Begin

i := left, j := mid, k := right

count := 0

while i <= mid -1 and j <= right, do

if array[i] <= array[j], then

tempArray[k] := array[i]

increase i and k by 1

else

tempArray[k] := array[j]

increase j and k by 1

count := count + (mid - i)

done

while left part of the array has some extra element, do

tempArray[k] := array[i]

increase i and k by 1

done

while right part of the array has some extra element, do

tempArray[k] := array[j]

increase j and k by 1

done

return count

End

**mergeSort(array, tempArray, left, right)**

**Input:** Given an array and temporary array, left and right index of the array.

**Output:** Number of inversions after sorting.

Begin

count := 0

if right > left, then

mid := (right + left)/2

count := mergeSort(array, tempArray, left, mid)

count := count + mergeSort(array, tempArray, mid+1, right)

count := count + merge(array, tempArray, left, mid+1, right)

return count

End

//////////////////////////////////////////

**a)List the five inversions of the array [2,3,8,6,1]**

1. (2,1)
2. (3,1)
3. (8,1)
4. (6,1)
5. (8,6)

**b)Give a worst-case O(nlogn) algorithm that determines the number of inversions in A. Explain your algorithm, give an example and derive its running time.**

Our logic is very similar to merge sort, where we divide whole array into sub parts and get the result from smallest possible array (here size <=2) and combine them all together to get the actual result for whole array.

Below is the java code to find out the number of inversions of an array.

|  |
| --- |
| **import** java.util.Arrays;  **public** **class** InversePairCount {  **static** **int** merge(**int**[] arr, **int**[] left, **int**[] right) {              //Merge sort: left and right arrays are already got sorted in previous recursions  **int** i = 0, j = 0, count = 0;  **while** (i < left.length || j < right.length) {  **if** (i == left.length) {                    //left array finished, so add all right to main array one by one                    arr[i+j] = right[j];                    j++;                } **else** **if** (j == right.length) {                    //right array finished, so add all left to main array one by one                    arr[i+j] = left[i];                    i++;                } **else** **if** (left[i] <= right[j]) {                    //left arr element is smaller than of right element so add left element to main array and move ahead in left array                    arr[i+j] = left[i];                    i++;                } **else** {                    /\* left array element is greater than right array element, this is what we called as a inversion pair                    i<j and a[i]>a[j] matched, so increase our count to size of left array - position of current left element                    e.g., if first element of left array is greater than first element of right array then we can form inversion pairs for current right element with all left elements                                that means we can form inversion pairs for right[j] with elements of left from i to length of left array (because left, right are already sorted)                    add right element to main array and move ahead in right array                    \*/                    arr[i+j] = right[j];                    count += left.length-i;                    j++;                }            }  **return** count;        }  **static** **int** getinversionsCount(**int**[] a) {  **if** (a.length < 2){  **return** 0; // No possibility to have a pair            }  **int** mid = (a.length + 1) / 2; //Splitting array into two parts using mid  **int** left[] = Arrays.*copyOfRange*(a, 0, mid);  **int** right[] = Arrays.*copyOfRange*(a, mid, a.length);  **return** *getinversionsCount*(left) + *getinversionsCount*(right) + *merge*(a, left, right);        }    **public** **static** **void** main(String args[]){  **int** a[] = {2,3,8,6,1};              System.***out***.println("Number of Inversions Count: "+*getinversionsCount*(a));        }  } |

In Worst/best/any case whole array will split into half in each iteration and merge happens back, during that process it self we are calculating the inversions count. So as similar to merge sort, complexity is o(nlogn).

**Explanation: (This can be better represented in tree way)**

Our Example input here is [2,3,8,6,1]

*Step1: getinversionsCount([2,3,8,6,1])*

*Step2: getinversionsCount([2,3,8]) + getinversionsCount([6,1]) + merge([2,3,8,6,1], [2,3,8], [6,1])*

*Step2(left)- step 3: getinversionsCount([2,3]) + getinversionsCount([8]) + merge([2,3,8], [2,3], [8])*

*Step3(left) getinversionsCount([2]) + getinversionsCount([3]) + merge([2,3], [2], [3])*

1. *0+0+0 – array becomes[2,3]*

*Step3(right) getinversionsCount([8]) + getinversionsCount([]) + merge([8], [8], [])*

1. 0+0+0 – array becomes [8]

0+0+0 – array becomes [2,3,8]

*Step2(right) –Step4: getinversionsCount([6]) + getinversionsCount([1]) + merge([6,1], [6], [1])*

*Step4(left): getinversionsCount([6]) + getinversionsCount([]) + merge([6], [], [6])*

1. *0+0+0 – array becomes [6]*

*Step4(right): getinversionsCount([1]) + getinversionsCount([]) + merge([1], [], [1])*

1. *0+0+0 – array becomes [1]*

     0+0+1 – array becomes [1,6]

     Step2- *merge([2,3,8,6,1], [2,3,8], [1,6]) (here as a result of Step2(right) right array became [1,6] )*

