**Exercise 7:**

**Divide and Conquer**

1. **Count Inversions in an array using Merge Sort**

*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 the reverse order, the inversion count is the maximum.

Formally speaking, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j

**METHOD (Enhance Merge Sort)**

* **Approach:**   
  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]

**Algorithm:**

* + 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 is the first half, the number of inversion 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

**Example:**

**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[] = {3, 1, 2}

**Output:** 2

**Explanation:** Given array has two inversions:

(3, 1), (3, 2)

1. **Closest Pair of Points using Divide and Conquer algorithm**

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.

  
The Brute force solution is O(n^2), compute the distance between each pair and return the smallest. We can calculate the smallest distance in O(nLogn) time using Divide and Conquer strategy.

**Algorithm**   
Following are the detailed steps of a O(n (Logn)^2) algorithm.   
*Input:* An array of n points *P[]*

*Output:* The smallest distance between two points in the given array.  
As a pre-processing step, the input array is sorted according to x coordinates.

**1)** Find the middle point in the sorted array, we can take *P[n/2]* as middle point.   
**2)**Divide the given array in two halves. The first subarray contains points from P[0] to P[n/2].

The second subarray contains points from P[n/2+1] to P[n-1].  
**3)** Recursively find the smallest distances in both subarrays. Let the distances be dl and dr. Find the minimum of dl and dr. Let the minimum be d.

**4)**From the above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from the left half and the other is from the right half. Consider the vertical line passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip[] of all such points.

**5)**Sort the array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by recursively sorting and merging.

**6)** Find the smallest distance in strip[]. This is tricky. From the first look, it seems to be a O(n^2) step, but it is actually O(n). It can be proved geometrically that for every point in the strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate).

**7)** Finally return the minimum of d and distance calculated in the above step (step 6)

**Example:**

**Input**: Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};

**Output**: The smallest distance is 1.414214