**Practical 10**

**Problem statement:** Given an unsorted array of integers, design an algorithm and implement it using a program to sort an array of elements by dividing the array into two subarrays and combining these subarrays after sorting each one of them. Your program should also find number of comparisons and inversions during sorting the array.

**Algorithm:** To sort an array of elements by dividing the array into two subarrays and combining these subarrays after sorting each one of them.

Step 1: Start

Step 2: merge(arr, left, mid, right, comparisons)

1. Initialize the size of left and right sub-arrays as n1 = mid - left + 1 and n2 = right –

mid.

2. Create two temporary sub-arrays: leftArr of size n1 and rightArr of size n2.

3. Copy the elements from the original array arr into leftArr and rightArr.

4. Initialize indices i = 0, j = 0, and k = left to keep track of positions in leftArr, rightArr,

and the original array.

5. Initialize a counter invCount = 0 to store the number of inversions.

6. While loop: Compare elements in leftArr and rightArr:

a- If leftArr[i] <= rightArr[j], increment i and place leftArr[i] into the main array at

position k.

b- Otherwise, increment j and place rightArr[j] into the main array at position k,and

k,and update the inversion count invCount += (n1 - i)

c- Increment comparisons after each comparison.

7. Copy any remaining elements from leftArr or rightArr to the original array.

8. Return the inversion count invCount.

Step 3: mergeAndCount(arr, left, right, comparisons)

1. Base condition: If left >= right, return 0 (no inversions).

2. Calculate mid: mid = left + (right - left) / 2.

3. Recursively call mergeAndCount(arr, left, mid, comparisons) and store the result in

invCount.

4. Recursively call mergeAndCount(arr, mid + 1, right, comparisons) and add the result.

to invCount

5. Call the merge function to merge the two halves and add the result to invCount.

6. Return the total inversion count invCount.

Step 4: mergeSortAndCount(arr, n, comparisons, inversions)

1. Initialize comparisons = 0 and inversions = 0.

2. Call mergeAndCount(arr, 0, n - 1, comparisons) to start the merge sort and inversion

Count.

3. Return the total inversion count.

Step 5: main()

1. Read the number of test cases T.

2. For each test case:

a- Read the number of elements n.

b-Declare an array arr of size n. o Read the n elements of the array.

c-Initialize comparisons = 0 and inversions = 0.

d- Call mergeSortAndCount(arr, n, comparisons, inversions) to perform merge sort and

count comparisons and inversions

e-Output the sorted array.

d-Output the total number of comparisons and inversions.

Step 6: Stop

**Source code:**

#include <bits/stdc++.h>

using namespace std;

int merge(vector<int> &arr, int left, int mid, int right, int &comparisons)

{

    int n1 = mid - left + 1;

    int n2 = right - mid;

    vector<int> leftArr(n1), rightArr(n2);

    for (int i = 0; i < n1; i++)

    {

        leftArr[i] = arr[left + i];

    }

    for (int i = 0; i < n2; i++)

    {

        rightArr[i] = arr[mid + 1 + i];

    }

    int i = 0, j = 0, k = left;

    int invCount = 0;

    while (i < n1 && j < n2)

    {

        comparisons++;

        if (leftArr[i] <= rightArr[j])

        {

            arr[k++] = leftArr[i++];

        }

        else

        {

            arr[k++] = rightArr[j++];

            invCount += (n1 - i);

        }

    }

    while (i < n1)

    {

        arr[k++] = leftArr[i++];

    }

    while (j < n2)

    {

        arr[k++] = rightArr[j++];

    }

    return invCount;

}

int mergeAndCount(vector<int> &arr, int left, int right, int &comparisons)

{

    if (left >= right)

         return 0;

    int mid = left + (right - left) / 2;

    int invCount = 0;

    invCount += mergeAndCount(arr, left, mid, comparisons);

    invCount += mergeAndCount(arr, mid + 1, right, comparisons);

    invCount += merge(arr, left, mid, right, comparisons);

    return invCount;

}

void mergeSortAndCount(vector<int> &arr, int n, int &comparisons, int &inversions)

{

    comparisons = 0;

    inversions = mergeAndCount(arr, 0, n - 1, comparisons);

}

int main()

{

    int T;

    cin >> T;

    for (int i = 0; i < T; i++)

    {

        int n;

        cin >> n;

        vector<int> arr(n);

        for (int i = 0; i < n; i++)

        {

            cin >> arr[i];

        }

        int comparisons = 0, inversions = 0;

        mergeSortAndCount(arr, n, comparisons, inversions);

        for (int i = 0; i < n; i++)

