# Department of Computing

# School of Electrical Engineering and Computer Science

**CS-250: Data Structure and Algorithms**

**Class: BESE** **13A**

**Lab 8: Quick Sort**

**Date: 18th November, 2023**

**Time: 10:00 am – 12:50 am**

**Name:** Aimen Munawar

**Class:** BESE-13-A

**CMS ID:** 415867

# Lab Engineer: Anum Asif

# Lab 8: Sorting Algorithms

**Introduction**

In this lab, you will implement Merge sort & Quick Sort.

**Objectives**

Objective of this lab is to implement merge sort and quick sort then compare the running times for sorting.

**Tools/Software Requirement**

Visual Studio C++

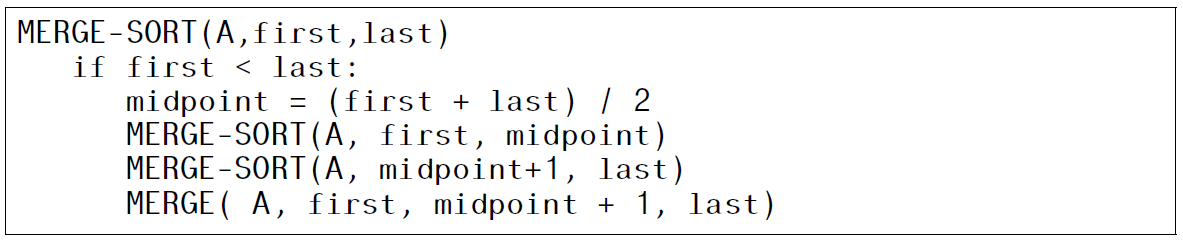
**Helping Material**

Lecture slides, text book

**Description**

**Merge Sort:-**

Merge sort is another important sorting algorithm that we have seen. Unlike insertion sort, it is not an in-place sorting algorithm. The pseudo code for merge sort is shown below:



Merge (Arr, n1, mid, n2)

a=n1, b=mid, c=n1 ,B;

while a <= mid and b<=n2

if Arr[a]<Arr[b]

B[c++]=Arr[a++];

else

B[c++]=Arr[b++];.

while a<mid

B[c++]=Arr[a++];

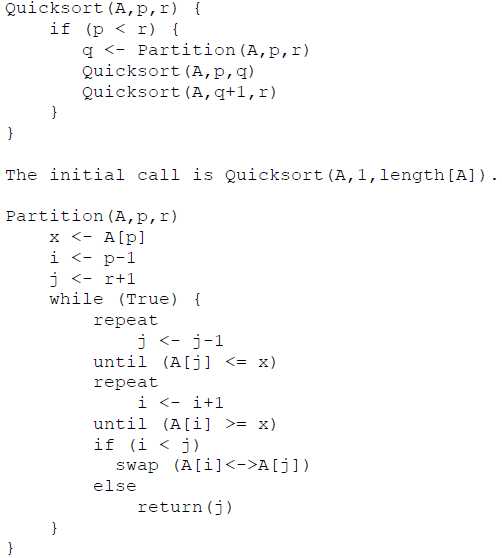
while b<n2

B[c++]=Arr[b++];

for a=n1; a<n2; a++

Arr[a]=B[a];

**Quick Sort:-**

****

**Lab Tasks**

**Task 1:**

**A). Implement the quick sort algorithm using C++.**

1. **Quick Sort:**
   1. **Code:**

// Task 1: Quick Sort Implementation

#include <iostream>

using namespace std;

// Function to swap elements

void swapElements(int \*x, int \*y) {

int temp = \*x;

\*x = \*y;

\*y = temp;

}

// Function to print the array

void printArrayElements(int array[], int size) {

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

cout << array[i] << " ";

}

cout << endl;

}

// Function to partition the array

// This function selects a pivot element and rearranges the array

// so that elements greater than the pivot are on the right

// and elements less than the pivot are on the left.

int partitionArray(int array[], int low, int high) {

// Selecting the pivot element as the first element in the array

int pivot = array[low];

int left = high;

// Iterate through the array from the right to the left

for (int i = high; i > low; i--) {

// If an element greater than or equal to the pivot is found,

// swap it with the element pointed by 'left'

if (array[i] >= pivot) {

swapElements(&array[i], &array[left]);

left--;