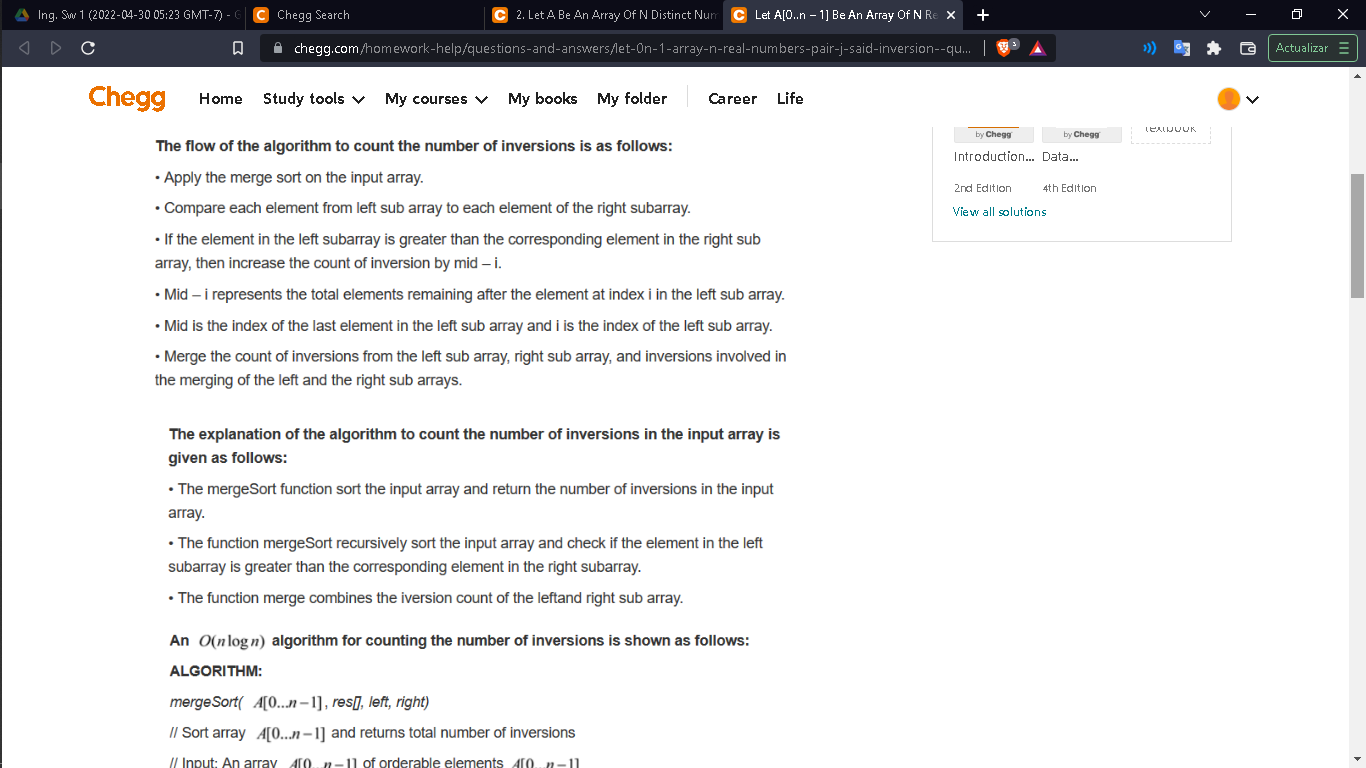
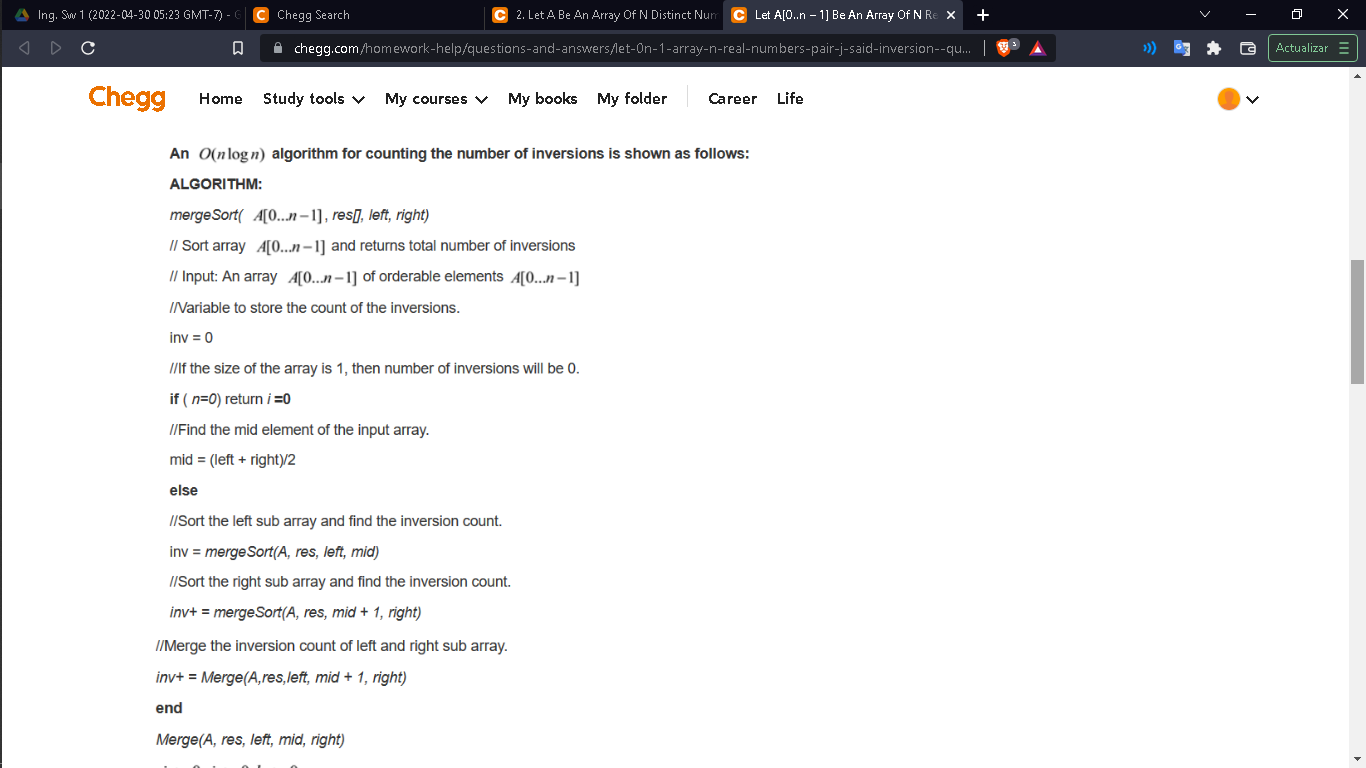
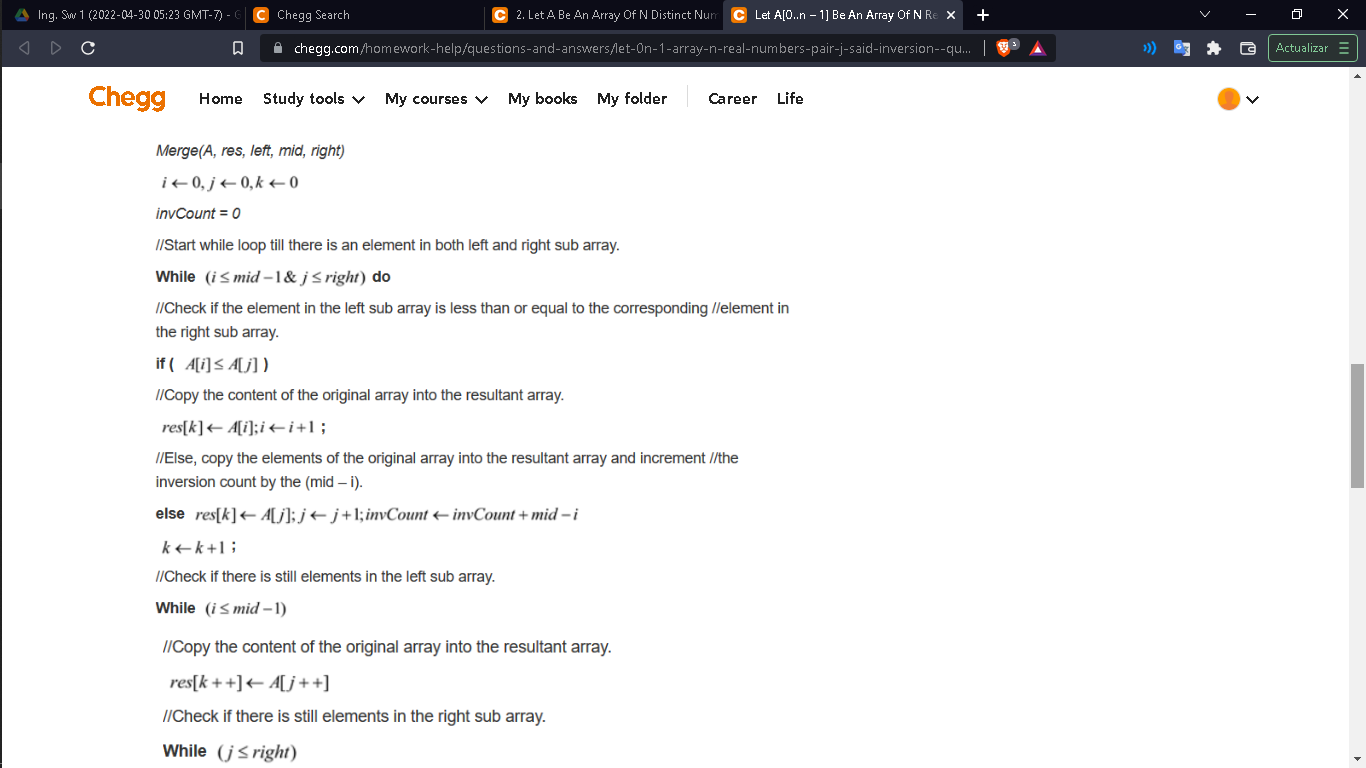
*when i=0, j=0 l[i]is >r[j] (2 is greater than 1) - here count becomes 0+3*

