

**SORTING**

**Sorting** is arranging data in a particular order, typically ascending or descending. This ordering can apply to numbers, letters, dates, or any data that can be systematically compared.

### Importance of Sorting

1. **Improves Data Accessibility**: Sorted data is easier to search, analyze, and interpret, making information retrieval faster and more efficient.
2. **Optimizes Algorithm Performance**: Many algorithms run faster and more effectively on sorted data, reducing computational time.
3. **Enhances User Experience**: Sorted lists help users quickly find the information they need in applications and websites, improving overall usability.
4. **Facilitates Data Organization**: Sorting provides structure, critical in fields like databases and data analysis, where order impacts how effectively data can be managed and processed.

**Four Sorting Algorithms**

The algorithms I will implement and compare in this assignment are

* Bubble Sort
* Selection Sort
* Merge Sort
* Quick Sort

**Bubble Sort**

Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is fully sorted.

### 

### How Bubble Sort Works

1. **Iterate through the list**: Start at the beginning of the list and compare each pair of adjacent elements.
2. **Swap elements if necessary**: If the current element is greater than the adjacent element, swap them. This pushes the larger element towards the end of the list.
3. **Repeat the process**: After the first pass, the largest element has “bubbled up” to its correct position at the end of the list. Repeat the process for the remaining unsorted elements.
4. **Optimization with flag**: An optimization can include a flag that checks if any swaps were made in the current pass. If no swaps are made, the list is already sorted, and the algorithm can stop early.

**Bubble Sort:**

Array

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 45 | 49 | 11 | 7 |

After Pass 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 45 | 49 | 11 | 7 |

After Pass 2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 49 | 11 | 45 |

After Pass 3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 11 | 49 | 45 |

After Pass 4:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 11 | 49 | 45 |

After Pass 4: Sorted Array

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 11 | 45 | 49 |

### Step-by-Step Bubble Sort Process

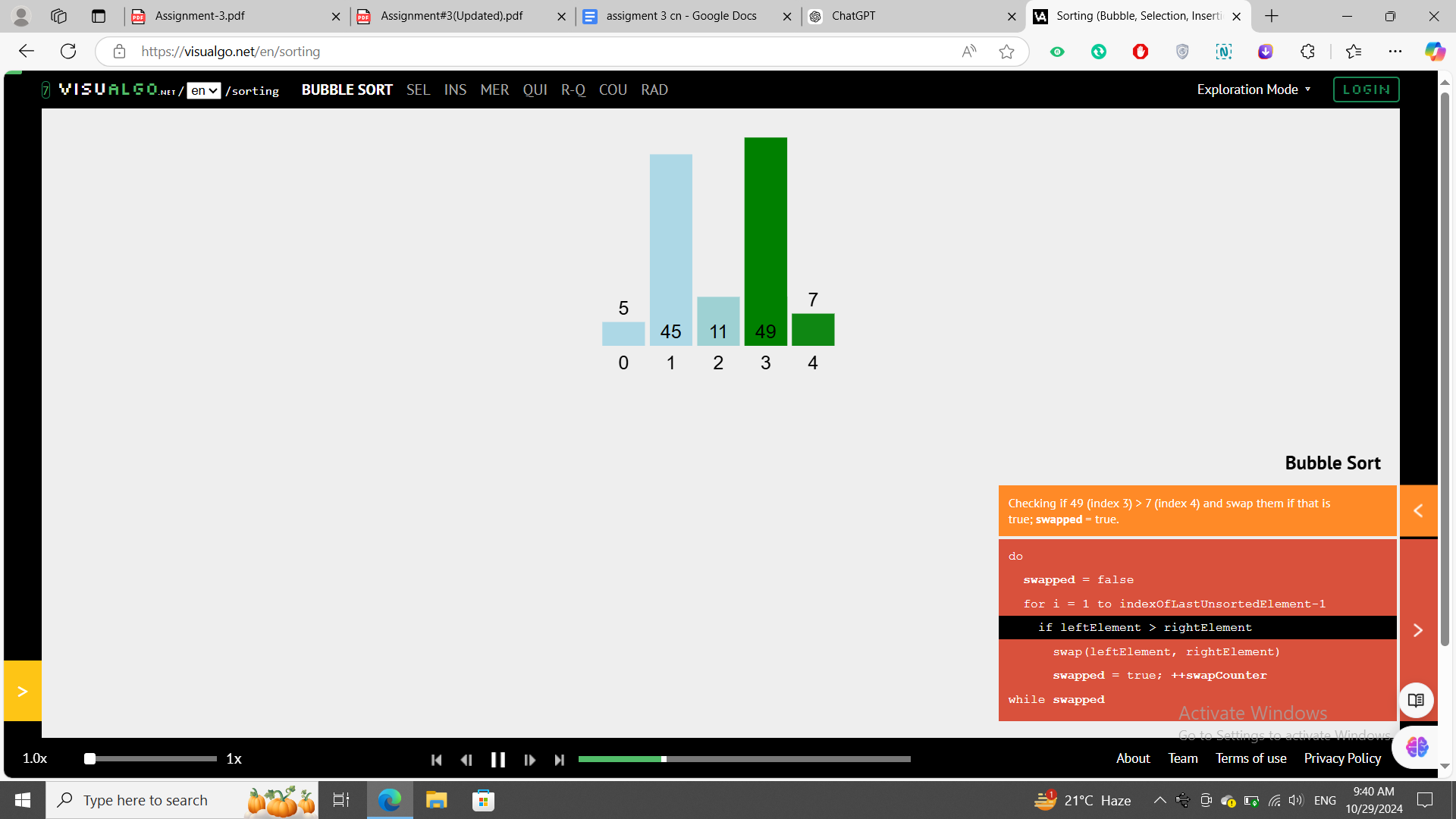
1. **First Pass**:
   * Compare **5** and **45**: No swap needed ➔ [5, 45, 49, 11, 7]
   * Compare **45** and **49**: No swap needed ➔ [5, 45, 49, 11, 7]
   * Compare **49** and **11**: Swap ➔ [5, 45, 11, 49, 7]
   * Compare **49** and **7**: Swap ➔ [5, 45, 11, 7, 49]
2. After the first pass, the largest element, **49**, has moved to the end of the array.
3. **Second Pass**:
   * Compare **5** and **45**: No swap needed ➔ [5, 45, 11, 7, 49]
   * Compare **45** and **11**: Swap ➔ [5, 11, 45, 7, 49]
   * Compare **45** and **7**: Swap ➔ [5, 11, 7, 45, 49]
4. Now, the second-largest element, **45**, has moved to its correct position.
5. **Third Pass**:
   * Compare **5** and **11**: No swap needed ➔ [5, 11, 7, 45, 49]
   * Compare **11** and **7**: Swap ➔ [5, 7, 11, 45, 49]
6. The third-largest element, **11**, is now in place.
7. **Fourth Pass**:
   * Compare **5** and **7**: No swap needed ➔ [5, 7, 11, 45, 49]
8. At this point, all elements are sorted in ascending order, so the algorithm stops.

