

1. Given an array of strings words, return the first palindromic string in the array. If there is no such string, return an empty string "". A string is palindromic if it reads the same forward and backward. aim algorithm.

Aim

To find and return the first palindromic string from a given array of strings. If no palindrome exists, return an empty string.

Algorithm

1. Start
2. Input an array of strings words[].
3. For each string word in words[]:
 - a. Check if word is a palindrome:
 - b. If they are equal, return this string (it is the first palindrome).
4. If no palindrome is found after checking all strings, return "".
5. End

The screenshot shows a C++ IDE with the following code in `Untitled2.cpp`:

```
1 #include <stdio.h>
2 #include <string.h>
3
4 // Function to check if a string is palindrome
5 int isPalindrome(char str[]) {
6     int left = 0, right = strlen(str) - 1;
7     while (left < right) {
8         if (str[left] != str[right])
9             return 0; // Not a palindrome
10        left++;
11        right--;
12    }
13    return 1; // Palindrome
14 }
15
16 int main() {
17     int n, i;
18     char words[50][50]; // Max 50 words, each max length 50
19
20     printf("Enter number of words: ");
21     scanf("%d", &n);
22
23     printf("Enter %d words:\n", n);
24     for (i = 0; i < n; i++) {
25         scanf("%s", words[i]);
26     }
27
28     // Find first palindromic string
```

The console output shows the program execution:

```
Enter number of words: 3
Enter 3 words:
abc racecar mom
First palindromic string: racecar

Process exited after 50.41 seconds with return value 0
Press any key to continue . . .
```

The IDE status bar at the bottom indicates: Line: 38 Col: 2 Sel: 0 Lines: 39 Length: 914 Insert Done parsing in 0.047 seconds.

2. You are given two integer arrays nums1 and nums2 of sizes n and m, respectively. Calculate the following values: answer1 : the number of indices i such that nums1[i] exists in nums2. answer2 : the number of indices i such that nums2[i] exists in nums1. Return [answer1,answer2].

Aim

To find:

- answer1: the count of indices i where nums1[i] exists in nums2.
- answer2: the count of indices i where nums2[i] exists in nums1.

Return [answer1, answer2].

Algorithm

1. Start
2. Input two arrays nums1 (size n) and nums2 (size m).
3. Initialize answer1 = 0, answer2 = 0.
4. For each element in nums1:
 - Check if it exists in nums2.
 - If yes, increment answer1.
5. For each element in nums2:
 - Check if it exists in nums1.
 - If yes, increment answer2.
6. Print [answer1, answer2].
7. End

The screenshot shows a C++ IDE with the following code in `Untitled2.cpp`:

```
1 #include <stdio.h>
2
3 int main() {
4     int n, m, i, j, answer1 = 0, answer2 = 0;
5
6     printf("Enter size of nums1: ");
7     scanf("%d", &n);
8     int nums1[n];
9     printf("Enter %d elements of nums1:\n", n);
10    for (i = 0; i < n; i++) scanf("%d", &nums1[i]);
11
12    printf("Enter size of nums2: ");
13    scanf("%d", &m);
14    int nums2[m];
15    printf("Enter %d elements of nums2:\n", m);
16    for (i = 0; i < m; i++) scanf("%d", &nums2[i]);
17
18    // Calculate answer1
19    for (i = 0; i < n; i++) {
20        for (j = 0; j < m; j++) {
21            if (nums1[i] == nums2[j]) {
22                answer1++;
23                break; // avoid double counting same index i
24            }
25        }
26    }
27
28    // Calculate answer2
```

The execution output in the console window is as follows:

```
Enter size of nums1: 3
Enter 3 elements of nums1:
2 3 2
Enter size of nums2: 2
Enter 2 elements of nums2:
1 2
[2, 1]

Process exited after 19.24 seconds with return value 0
Press any key to continue . . .
```

3. You are given a 0-indexed integer array `nums`. The distinct count of a subarray of `nums` is defined as: Let `nums[i..j]` be a subarray of `nums` consisting of all the indices from `i` to `j` such that $0 \leq i \leq j < \text{nums.length}$. Then the number of distinct values in `nums[i..j]` is called the distinct count of `nums[i..j]`. Return the sum of the squares of distinct counts of all subarrays of `nums`. A subarray is a contiguous non-empty sequence of elements within an array.

