**WEEK 1**

**Practical-1**

Given an array of nonnegative integers, design a linear algorithm and implement it using a program to find whether given key element is present in the array or not. Also, find total number of comparisons for each input case. (Time Complexity = O(n), where n is the size of input)

**Algorithm**

* Initialize a counter called "comparisons" to 0 and a flag "found" to false.
* Loop through each element of the array.
* For each element, increment the "comparisons" counter by 1.
* Check if the current element is equal to the key:
* If found set "found" to true and exit the loop immediately.
* After the loop, check whether the key was found and display the total number of comparisons.

**Code**

#include <iostream>

#include <vector>

using namespace std;

bool search(const vector<int>& arr, int key, int &comparisons) {

comparisons = 0;

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

comparisons++;

if (arr[i] == key)

return true;

}

return false;

}

int main() {

int n, key, comparisons;

cout << "Enter number of elements"<<endl;

cin >> n;

vector<int> arr(n);

cout << "Enter " << n << " numbers "<<endl;

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

cin >> arr[i];

}

cout << "Enter key to search"<<endl;

cin >> key;

bool found = search(arr, key, comparisons);

if (found)

cout << "Key " << key << " is present in the array" << endl;

else

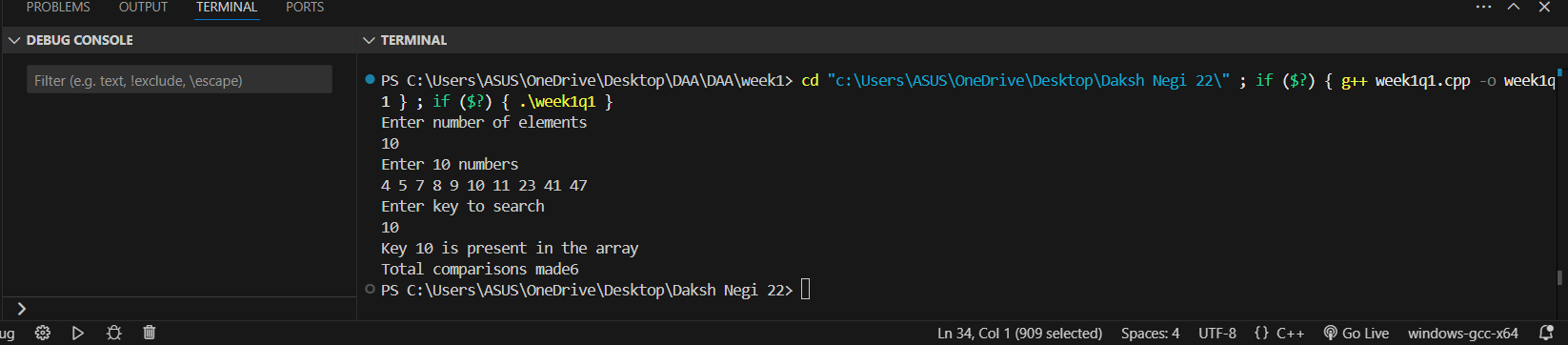
cout << "Key " << key << " is not present in the array" << endl;

cout << "Total comparisons made" << comparisons << endl;

return 0;

}

Output



**Practical-2**

Given an already sorted array of positive integers, design an algorithm and implement it using a program to find whether given key element is present in the array or not. Also, find total number of comparisons for each input case. (Time Complexity = O(nlogn), where n is the size of input).

**Algorithm**

* Initialize a counter comparisons to 0.
* Set left to 0 and right to n-1 (where n is the array size).
* While left is less than or equal to right:

Increment comparisons by 1.

Calculate mid as (left + right) / 2.

If the element at mid equals the key, return true.

Otherwise, if the key is less than the element at mid, update right to mid - 1.

Otherwise, update left to mid + 1.

* If the loop ends without finding the key, return false.
* Check whether the key was found and display the total number of comparisons made.

**Code**

#include <iostream>

#include <vector>

using namespace std;

bool binarySearch(vector<int>& arr, int key, int &comparisons) {

int left = 0, right = arr.size() - 1;

comparisons = 0;

while (left <= right) {

comparisons++;

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

if (arr[mid] == key)

return true;

else if (key < arr[mid])

right = mid - 1;

else

left = mid + 1;

}

return false;

}

int main() {

int n, key, comparisons;

cout << "Enter number of elements: ";

cin >> n;

vector<int> arr(n);

cout << "Enter " << n << " sorted positive integers: ";

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

cin >> arr[i];

}

cout << "Enter key to search: ";

cin >> key;

bool found = binarySearch(arr, key, comparisons);

if (found)

cout << "Key " << key << " is present in the array." << endl;

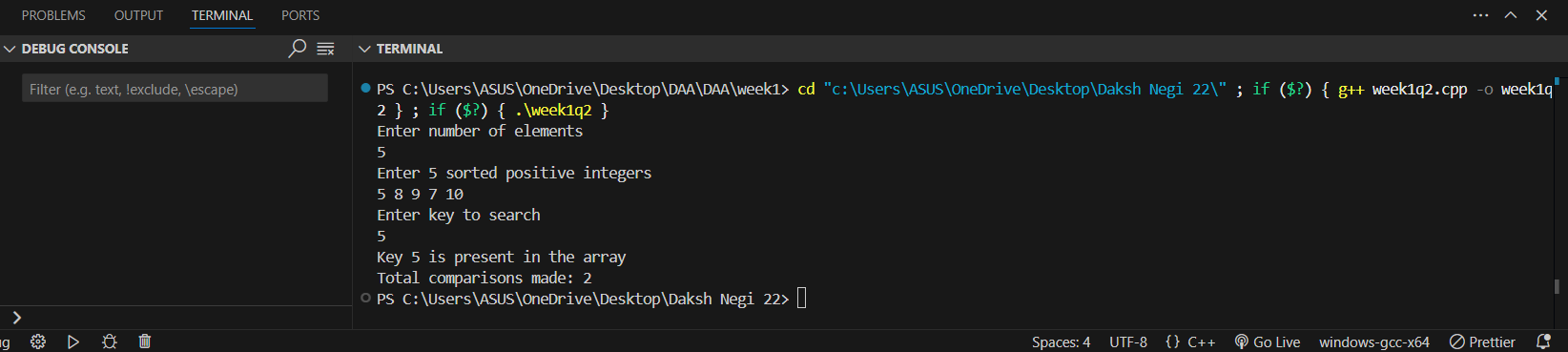
else

cout << "Key " << key << " is not present in the array." << endl;

cout << "Total comparisons made: " << comparisons << endl;

return 0;