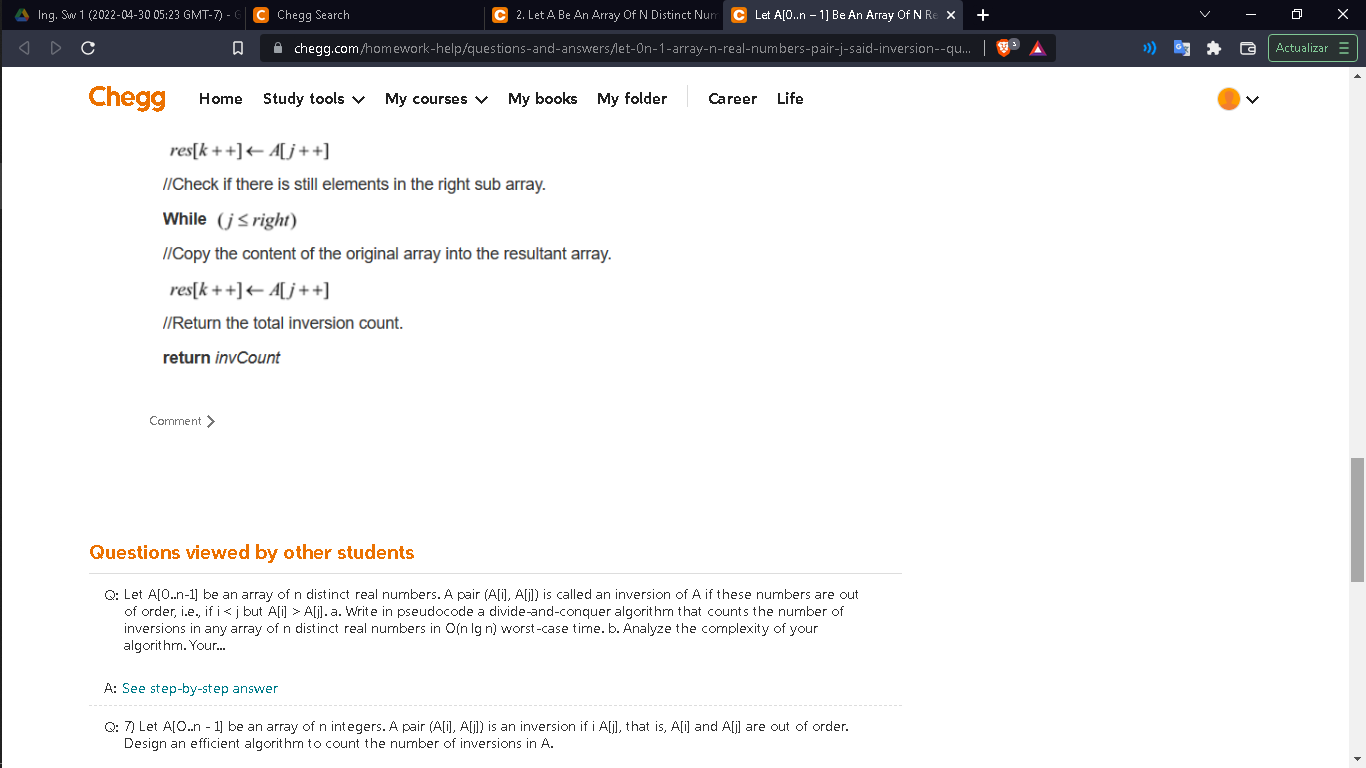
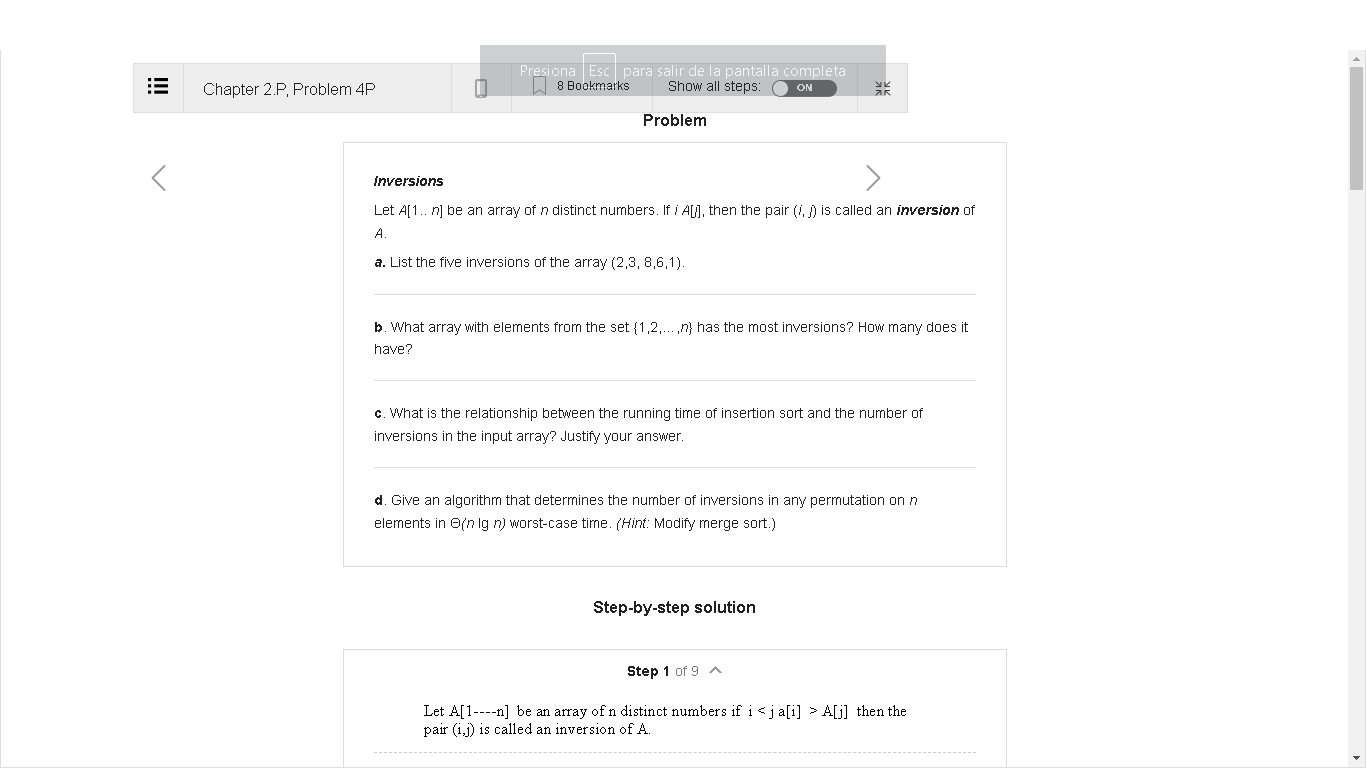