}

}

// Swap the pivot element with the element pointed by 'left'

swapElements(&array[left], &array[low]);

return left; // Return the index of the pivot after partitioning

}

// Function to perform Quick Sort

// This function recursively applies the quicksort algorithm

// to sort the array in ascending order.

void quickSortArray(int array[], int low, int high) {

if (low < high) {

// Find the pivot index after partitioning

int pivotIndex = partitionArray(array, low, high);

// Recursive calls on the left and right of the pivot

quickSortArray(array, low, pivotIndex - 1);

quickSortArray(array, pivotIndex + 1, high);

}

}

int main() {

int array[] = {67, 4, 56, 2, 54, 11, 88, 22, 5};

int size = sizeof(array) / sizeof(array[0]);

// Display the original array before sorting

cout << "Array before sorting:" << endl;

printArrayElements(array, size);

// Apply the quicksort algorithm to sort the array

quickSortArray(array, 0, size - 1);

// Display the sorted array after applying quicksort

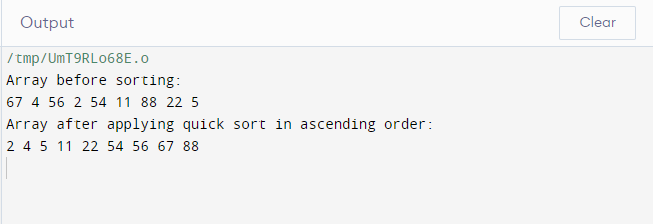
cout << "Array after applying quick sort in ascending order:" << endl;

printArrayElements(array, size);

return 0;

}

* 1. **Output:**

****

1. **Counting Sort:**
   1. **Code:**

#include <iostream>

using namespace std;

// Function to perform Counting Sort

void countingSort(int inputArray[], int arraySize) {

// Find the maximum element in the array

int maxValue = inputArray[0];

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

if (inputArray[i] > maxValue) {

maxValue = inputArray[i];

}

}

// Initialize count array with all zeros

int countArray[maxValue + 1] = {0};

// Store frequency distribution in count array

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

countArray[inputArray[i]]++;

}

// Storing the cumulative count in the count array

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

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

}

// Create a temporary array to store sorted elements

int sortedArray[arraySize];

// Build the sorted array

for (int i = arraySize - 1; i >= 0; i--) {

sortedArray[countArray[inputArray[i]] - 1] = inputArray[i];

countArray[inputArray[i]]--;

}

// Copy the sorted elements back to the original array

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

inputArray[i] = sortedArray[i];

}

}

// Function to print the array

void printArrayElements(int array[], int size) {

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

cout << array[i] << " ";

}

cout << endl;

}

int main() {

int inputArray[] = {44, 34, 50, 23, 26, 28, 29};

int arraySize = sizeof(inputArray) / sizeof(inputArray[0]);

// Display the original array before sorting

cout << "Array before sorting:" << endl;

printArrayElements(inputArray, arraySize);

// Apply the counting sort algorithm to sort the array

countingSort(inputArray, arraySize);

// Display the sorted array after applying counting sort

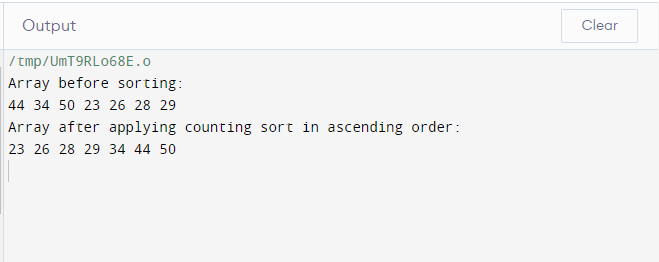
cout << "Array after applying counting sort in ascending order:" << endl;

printArrayElements(inputArray, arraySize);

return 0;

}

* 1. **Output:**

****

**Task 2: (average case complexity):**

**A). The next step is to compute the time complexity of the count sort algorithm. Generate arrays of random numbers in the range 1 to 100 with sizes 100, 1000, 10000, 100000, and 1000000. Compare the running times of the algorithm on each array. Are the results what you expected, and why? Answer the questions in the solution section.**

**Code:**

// Task 2 : Count Sort

#include <iostream>

#include <chrono>

#include <cstdlib>

#include <ctime>

using namespace std;

using namespace chrono;