### Final Sorted Array

[5, 7, 11, 45, 49]

### Visual Algo:

### Array=[5, 45, 49, 11, 7]

**Screenshot:  
**

### Bubble Sort Code:

#include <iostream>

#include <chrono> // For timing the execution

using namespace std;

using namespace std::chrono;

int main() {

int arr[] = {5, 45, 49, 11, 7};

int n = sizeof(arr) / sizeof(arr[0]);

// Start time

auto start = high\_resolution\_clock::now();

// Bubble Sort Algorithm

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap arr[j] and arr[j + 1]

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

// End time

auto end = high\_resolution\_clock::now();

// Calculate the duration of sorting

auto duration = duration\_cast<microseconds>(end - start);

// Output the sorted array

cout << "Sorted array: ";

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

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

}

cout << endl;

// Print the execution time

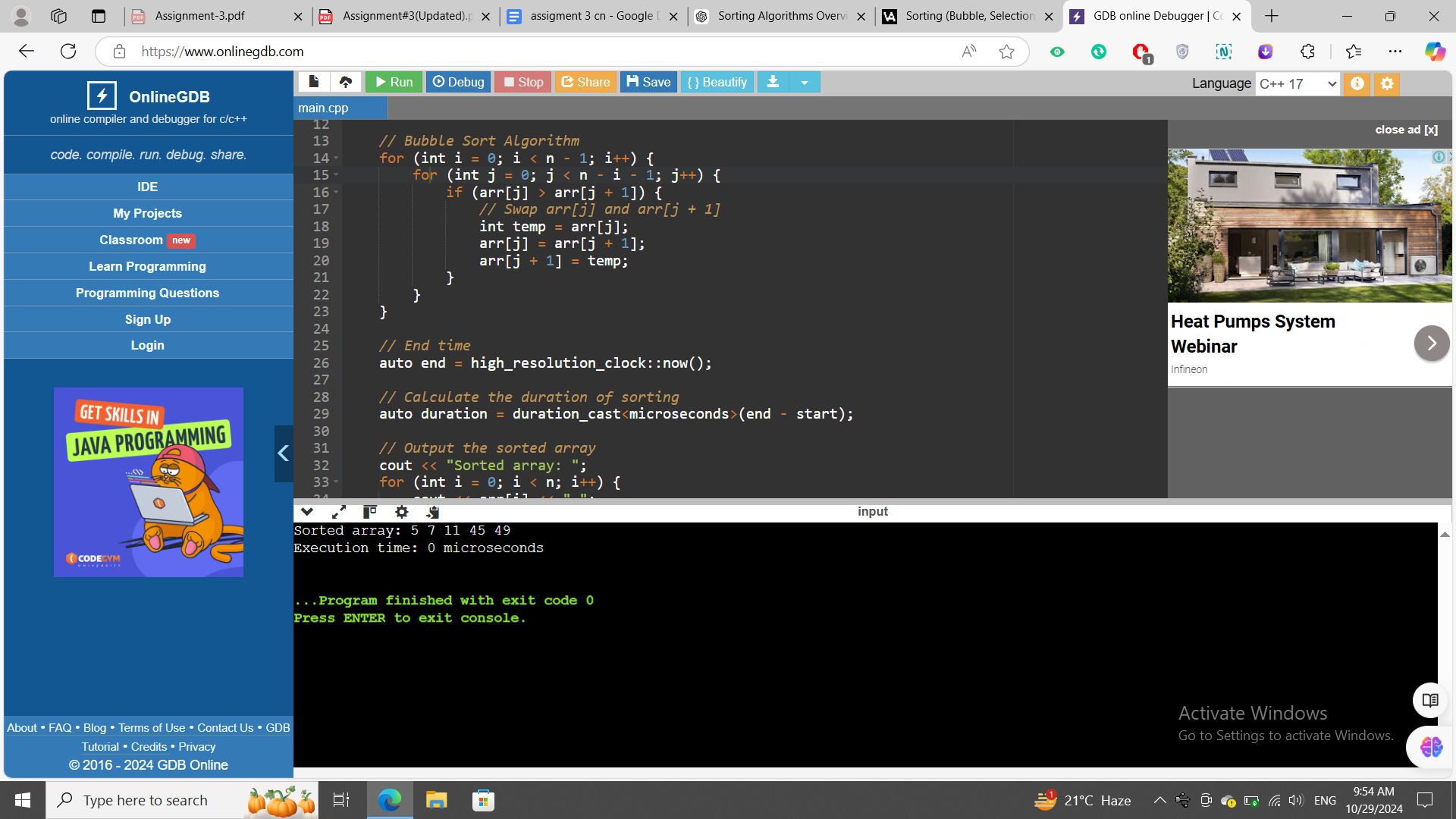
cout << "Execution time: " << duration.count() << " microseconds" << endl;

return 0;

}

**Compiler used:**

[**GDB online Debugger | Compiler - Code, Compile, Run, Debug online C, C++**](https://www.onlinegdb.com/)

**Running time screenshot:  
**

**Time complexity**

In the context of Bubble Sort, the best, worst, and average cases refer to the time complexity of the algorithm under different initial conditions of the input array.

