Divide & Conquer
(Assignment Solutions)

Solution 1 :

```
void merge(string arr[], int lo, int mid, int hi) {
      int m = mid;
      int n = hi;
      vector<string> temp;

      int i = lo;
      int j = mid+1;

      while (i <= m && j <= n) {
          if (arr[i] <= arr[j]) {
              temp.push_back(arr[i]);
              i++;
          }
          else {
              temp.push_back(arr[j]);
              j++;
          }
      }
      while (i <= m) {
           temp.push_back(arr[i]);
           i++;
      }
      while (j <= n) {
           temp.push_back(arr[j]);
           j++;
      }

      for(int idx=0, x=lo; x<=hi; x++) {
           arr[x] = temp[idx++];
      }
  }

void mergeSort(string arr[], int lo, int hi) {
      if (lo >= hi) {
          return;
```

```
    }

    int mid = lo + (hi - lo) / 2;
    mergeSort(arr, lo, mid);
    mergeSort(arr, mid + 1, hi);


    merge(arr, lo, mid, hi);
}

int main() {
    string arr[4] = { "sun", "earth", "mars", "mercury" };
    mergeSort(arr, 0, 3);

    for (int i = 0; i < 4; i++) {
        cout << arr[i] << endl;
    }

  return 0;
}
```


Question 2 :

```
int countInRange(int nums[], int num, int lo, int hi) {
    int count = 0;
    for (int i = lo; i <= hi; i++) {
        if (nums[i] == num) {
            count++;
        }
    }
    return count;
}


int majorityElementRec(int nums[], int lo, int hi) {
    // base case; the only element in an array of size 1 is the majority
    // element.
    if (lo == hi) {
        return nums[lo];
```

```
    }


    // recurse on left and right halves of this slice.
    int mid = (hi-lo)/2 + lo;
    int left = majorityElementRec(nums, lo, mid);
    int right = majorityElementRec(nums, mid+1, hi);


    // if the two halves agree on the majority element, return it.
    if (left == right) {
        return left;
    }


    // otherwise, count each element and return the "winner".
    int leftCount = countInRange(nums, left, lo, hi);
    int rightCount = countInRange(nums, right, lo, hi);


    return leftCount > rightCount ? left : right;
}


int majorityElement(int nums[], int n) {
    return majorityElementRec(nums, 0, n-1);
}


int main() {
    int nums[] = {2,2,1,1,1,2,2};
    int n = 7;
    cout << majorityElement(nums, n) << endl;
    return 0;
}
```

Time Complexity of solution : O(nlogn)

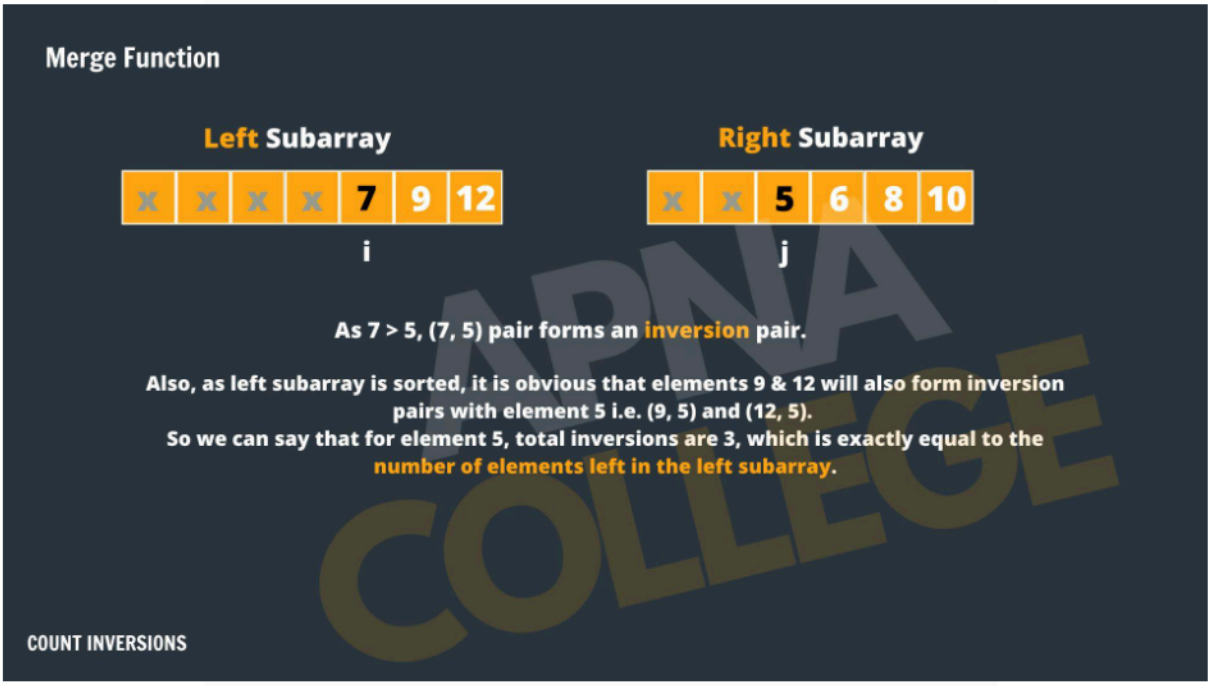
Question 3 :