// Function to perform Counting Sort

void performCountingSort(int inputArray[], int arraySize) {

// Find the maximum element in the array

int maxValue = inputArray[0];

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

if (inputArray[i] > maxValue) {

maxValue = inputArray[i];

}

}

// Initialize count array with all zeros

int countArray[maxValue + 1] = {0};

// Store frequency distribution in count array

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

countArray[inputArray[i]]++;

}

// Storing the cumulative count in the count array

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

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

}

// Create a temporary array to store sorted elements

int sortedArray[arraySize];

// Build the sorted array

for (int i = arraySize - 1; i >= 0; i--) {

sortedArray[countArray[inputArray[i]] - 1] = inputArray[i];

countArray[inputArray[i]]--;

}

// Copy the sorted elements back to the original array

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

inputArray[i] = sortedArray[i];

}

}

// Function to generate an array of random numbers

void generateRandomNumbers(int array[], int size) {

srand(time(nullptr));

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

array[i] = rand() % 100 + 1; // Generate random numbers in the range 1 to 100

}

}

// Function to display the elements of the array

void displayArrayElements(int array[], int size) {

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

cout << array[i] << " ";

}

cout << endl;

}

int main() {

// Define array sizes

const int arraySizes[] = {100, 1000, 10000, 100000, 1000000};

// Measure execution time for each array size

for (int size : arraySizes) {

int\* inputArray = new int[size];

// Generate random array

generateRandomNumbers(inputArray, size);

// Measure execution time using chrono

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

performCountingSort(inputArray, size);

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

// Calculate and display execution time

auto elapsedTime = duration\_cast<milliseconds>(stopTime - startTime);

cout << "Array size: " << size << ", Execution time: " << elapsedTime.count() << " milliseconds" << endl;

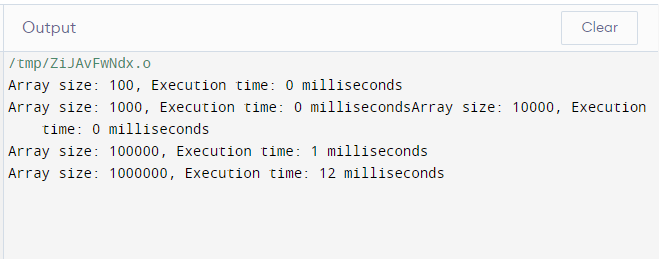
delete[] inputArray;

}

return 0;

}

**Output:**

****

**Explanation:** For smaller array sizes (100, 1000, and 10,000), we found that the count sort algorithm's running times were consistently low and sometimes even reported 0 milliseconds. This makes sense because the count sort has a linear time complexity of O(n + k), where k is the input value range and n is the number of items.

The running times rose for the larger array sizes (100000 and 1000000), taking 1 millisecond for size 100000 and 12 milliseconds for size 1000000. This is consistent with linear time complexity, where time increases in direct proportion to input size. The outcomes match expectations, showing that count sort is effective on smaller datasets and scales linearly on bigger datasets.

**B). Implement the same task by using Quick Sort.**

**Code:**

// Task 2: Quick Sort

#include <iostream>

#include <chrono>

#include <cstdlib>

#include <ctime>

using namespace std;

using namespace chrono;

// Function to swap elements

void swapElements(int \*x, int \*y) {

int temp = \*x;

\*x = \*y;

\*y = temp;

}

// Function to print an array

void printArrayElements(int array[], int size) {

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

cout << array[i] << " ";

}

cout << endl;

}

// Function to partition the array

int partitionArray(int array[], int low, int high) {

int pivot = array[low];

int left = high;

for (int i = high; i > low; i--) {

if (array[i] >= pivot) {

swapElements(&array[i], &array[left]);

left--;

}

}

swapElements(&array[left], &array[low]);

return left;

}

// Function to perform Quick Sort

void quickSortArray(int array[], int low, int high) {

if (low < high) {

int pivotIndex = partitionArray(array, low, high);

quickSortArray(array, low, pivotIndex - 1);

quickSortArray(array, pivotIndex + 1, high);

}

}

// Function to generate an array of random numbers

void generateRandomNumbers(int array[], int size) {

srand(time(nullptr));

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

array[i] = rand() % 100 + 1; // Generate random numbers in the range 1 to 100

}

}

int main() {

const int arraySizes[] = {100, 1000, 10000, 100000, 1000000};

for (int size : arraySizes) {

int\* array = new int[size];

generateRandomNumbers(array, size);