Aim

To calculate the sum of squares of the number of distinct elements in all subarrays of a given array.

Algorithm

1. Start.
2. Input array `nums` of size `n`.
3. Initialize `sum = 0`.
4. For each starting index `i` from 0 to `n-1`:
5. Print `sum`.
6. End.

```

1 #include <stdio.h>
2
3 int main() {
4     int n, i, j, k;
5     printf("Enter size of array: ");
6     scanf("%d", &n);
7
8     int nums[n];
9     printf("Enter %d elements:\n", n);
10    for (i = 0; i < n; i++) {
11        scanf("%d", &nums[i]);
12    }
13
14    long long sum = 0; // result can be large
15
16    // Loop over all subarrays
17    for (i = 0; i < n; i++) {
18        int freq[1000] = {0}; // assume values are <= 999
19        int distinct = 0;
20
21        for (j = i; j < n; j++) {
22            if (freq[nums[j]] == 0) {
23                distinct++;
24            }
25            freq[nums[j]]++;
26
27            sum += (distinct * distinct);
28        }
29    }
30
31    printf("Sum of squares of distinct counts = %lld\n", sum);
32    return 0;
33 }

```

Compiler: TDM-GCC 9.2.0 64-bit Release

Console Output:

```

D:\Untitled2.exe
Enter size of array: 3
Enter 3 elements:
1 2 1
Sum of squares of distinct counts = 15

Process exited after 47.85 seconds with return value 0
Press any key to continue . . .

```

Compiler Output:

```

- Errors: 0
- Warnings: 0
- Output Filename: D:\Untitled2.exe
- Output Size: 323.2841796875 KiB
- Compilation Time: 0.41s

```

Line: 34 Col: 1 Sel: 0 Lines: 34 Length: 780 Insert Done parsing in 0.031 seconds

4. Given a 0-indexed integer array **nums** of length **n** and an integer **k**, return *the number of pairs (i, j) where $0 \leq i < j < n$, such that $\text{nums}[i] == \text{nums}[j]$ and $(i * j)$ is divisible by k .*

Aim

To count the number of pairs (i, j) in an array **nums** such that:

- $0 \leq i < j < n$
- $\text{nums}[i] == \text{nums}[j]$
- $(i * j)$ is divisible by k .

Algorithm

1. Start.
2. Input array **nums** of size **n** and integer **k**.
3. Initialize **count = 0**.
4. For each pair (i, j) where $0 \leq i < j < n$:
 - If $\text{nums}[i] == \text{nums}[j]$ and $(i * j) \% k == 0$, then increment **count**.
5. Print **count**.
6. End.

The screenshot displays a C++ IDE with the following components:

- Source File (Untitled2.cpp):**

```
3 int main() {
4     int n, k, i, j, count = 0;
5
6     printf("Enter size of array: ");
7     scanf("%d", &n);
8
9     int nums[n];
10    printf("Enter %d elements:\n", n);
11    for (i = 0; i < n; i++) {
12        scanf("%d", &nums[i]);
13    }
14
15    printf("Enter value of k: ");
16    scanf("%d", &k);
17
18    // Check all pairs
19    for (i = 0; i < n; i++) {
20        for (j = i + 1; j < n; j++) {
21            if (nums[i] == nums[j] && ((i * j) % k == 0)) {
22                count++;
23            }
24        }
25    }
26
27    printf("Number of valid pairs = %d\n", count);
28    return 0;
29 }
30
```
- Compiler Output:**
 - Errors: 0
 - Warnings: 0
 - Output Filename: D:\Untitled2.exe
 - Output Size: 323,284,796,875 KiB
 - Compilation Time: 0.36s
- Console Output (D:\Untitled2.exe):**

```
Enter size of array: 4
Enter 4 elements:
3 1 2 2
Enter value of k: 2
Number of valid pairs = 1

-----
Process exited after 33.37 seconds with return value 0
Press any key to continue . . .
```