}



**Practical-3**

Given an already sorted array of positive integers, design an algorithm and implement it using a program to find whether a given key element is present in the sorted array or not. For an array arr[n], search at the indexes arr[0], arr[2], arr[4],.....,arr[2k] and so on. Once the interval (arr[2k] < key < arr[ 2k+1] ) is found, perform a linear search operation from the index 2k to find the element key. (Complexity < O(n), where n is the number of elements need to be scanned for searching): Jump Search

**Algorithm**

* Start by initializing a counter comparisons to 0.
* Let n be the size of the sorted array.
* Set an index prev to 0 and define a fixed jump size of 2.
* While prev is less than n and the element at index prev is less than the key:

Increment comparisons.

Check if prev + jump is beyond the array bounds or if the element at index prev + jump is greater than or equal to the key:

If so, break out of the loop; this defines the interval [prev, min(prev + jump, n)] where the key could be.

Otherwise, update prev by adding the jump size.

* Perform a linear search starting from index prev up to the smaller of prev + jump or n:

For each element in this interval, increment comparisons.

If the element equals the key, return that the key is found.

* If the key is not found in the interval, return that the key is not present.
* The total comparisons count records the number of comparisons made during both the jump and linear search phases.

**Code**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

bool jumpSearch(vector<int>& arr, int key, int &comparisons) {

comparisons = 0;

int n = arr.size();

int jump = 2, prev = 0;

while (prev < n && arr[prev] < key) {

comparisons++;

prev += jump;

}

int start = max(0, prev - jump);

for (int i = start; i < n && i <= prev; i++) {

comparisons++;

if (arr[i] == key)

return true;

}

return false;

}

int main() {

int n, key, comparisons;

cin >> n;

vector<int> arr(n);

for (int i = 0; i < n; i++)

cin >> arr[i];

cin >> key;

bool found = jumpSearch(arr, key, comparisons);

if(found) {

cout<<"Key is present in the array"<<endl;

}

else{

cout<< "Key is not present in the array" << endl;

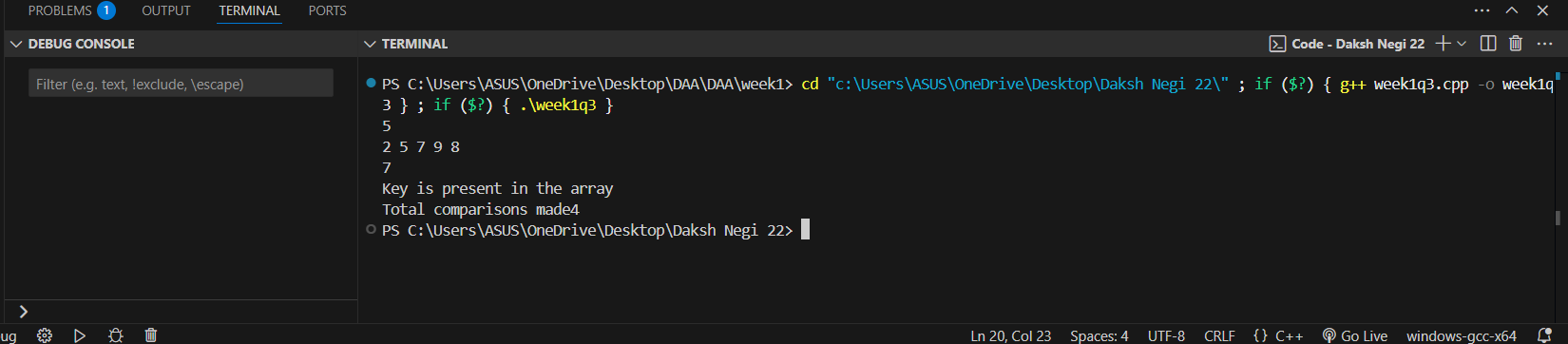
}

cout << "Total comparisons made" << comparisons << endl;

return 0;

}

**Output**

****

**Week-2**

**Practical-1**

Given a sorted array of positive integers containing few duplicate elements, design an algorithmand implement it using a program to find whether the given key element is present in the array or not. If present, then also find the number of copies of given key. (Time Complexity = O(log n))

**Algorithm**

1. Start
2. Input the size of the array n.
3. Input the array elements arr[ ] in sorted order.
4. Input the key element key to search.
5. Call the function firstoccurrence(arr, n, key) to find the first index of the key:
   * Initialize low = 0, high = n-1, and first = -1
   * While low <= high
     + Set mid = (low + high) / 2
     + If arr[mid] == key
       - Store first = mid
       - Move high = mid - 1 (Search in left half)
     + Else if arr[mid] < key
       - Set low = mid + 1
     + Else
       - Set high = mid - 1
   * Return first
6. If first == -1
   * Print "Key not found"
   * Exit
7. Call the function lastoccurrence(arr, n, key) to find the last index of the key using the same binary search method (Move low = mid + 1 for right half).
8. Calculate the total number of occurrences:

count=last−first+1\text{count} = \text{last} - \text{first} + 1count=last−first+1