### 1. Best Case

* Condition: The array is already sorted in ascending order.
* Time Complexity: O(n)
* Explanation: In the best case, Bubble Sort only needs one pass through the array to confirm that it is already sorted. The algorithm can use an optimization (a flag to track swaps) to stop early when no swaps are made in a pass, resulting in a linear time complexity.

### 2. Worst Case

* Condition: The array is sorted in descending order (completely reversed).
* Time Complexity: O(n²)
* Explanation: In the worst case, every pair of adjacent elements needs to be swapped in each pass. For an array of n elements, Bubble Sort will need n-1 passes, where each pass involves multiple swaps. This results in the maximum number of comparisons and swaps, leading to quadratic time complexity.

### 3. Average Case

* Condition: The array elements are in random order.
* Time Complexity: O(n²)
* Explanation: In the average case, Bubble Sort will require a number of comparisons and swaps that is roughly halfway between the best and worst cases. However, since the algorithm is still making multiple passes with comparisons and swaps, the time complexity remains O(n²).

### Summary of Time Complexities

* Best Case: O(n)
* Worst Case: O(n²)
* Average Case: O(n²)

### Space Complexity

* Space Complexity: O(1) (Bubble Sort is an in-place sorting algorithm, meaning it requires only a constant amount of additional memory for swapping).

**Selection Sort**

Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly finding the minimum (or maximum, for descending order) element from the unsorted portion of the array and placing it in its correct position in the sorted portion.

### How Selection Sort Works

1. Iterate through the list: Start from the beginning and find the minimum element from the unsorted part of the array.
2. Swap elements: Swap the minimum element found with the first unsorted element. This places it in its correct position.
3. Repeat the process: Move to the next unsorted element, treating it as the new starting point, and repeat until the entire list is sorted.

Array

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 45 | 49 | 11 | 7 |

After Pass 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 45 | 49 | 11 | 7 |

After Pass 2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 49 | 11 | 45 |

After Pass 3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 11 | 49 | 45 |

After Pass 4:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 11 | 49 | 45 |

After Pass 4: Sorted Array

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 11 | 45 | 49 |

### 

### Step-by-Step Selection Sort Process

#### First Pass:

* Unsorted array: **[5, 45, 49, 11, 7]**
* Find the minimum element (5) in the unsorted part, which is already in the correct place.
* After Pass 1: **[5, 45, 49, 11, 7]**

#### Second Pass:

* Unsorted array: **[45, 49, 11, 7]**
* Find the minimum element (7) and swap it with the first element of the unsorted part.
* After Pass 2: **[5, 7, 49, 11, 45]**

#### Third Pass:

* Unsorted array: **[49, 11, 45]**
* Find the minimum element (11) and swap it with the first element of the unsorted part.
* After Pass 3**: [5, 7, 11, 49, 45]**

#### Fourth Pass:

Unsorted array**: [49, 45]**

Find the minimum element (45) and swap it with the first element of the unsorted part.