        {

            cout << arr[i] << " ";

        }

        cout << endl;

        cout << "comparisons = " << comparisons << endl;

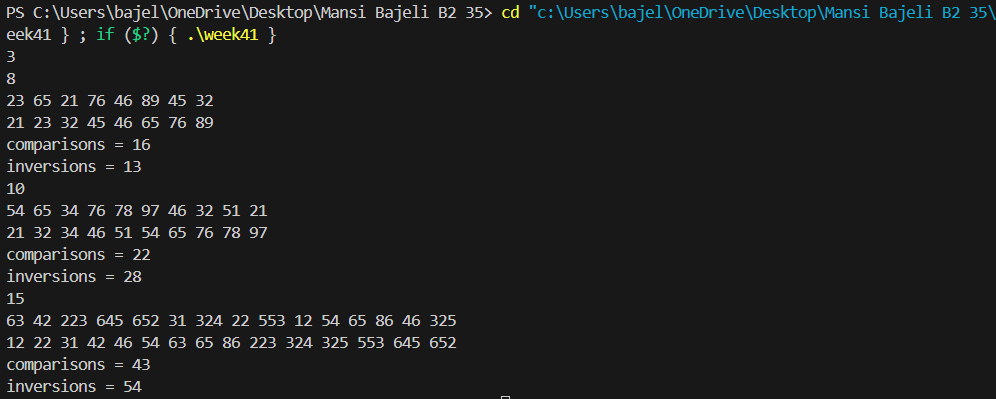
        cout << "inversions = " << inversions << endl;

    }

    return 0;

}

**OUTPUT:**

****

**Practical 11**

**Problem statement :** Given an unsorted array of integers, design an algorithm and implement it using a program to find Kth smallest or largest element in the array. (Worst case Time Complexity = O(n))

**Algorithm :** To find Kth smallest or largest element in the array.

Step 1: Start

Step 2: pivot(arr, low, high, swaps ,comp)

1- Choose the last element as the pivot.

2- Initialize an index **i** to track the partition position.

3-Iterate through the array, comparing each element with the pivot

1. If an element is smaller than the pivot, swap it to the left partition and update

swap count.

b-Increase the comparison count for each comparison made.

4- Finally, swap the pivot into its correct position and increment the swap count.

5- Return the pivot index.

Step 3: quick\_Sort(arr, low, high, swaps, comp)

1- If low < high, recursively:

a-Partition the array using the pivot function.

b-Call quick\_Sort on the left and right subarrays

Step 4: main()

1- Read the number of test cases T.

2-For each test case:

a-Read the size of the array n.

b-Read the n elements into an array.

3- Initialize swap and comparison counters to zero.

4-Call quick\_Sort on the array.

5-Print the sorted array.

6-Print the total comparisons and swaps made during sorting.

Step 5: Stop

**Source code:**

#include <bits/stdc++.h>

using namespace std;

int pivot(vector<int> &arr, int low, int high, int &swaps, int &comp)

{

    int temp;

    int i = low - 1;

    int p = arr[high];

    for (int j = low; j < high; j++)

    {

        if (arr[j] < p)

        {

            i++;

            temp = arr[i];

            arr[i] = arr[j];

            arr[j] = temp;

            swaps++;

        }

        comp++;

    }

    i++;

    temp = arr[i];

    arr[i] = arr[high];

    arr[high] = temp;

    swaps++;

    return i;

}

void quick\_Sort(vector<int> &arr, int low, int high, int &swaps, int &comp)

{

    if (low < high)

    {

        int pi = pivot(arr, low, high, swaps, comp);

        quick\_Sort(arr, low, pi - 1, swaps, comp);

        quick\_Sort(arr, pi + 1, high, swaps, comp);

    }}

int main(){

    int n, val, T;

    cin >> T;

    for (int i = 0; i < T; i++)

    {

        cin >> n;

        vector<int> arr;

        for (int i = 0; i < n; i++)

        {

            cin >> val;

            arr.push\_back(val);

        }

        int comp = 0, swaps = 0;

        quick\_Sort(arr, 0, n - 1, swaps, comp);

        for (int i = 0; i < n; i++)

        {

            cout << arr[i] << " ";

        }

        cout << endl;