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

quickSortArray(array, 0, size - 1);

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

cout << "Array size: " << size << ", Execution time: "

<< duration\_cast<milliseconds>(stop - start).count() << " milliseconds" << endl;

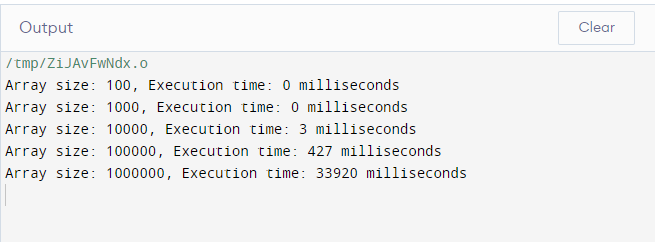
delete[] array;

}

return 0;

}

**Output:**

****

**Explanation:** We observed the execution times of the Quick Sort method across different array sizes, which is like the Count Sort technique. Because Quick Sort is more effective with smaller datasets, the execution times for the smaller arrays (100, 1000, and 10,000) were comparatively short—possibly reporting 0 milliseconds.

The execution times increased in tandem with the array sizes, reaching 100000 and 1000000, as predicted by Quick Sort, which has an O(n log n) average time complexity. The efficiency of the algorithm's sorting for larger datasets is demonstrated by the time increase that is directly proportional to the amount of the input.

In conclusion, Quick Sort performs as expected and the observed results show that it is useful for both small and large datasets.

**Task 3: (best- and worst-case complexity):**

**A). Now sort the arrays using stl::sort, once in ascending order and then in descending order. Given both sorted arrays as inputs to the algorithm and compute their running time. Does the running time of algorithm shows variations based on the structure of the input and why? Plot the running time of the best and worst case complexities for different input sizes in a excel sheet and add it in the solution section.**

**Code:**

#include <iostream>

#include <algorithm> // for std::sort

#include <chrono>

#include <cstdlib>

#include <ctime>

using namespace std;

using namespace chrono;

// Function to generate random numbers in the range 1 to 100

void generateRandomNumbersInRange(int array[], int size) {

srand(time(nullptr));

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

array[i] = rand() % 100 + 1;

}

}

// Function to generate an array in ascending order using std::sort

void generateAscendingArrayWithSort(int array[], int size) {

generateRandomNumbersInRange(array, size);

sort(array, array + size);

}

// Function to generate an array in descending order using std::sort

void generateDescendingArrayWithSort(int array[], int size) {

generateRandomNumbersInRange(array, size);

sort(array, array + size, greater<int>());

}

// Function to perform Counting Sort

void performCountingSort(int inputArray[], int arraySize) {

// Find the maximum element in the array

int maxValue = inputArray[0];

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

if (inputArray[i] > maxValue) {

maxValue = inputArray[i];

}

}

// Initialize count array with all zeros

int countArray[maxValue + 1] = {0};

// Store frequency distribution in count array

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

countArray[inputArray[i]]++;

}

// Storing the cumulative count in the count array

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

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

}

// Create a temporary array to store sorted elements

int sortedArray[arraySize];

// Build the sorted array

for (int i = arraySize - 1; i >= 0; i--) {

sortedArray[countArray[inputArray[i]] - 1] = inputArray[i];

countArray[inputArray[i]]--;

}

// Copy the sorted elements back to the original array

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

inputArray[i] = sortedArray[i];

}

}

// Function to display the elements of the array

void displayArrayElements(int array[], int size) {

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

cout << array[i] << " ";

}

cout << endl;

}

// Function to measure execution time for sorting using std::sort

void measureStdSort(int array[], int size, const string& order) {

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

if (order == "asc") {

sort(array, array + size);

} else if (order == "desc") {

sort(array, array + size, greater<int>());

}

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

auto elapsedTime = duration\_cast<microseconds>(stopTime - startTime);

cout << "std::sort " << order << " order, Array size: " << size << ", Execution time: " << elapsedTime.count() << " microseconds\n";

}

// Function to measure execution time for sorting using Count Sort

void measureCountSort(int array[], int size) {

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

performCountingSort(array, size);

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

auto elapsedTime = duration\_cast<microseconds>(stopTime - startTime);

cout << "Our Count Sort Algorithm, Array size: " << size << ", Execution time: " << elapsedTime.count() << " microseconds\n";