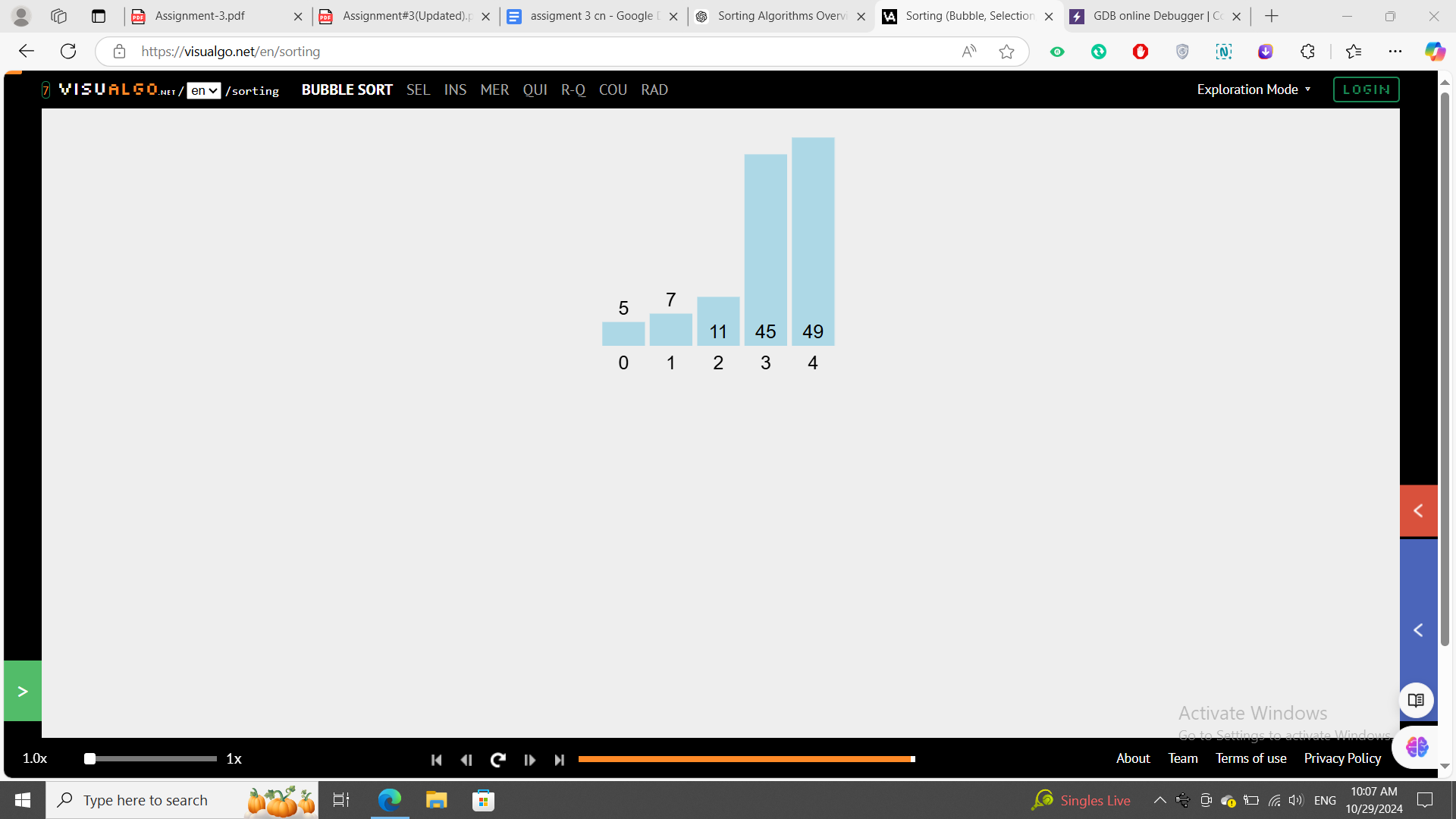
* After Pass 4: **[5, 7, 11, 45, 49]**

### Final Sorted Array

**[5, 7, 11, 45, 49]**

### Visual Algo:

### Array=[5, 45, 49, 11, 7]

**Screenshot:  
**

### Selection Sort Code:

#include <iostream>

#include <chrono> // For timing the execution

using namespace std;

using namespace std::chrono;

int main() {

int arr[] = {5, 45, 49, 11, 7};

int n = sizeof(arr) / sizeof(arr[0]);

// Start time

auto start = high\_resolution\_clock::now();

// Bubble Sort Algorithm

for (int i = 0; i < n - 1; i++) {

bool swapped = false; // Optimization flag

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap arr[j] and arr[j + 1]

int temp = arr[j];

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

arr[j + 1] = temp;

swapped = true; // A swap occurred

}

}

// If no swaps occurred in this pass, array is already sorted

if (!swapped) break;

}

// End time

auto end = high\_resolution\_clock::now();

// Calculate the duration of sorting

auto duration = duration\_cast<microseconds>(end - start);

// Output the sorted array

cout << "Sorted array: ";

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

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

}

cout << endl;

// Print the execution time

cout << "Execution time: " << duration.count() << " microseconds" << endl;

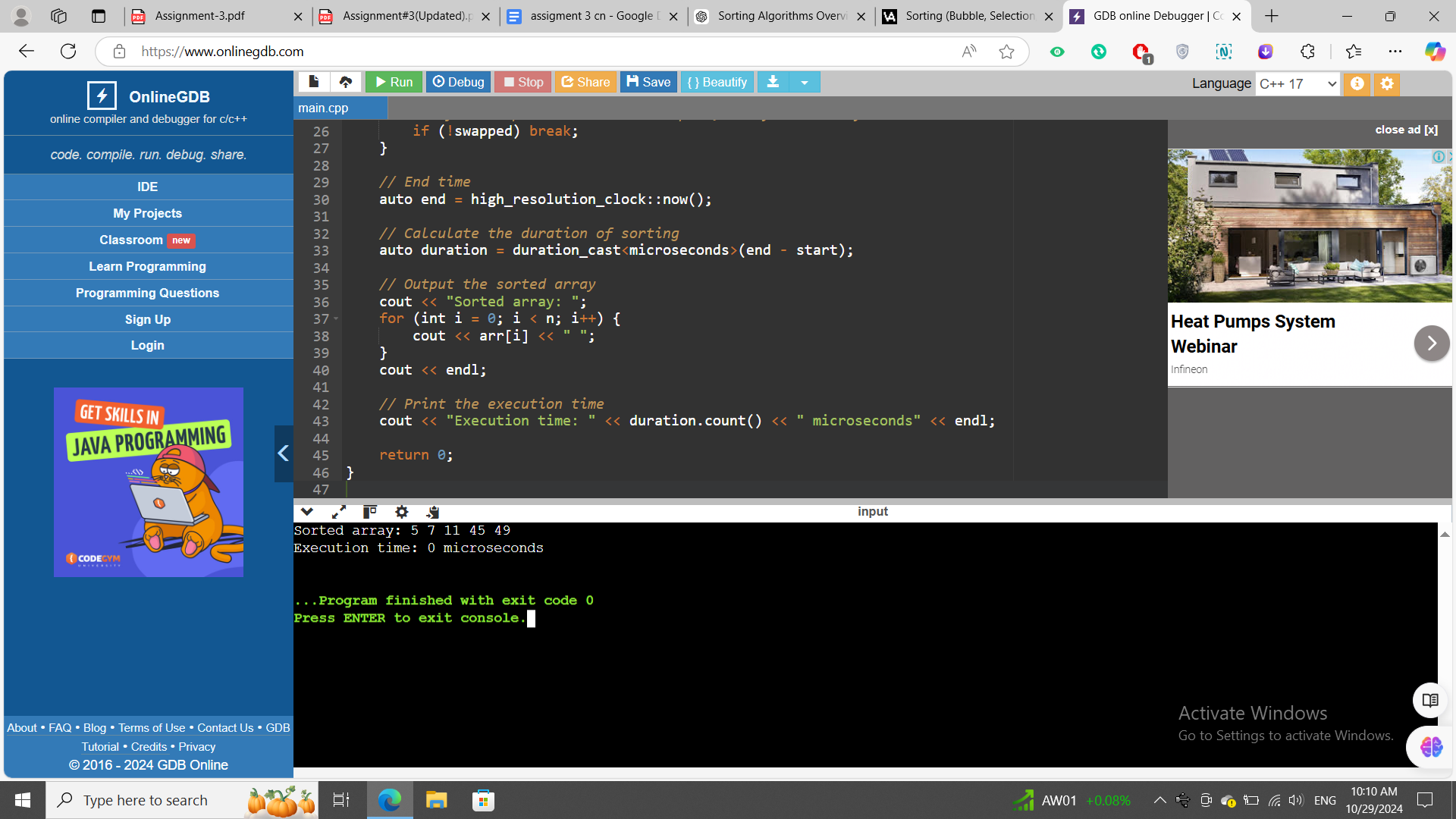
return 0;

}

**Compiler used:**

[**GDB online Debugger | Compiler - Code, Compile, Run, Debug online C, C++**](https://www.onlinegdb.com/)

**Running time screenshot:**

****

**Time complexity:**Same as Bubble sort.

**Merge Sort**

Mergesort algorithm is one of two important divide-and-conquer sorting algorithms (the other one is quicksort).