5. Write a program FOR THE BELOW TEST CASES with least time complexity
Test Cases: -

Input: {1, 2, 3, 4, 5} Expected Output: 5

Input: {7, 7, 7, 7, 7} Expected Output: 7

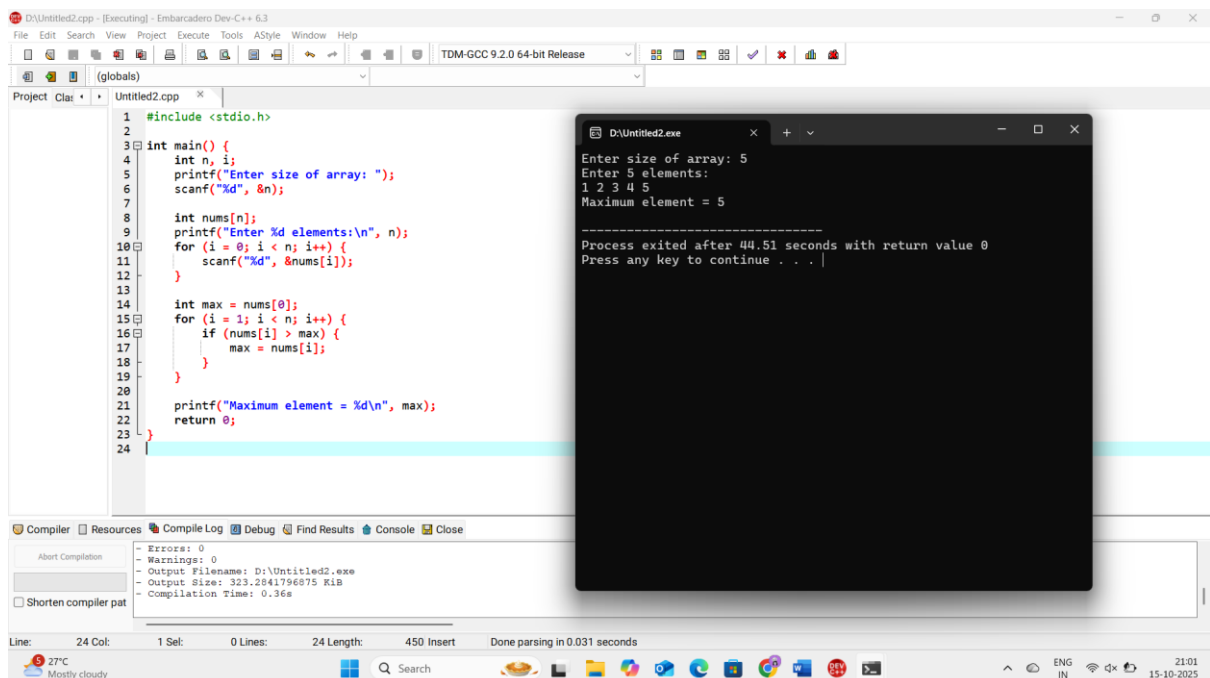
Input: {-10, 2, 3, -4, 5} Expected Output: 5

Aim

Write a program to find the maximum element of an array with least time complexity.

Algorithm

1. Start.
2. Input array of size n.
3. Initialize max = nums[0].
4. Traverse the array from index 1 to n-1:
 - If nums[i] > max, update max = nums[i].
5. Print max.
6. End.



```
1 #include <stdio.h>
2
3 int main() {
4     int n, i;
5     printf("Enter size of array: ");
6     scanf("%d", &n);
7
8     int nums[n];
9     printf("Enter %d elements:\n", n);
10    for (i = 0; i < n; i++) {
11        scanf("%d", &nums[i]);
12    }
13
14    int max = nums[0];
15    for (i = 1; i < n; i++) {
16        if (nums[i] > max) {
17            max = nums[i];
18        }
19    }
20
21    printf("Maximum element = %d\n", max);
22    return 0;
23 }
24
```

Enter size of array: 5
Enter 5 elements:
1 2 3 4 5
Maximum element = 5

Process exited after 44.51 seconds with return value 0
Press any key to continue . . .