}

int main() {

srand(time(nullptr)); // Seed for random number generation

int arraySizes[] = {100, 1000, 10000, 100000, 1000000};

for (int i = 0; i < 5; i++) {

int\* ascendingArray = new int[arraySizes[i]];

int\* descendingArray = new int[arraySizes[i]];

// Generate ascending and descending arrays

generateAscendingArrayWithSort(ascendingArray, arraySizes[i]);

generateDescendingArrayWithSort(descendingArray, arraySizes[i]);

// Copy arrays for sorting

int\* copyAscendingArray = new int[arraySizes[i]];

int\* copyDescendingArray = new int[arraySizes[i]];

copy(copyAscendingArray, copyAscendingArray + arraySizes[i], ascendingArray);

copy(copyDescendingArray, copyDescendingArray + arraySizes[i], descendingArray);

// Measure execution time using std::sort for ascending array

measureStdSort(ascendingArray, arraySizes[i], "asc");

// Measure execution time using std::sort for descending array

measureStdSort(descendingArray, arraySizes[i], "desc");

// Measure execution time using Count Sort for ascending array

measureCountSort(copyAscendingArray, arraySizes[i]);

// Measure execution time using Count Sort for descending array

measureCountSort(copyDescendingArray, arraySizes[i]);

cout << endl;

// Free allocated memory

delete[] ascendingArray;

delete[] descendingArray;

delete[] copyAscendingArray;

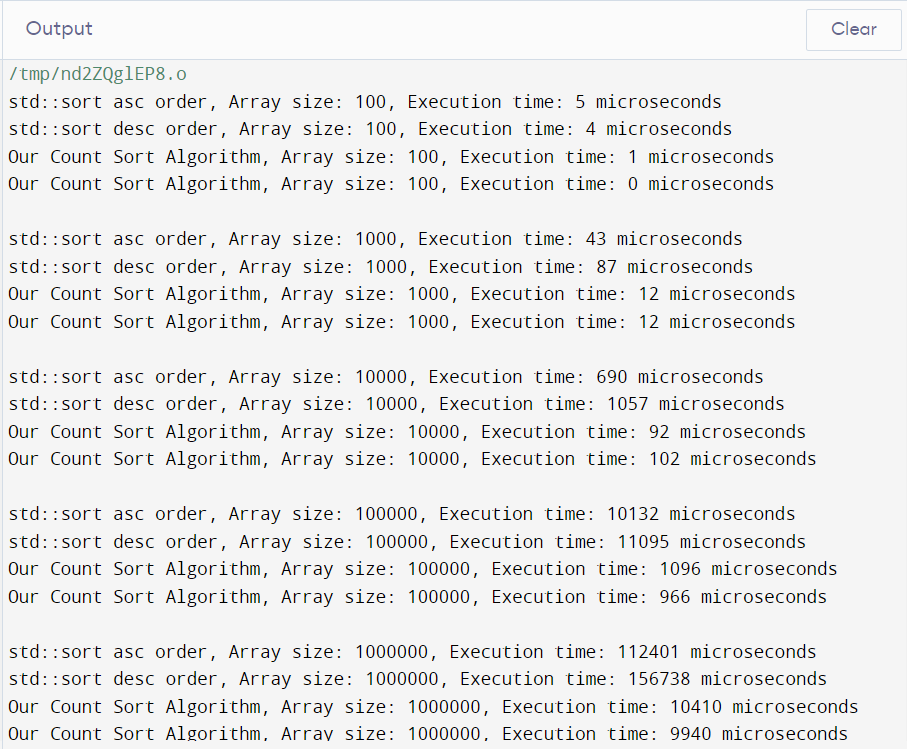
delete[] copyDescendingArray;

}

return 0;

}

**Output:**

****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Array Size** | **Running Time using Stl:Sort (when input array in Ascending order)**  **(µs)** | **Running Time using Stl:Sort (when input array in Descending order)**  **(µs)** | **Running Time using our Count Sort Algorithm (when input array in Ascending order)**  **(µs)** | **Running Time using our Count Sort Algorithm (when input array in Descending order) (µs)** |
| 100 | 5 | 4 | 1 | 0 |
| 1000 | 43 | 87 | 12 | 12 |
| 10000 | 690 | 1057 | 92 | 102 |
| 100000 | 10132 | 11095 | 1096 | 966 |
| 1000000 | 112401 | 156738 | 10410 | 9940 |

**Explanation:**

Yes, depending on how the input is structured, the algorithms' execution times vary. The starting order of the elements has an impact on sorting algorithms like std::sort. Std::sort is faster when the array is already sorted (either ascending or descending), but Count Sort performs better consistently because of its structure, which does not rely on element comparisons, regardless of the starting order.