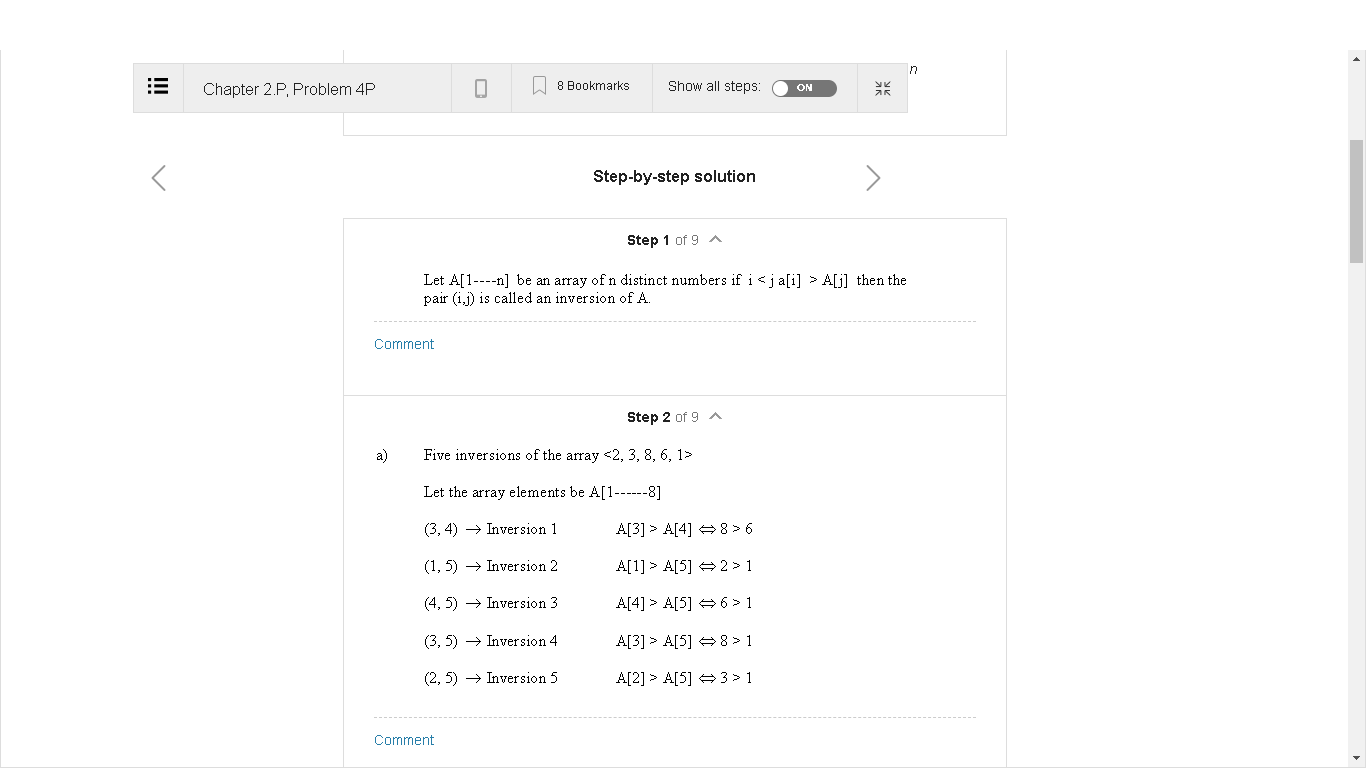
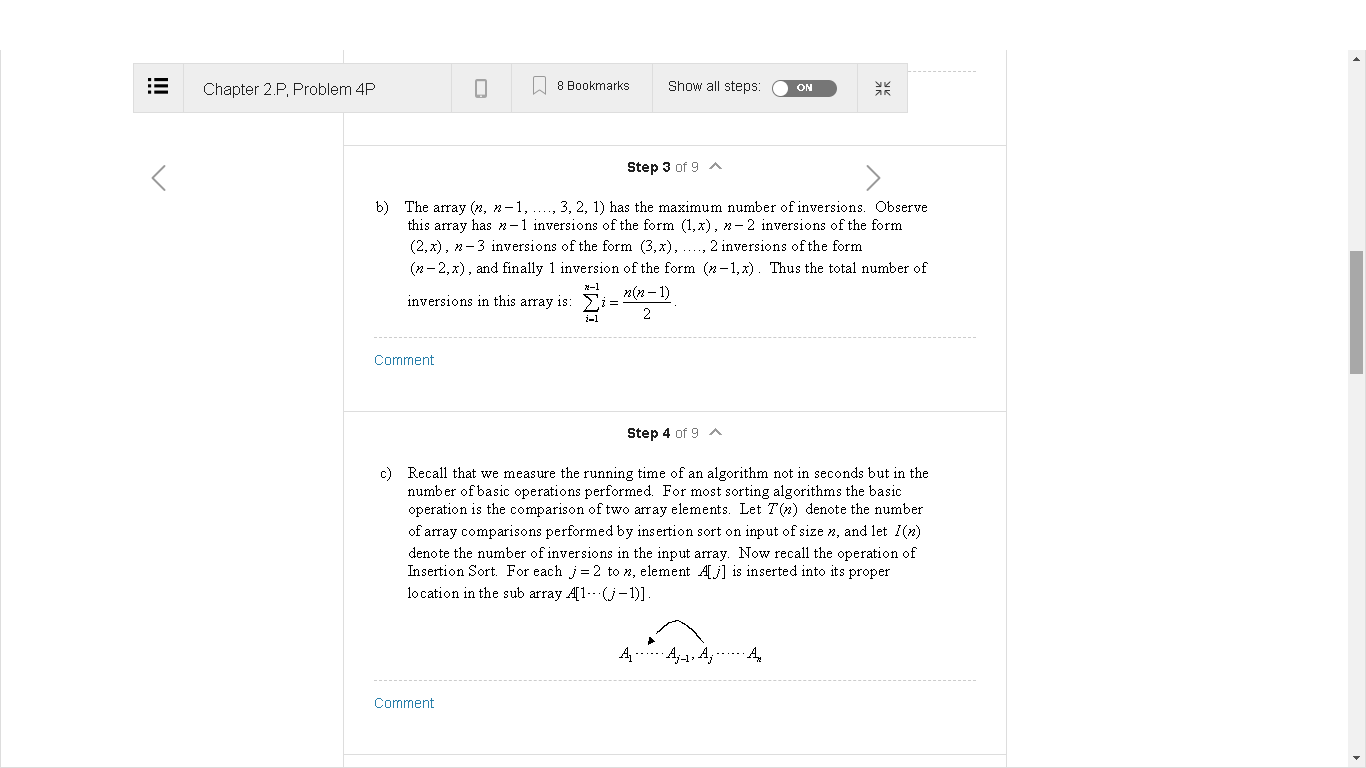