6. You have an algorithm that process a list of numbers. It firsts sorts the list using an efficient sorting algorithm and then finds the maximum element in sorted list. Write the code for the same.

Aim

To design a program that first sorts a list of numbers using an efficient sorting algorithm (like QuickSort or MergeSort) and then finds the maximum element in the sorted list.

Algorithm

1. Start
2. Input the size n of the array and the n elements.
3. Sort the array using an efficient sorting algorithm (QuickSort or MergeSort).
4. After sorting, the array is in ascending order.
5. The maximum element will be at the last index (arr[n-1]).
6. Print arr[n-1].
7. End

The screenshot shows a C++ IDE with the following code in `Untitled2.cpp`:

```
1 #include <stdio.h>
2
3 // Function to swap two numbers
4 void swap(int *a, int *b) {
5     int temp = *a;
6     *a = *b;
7     *b = temp;
8 }
9
10 // Partition function for QuickSort
11 int partition(int arr[], int low, int high) {
12     int pivot = arr[high];
13     int i = low - 1;
14     for (int j = low; j < high; j++) {
15         if (arr[j] < pivot) {
16             i++;
17             swap(&arr[i], &arr[j]);
18         }
19     }
20     swap(&arr[i + 1], &arr[high]);
21     return i + 1;
22 }
23
24 // QuickSort function
25 void quickSort(int arr[], int low, int high) {
26     if (low < high) {
27         int pi = partition(arr, low, high);
28         quickSort(arr, low, pi - 1);
29     }
30 }
```

The console output shows the program execution:

```
Enter size of array: 5
Enter 5 elements:
1 2 3 4 5
Maximum element = 5

-----
Process exited after 23.77 seconds with return value 0
Press any key to continue . . .
```

The bottom status bar indicates: Line: 52 Col: 1 Sel: 0 Lines: 52 Length: 1115 Insert Done parsing in 0.031 seconds.

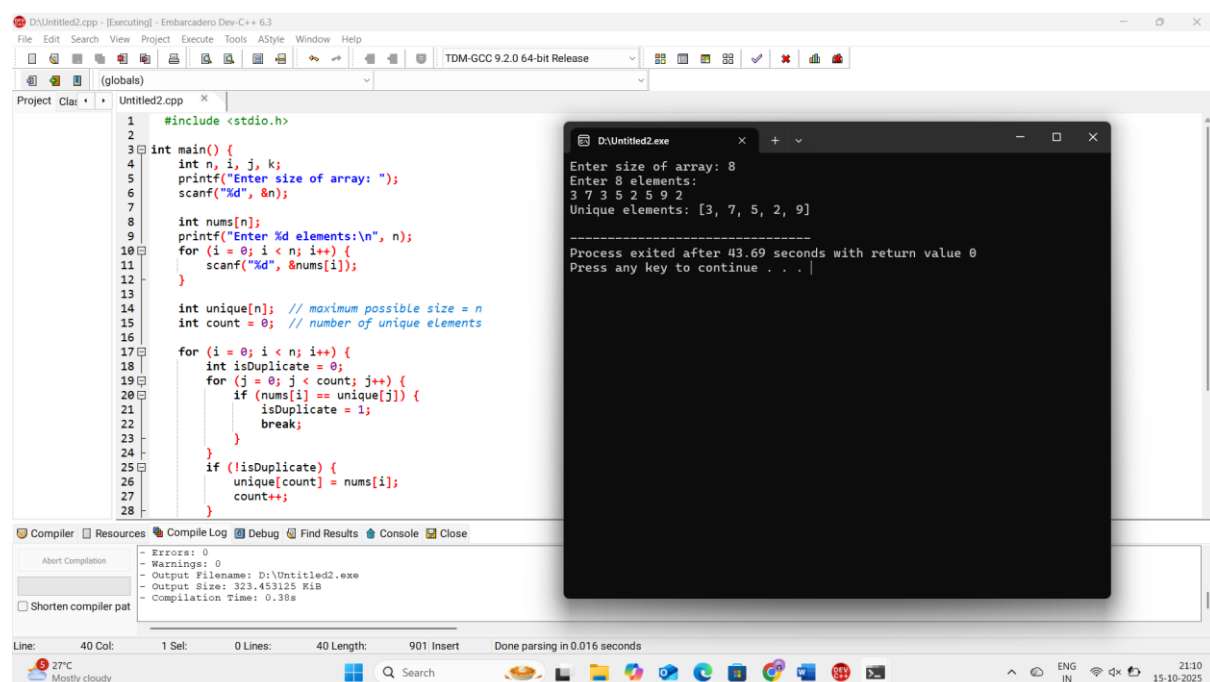
7. Write a program that takes an input list of n numbers and creates a new list containing only the unique elements from the original list. What is the space complexity of the algorithm?