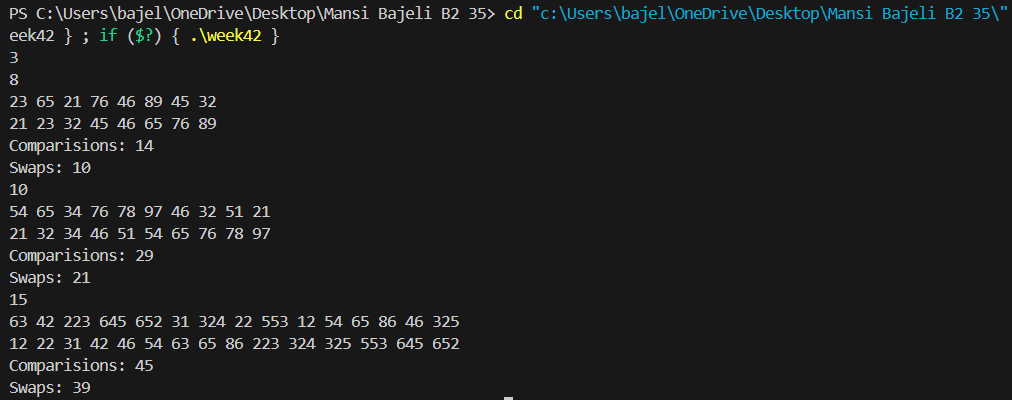
        cout << "Comparisions: " << comp << endl;

        cout << "Swaps: " << swaps << endl;

    }

    return 0;}

**OUTPUT:**

****

**Practical 12**

**Problem statement**: Given an unsorted array of integers, design an algorithm and implement it using a program to find Kth smallest or largest element in the array. (Worst case Time Complexity = O(n))

**Algorithm:** To find Kth smallest or largest element in the array.

Step 1: Start

Step 2: kthElement(arr, n, k)

1-if k > n, print "Invalid key" and return.

2-Find Minimum and Maximum Elements:

3-Initialize max = arr[0] and min = arr[0].

4-Iterate through the array:

a-If arr[i] > max, update max.

b-If arr[i] < min, update min.

5-Compute the range as max - min + 1.

6-Create a count array of size range, initialized to 0.

7-Create an auxiliary output array b of size n.

8-Traverse the input array and update the count array:

a-count[arr[i] - min]++ (to store occurrences).

9-Modify the count array to store prefix sums:

a-count[i] = count[i] + count[i - 1].

10-Traverse arr from right to left:

a-Compute index = --count[arr[i] - min].

b-Place arr[i] at b[index].

11-Kth Smallest = b[k - 1].

12-Kth Largest = b[n - k].

13-Print both values.

Step 3: main()

1-Read the number of test cases **T**.

2-For each test case:

3-Read **n**, the size of the array.

4-Read **n** elements into an array **arr**.

5-Read the value **k**.

6-call function kthelement()

Step 4: Stop

**Source code:**

#include <bits/stdc++.h>

using namespace std;

void kthElement(vector<int> &arr, int n, int k)

{

    if (k > n)

    {

        cout << "Invalid key";

    }

    int max = arr[0], min = arr[0];

    for (int i = 0; i < n; i++)

    {

        if (arr[i] > max)

        {

            max = arr[i];

        }

        if (arr[i] < min)

        {

            min = arr[i];

        }

    }

    int range = max - min + 1, index;

    int count[range] = {0};

    int b[n];

    for (int i = 0; i < n; i++)

    {

        count[arr[i] - min]++;

    }

    for (int i = 1; i < range; i++)

    {

        count[i] = count[i] + count[i - 1];

    }

    for (int i = n - 1; i >= 0; i--)

    {

        index = --count[arr[i] - min];

        b[index] = arr[i];

    }

    cout << "Kth Smallest Number:" << b[k - 1] << endl;

    cout << "Kth Largest Number:" << b[n - k]<<endl;

}

int main()

{

    int n, val, T;

    cin >> T;

    for (int i = 0; i < T; i++)

    {

        cin >> n;

        vector<int> arr;

        for (int i = 0; i < n; i++)

        {

            cin >> val;

            arr.push\_back(val);

        }

        int k;

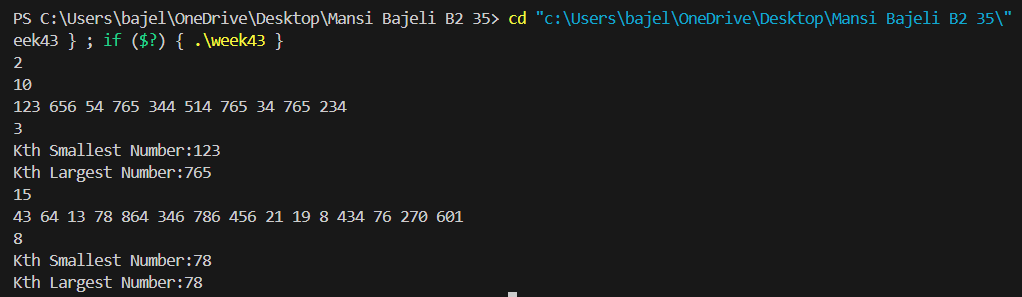
        cin >> k;

        kthElement(arr, n, k);

    }

    return 0;}

**OUTPUT:**

****

**Practical 13**

**Problem statement:** Given an unsorted array of alphabets containing duplicate elements. Design an algorithm and implement it using a program to find which alphabet has maximum number of occurrences and print it. (Time Complexity = O(n)) (Hint: Use counting sort).