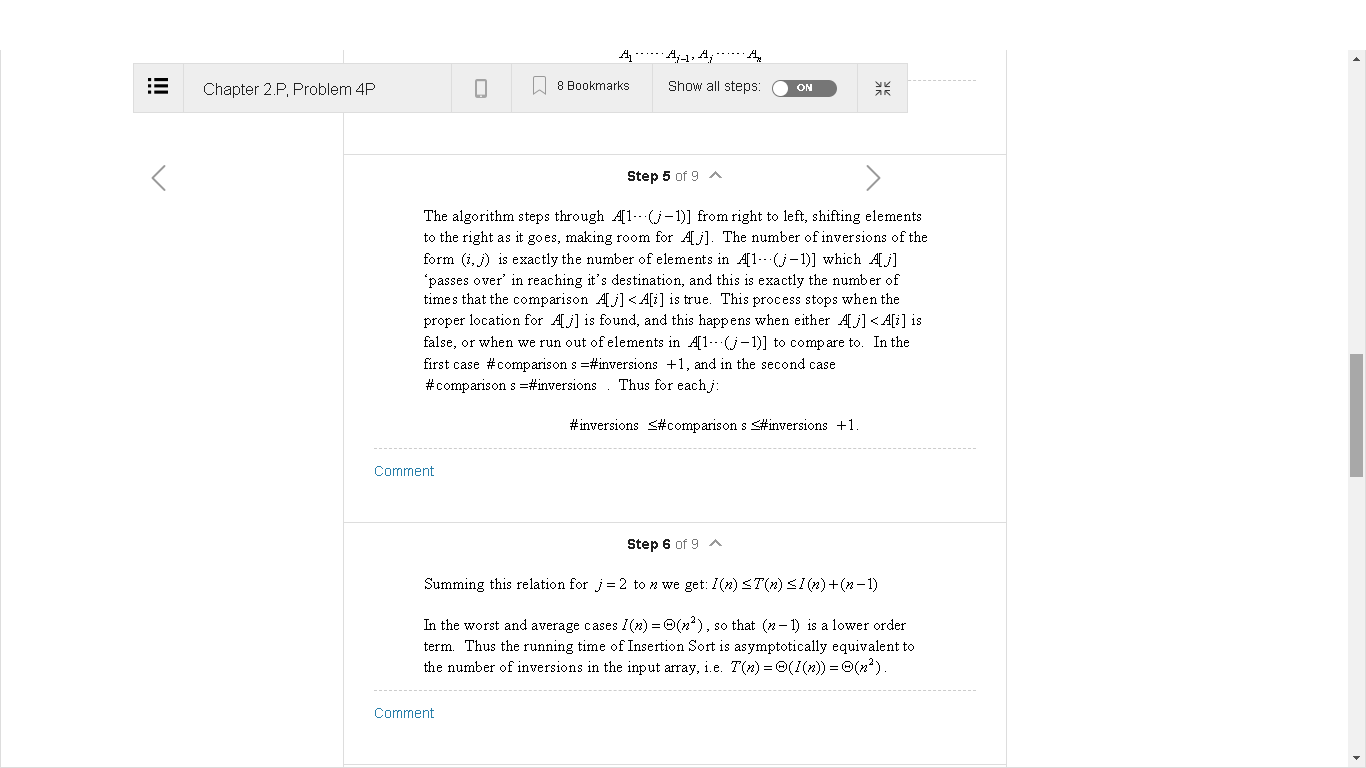
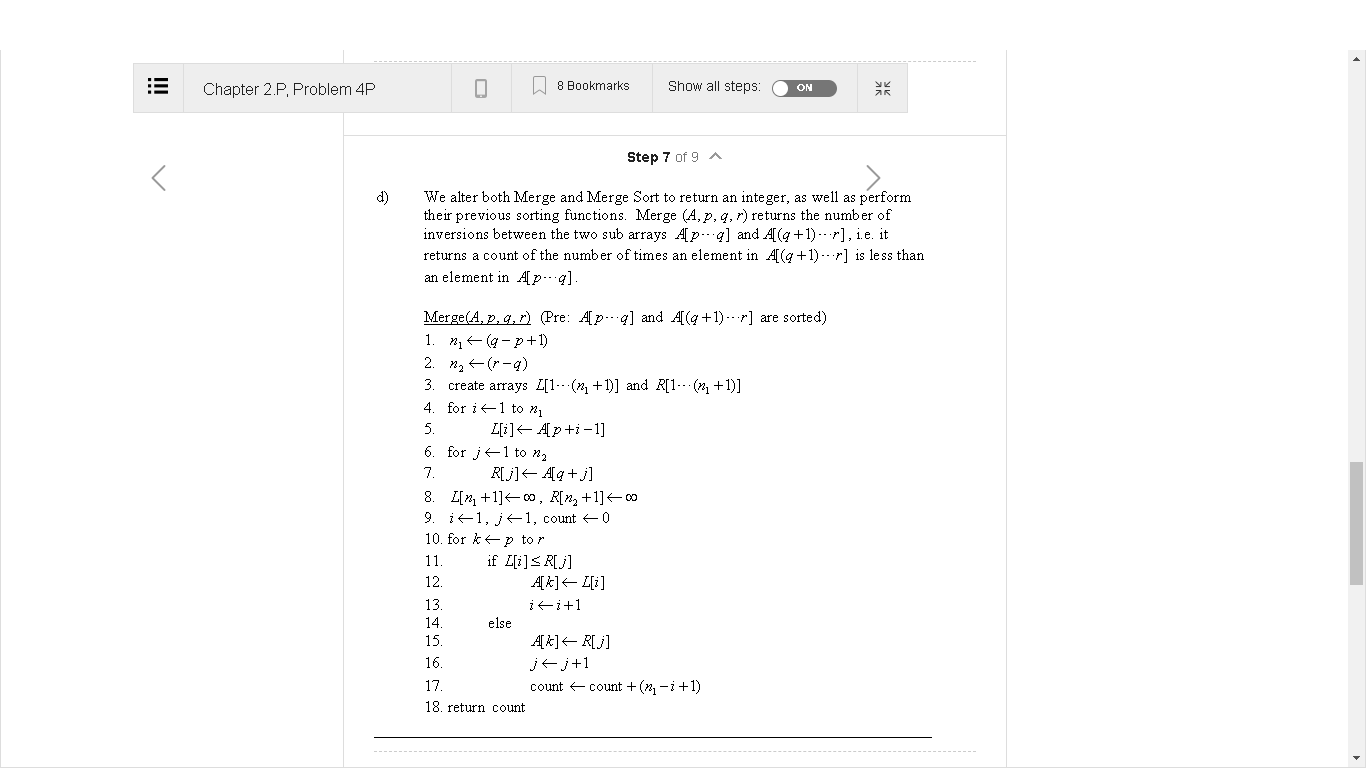