Aim

To take an input list of n numbers and create a new list containing only the unique elements from the original list.

Algorithm

1. Start
2. Input the size n of the array and the array elements `nums[]`.
3. Create an empty array `unique[]` to store unique elements.
4. For each element in `nums[]`:
 - Check if it already exists in `unique[]`.
 - If not, add it to `unique[]`.
5. Print the `unique[]` array.
6. End



```
#include <stdio.h>

int main() {
    int n, i, j, k;
    printf("Enter size of array: ");
    scanf("%d", &n);

    int nums[n];
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &nums[i]);
    }

    int unique[n]; // maximum possible size = n
    int count = 0; // number of unique elements

    for (i = 0; i < n; i++) {
        int isDuplicate = 0;
        for (j = 0; j < count; j++) {
            if (nums[i] == unique[j]) {
                isDuplicate = 1;
                break;
            }
        }
        if (!isDuplicate) {
            unique[count] = nums[i];
            count++;
        }
    }

    printf("Unique elements: ");
    for (i = 0; i < count; i++) {
        printf("%d ", unique[i]);
    }
    printf("\n");
}
```

Enter size of array: 8
Enter 8 elements:
3 7 5 2 5 9 2
Unique elements: [3, 7, 5, 2, 9]

Process exited after 43.69 seconds with return value 0
Press any key to continue . . .

8. Sort an array of integers using the bubble sort technique. Analyze its time complexity using Big-O notation. Write the code

Aim

To sort an array of integers in ascending order using the Bubble Sort technique and analyze its time complexity.

Algorithm

1. Start
2. Input array arr[] of size n.
3. Repeat for i from 0 to n-2:
 - For j from 0 to n-i-2:
 - If $\text{arr}[j] > \text{arr}[j+1]$, swap $\text{arr}[j]$ and $\text{arr}[j+1]$.
4. After all iterations, the array is sorted in ascending order.
5. Print the sorted array.
6. End

The screenshot displays a C++ IDE with the following components:

- Source Code (Untitled2.cpp):**

```
1 #include <stdio.h>
2
3 int main() {
4     int n, i, j, temp;
5
6     printf("Enter size of array: ");
7     scanf("%d", &n);
8
9     int arr[n];
10    printf("Enter %d elements:\n", n);
11    for (i = 0; i < n; i++) {
12        scanf("%d", &arr[i]);
13    }
14
15    // Bubble Sort
16    for (i = 0; i < n - 1; i++) {
17        for (j = 0; j < n - i - 1; j++) {
18            if (arr[j] > arr[j + 1]) {
19                // Swap arr[j] and arr[j+1]
20                temp = arr[j];
21                arr[j] = arr[j + 1];
22                arr[j + 1] = temp;
23            }
24        }
25    }
26
27    printf("Sorted array: ");
28    for (i = 0; i < n; i++) {
29        printf("%d ", arr[i]);
30    }
31}
```
- Compiler Output:**
 - Errors: 0
 - Warnings: 0
 - Output Filename: D:\Untitled2.exe
 - Output Size: 323.4560546875 KiB
 - Compilation Time: 0.39s
- Execution Output (D:\Untitled2.exe):**

```
Enter size of array: 5
Enter 5 elements:
5 2 8 1 3
Sorted array: 1 2 3 5 8

-----
Process exited after 20.3 seconds with return value 0
Press any key to continue . . .
```


