**Department of Computing**

**School of Electrical Engineering and Computer Science**

**CS250 – Data Structures and Algorithms**



**Lab 11: Implementation of Sorting Algorithms and   
Complexity Analysis**

**Submission Details**

|  |  |
| --- | --- |
| Name | CMS ID |
| Muhammad Umer | 345834 |
| Group | GP – 1 |
| Lab Engineer | Anum Asif |
| Faculty Member | Bostan Khan |
| Class | BEE12 |
| Date | 22/04/2024 |
| Time | 10:00 am – 12:50 pm |

**Table of Contents**

[2 Implementation of Sorting Algorithms and Complexity Analysis 3](#_Toc164673615)

[2.1 Introduction 3](#_Toc164673616)

[2.2 Objectives 3](#_Toc164673617)

[2.3 Tools/Software Requirement 3](#_Toc164673618)

[2.4 Deliverables 3](#_Toc164673619)

[2.5 Description 3](#_Toc164673620)

[3 Lab Tasks 5](#_Toc164673621)

[3.1 Task 1 5](#_Toc164673622)

[3.2 Task 2 (average case complexity) 6](#_Toc164673623)

[3.3 Task 3 (best and worst case complexity) 8](#_Toc164673624)

# Implementation of Sorting Algorithms and Complexity Analysis

## Introduction

In this lab, you will implement three sorting algorithms and compare them.

## Objectives

Objective of this lab is to implement insertion sort and merge sort and compare the running times for both sorting algorithms.

## Tools/Software Requirement

* Visual Studio C++

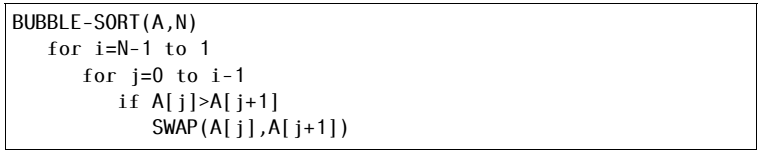
## Deliverables

Compile a single word document by filling in the solution parts and submit this file on LMS. The name of word document should follow this format. i.e., YourFullName(reg)\_Lab#. You must show the implementation of the tasks in a complete manner to get your work graded.

## Description

**Bubble Sort**

Bubble sort is a popular sorting algorithm, which is quite simple to implement. The pseudo code is as follows:



**Selection Sort**

Selection sort is a popular sorting algorithm, which is quite simple to implement. The pseudo code is as follows:



**Insertion Sort**

Insertion sort is a popular sorting algorithm, which is quite simple to implement. The pseudo code is as follows:



**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];

# Lab Tasks

## Task 1

Implement **B**ubble sort, **S**election sort, **I**nsertion sort and **M**erge sort algorithms in C++.

Code: Bubble Sort

*void* bubbleSort(*int* *arr*[], *int* *n*) {

    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], *arr*[j + 1]);

            }

        }

    }

}

Code: Selection Sort

*void* selectionSort(*int* *arr*[], *int* *n*) {

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

*int* minIndex = i;

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

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

                minIndex = j;

            }

        }

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

    }

}

Code: Insertion Sort

*void* insertionSort(*int* *arr*[], *int* *n*) {

    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--;

        }

*arr*[j + 1] = key;

    }

}

Code: Merge Sort

Left out as we have not studied it yet.

Main Function

*int* main() {

*int* arr[] = {5, 2, 3, 1, 4};

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

    cout << "Bubble Sort: ";

    bubbleSort(arr, n);

    for (*int* i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << "\nSelection Sort: ";

    selectionSort(arr, n);

    for (*int* i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << "\nInsertion Sort: ";

    insertionSort(arr, n);

    for (*int* i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    return 0;

}

Output

root@Zonularity:/home/zonularity/dsa/lab\_11# ./task\_1

Bubble Sort: 1 2 3 4 5

Selection Sort: 1 2 3 4 5

Insertion Sort: 1 2 3 4 5

## Task 2 (average case complexity)

The next step is to compare the two algorithms. Generate arrays of random numbers in the range 1 to 100 with sizes 100, 1000, 10000, and 100000. Compare the running times of the four algorithms on each array. How do they compare? Are the results what you expected, and why? Answer the questions in the solution section.