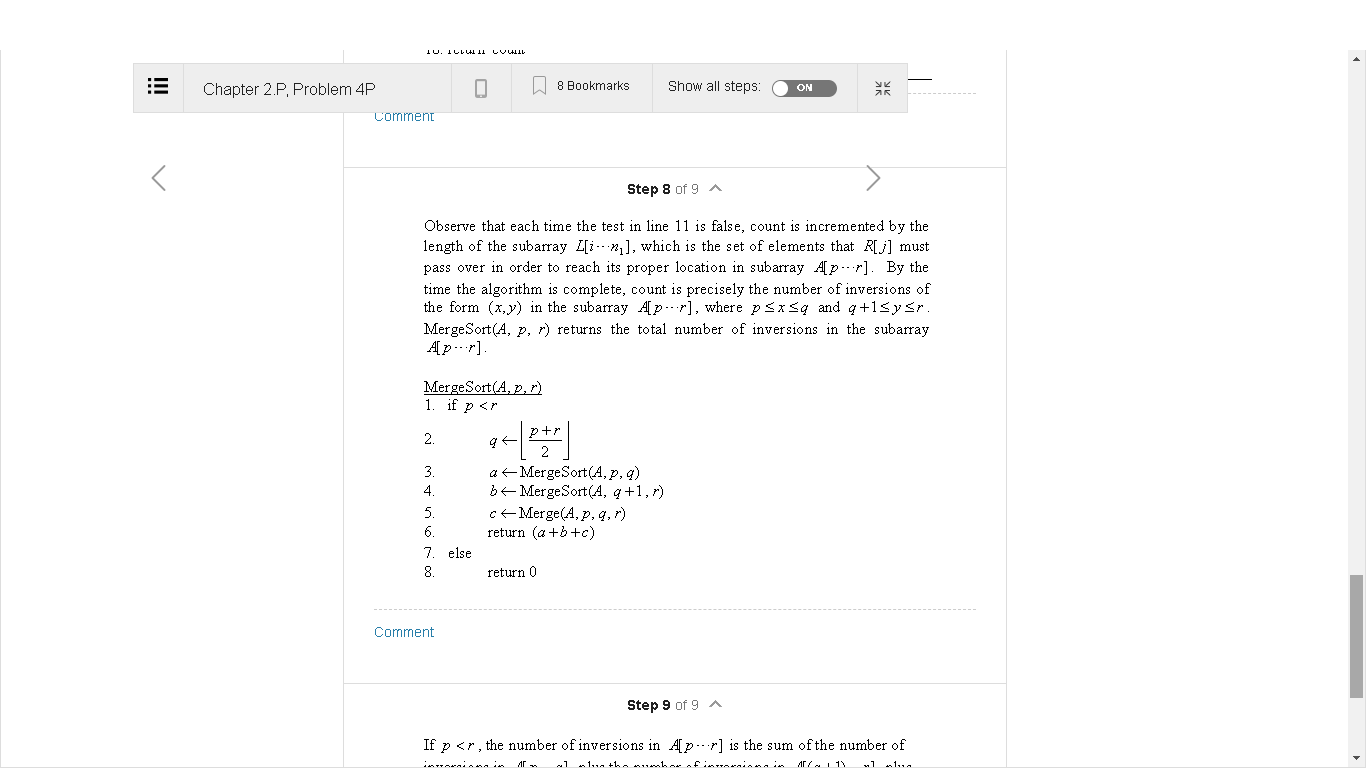
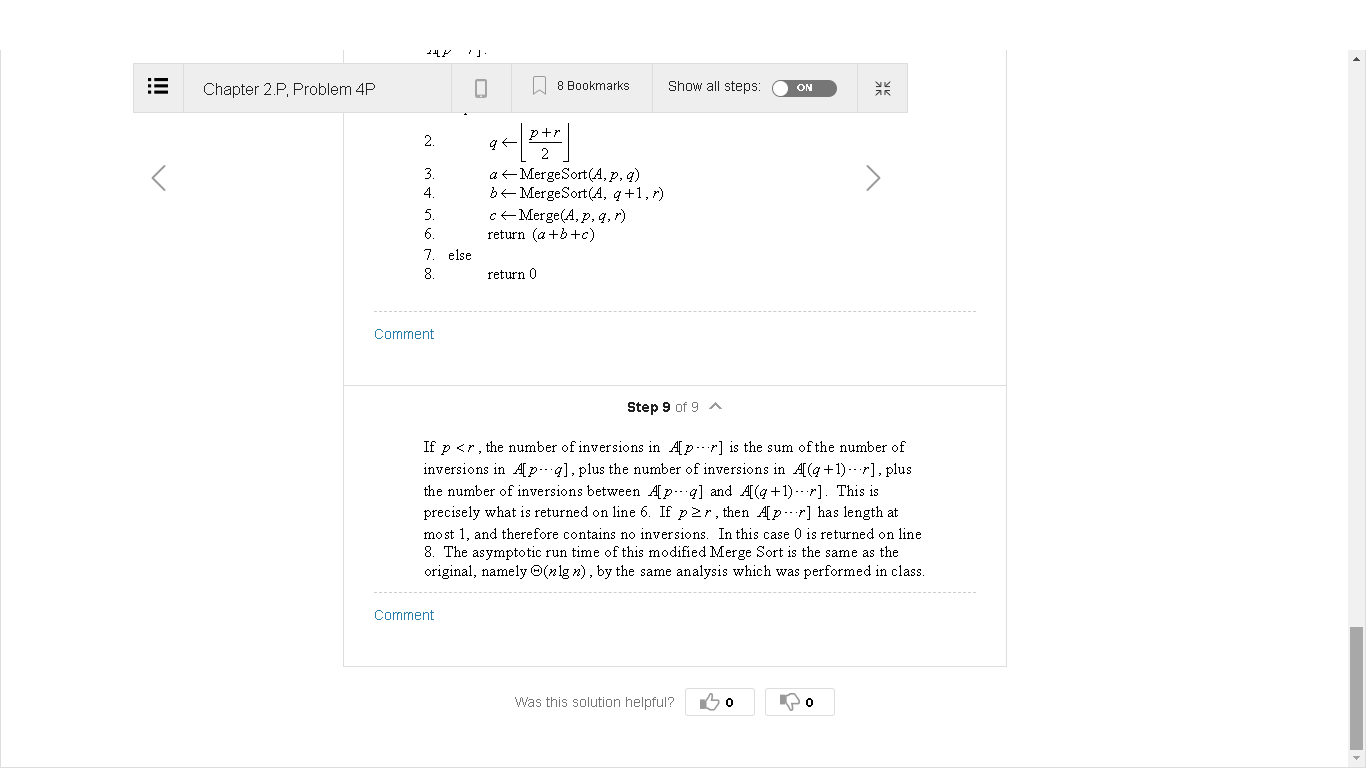
*when i=2, j=1 l[i]is >r[j] (8 is greater than 6) - here count becomes 3+1=4*