9. Checks if a given number x exists in a sorted array arr using binary search. Analyze its time complexity using Big-O notation.

Aim

To check if a given number key exists in a sorted array using Binary Search and analyze its time complexity.

Algorithm

1. Start
2. Input array $arr[]$ of size n and number key .
3. Sort the array in ascending order (use any efficient sorting like QuickSort or Bubble Sort).
4. Initialize $low = 0$ and $high = n - 1$.
5. Repeat while $low \leq high$:
6. If loop ends without finding key , element is not found.
7. End

The screenshot shows a C++ IDE with a source file named 'Untitled2.cpp' and an executable named 'D:\Untitled2.exe'. The source code implements a bubble sort and a binary search function. The binary search function takes an array, its size, and a key as input. It initializes low and high pointers and enters a while loop that continues as long as low is less than or equal to high. Inside the loop, it calculates the mid index and compares the element at mid with the key. If they are equal, it returns mid. If the element at mid is less than the key, it updates low to mid + 1. If the element at mid is greater than the key, it updates high to mid - 1. If the loop ends without finding the key, it returns -1. The executable window shows the program's output: 'Enter size of array: 8', 'Enter 8 elements: 3 4 6 -9 10 8 9 30', 'Enter key to search: 100', and 'Element 100 is not found'. It also shows the process exit time and return value.

```
1 #include <stdio.h>
2
3 // Swap function for sorting
4 void swap(int *a, int *b) {
5     int temp = *a;
6     *a = *b;
7     *b = temp;
8 }
9
10 // Simple Bubble Sort
11 void bubbleSort(int arr[], int n) {
12     for (int i = 0; i < n - 1; i++) {
13         for (int j = 0; j < n - i - 1; j++) {
14             if (arr[j] > arr[j + 1])
15                 swap(&arr[j], &arr[j + 1]);
16         }
17     }
18 }
19
20 // Binary Search Function
21 int binarySearch(int arr[], int n, int key) {
22     int low = 0, high = n - 1;
23     while (low <= high) {
24         int mid = (low + high) / 2;
25         if (arr[mid] == key)
26             return mid;
27         else if (arr[mid] < key)
28             low = mid + 1;
29     }
30     return -1;
31 }
```

Enter size of array: 8
Enter 8 elements:
3 4 6 -9 10 8 9 30
Enter key to search: 100
Element 100 is not found

Process exited after 35.88 seconds with return value 0
Press any key to continue . . .