1. Print:
   * "Key found"
   * Number of copies = count
2. End

**Code**

**Code**

#include <iostream>

using namespace std;

int firstoccurrence(int arr[], int n, int key) {

    int low = 0, high = n - 1, first = -1;

    while (low <= high) {

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

        if (arr[mid] == key) {

            first = mid;

            high = mid - 1;

        } else if (arr[mid] < key) {

            low = mid + 1;

        } else {

            high = mid - 1;

        }

    }

    return first;

}

int lastoccurrence(int arr[], int n, int key) {

    int low = 0, high = n - 1, last = -1;

    while (low <= high) {

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

        if (arr[mid] == key) {

            last = mid;

            low = mid + 1;

        } else if (arr[mid] < key) {

            low = mid + 1;

        } else {

            high = mid - 1;

        }

    }

    return last;

}

int main() {

    int n, key;

    cout << "Enter number of elements";

    cin >> n;

    int arr[n];

    cout << "Enter sorted array elements";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    cout << "Enter key to search";

    cin >> key;

    int first = firstoccurrence(arr, n, key);

    int last = lastoccurrence(arr, n, key);

    if (first == -1) {

        cout << "Key not found" << endl;

    } else {

        int count = last - first + 1;

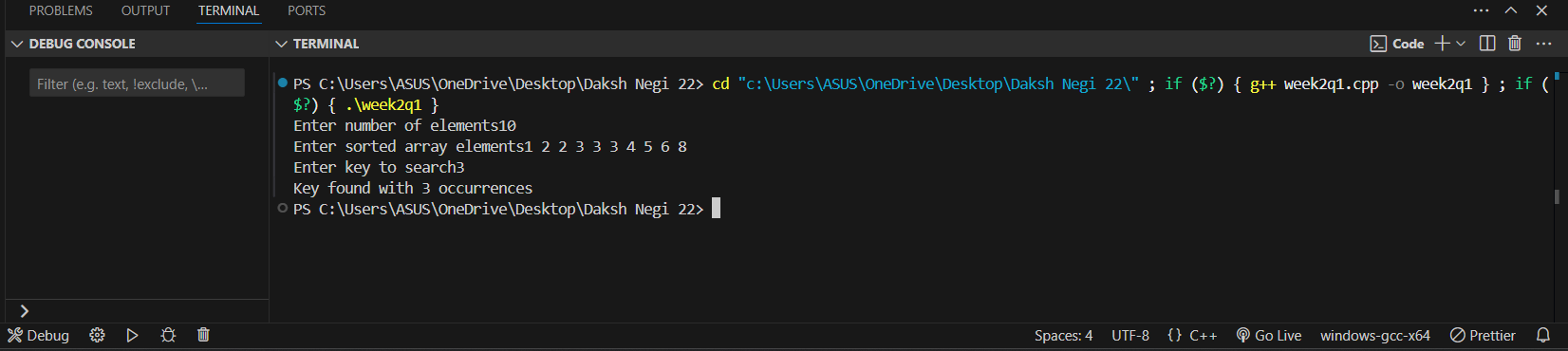
        cout << "Key found with " << count << " occurrences" << endl;

    }

    return 0;

}

**Output**

****

**Practical-2**

Given a sorted array of positive integers, design an algorithm and implement it using a program to find three indices i, j, k such that arr[i] + arr[j] = arr[k].

**Algorithm**

1. Input the size n of the array.
2. Input the sorted array arr[].
3. Iterate k from 2 to n-1.
4. Initialize two pointers i = 0 and j = k-1.
5. While i < j:
   * If arr[i] + arr[j] == arr[k], print i, j, k and return.
   * If arr[i] + arr[j] < arr[k], increment i.
   * If arr[i] + arr[j] > arr[k], decrement j.
6. If no such indices are found, print "No triplet found".

**Code**

#include<bits/stdc++.h>

using namespace std;

int main() {

int n;

cin >> n;

vector<int> arr(n);

for(int i = 0; i < n; i++)

cin >> arr[i];

for(int k = 2; k < n; k++) {

int i = 0, j = k - 1;

while(i < j) {

if(arr[i] + arr[j] == arr[k]) {

cout << i << " " << j << " " << k;

return 0;

}

else if(arr[i] + arr[j] < arr[k])

i++;

else

j--;

}

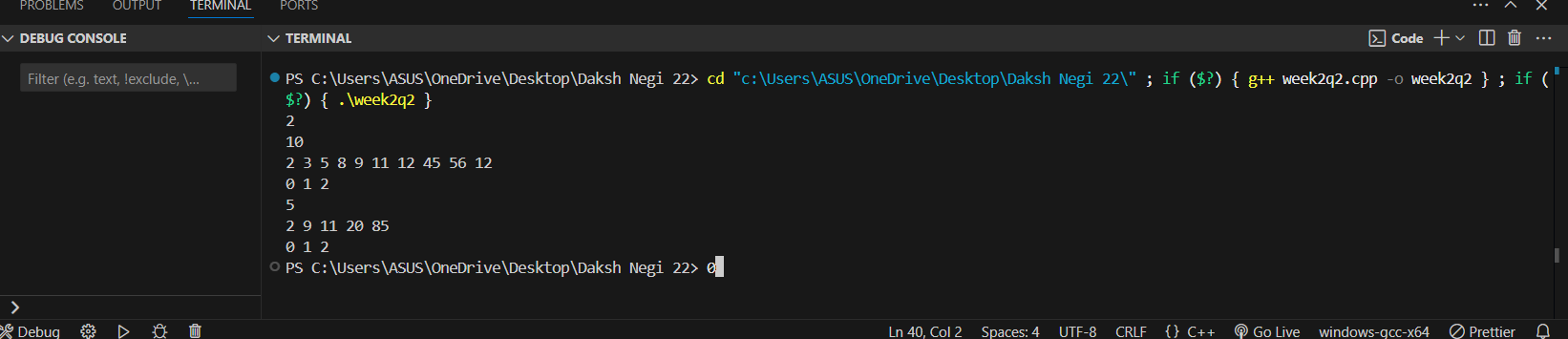
}

cout << "No triplet found";

return 0;

}

**Output**

****

Given an array of nonnegative integers, design an algorithm and a program to count the number of pairs of integers such that their difference is equal to a given key, K.