Code

*int* main() {

    srand(time(NULL));

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

    for (*int* size : sizes) {

*int* arr[size];

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

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

        }

*clock\_t* start = clock();

        bubbleSort(arr, size);

*clock\_t* end = clock();

        cout << "Bubble sort on size " << size << " took "

             << (*double*)(end - start) / CLOCKS\_PER\_SEC << " seconds" << endl;

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

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

        }

        start = clock();

        selectionSort(arr, size);

        end = clock();

        cout << "Selection sort on size " << size << " took "

             << (*double*)(end - start) / CLOCKS\_PER\_SEC << " seconds" << endl;

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

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

        }

        start = clock();

        insertionSort(arr, size);

        end = clock();

        cout << "Insertion sort on size " << size << " took "

             << (*double*)(end - start) / CLOCKS\_PER\_SEC << " seconds" << endl

             << endl;

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

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

        }

    }

    return 0;

}

Output

root@Zonularity:/home/zonularity/dsa/lab\_11# ./task\_2

Bubble sort on size 100 took 2.5e-05 seconds

Selection sort on size 100 took 1e-05 seconds

Insertion sort on size 100 took 7e-06 seconds

Bubble sort on size 1000 took 0.002162 seconds

Selection sort on size 1000 took 0.000619 seconds

Insertion sort on size 1000 took 0.000361 seconds

Bubble sort on size 10000 took 0.205059 seconds

Selection sort on size 10000 took 0.05945 seconds

Insertion sort on size 10000 took 0.037536 seconds

Bubble sort on size 100000 took 28.0958 seconds

Selection sort on size 100000 took 5.69118 seconds

Insertion sort on size 100000 took 3.87243 seconds

## Task 3 (best- and worst-case complexity)

Now sort the arrays using stl::sort, once in ascending order and then in descending order. Given both sorted arrays as inputs to all three algorithms and compute their running time. The running time of which algorithm shows most variations based on the structure of the input and why? Answer the questions in the solution section.

Code

*int* main() {

*int* n = 10000;

*int* arr1[n], arr2[n];

    for (*int* i = 0; i < n; i++) {

        arr1[i] = rand() % n;

        arr2[i] = arr1[i];

    }

*clock\_t* start, end;

*double* timeTaken;

    start = clock();

    sort(arr1, arr1 + n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "STL sort in ascending order: " << timeTaken << endl;

    start = clock();

    sort(arr2, arr2 + n, greater<*int*>());

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "STL sort in descending order: " << timeTaken << endl

         << endl;

    for (*int* i = 0; i < n; i++) {

        arr1[i] = rand() % n;

        arr2[i] = arr1[i];

    }

    sort(arr1, arr1 + n);

    sort(arr2, arr2 + n, greater<*int*>());

    start = clock();

    bubbleSort(arr1, n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "Time taken by bubble sort (ascending): " << timeTaken

<< endl;

    start = clock();

    bubbleSort(arr2, n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "Time taken by bubble sort (descending): " << timeTaken

<< endl << endl;

    for (*int* i = 0; i < n; i++) {

        arr1[i] = rand() % n;

        arr2[i] = arr1[i];

    }

    sort(arr1, arr1 + n);

    sort(arr2, arr2 + n, greater<*int*>());

    start = clock();

    selectionSort(arr1, n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "Time taken by selection sort (ascending): "

<< timeTaken << endl;

    start = clock();

    selectionSort(arr2, n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "Time taken by selection sort (descending): "

<< timeTaken << endl

         << endl;

    for (*int* i = 0; i < n; i++) {

        arr1[i] = rand() % n;

        arr2[i] = arr1[i];

    }

    sort(arr1, arr1 + n);

    sort(arr2, arr2 + n, greater<*int*>());

    start = clock();

    insertionSort(arr1, n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "Time taken by insertion sort (ascending): "

<< timeTaken << endl;

    start = clock();

    insertionSort(arr2, n);

    end = clock();

    timeTaken = *double*(end - start) / *double*(CLOCKS\_PER\_SEC);

    cout << "Time taken by insertion sort (descending): "

<< timeTaken << endl

         << endl;

    return 0;

}

Output

root@Zonularity:/home/zonularity/dsa/lab\_11# ./task\_3

STL sort in ascending order: 0.001029

STL sort in descending order: 0.001088

Time taken by bubble sort (ascending): 0.066926

Time taken by bubble sort (descending): 0.296401

Time taken by selection sort (ascending): 0.056821

Time taken by selection sort (descending): 0.061838

Time taken by insertion sort (ascending): 2.2e-05

Time taken by insertion sort (descending): 0.077024

Reason

The running time of bubble sort shows the most variations based on the structure of the input. This is because bubble sort has a time complexity of O(n^2) in the worst case. The worst case occurs when the array is sorted in descending order. In this case, the inner loop will run n - 1 times in the first iteration, n - 2 times in the second iteration, and so on. This results in a total of (n - 1) + (n - 2) + ... + 1 = n(n - 1) / 2 = O(n^2) comparisons and swaps.

On the other hand, the best case occurs when the array is already sorted in ascending order. In this case, the inner loop will run only once in each iteration, resulting in O(n) comparisons and swaps. Therefore, the running time of bubble sort shows the *most variations* based on the structure of the input.

In contrast, the running time of selection sort, and insertion sort is more consistent across different input structures. This is because selection sort, and insertion sort have time complexities of O(n^2), and O(n^2) respectively, regardless of the input structure. As a result, the running time of these algorithms is more predictable and less affected by the structure of the input compared to bubble sort.