### How Selection Sort Works

* It is a recursive algorithm
* Divides the list into halves
* Sort each halve separately
* Then merge the sorted halves into one sorted array

Array

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 45 | 49 | 11 | 7 |

divide array  
1:

|  |  |  |
| --- | --- | --- |
| 5 | 45 | 49 |

2:

|  |  |
| --- | --- |
| 11 | 7 |

**This’s already sorted:**

|  |  |  |
| --- | --- | --- |
| 5 | 45 | 49 |

it’s not sorted so first we sort it

|  |  |
| --- | --- |
| 11 | 7 |

|  |  |
| --- | --- |
| 7 | 11 |

Now merge it

Sorted Array

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 11 | 45 | 49 |

### 

### Step-by-Step Merge Sort Process

#### Step 1: Divide the Array

* Initial Array: [5, 45, 49, 11, 7]
* Split the array into two halves:
  + Left: [5, 45]
  + Right: [49, 11, 7]

#### Step 2: Further Divide the Left Half

* Left: [5, 45]
  + Split into [5] and [45]

#### Step 3: Further Divide the Right Half

* Right: [49, 11, 7]
  + Split into [49] and [11, 7]

#### Step 4: Split [11, 7]

* Split [11, 7] into [11] and [7]

#### Step 5: Merge [11] and [7]

* Compare and merge:
  + Compare 11 and 7:
    - Since 7 < 11, merge them: [7, 11]

#### Step 6: Merge [49] and [7, 11]

* Compare and merge:
  + Compare 49 and 7:
    - 7 < 49, so place 7 first.
  + Compare 49 and 11:
    - 11 < 49, so place 11 next.
  + Place 49 last.
* Resulting in: [7, 11, 49]

#### Step 7: Merge [5] and [45]

* Compare and merge:
  + Since 5 < 45, result: [5, 45]

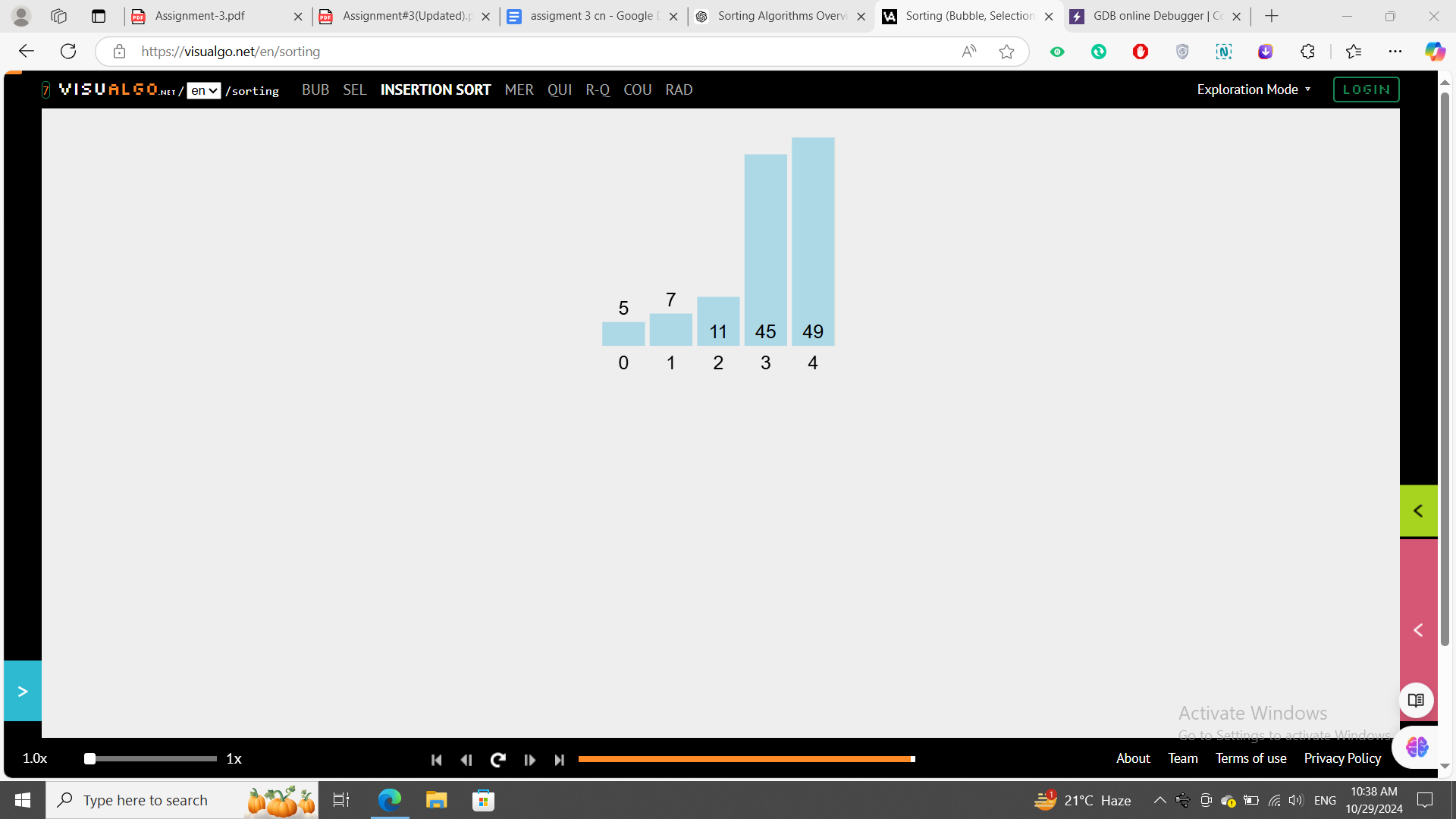
#### Step 8: Merge [5, 45] and [7, 11, 49]

* Compare and merge:
  + Compare 5 and 7:
    - 5 < 7, place 5 first.
  + Compare 45 and 7:
    - 7 < 45, place 7 next.
  + Compare 45 and 11:
    - 11 < 45, place 11 next.
  + Place 49 after 11.
  + Finally, place 45 last.