**Algorithm**

1. Read the number of test cases.
2. For Each Test Case
   * Input n (size of the array) and k (the key difference).
   * Input n elements into the vector arr.
3. Count Pairs
   * Initialize count to zero.
   * Use two nested loops:
     + Outer loop iterates from 0 to n-1.
     + Inner loop iterates from i+1 to n-1.
     + If the absolute difference |arr[i] - arr[j]| == k, increment the count.
4. Output the Count
   * Print the count of such pairs.
5. Repeat the process for all test cases.

**Code**

#include<iostream>

#include<vector>

using namespace std;

int countpairs(vector<int>& arr, int n, int k) {

    int count = 0;

    for(int i = 0; i < n; i++) {

        int j = i + 1;

        while(j < n) {

            if(abs(arr[i] - arr[j]) == k) {

                count++;

            }

            j++;

        }

    }

    return count;

}

int main() {

    int T;

    cin >> T;

    while(T--) {

        int n;

        cin >> n;

        vector<int> arr(n);

        for(int i = 0; i < n; i++) {

            cin >> arr[i];

        }

        int k;

        cin >> k;

        cout << countpairs(arr, n, k) << endl;

    }

    return 0;

}

**Output**



**Week-3**

**Practical-1**

Given an unsorted array of integers, design an algorithm and a program to sort the array using insertion sort. Your program should be able to find number of comparisons and shifts ( shifts total number of times the array elements are shifted from their place) required for sorting the array

**Algorithm**

1. Initialize comparisons = 0, shifts = 0.
2. Loop i from 1 to n-1:
   * Store key = arr[i].
   * Set j = i - 1.
   * While j >= 0 and arr[j] > key:
     + Increment comparisons.
     + Shift arr[j] to arr[j + 1].
     + Increment shifts.
     + Decrement j.
   * Place key at arr[j + 1].
   * If any comparison happened but no shift, increment comparisons.
   * If no shifting happens, increment comparisons.
3. Print the sorted array.
4. Print comparisons and shifts.

**Code**

#include<iostream>

#include<vector>

using namespace std;

void insertionSort(vector<int>& arr, int n) {

int comparisons = 0, shifts = 0;

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

shifts++;

comparisons++;

}

if (j >= 0) {

comparisons++;

}

arr[j + 1] = key;

shifts++;

}

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

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

}

cout << endl << "Comparisons: " << comparisons << endl;

cout << "Shifts: " << shifts << endl;

}

int main() {

int T;

cin >> T;

while(T--) {

int n;

cin >> n;

vector<int> arr(n);

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

cin >> arr[i];

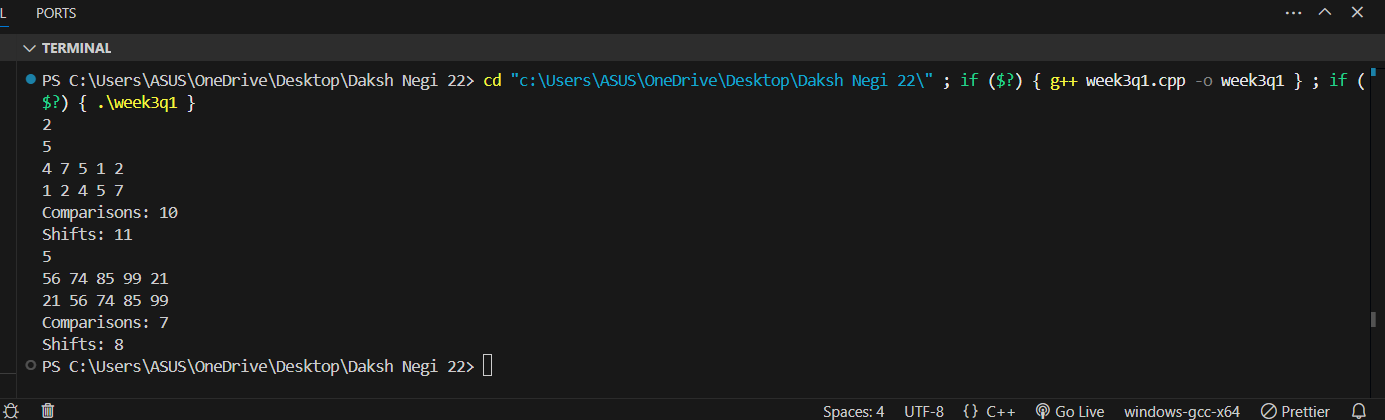
}

insertionSort(arr, n);

}

return 0;

**Output**

****

**Practical-2**

Given an unsorted array of integers, design an algorithm and implement a program to sort this array using selection sort. Your program should also find number of comparisons and number of swaps required

**Algorithm**

Read the number of test cases.

For each test case:

* Read the size of the array.
* Read the array elements.
* Initialize comparisons and swaps to zero.
* Loop through the array from 0 to n-1:
  + Set the current index as minIndex.
  + Loop through the remaining elements to find the minimum element:
    - Increment comparisons.
    - If a smaller element is found, update minIndex.
  + If minIndex is not equal to the current index, swap the elements and increment swaps.
* Print the sorted array.
* Print the number of comparisons and swaps.

End.