*overall for step 2 result is 0+1+4=5*

*Overall result is 5*

Thank you, Hope this answer explained you in detail.

        #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 funt 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; /\* i is index for left subarray\*/

    j = mid; /\* j is index for right subarray\*/

    k = left; /\* k is index for resultant merged subarray\*/

    while ((i <= mid - 1) && (j <= right)) {

        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[] = { 20,-3,17,9,10};

    int n = sizeof(arr) / sizeof(arr[0]);

    int ans = mergeSort(arr, n);

    cout << " Number of inversions are " << ans;

    return 0;

}

output

Number of inversions are 6

**Pseduocode**

int merge(int originalArr[], int temp[], int start, int mid, int end)  
{  
int i = start;  
int j = mid;  
int k = start;  
int inversion\_count = 0;  
while(i <= mid-1 && j<= end)  
{  
if(originalArr[i]< originalArr[j])  
temp[k++] = originalArr[i++]  
else{  
inversion\_count += mid-i;  
temp[k++] = originalArr[j++]  
}  
}  
while(i <= mid-1)  
temp[k++] = originalArr[i++]  
while(j <= end)  
temp[k++] = originalArr[j++]  
for(int i = start; i <= end; i++)  
originalArr[i] = temp[i]  
return inversion\_count;  
}  
int mergeSort(int originalArr[], int temp[], int start, int end)  
{  
int inversion\_count = 0;  
if(start < end) {  
int mid = (start+end)/2;  
inversion\_count += mergeSort(originalArr, temp, start, mid);  
inversion\_count += mergeSort(originalArr, temp, mid+1, end);  
inversion\_count += merge(originalArr, temp, start, mid+1, end);  
}  
return inversion\_count;  
    
}  
int inversionCount(int A[], int size)  
{  
int temp[size]  
return mergeSort(A, temp, 0, size-1)  
}

Detail description

function **countInversions**(**array**) {

if length of **array** is equal to 1

return **array midPoint** = length of array \ 2

**firstHalf, inversions1** = **mergeSort**(first half of the array)

**secondHalf, inversions2** = **mergeSort**(second half of the array) initialize **sortedArray** to an empty array

initialize **i** to 0

initialize **j** to 0

initialize **inversions** to 0 while **i** < length of **firstHalf** and j < length of s**econdHalf**

if **firstHalf[i]** > **secondHalf[j]**

append **secondHalf[j]** to **sortedArray**

**inversions** = **inversions** + (**midPoint** - **i)**

else

append **firstHalf[i]** to **sortedArray** append the remaining elements of **firstHalf** to **sortedArray**

append the remaining elements of **secondHalf** to **sortedArray** **totalInversions = inversions1 + inversions2 + inversions**

return **sortedArray, totalInversions**