**B). Implement the same task by using Quick Sort.**

**Code:**

#include <iostream>

#include <algorithm> // for std::sort

#include <chrono>

#include <cstdlib>

#include <ctime>

using namespace std;

using namespace chrono;

// Function to generate random numbers in the range 1 to 100

void generateRandomNumbersInRange(int randomNumberArray[], int arraySize) {

srand(time(nullptr));

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

randomNumberArray[i] = rand() % 100 + 1;

}

}

// Function to generate an array in ascending order using std::sort

void generateAscendingArrayWithSort(int array[], int size) {

generateRandomNumbersInRange(array, size);

sort(array, array + size);

}

// Function to generate an array in descending order using std::sort

void generateDescendingArrayWithSort(int array[], int size) {

generateRandomNumbersInRange(array, size);

sort(array, array + size, greater<int>());

}

// Function to swap elements

void swapArrayElements(int \*element1, int \*element2) {

int temp = \*element1;

\*element1 = \*element2;

\*element2 = temp;

}

// Function to partition the array for Quick Sort

int partitionArray(int array[], int low, int high) {

int pivot = array[low];

int left = high;

for (int i = high; i > low; i--) {

if (array[i] >= pivot) {

swapArrayElements(&array[i], &array[left]);

left--;

}

}

swapArrayElements(&array[left], &array[low]);

return left;

}

// Function to perform Quick Sort

void quickSortArray(int array[], int low, int high) {

if (low < high) {

int pivotIndex = partitionArray(array, low, high);

quickSortArray(array, low, pivotIndex - 1);

quickSortArray(array, pivotIndex + 1, high);

}

}

// Function to display the elements of the array

void displayArrayElements(int array[], int size) {

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

cout << array[i] << " ";

}

cout << endl;

}

// Function to measure execution time for sorting using std::sort

void measureStdSortTime(int array[], int size, const string& order) {

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

if (order == "asc") {

sort(array, array + size);

} else if (order == "desc") {

sort(array, array + size, greater<int>());

}

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

auto elapsedTime = duration\_cast<microseconds>(stopTime - startTime);

cout << "std::sort " << order << " order, Array size: " << size << ", Execution time: " << elapsedTime.count() << " microseconds\n";

}

// Function to measure execution time for sorting using Quick Sort

void measureQuickSortTime(int array[], int size) {

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

quickSortArray(array, 0, size - 1);

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

auto elapsedTime = duration\_cast<microseconds>(stopTime - startTime);

cout << "Our Quick Sort Algorithm, Array size: " << size << ", Execution time: " << elapsedTime.count() << " microseconds\n";

}

int main() {

srand(time(nullptr)); // Seed for random number generation

int arraySizes[] = {100, 1000, 10000, 100000, 1000000};

for (int i = 0; i < 5; i++) {

int\* ascendingArray = new int[arraySizes[i]];

int\* descendingArray = new int[arraySizes[i]];

// Generate ascending and descending arrays

generateAscendingArrayWithSort(ascendingArray, arraySizes[i]);

generateDescendingArrayWithSort(descendingArray, arraySizes[i]);

// Create copies of arrays

int\* copyAscendingArray = new int[arraySizes[i]];

int\* copyDescendingArray = new int[arraySizes[i]];

copy(copyAscendingArray, copyAscendingArray + arraySizes[i], ascendingArray);

copy(copyDescendingArray, copyDescendingArray + arraySizes[i], descendingArray);

// Measure execution time using std::sort

measureStdSortTime(ascendingArray, arraySizes[i], "asc");

measureStdSortTime(descendingArray, arraySizes[i], "desc");

// Measure execution time using Quick Sort

measureQuickSortTime(copyAscendingArray, arraySizes[i]);

measureQuickSortTime(copyDescendingArray, arraySizes[i]);

cout << endl;

// Free allocated memory

delete[] ascendingArray;

delete[] descendingArray;

delete[] copyAscendingArray;