**Code**

#include<iostream>

#include<vector>

using namespace std;

void selectionsort(vector<int>& arr, int n, int& comparisons, int& swaps) {

    comparisons = 0;

    swaps = 0;

    for (int i = 0; i < n - 1; i++) {

        int minIndex = i;

        for (int j = i + 1; j < n; j++) {

            comparisons++;

            if (arr[j] < arr[minIndex]) {

                minIndex = j;

            }

        }

        if (minIndex != i) {

            swap(arr[i], arr[minIndex]);

            swaps++;

        }

    }

}

int main() {

    int T;

    cin >> T;

    while(T--) {

        int n;

        cin >> n;

        vector<int> arr(n);

        for(int i = 0; i < n; i++) {

            cin >> arr[i];

        }

        int comparisons, swaps;

        selectionsort(arr, n, comparisons, swaps);

        for(int i = 0; i < n; i++) {

            cout << arr[i] << " ";

        }

        cout << endl << "Comparisons: " << comparisons << endl;

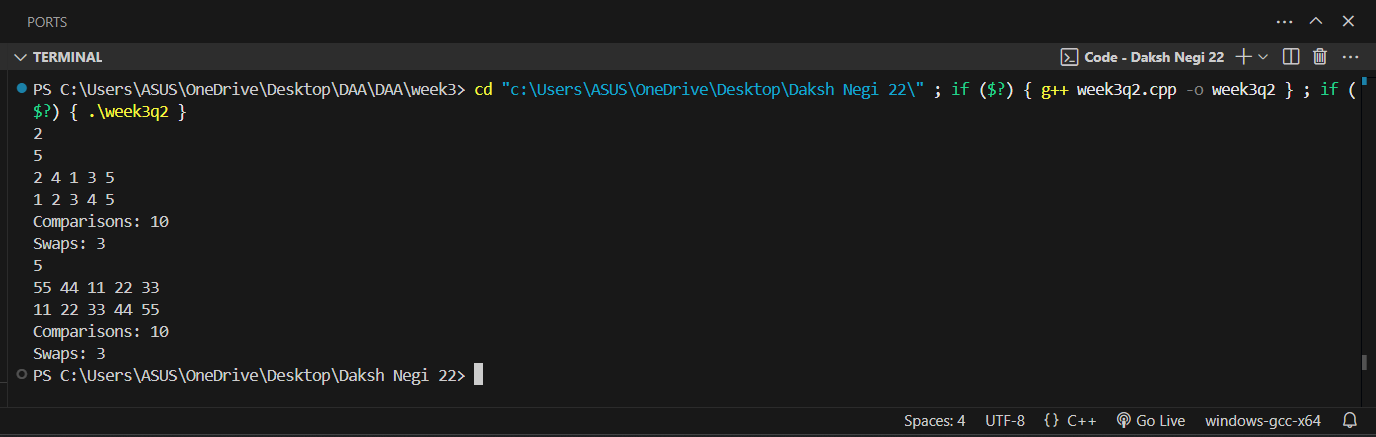
        cout << "Swaps: " << swaps << endl;

    }

    return 0;

}

**Output**

****

**Practical-3**

Given an unsorted array of positive integers, design an algorithm and implement it using a program to find whether there are any duplicate elements in the array or not. (use sorting) (Time Complexity = O(n log n))

**Algorithm**

**Merge Sort:**

1. Input: Array arr[], Left index l, Right index r.
2. If l >= r, return (Base case).
3. Find middle index m = (l + r) / 2.
4. Recursively sort the left subarray mergeSort(arr, l, m).
5. Recursively sort the right subarray mergeSort(arr, m+1, r).
6. Merge the two sorted halves using the merge() function.

**Duplicate Detection:**

1. Input: Array arr[] and size n.
2. Sort the array using Merge Sort.
3. Traverse the array:
   * If arr[i] == arr[i-1], return YES.
4. If no duplicates found, return NO.

**Code**

#include<iostream>

using namespace std;

void merge(int arr[], int l, int m, int r) {

    int n1 = m - l + 1;

    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (int j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k++] = L[i++];

        } else {

            arr[k++] = R[j++];

        }

    }

    while (i < n1) {

        arr[k++] = L[i++];

    }

    while (j < n2) {

        arr[k++] = R[j++];

    }

}

void mergesort(int arr[], int l, int r) {

    if (l < r) {

        int m = l + (r - l) / 2;

        mergesort(arr, l, m);

        mergesort(arr, m + 1, r);

        merge(arr, l, m, r);

    }

}

bool hasduplicate(int arr[], int n) {

    mergesort(arr, 0, n - 1);

    for (int i = 1; i < n; i++) {

        if (arr[i] == arr[i - 1]) {

            return true;

        }

    }

    return false;

}

int main() {

    int T;

    cin >> T;

    while (T--) {

        int n;

        cin >> n;

        int arr[n];

        for (int i = 0; i < n; i++) {

            cin >> arr[i];

        }

        if (hasduplicate(arr, n)) {

            cout << "YES" << endl;

        } else {

            cout << "NO" << endl;

        }

    }

    return 0;

}

**Output**

**Week-4**

**Practical-1**

. 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**

Divide:

* If the array has more than one element, divide the array into two halves.
* Find the middle index of the array.
* Recursively divide both halves until each subarray contains only one element.

Conquer:

* Merge the two sorted halves by comparing elements from both subarrays.
* Copy the smaller element into the resultant array.
* Continue the process until one of the subarrays is fully traversed.

Combine:

* Copy the remaining elements from the non-empty subarray to the resultant array.
* During merging, count:
  + Comparisons: Every time two elements are compared.
  + Inversions: When an element from the left subarray is greater than an element from the right subarray.

Base Case:

* If the array has only one element, return it as sorted.

Output:

* Return the sorted array along with the total number of comparisons and inversions.