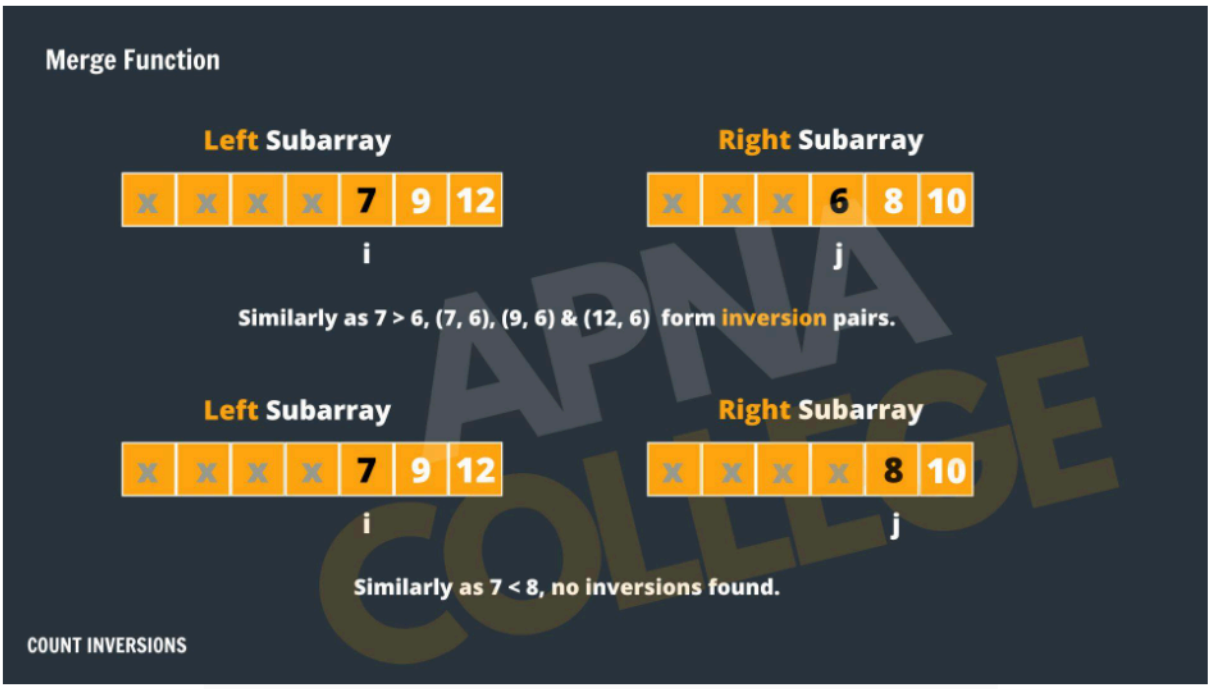
Approach - Modified Merge Sort

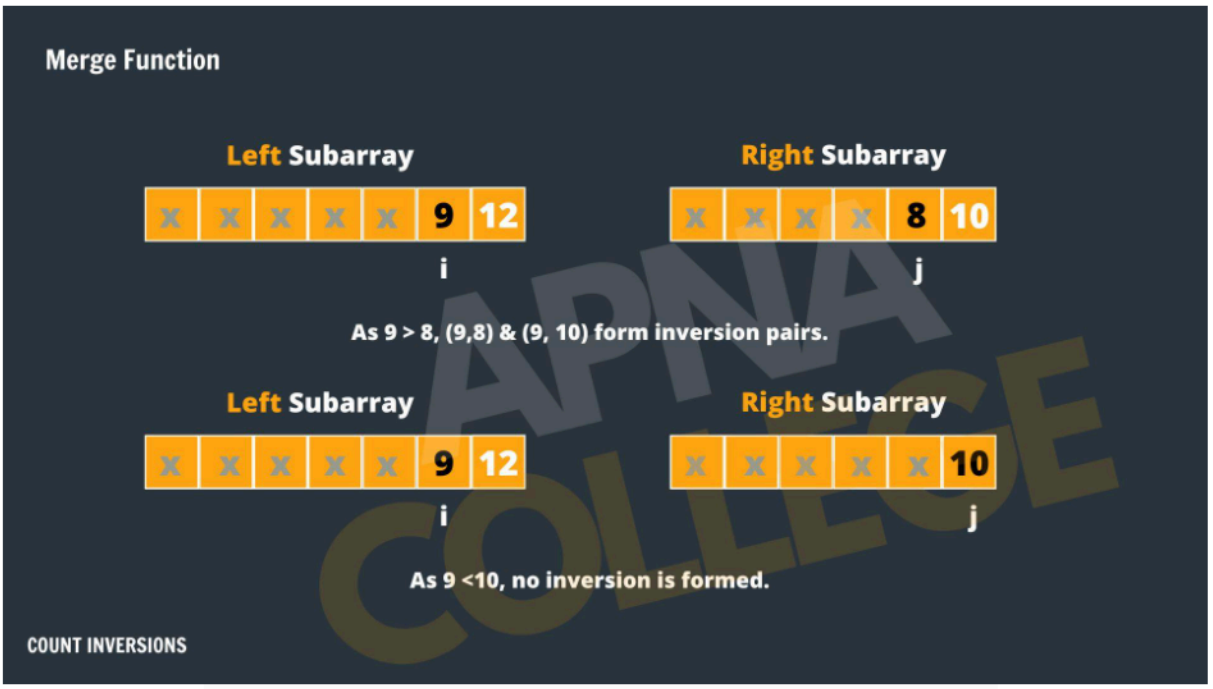
Idea : Suppose the number of inversions in the left half and right half of the array (let be inv1 and inv2); what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is – the inversions that need to be counted during the merge step. Therefore, to get the total number of inversions that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