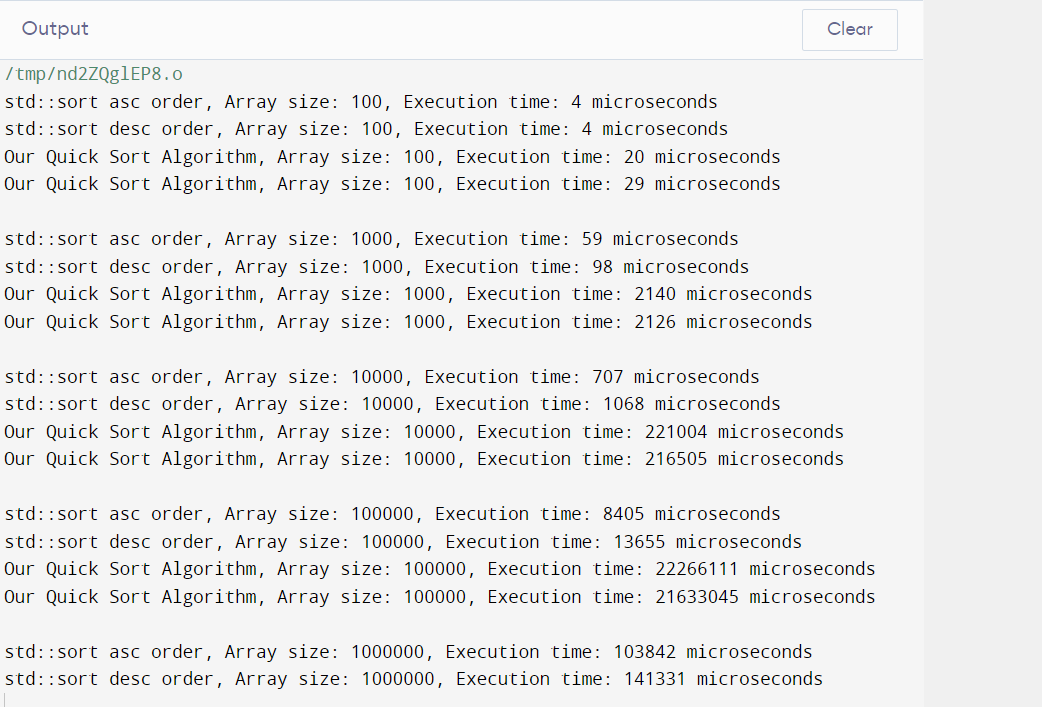
delete[] copyDescendingArray;

}

return 0;

}

**Output:**

****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Array Size** | **Running Time using Stl:Sort (when input array in Ascending order)**  **(µs)** | **Running Time using Stl:Sort (when input array in Descending order)**  **(µs)** | **Running Time using our Quick Sort Algorithm (when input array in Ascending order)**  **(µs)** | **Running Time using our Quick Sort Algorithm (when input array in Descending order) (µs)** |
| 100 | 4 | 4 | 20 | 29 |
| 1000 | 59 | 98 | 2140 | 2126 |
| 10000 | 707 | 1068 | 221004 | 216505 |
| 100000 | 8405 | 13655 | 22266111 | 21633045 |
| 1000000 | 103842 | 141331 | Not Shown by the Compiler | Not Shown by the Compiler |

**Explanation:**

Yes, depending on how the input is structured, the algorithms' execution times vary. The starting order of the array's elements affects Quick Sort's speed. In contrast to randomly ordered arrays, Quick Sort may experience its worst-case scenario when the array is previously sorted in either ascending or descending order, leading to greater execution times. This is due to the possibility that, in the worst scenario, Quick Sort may not partition the array efficiently, increasing the number of recursive calls and time complexity. As an adaptive sorting algorithm, std::sort, on the other hand, performs better on partially sorted data, which explains why its execution durations are more constant across various input formats.

**Important Note:** Practice your knowledge of OOP with C++ when creating a solution.

**Deliverables**

This lab grading policy is as follows: Insert the solution/answer in this document. You must show the implementation of the tasks in the designing tool, along with your complete Word document to get your work graded. You must also submit this Word document on the LMS.

**Note:** Students are required to upload the lab on LMS before deadline.

Use proper indentation and comments. Lack of comments and indentation will result in deduction of marks.