**Code**

#include <iostream>

using namespace std;

void merge(int arr[], int left, int mid, int right, int &comparisons, int &inversions) {

    int n1 = mid - left + 1;

    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)

        L[i] = arr[left + i];

    for (int j = 0; j < n2; j++)

        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {

        comparisons++;

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

            inversions += (n1 - i);

        }

        k++;

    }

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

void mergeSort(int arr[], int left, int right, int &comparisons, int &inversions) {

    if (left < right) {

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

        mergeSort(arr, left, mid, comparisons, inversions);

        mergeSort(arr, mid + 1, right, comparisons, inversions);

        merge(arr, left, mid, right, comparisons, inversions);

    }

}

int main() {

    int n;

    cout << "Enter number of elements: ";

    cin >> n;

    int arr[n];

    cout << "Enter elements: ";

    for (int i = 0; i < n; i++)

        cin >> arr[i];

    int comparisons = 0, inversions = 0;

    mergeSort(arr, 0, n - 1, comparisons, inversions);

    cout << "Sorted array: ";

    for (int i = 0; i < n; i++)

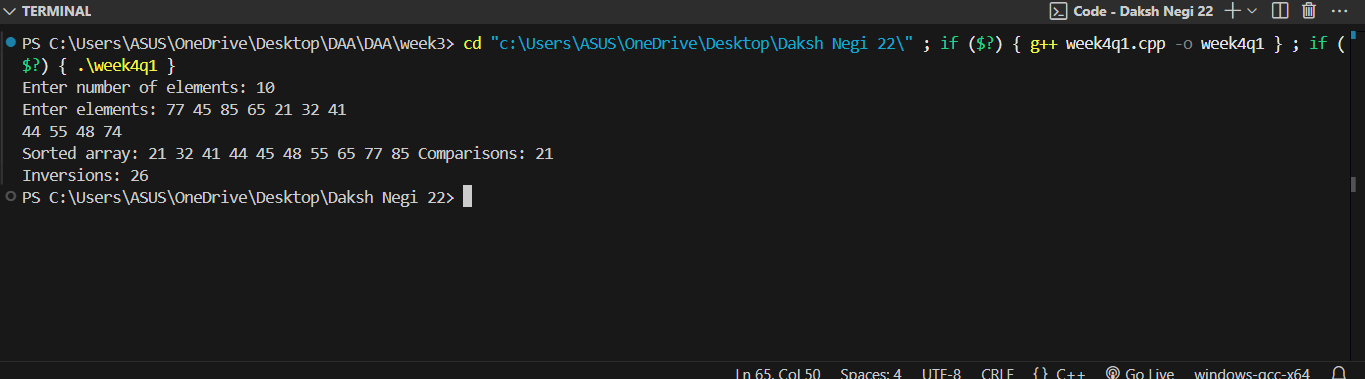
        cout << arr[i] << " ";

    cout << "Comparisons: " << comparisons<<endl;

    cout << "Inversions: " << inversions << endl;

    return 0;

}

**Output**

**Practical-2**

Given an unsorted array of integers, design an algorithm and implement it using a program to sort an array of elements by partitioning the array into two subarrays based on a pivot element such that one of the sub array holds values smaller than the pivot element while another sub array holds values greater than the pivot element. Pivot element should be selected randomly from the array. Your program should also find number of comparisons and swaps required for sorting the array

**Algorithm**

Input:

* An unsorted array arr[] of size n.

Partition Function:

* Select a random pivot element from the array.
* Swap the random pivot element with the last element to simplify partitioning.
* Initialize i = low - 1.
* Traverse the array from low to high - 1.
  + If arr[j] < pivot, increment i and swap arr[i] with arr[j].
  + Count the comparisons during each iteration.
* Swap arr[i + 1] with arr[high] to place the pivot in its correct position.
* Return the partition index.

QuickSort Function:

* If low < high:
  + Call the partition function to place the pivot at the correct position.
  + Recursively apply QuickSort to the left (low to pi-1) and right (pi+1 to high) subarrays.

Swap Function:

* Swap two elements and count the swaps.

Base Case:

* If the array contains one or zero elements, return without sorting.

Output:

* The sorted array.
* The total number of comparisons and swaps.

**Code**

#include<iostream>

#include<cstdlib>

#include<ctime>

using namespace std;

int comparisons = 0, swaps = 0;

void swap(int &a, int &b) {

    int temp = a;

    a = b;

    b = temp;

    swaps++;

}

int partition(int arr[], int low, int high) {

    srand(time(0));

    int random = low + rand() % (high - low + 1);

    swap(arr[random], arr[high]);

    int pivot = arr[high];

    int i = low - 1;

    for(int j = low; j <= high - 1; j++) {

        comparisons++;

        if(arr[j] < pivot) {

            i++;

            swap(arr[i], arr[j]);

        }

    }

    swap(arr[i + 1], arr[high]);

    return i + 1;

}

void quicksort(int arr[], int low, int high) {

    if(low < high) {

        int pi = partition(arr, low, high);

        quicksort(arr, low, pi - 1);

        quicksort(arr, pi + 1, high);

    }

}

int main() {

    int n;

    cout << "Enter number of elements: ";

    cin >> n;

    int arr[n];

    cout << "Enter array elements: ";

    for(int i = 0; i < n; i++) cin >> arr[i];

    quicksort(arr, 0, n - 1);

    cout << "Sorted array: ";

    for(int i = 0; i < n; i++) cout << arr[i] << " ";

    cout << "Comparisons: " << comparisons<<endl;

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

    return 0;

}

**Output**

****

**Practical-3**

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**

Input:

An unsorted array arr[] of size n.

* An integer k representing the position of the smallest element to find.

Find Min and Max Values:

* Initialize max = arr[0] and min = arr[0].
* Traverse the array to find the maximum and minimum elements.

Create Count Array:

* Calculate the range as max - min + 1.
* Create a count array of size range initialized to zero.

Fill Count Array:

* Traverse the array and increment count[arr[i] - min] for each element.

Cumulative Frequency:

* Traverse the count array and calculate the cumulative frequency.
* If the cumulative frequency becomes greater than or equal to k, return the index adjusted by min.

Output:

* If k is valid, print the Kth smallest element.
* If k is invalid, print an error message.