### Final Sorted Array [5, 7, 11, 45, 49]

### Visual Algo:

### Array=[5, 45, 49, 11, 7]

**Screenshot:  
**

### Merge Sort Code:

#include <iostream>

#include <chrono> // For timing the execution

using namespace std;

using namespace std::chrono;

// Function to merge two halves

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

int n1 = mid - left + 1; // Size of left subarray

int n2 = right - mid; // Size of right subarray

// Create temporary arrays

int\* L = new int[n1];

int\* R = new int[n2];

// Copy data to temporary arrays

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];

// Merge the temporary arrays back into arr[left..right]

int i = 0; // Initial index of the first subarray

int j = 0; // Initial index of the second subarray

int k = left; // Initial index of merged subarray

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy remaining elements of L[], if any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy remaining elements of R[], if any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

// Free the temporary arrays

delete[] L;

delete[] R;

}

// Function to implement merge sort

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

if (left < right) {

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

// Sort first and second halves

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

// Merge the sorted halves

merge(arr, left, mid, right);

}

}

int main() {

int arr[] = {5, 45, 49, 11, 7};

int n = sizeof(arr) / sizeof(arr[0]);

// Start time

auto start = high\_resolution\_clock::now();

// Perform merge sort

mergeSort(arr, 0, n - 1);

// End time

auto end = high\_resolution\_clock::now();

// Calculate the duration of sorting

auto duration = duration\_cast<microseconds>(end - start);

// Output the sorted array

cout << "Sorted array: ";

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

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

}

cout << endl;

// Print the execution time

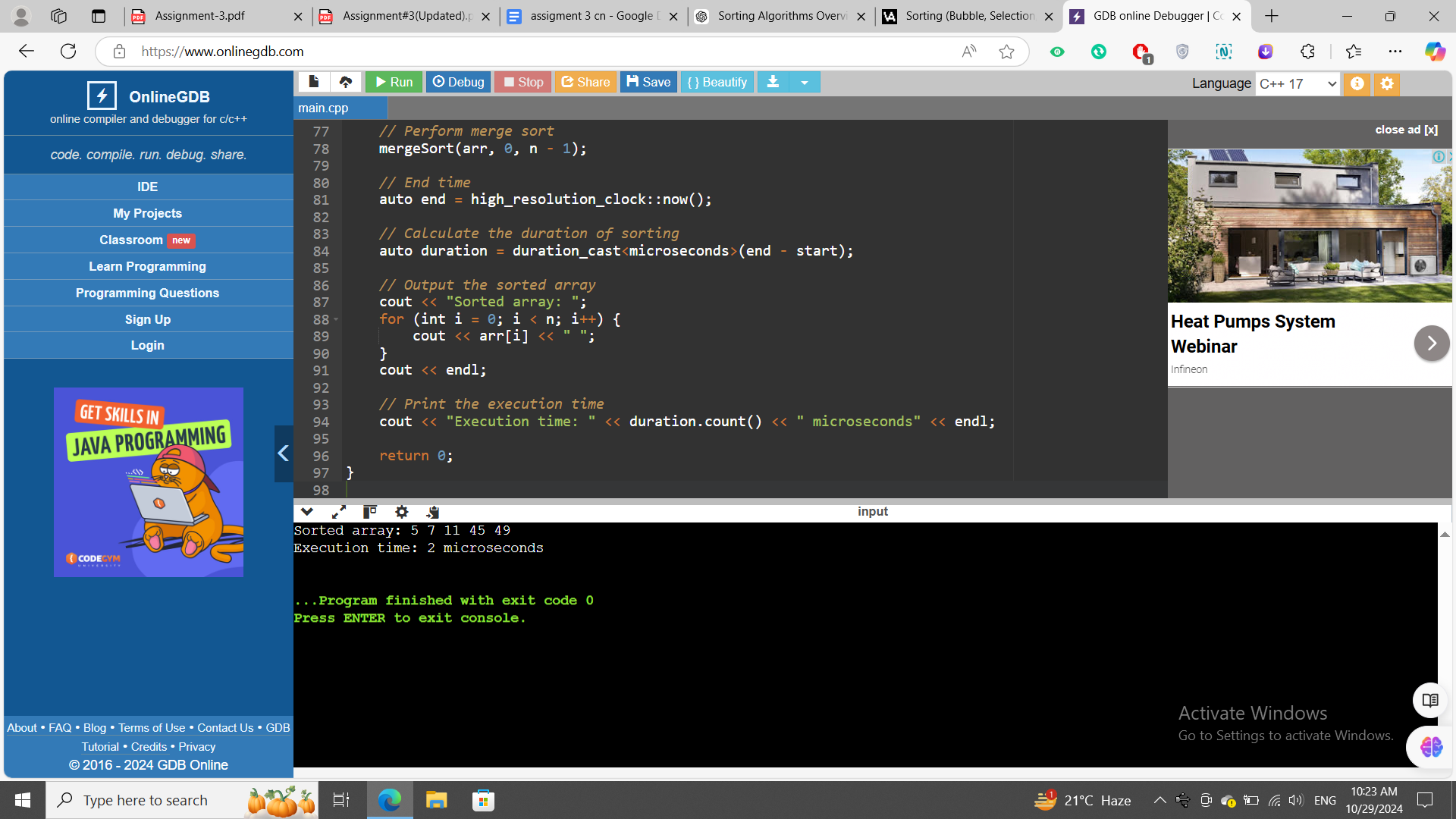
cout << "Execution time: " << duration.count() << " microseconds" << endl;

return 0;

}

**Compiler used:**

[**GDB online Debugger | Compiler - Code, Compile, Run, Debug online C, C++**](https://www.onlinegdb.com/)

**Running time screenshot:  
**

**Time complexity**

#### 1. Best Case: O(n log n)

* Condition: The array is already sorted.
* Explanation: Even if the array is sorted, Merge Sort will still divide the array and merge the subarrays, resulting in the same number of operations.