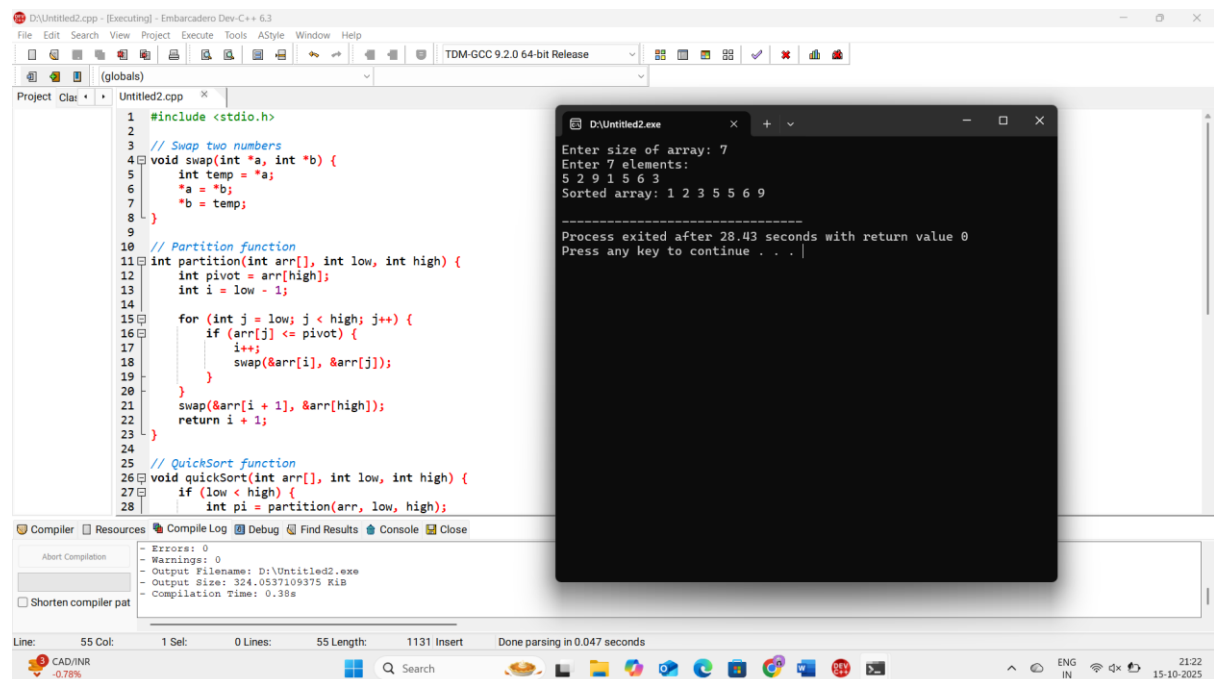
10. Given an array of integers nums, sort the array in ascending order and return it. You must solve the problem without using any built-in functions in $O(n \log(n))$ time complexity and with the smallest space complexity possible.

Aim

To sort an array of integers in ascending order using QuickSort in $O(n \log n)$ time without using built-in functions, with minimal space.

Algorithm

1. Start
2. Input array nums[] of size n.
3. Call QuickSort(nums, 0, n-1)
4. After recursion ends, array is sorted in ascending order.
5. Print the sorted array.
6. End



The screenshot displays an IDE window titled 'D:\Untitled2.cpp - [Executing] - Embarcadero Dev-C++ 6.3'. The code implements a QuickSort algorithm. The 'swap' function swaps two integers. The 'partition' function selects the last element as a pivot and rearranges the array. The 'quickSort' function recursively sorts the array. The output window shows the execution results.

```
1 #include <stdio.h>
2
3 // Swap two numbers
4 void swap(int *a, int *b) {
5     int temp = *a;
6     *a = *b;
7     *b = temp;
8 }
9
10 // Partition function
11 int partition(int arr[], int low, int high) {
12     int pivot = arr[high];
13     int i = low - 1;
14
15     for (int j = low; j < high; j++) {
16         if (arr[j] <= pivot) {
17             i++;
18             swap(&arr[i], &arr[j]);
19         }
20     }
21     swap(&arr[i + 1], &arr[high]);
22     return i + 1;
23 }
24
25 // QuickSort function
26 void quickSort(int arr[], int low, int high) {
27     if (low < high) {
28         int pi = partition(arr, low, high);
```

Output window (D:\Untitled2.exe):

```
Enter size of array: 7
Enter 7 elements:
5 2 9 1 5 6 3
Sorted array: 1 2 3 5 5 6 9

-----
Process exited after 28.43 seconds with return value 0
Press any key to continue . . .
```

Compiler output:

```
Errors: 0
Warnings: 0
Output Filename: D:\Untitled2.exe
Output Size: 324.0537109375 Kib
Compilation Time: 0.39s
```