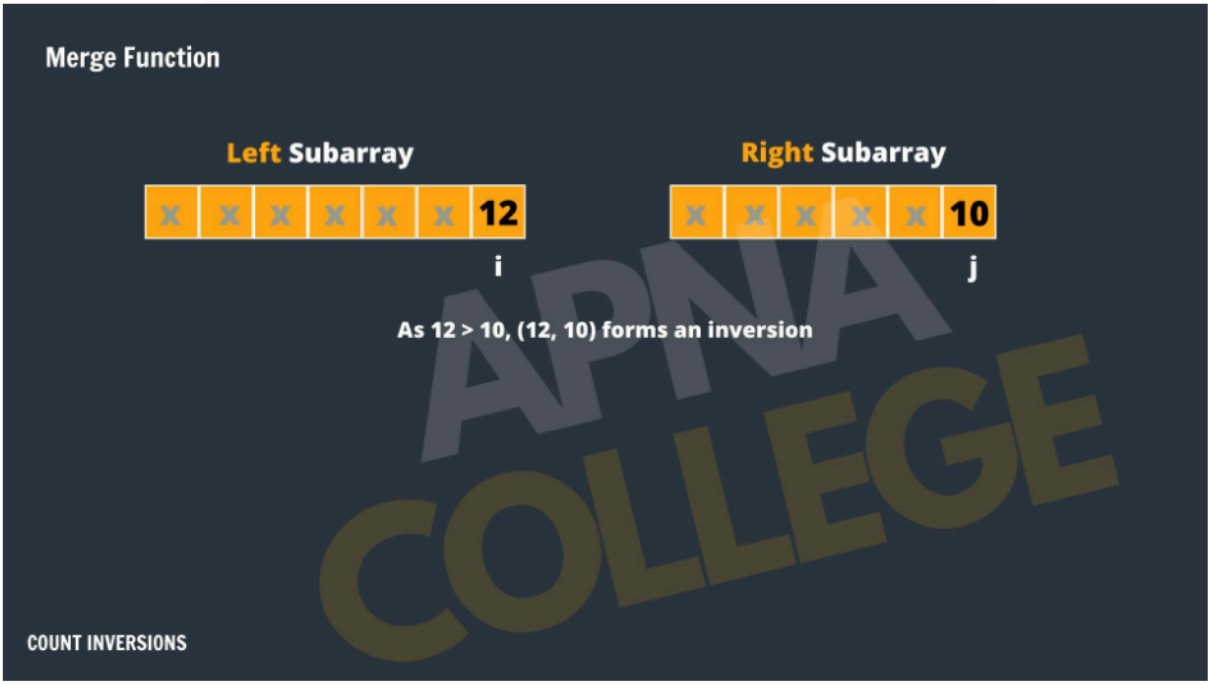
Basically, for each array element, count all elements more than it to its left and add the count to the output. This whole magic happens inside the merge function of merge sort.

Let's consider two subarrays involved in the merge process:









Algorithm:

- Split the given input array into two halves, left and right similar to merge sort recursively.
- Count the number of inversions in the left half and right half along with the inversions found during the merging of the two halves.
- Stop the recursion, only when 1 element is left in both halves.
- To count the number of inversions, we will use a two pointers approach. Let us consider two pointers i and j, one pointing to the left half and the other pointing towards the right half.

- While iterating through both the halves, if the current element A[i] is less than A[j], add it to the sorted list, else increment the count by mid – i.

Time complexity - O(n* logn)

```
int merge(int arr[], int left, int mid, int right) {
      int i = left, j = mid+1;
      int invCount = 0;
      vector<int> temp;
       while ((i <= mid) && (j <= right)) {
          if (arr[i] <= arr[j]) {
              temp.push_back(arr[i]);
              i++;
          }
          else {
              temp.push_back(arr[j]);
              invCount += (mid - i);
              j++;
          }
      }
       while (i <= mid) {
          temp.push_back(arr[i]);
          i++;
      }
       while (j <= right) {
          temp.push_back(arr[j]);
          j++;
      }
       for (int x = left, k = 0; x <= right; x++, k++) {
          arr[x] = temp[k];
      }
       return invCount;
  }


int mergeSort(int arr[], int left, int right) {
      int invCount = 0;
       if (right > left) {
          int mid = (right + left) / 2;
           invCount = mergeSort(arr, left, mid);
```

```
            invCount += mergeSort(arr, mid + 1, right);
            invCount += merge(arr, left, mid + 1, right);
        }
        return invCount;
}
 int getInversions(int arr[], int n) {
        return mergeSort(arr, 0, n - 1);
}


int main() {
        int arr[] = {1, 20, 6, 4, 5};
        int n = 5;
        cout << "Inversion Count = " << getInversions(arr, n) << endl;
        return 0;
}
```