#### 2. Worst Case: O(n log n)

* Condition: The array is in reverse order.
* Explanation: Similar to the best case, regardless of the initial order of elements, Merge Sort performs the same sequence of operations, leading to a logarithmic number of splits (log n) and linear merging operations (n).

#### 3. Average Case: O(n log n)

* Condition: The array is in random order.
* Explanation: The average scenario is similar to both the best and worst cases. The array will still be divided log n times, and merging will always require O(n) time. Thus, the average time complexity remains O(n log n).

**Quick Sort**

Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is fully sorted.

### 

### How Quick Sort Works:

### Choose a Pivot: Select an element from the array to serve as the pivot. Various strategies exist for choosing the pivot (e.g., the first element, the last element, or the median).

### Partitioning: Rearrange the array so that all elements with values less than the pivot come before it, and all elements with values greater than the pivot come after it. The pivot is now in its final position.

### Recursively Apply: Recursively apply the above steps to the subarrays of elements with smaller and larger values.

### 

**Quick Sort:**

Array

pivot

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 45 | 49 | 11 | 7 |

pivot S1 S2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 45 | 49 | 11 | 7 |

pivot S1 S2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 49 | 11 | 45 |

pivot S1 S2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 49 | 11 | 45 |

pivot S1 S2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 11 | 49 | 45 |

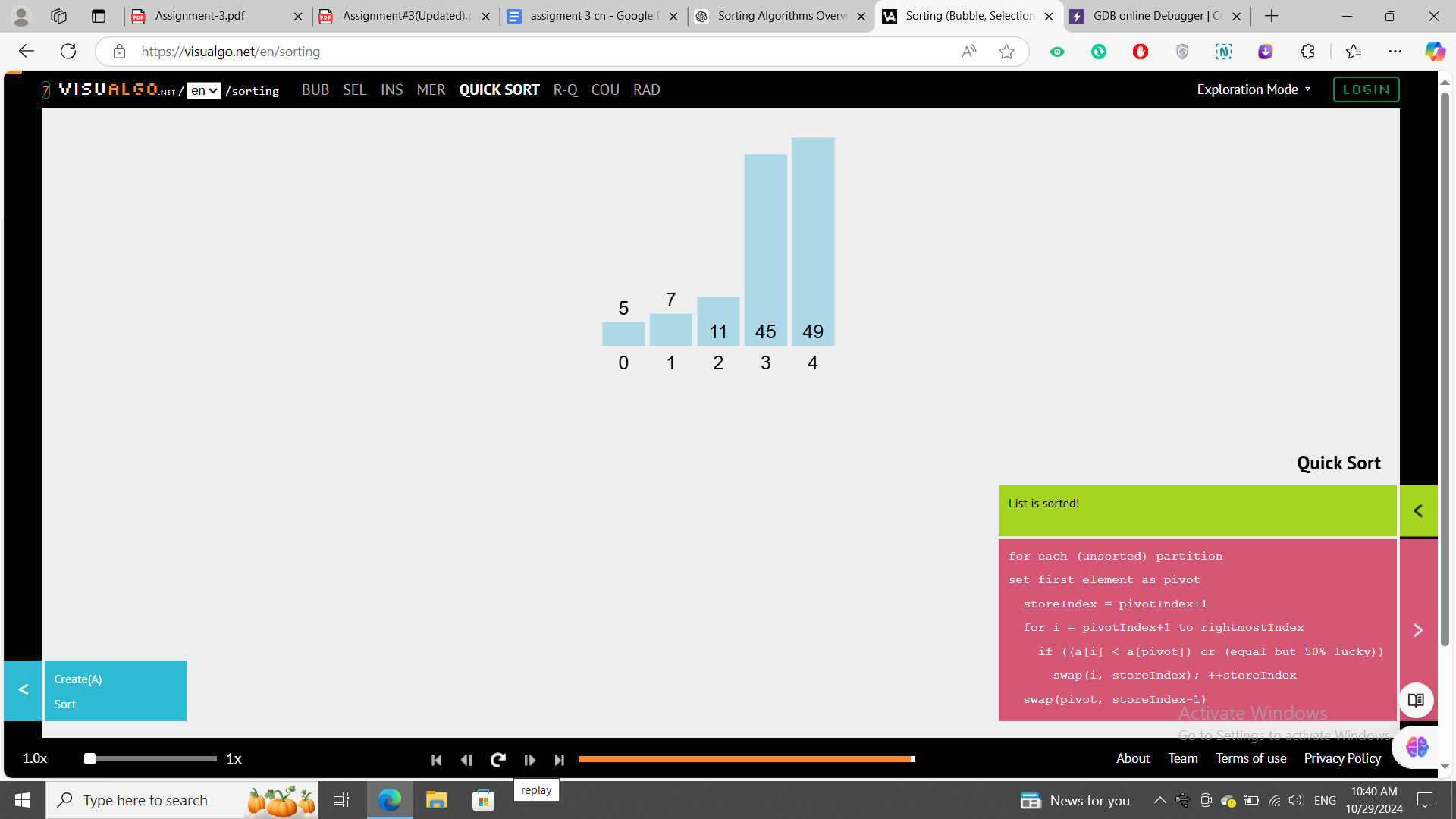
pivot S1 S2 S3 S4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 7 | 11 | 45 | 49 |

### 

### Visual Algo:

### Array=[5, 45, 49, 11, 7]

**Screenshot:  
**

### Qucik Sort Code:

#include <iostream>

#include <chrono>

using namespace std;

using namespace std::chrono;

// Function to partition the array

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

int pivot = arr[high]; // Choosing the last element as pivot

int i = (low - 1); // Index of smaller element

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

// If current element is smaller than or equal to pivot

if (arr[j] <= pivot) {

i++; // Increment index of smaller element

swap(arr[i], arr[j]); // Swap

}

}

swap(arr[i + 1], arr[high]); // Swap the pivot element with the element at i + 1

return (i + 1); // Return the partitioning index

}

// Function to implement Quick Sort

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

if (low < high) {

// Partition the array

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

// Recursively sort elements before and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int arr[] = {5, 45, 49, 11, 7}; // Sample array

int n = sizeof(arr) / sizeof(arr[0]);

