# 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**

# Lab Engineer: Anum Asif

# Lab 8: Sorting Algorithms

**Lab Task:**

**Code:**

#include <iostream>

#include <algorithm>

#include <ctime>

#include <cstdlib>

#include <functional>

#include <vector>

using namespace std;

class Sorter

{

public:

    // Function for quick sort

    void QuickSort(int arr[], int low, int high)

    {

        if (low < high)

        {

            // Partition the array and get the pivot index

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

            // Recursively sort the subarrays on both sides of the pivot

            QuickSort(arr, low, pi - 1);

            QuickSort(arr, pi + 1, high);

        }

    }

    // Partition function for quicksort

    int partition(int arr[], int left, int right)

    {

        int pivot = arr[right];

        int i = (left - 1);

        // Traverse through the array elements

        for (int j = left; j <= right - 1; j++)

        {

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

            if (arr[j] < pivot)

            {

                i++;

                // Swap arr[i] and arr[j]

                std::swap(arr[i], arr[j]);

            }

        }

        // Swap arr[i + 1] and arr[right] (placing the pivot in its correct position)

        std::swap(arr[i + 1], arr[right]);

        return i + 1;

    }

    // Function for counting sort

    void CountingSort(int arr[], int size)

    {

        int max = \*max\_element(arr, arr + size) + 1;

        vector<int> count(max, 0);

        // Count occurrences of each element

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

        {

            count[arr[i]]++;

        }

        // Update array with sorted values

        int index = 0;

        for (int i = 0; i < max; i++)

        {

            while (count[i] > 0)

            {

                arr[index++] = i;

                count[i]--;

            }

        }

    }

    // Function to display an array

    void displayArray(int arr[], int n)

    {

        for (int i = 0; i < n; i++)

        {

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

        }

        std::cout << std::endl;

    }

};

// Function to measure and display the running time for sorting using counting sort

void measureAndDisplayCountingSortTime(Sorter &s, int array[], int size)

{

    std::clock\_t start\_time = std::clock();

    // Sort the array using counting sort

    s.CountingSort(array, size);

    // Measure the running time

    std::clock\_t end\_time = std::clock();

    // Calculate elapsed time

    double elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;

    // Display the result

    std::cout << "Counting Sort for " << size << " elements took " << elapsed\_time << " seconds\n";

}

int main()

{

    Sorter s;

    // Array 1

    int a1[10] = {5, 9, 1, 3, 7, 2, 8, 4, 6, 10};

    int a2[10] = {90, 3, 1, 0, 4, 93, 33, 77, 2, 10};

    // Sort and display the array using QuickSort

    s.QuickSort(a1, 0, 9);

    std::cout << "The Result of Quick sort is: ";

    s.displayArray(a1, 10);

    // Sort and display the array using CountingSort

    s.CountingSort(a2,10);

    std::cout << "The Result of Counting sort is: ";

    s.displayArray(a2, 10);

    // Task 2: Measure and compare the running times of sorting algorithms on random arrays

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

    for (int size : sizes)

    {

        // Dynamically allocate memory for a random array

        int \*randomArray = new int[size];

        // Fill the array with random numbers

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

        {

            randomArray[i] = std::rand() % 100 + 1;

        }

        // Start the timer for QuickSort on the random array

        std::clock\_t start\_time\_quick = std::clock();

        // Test and measure QuickSort

        s.QuickSort(randomArray, 0, size - 1);

        // Stop the timer

        std::clock\_t end\_time\_quick = std::clock();

        // Calculate elapsed time

        double elapsed\_time\_quick = static\_cast<double>(end\_time\_quick - start\_time\_quick) / CLOCKS\_PER\_SEC;

        std::cout << "Quick Sort for " << size << " elements took " << elapsed\_time\_quick << " seconds\n";

        // Start the timer for CountingSort on the random array

        std::clock\_t start\_time\_counting = std::clock();

        // Test and measure CountingSort

        s.CountingSort(randomArray, size);

        // Stop the timer

        std::clock\_t end\_time\_counting = std::clock();