#include<iostream>

using namespace std;

int findKthsmallest(int arr[], int n, int k) {

    int max = arr[0], min = arr[0];

    for(int i = 1; i < n; i++) {

        if(arr[i] > max) max = arr[i];

        if(arr[i] < min) min = arr[i];

    }

    int range = max - min + 1;

    int count[range] = {0};

    for(int i = 0; i < n; i++) {

        count[arr[i] - min]++;

    }

    int total = 0;

    for(int i = 0; i < range; i++) {

        total += count[i];

        if(total >= k) return i + min;

    }

    return -1;

}

int main() {

    int n, k;

    cout << "Enter number of elements: ";

    cin >> n;

    int arr[n];

    cout << "Enter array elements: ";

    for(int i = 0; i < n; i++) cin >> arr[i];

    cout << "Enter K value: ";

    cin >> k;

    if(k > 0 && k <= n) {

        int result = findKthsmallest(arr, n, k);

        cout  << k << "th smallest element is: " << result << endl;

    } else {

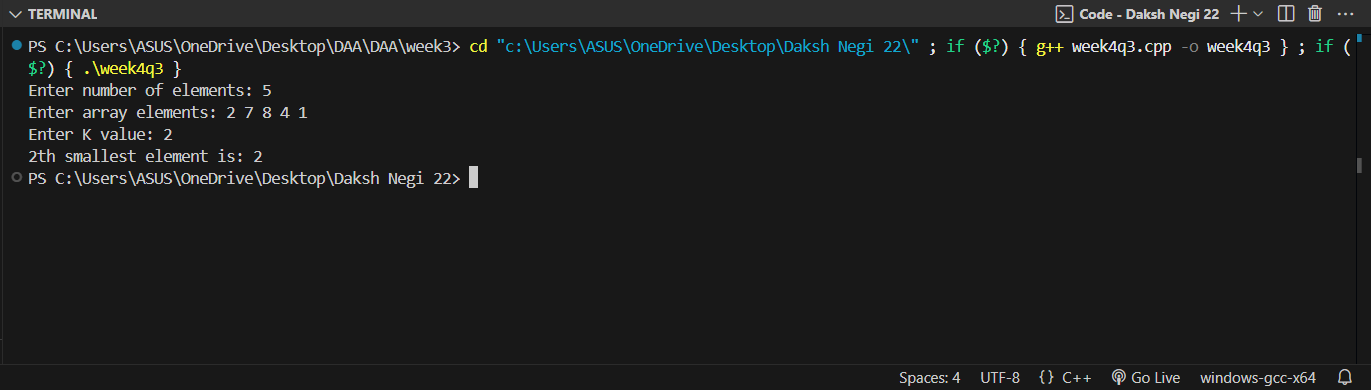
        cout << "Invalid K value" << endl;

    }

    return 0;

}

**Output**

****

**Week-5**

**Practical-1**

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**

Input:

* An array arr[] of size n containing lowercase alphabets.

Initialize Count Array:

* Create an array count[26] initialized with 0 to store the frequency of each alphabet.

Count Frequency:

* Traverse the input array from 0 to n-1.
* For each character arr[i], increment count[arr[i] - 'a'] by 1.

Find Maximum Frequency:

* Initialize maxCount = 0 and result = 'a'.
* Traverse the count[] array.
* If count[i] > maxCount, update:
  + maxCount = count[i]
  + result = i + 'a' (Convert index to corresponding alphabet).

Output:

* Return the alphabet stored in result as the alphabet with the maximum occurrences.

**Code**

#include<iostream>

#include<string>

using namespace std;

char findmaxoccurrence(char arr[], int n) {

    int count[26] = {0};

    for(int i = 0; i < n; i++) {

        count[arr[i] - 'a']++;

    }

    int maxCount = 0;

    char result = 'a';

    for(int i = 0; i < 26; i++) {

        if(count[i] > maxCount) {

            maxCount = count[i];

            result = i + 'a';

        }

    }

    return result;

}

int main() {

    int n;

    cout << "Enter number of alphabets: ";

    cin >> n;

    char arr[n];

    cout << "Enter alphabets: ";

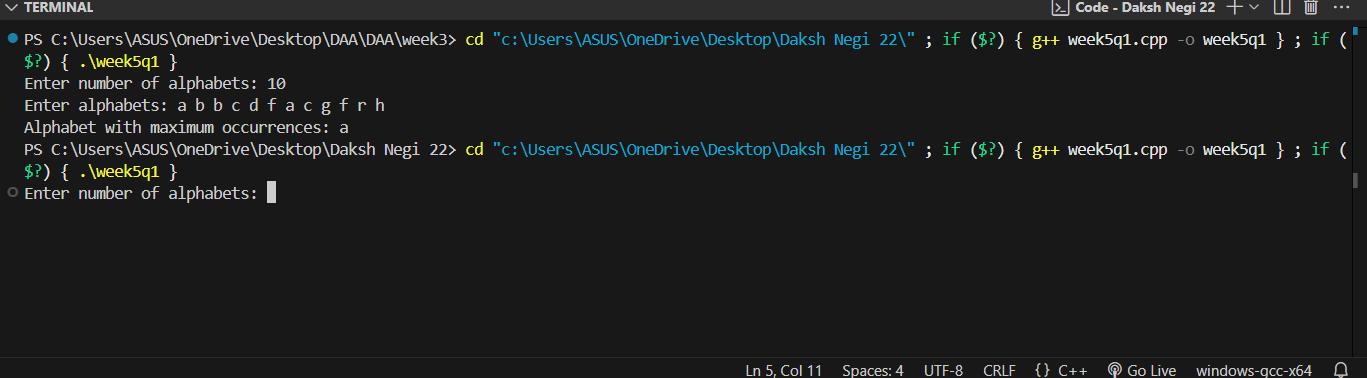
    for(int i = 0; i < n; i++) cin >> arr[i];

    char result = findmaxoccurrence(arr, n);

    cout << "Alphabet with maximum occurrences: " << result << endl;

    return 0;

}

**Output**

**Practical-2**

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**

Input the Array and Key:

* Accept the number of elements n and the array elements from the user.
* Accept the key element to find the pair.