**Algorithm:** To find which alphabet has maximum number of occurrences.

Step 1: Start

Step 2: Read the number of test cases T.

Step 3:For each test case:

a-Read the size of the array n.

b-Read n characters into an array arr.

Step 4: Initialize a frequency array count[26] to store frequencies of characters 'a' to 'z'.

Step 5: Count the frequency of each character in the array arr using the count[] array.

Step 6: Initialize the variables max to 1, ch to store the character with maximum frequency, and flag to 0.

Step 7: Loop through the count[] array:

a-For each character, if the frequency is greater than max, update max, ch, and set

flag=1.

Step 8: Check the flag:

a-If flag == 1, print the character ch with its frequency max.

b-Else, print "NO DUPLICATE FOUND".

Step 9: Repeat steps 3 to 8 for all test cases.

Step 10: Stop

**Source code:**

#include <bits/stdc++.h>

using namespace std;

void maxFrequency(vector<char> &arr)

{

    int count[26] = {0};

    for (int i = 0; i < arr.size(); i++)

    {

        count[arr[i] - 'a']++;

    }

    int max = 1;

    char ch;

    int flag = 0;

    for (int i = 0; i < 26; i++)

    {

        if (count[i] > max)

        {

            max = count[i];

            ch = i + 'a';

            flag = 1;

        }

    }

    if (flag == 1)

    {

        cout << ch << " - " << max << endl;

    }

    else

    {

        cout << "NO DUPLICATE FOUND"<<endl;

    }

}

int main()

{

    int n, T;

    char val;

    cin >> T;

    for (int i = 0; i < T; i++)

    {

        cin >> n;

        vector<char> arr;

        for (int i = 0; i < n; i++)

        {

            cin >> val;

            arr.push\_back(val);

        }

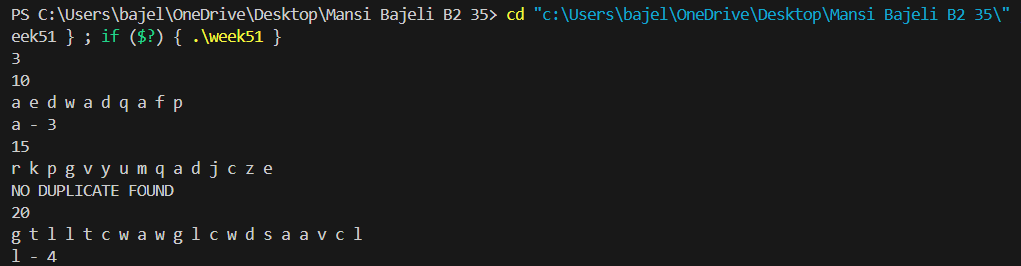
        maxFrequency(arr);

    }

    return 0;

}

**OUTPUT:**

****

**Practical 14**

**Problem statement:** Given an unsorted array of integers, design an algorithm and implement it using a program to find whether two elements exist such that their sum is equal to the given key element. (Time Complexity = O(n log n)).

**Algorithm:** To find whether two elements exist such that their sum is equal to the given key element.

Step 1: Start

Step 2: Read the number of test cases T.

Step 3: For each test case:

a-Read the size of the array n.

b-Read n integers and store them in an array arr.

c-Read the target sum key.

Step 4: Sort the Array

Step 5: Sort the array arr in ascending order.

Step 6: Find the Pair Using Binary Search

Step 7: Iterate through the array:

a-For each element arr[i], compute target = key - arr[i].

b-Perform a binary search on the remaining elements (i+1 to n-1) to check if target exists.

c-If found, print the pair arr[i] and target, then return.

Step 8: If no such pair exists, print "No Such Element Exist".

Step 9: Repeat the process for the next test case.