        // Calculate elapsed time

        double elapsed\_time\_counting = static\_cast<double>(end\_time\_counting - start\_time\_counting) / CLOCKS\_PER\_SEC;

        std::cout << "Counting Sort for " << size << " elements took " << elapsed\_time\_counting << " seconds\n";

        // Deallocate memory

        delete[] randomArray;

    }

    // Task 3: Sort the arrays using std::sort in ascending and descending order

    // and then test QuickSort and CountingSort on these sorted arrays.

    int array1[10] = {5, 9, 1, 3, 7, 2, 8, 4, 6, 10};

    int array2[10] = {12, 45, 23, 9, 3, 31, 7, 19, 2, 8};

    // Task 3: Best Case - Ascending Order

    std::sort(std::begin(array1), std::end(array1));

    // Start the timer for QuickSort on the sorted array

    std::clock\_t start\_time\_quick\_best = std::clock();

    // Test and measure QuickSort

    s.QuickSort(array1, 0, 9);

    // Stop the timer

    std::clock\_t end\_time\_quick\_best = std::clock();

    // Calculate elapsed time

    double elapsed\_time\_quick\_best = static\_cast<double>(end\_time\_quick\_best - start\_time\_quick\_best) / CLOCKS\_PER\_SEC;

    std::cout << "Quick Sort on the sorted array (Best Case) took " << elapsed\_time\_quick\_best << " seconds\n";

    // Start the timer for CountingSort on the sorted array

    std::clock\_t start\_time\_counting\_best = std::clock();

    // Test and measure CountingSort

    s.CountingSort(array1, 10);

    // Stop the timer

    std::clock\_t end\_time\_counting\_best = std::clock();

    // Calculate elapsed time

    double elapsed\_time\_counting\_best = static\_cast<double>(end\_time\_counting\_best - start\_time\_counting\_best) / CLOCKS\_PER\_SEC;

    std::cout << "Counting Sort on the sorted array (Best Case) took " << elapsed\_time\_counting\_best << " seconds\n";

    // Task 3: Worst Case - Descending Order

    std::sort(std::begin(array2), std::end(array2), std::greater<int>());

    // Start the timer for QuickSort on the sorted array

    std::clock\_t start\_time\_quick\_worst = std::clock();

    // Test and measure QuickSort

    s.QuickSort(array2, 0, 9);

    // Stop the timer

    std::clock\_t end\_time\_quick\_worst = std::clock();

    // Calculate elapsed time

    double elapsed\_time\_quick\_worst = static\_cast<double>(end\_time\_quick\_worst - start\_time\_quick\_worst) / CLOCKS\_PER\_SEC;

    std::cout << "Quick Sort on the sorted array (Worst Case) took " << elapsed\_time\_quick\_worst << " seconds\n";

    // Start the timer for CountingSort on the sorted array

    std::clock\_t start\_time\_counting\_worst = std::clock();

    // Test and measure CountingSort

    s.CountingSort(array2, 10);

    // Stop the timer

    std::clock\_t end\_time\_counting\_worst = std::clock();

    // Calculate elapsed time

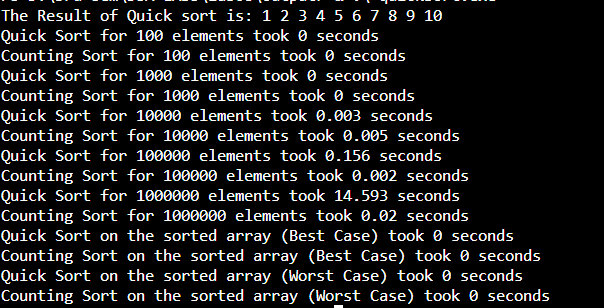
    double elapsed\_time\_counting\_worst = static\_cast<double>(end\_time\_counting\_worst - start\_time\_counting\_worst) / CLOCKS\_PER\_SEC;

    std::cout << "Counting Sort on the sorted array (Worst Case) took " << elapsed\_time\_counting\_worst << " seconds\n";

    return 0;

}

**Output:**

****

**Table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **100** | **1000** | **10000** | **100000** | **1000000** |
| **Quick Sort** | **0** | **0** | **0.003** | **0.156** | **14.593** |
| **Counting** | **0** | **0** | **0005** | **0.002** | **0.02** |

**Deliverables**