Sort the Array:

* Use the Quick Sort algorithm to sort the array in O(n log n) time.
* Select the last element as the pivot.
* Partition the array such that elements smaller than the pivot are on the left and larger elements are on the right.
* Recursively apply quick sort on the left and right partitions.

Initialize Two Pointers:

* Set left = 0 and right = n - 1.

Search for the Pair:

* Repeat while left < right:
  + Calculate the sum of arr[left] + arr[right].
  + If the sum equals the key, return true (pair found).
  + If the sum is smaller than the key, increment the left pointer.
  + If the sum is greater than the key, decrement the right pointer.

Result:

* If no pair is found, return false.

Output the Result:

* If the pair is found, print "Pair with the given sum exists."
* Otherwise, print "No such pair exists."

**Code**

#include<iostream>

using namespace std;

void quicksort(int arr[], int low, int high) {

    if (low < high) {

        int pivot = arr[high];

        int i = low - 1;

        for (int j = low; j < high; j++) {

            if (arr[j] <= pivot) {

                i++;

                int temp = arr[i];

                arr[i] = arr[j];

                arr[j] = temp;

            }

        }

        int temp = arr[i + 1];

        arr[i + 1] = arr[high];

        arr[high] = temp;

        int pi = i + 1;

        quicksort(arr, low, pi - 1);

        quicksort(arr, pi + 1, high);

    }

}

bool Pairsum(int arr[], int n, int key) {

    quicksort(arr, 0, n - 1);

    int left = 0, right = n - 1;

    while (left < right) {

        int sum = arr[left] + arr[right];

        if (sum == key) return true;

        else if (sum < key) left++;

        else right--;

    }

    return false;

}

int main() {

    int n, key;

    cout << "Enter number of elements: ";

    cin >> n;

    int arr[n];

    cout << "Enter array elements: ";

    for (int i = 0; i < n; i++) cin >> arr[i];

    cout << "Enter key element: ";

    cin >> key;

    if (Pairsum(arr, n, key)) {

        cout << "Pair with the given sum exists." << endl;

    } else {

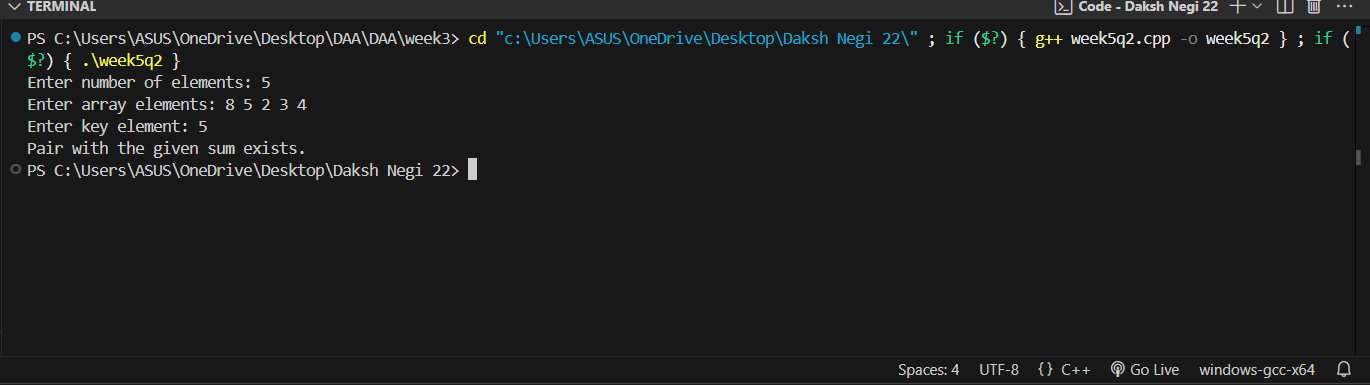
        cout << "No such pair exists." << endl;

    }

    return 0;

}

Output



**Practical-3**

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**

Input the Arrays:

* Accept the size m and elements of the first sorted array.
* Accept the size n and elements of the second sorted array.

Initialize Pointers:

* Set i = 0 for the first array.
* Set j = 0 for the second array.

Compare Elements:

* Repeat while i < m and j < n:
* If arr1[i] < arr2[j], increment i.
* If arr1[i] > arr2[j], increment j.
* If arr1[i] == arr2[j], print the element, increment both i and j.

Stop Condition:

* The loop ends when one of the arrays is fully traversed.

Output the Result:

* Print the common elements between both arrays.

**Code**

#include<iostream>

using namespace std;

void Commonelements(int arr1[], int m, int arr2[], int n, int result[], int &resSize) {

    int i = 0, j = 0;

    resSize = 0;

    cout << "Common elements: ";

    while (i < m && j < n) {

        if (arr1[i] < arr2[j]) {

            i++;

        } else if (arr1[i] > arr2[j]) {

            j++;

        } else {

            cout << arr1[i] << " ";

            result[resSize++] = arr1[i];

            i++;

            j++;

        }

    }

    cout << endl;

}

int main() {

    int m, n;

    cout << "Enter size of first array: ";

    cin >> m;

    int arr1[m];

    cout << "Enter elements of first sorted array: ";

    for (int i = 0; i < m; i++) cin >> arr1[i];

    cout << "Enter size of second array: ";

    cin >> n;

    int arr2[n];

    cout << "Enter elements of second sorted array: ";

    for (int i = 0; i < n; i++) cin >> arr2[i];

    int result[min(m, n)], resSize;

    Commonelements(arr1, m, arr2, n, result, resSize);

    cout << "Common elements stored in new array: ";

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

        cout << result[i] << " ";

    }

    cout << endl;

    return 0;

}

**Output**