// Start time measurement

auto start = high\_resolution\_clock::now();

// Perform Quick Sort

quickSort(arr, 0, n - 1);

// End time measurement

auto end = high\_resolution\_clock::now();

// Calculate the duration of sorting

auto duration = duration\_cast<microseconds>(end - start);

// Output the sorted array

cout << "Sorted Array: ";

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

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

}

cout << endl;

// Print the execution time

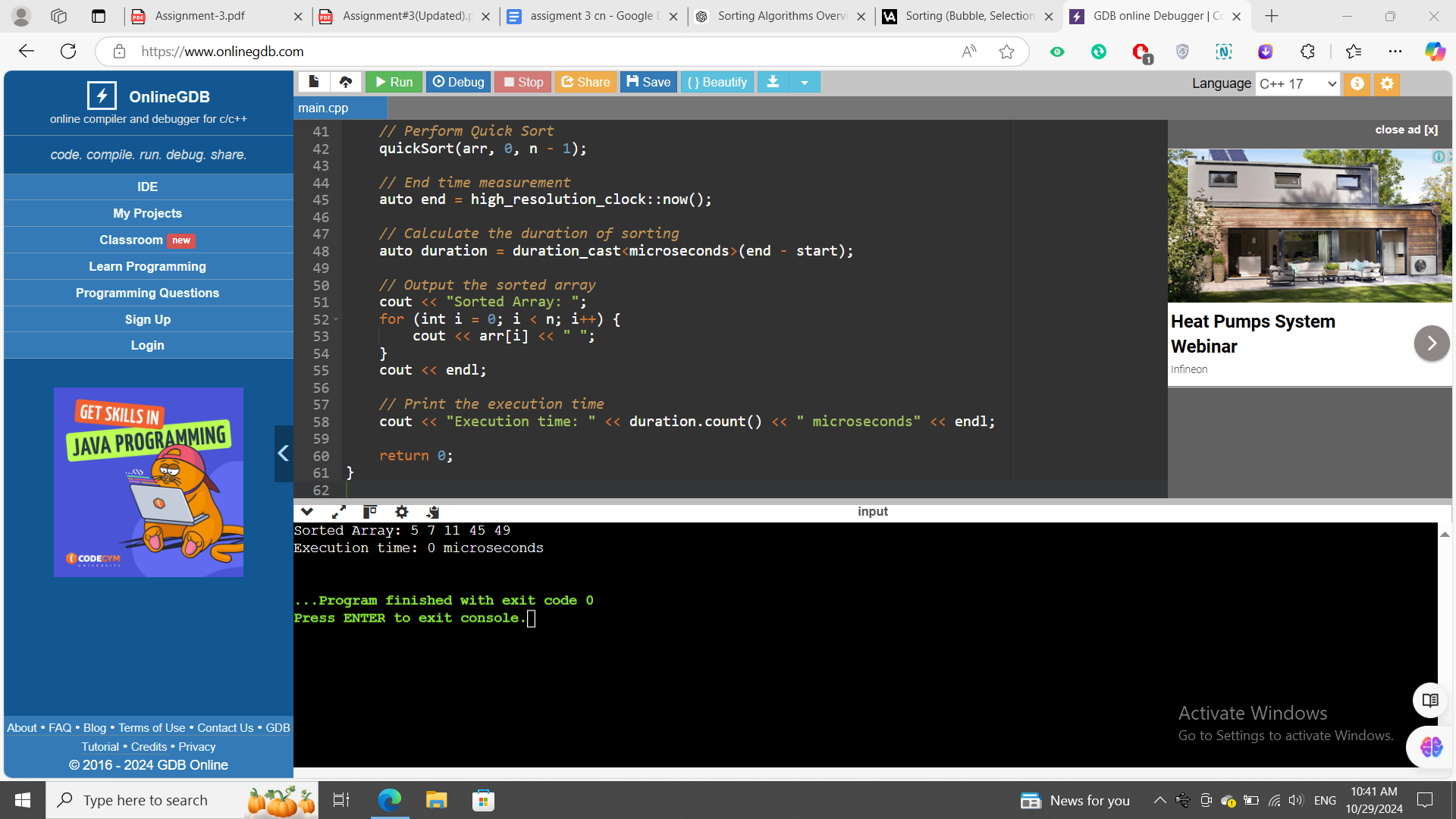
cout << "Execution time: " << duration.count() << " microseconds" << endl;

return 0;

}

**Compiler used:**

[**GDB online Debugger | Compiler - Code, Compile, Run, Debug online C, C++**](https://www.onlinegdb.com/)

**Running time screenshot:  
**

**Time complexity**

In the context of Bubble Sort, the best, worst, and average cases refer to the time complexity of the algorithm under different initial conditions of the input array.

### 1. Best Case

* Condition: The array is already sorted in ascending order.
* Time Complexity: O(n)
* Explanation: In the best case, Bubble Sort only needs one pass through the array to confirm that it is already sorted. The algorithm can use an optimization (a flag to track swaps) to stop early when no swaps are made in a pass, resulting in a linear time complexity.

### 2. Worst Case

* Condition: The array is sorted in descending order (completely reversed).
* Time Complexity: O(n²)
* Explanation: In the worst case, every pair of adjacent elements needs to be swapped in each pass. For an array of n elements, Bubble Sort will need n-1 passes, where each pass involves multiple swaps. This results in the maximum number of comparisons and swaps, leading to quadratic time complexity.

### 3. Average Case

* Condition: The array elements are in random order.
* Time Complexity: O(n²)
* Explanation: In the average case, Bubble Sort will require a number of comparisons and swaps that is roughly halfway between the best and worst cases. However, since the algorithm is still making multiple passes with comparisons and swaps, the time complexity remains O(n²).

### Summary of Time Complexities

* Best Case: O(n)
* Worst Case: O(n²)
* Average Case: O(n²)

### Space Complexity

* Space Complexity: O(1) (Bubble Sort is an in-place sorting algorithm, meaning it requires only a constant amount of additional memory for swapping).