Step 10: Stop

**Source code:**

#include <bits/stdc++.h>

using namespace std;

bool binarysearch(vector<int> &arr, int low, int high, int target)

{

    while (low <= high)

    {

        int mid = low + (high - low) / 2;

        if (target == arr[mid])

        {

            return true;

        }

        else if (target < arr[mid])

        {

            high = mid - 1;

        }

        else

        {

            low = mid + 1;

        }

    }

    return false;

}

void twoSum(vector<int> &arr, int n, int key)

{

    sort(arr.begin(), arr.end());

    for (int i = 0; i < n; i++)

    {

        int target = key - arr[i];

        if (binarysearch(arr, i + 1, n - 1, target))

        {

            cout << "The pair is :" << endl;

            cout << arr[i] << " " << target << endl;

            return;

        }

    }

    cout << "No Such Element Exist";

}

int main()

{

    int n, T, val;

    cin >> T;

    for (int i = 0; i < T; i++)

    {

        cin >> n;

        vector<int> arr;

        for (int i = 0; i < n; i++)

        {

            cin >> val;

            arr.push\_back(val);

        }

        int key;

        cin >> key;

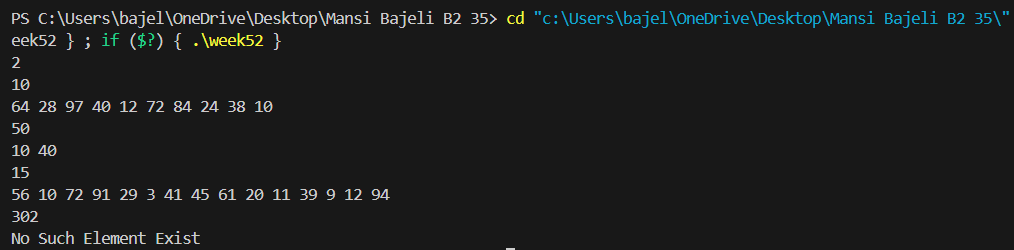
        twoSum(arr, n, key);

    }

    return 0;

}

**OUTPUT:**

****

**Practical 15**

**Problem statement:** You have been given two sorted integer arrays of size m and n. Design an algorithm and implement it using a program to find list of elements which are common to both. (Time Complexity = O(m+n))

**Algorithm:** To find list of elements which are common to both.

Step 1: Start

Step 2: findCommonElements(arr1, arr2, m, n)

1. Initialize two pointers i = 0 and j = 0.

2. While both pointers are within bounds (i < m and j < n):

a. If arr1[i] == arr2[j] ,print arr1[i] and increment both i and j.

b. If arr1[i] < arr2[j], increment i.

c. If arr1[i] > arr2[j], increment j.

Step 3: main()

1. Read the size m of the first array and input the elements of arr1.

2. Read the size n of the second array and input the elements of arr2.

3. Call findCommonElements(arr1, arr2, m, n) to print the common elementsbetween

arr1 and arr2.

Step 4: Stop

**Source code:**

#include <bits/stdc++.h>

using namespace std;

void findCommonElements(const vector<int> &arr1, const vector<int> &arr2, int m, int n)

{

    int i = 0, j = 0;

    while (i < m && j < n)

    {

        if (arr1[i] == arr2[j])

        {

            cout << arr1[i] << " ";

            i++;

            j++;

        }

        else if (arr1[i] < arr2[j])

        {

            i++;

        }

        else

        {

            j++;

        }

    }

}

int main()

{

    int m, n;

    cin >> m;

    vector<int> arr1(m);

    for (int i = 0; i < m; i++)

    {

        cin >> arr1[i];

    }

    cin >> n;

    vector<int> arr2(n);

    for (int i = 0; i < n; i++)

    {

        cin >> arr2[i];

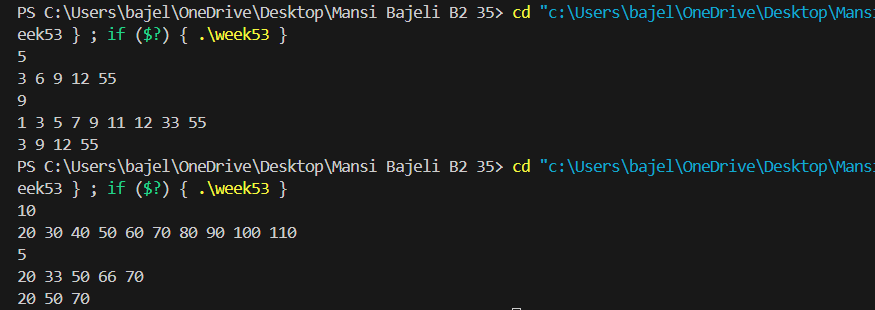
    }

    findCommonElements(arr1, arr2, m, n);

    return 0;

}

**OUTPUT:**

****