National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

School of Electrical Engineering and Computer Science

CS-250: Data Structure and Algorithms

Class: BESE 13A

Name: Haris Rehman

CMS ID: 410937

Lab 7: Implementation of Sorting Algorithms and Complexity Analysis

Date: 3rd November, 2023

Time: 10 am - 12 pm

Lab Instructor: Anum Asif

Lab 7: 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++

Helping Material
Lecture slides, text book

Description:

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

```
BUBBLE-SORT(A,N)
   for i=N-1 to 1
      for j=0 to i-1
         if A[j]>A[j+1]
            SWAP(A[j],A[j+1])
```

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

```
SELECTION-SORT(A,N)
   for i=N-1 to 0
      max = i
      for j=0 to i-1
         if A[j]>A[max]
            max = j
      SWAP(A[i],A[max])
```

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

```
INSERTION-SORT(A,N)
   for i=1 to N-1
      key = A[i]
      j = i - 1
      while j >= 0 and A[j]>key
         A[j+1] = A[j]
         j = j - 1
      A[j+1] = key
```

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-SORT(A,first,last)
   if first < last:
      midpoint = (first + last) / 2
      MERGE-SORT(A, first, midpoint)
      MERGE-SORT(A, midpoint+1, last)
      MERGE( A, first, midpoint + 1, last)
```

```
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 Bubble sort, Selection sort, Insertion sort and Merge sort algorithms in C++.

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, 100000, and 1000000. Compare the running times of the three algorithms on each array. How do they compare? Are the results what you expected, and why? Answer the questions in the solution section.

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.

Important Note: Practice your knowledge of OOP with C++ when creating a solution. Remember to comment your code properly. Inappropriate or no comment may result in deduction of marks.

Solution:

Solution

Task 1:
 	
For the inputs
 	
`int arr[] = {64, 34, 25, 12, 22, 11, 90};`
`int arr2[] = {12, 11, 13, 5, 6};`

```
int arr3[] = {64, 25, 12, 22, 11};
int arr4[] = {64, 34, 25, 12, 22, 11, 90, 23, 55, 12, 55};
```

The outputs are:

```
BUBBLE SORT:
Sorted array: 11 12 22 25 34 64 90

INSERTION SORT:
Sorted array: 5 6 11 12 13

SELECTION SORT:
Sorted array: 11 12 22 25 64

MERGE SORT:
Sorted array: 11 12 12 22 23 25 34 55 55 64 90
```

Task 2 :

Based on the outputs I got, Bubble Sort took longest as it is a simple sorting algorithm with a time complexity of O(n^2). It is not efficient for larger arrays. As the array size increases, the running time increases significantly. For larger arrays, Bubble Sort becomes impractical.

Selection Sort also has a time complexity of O(n^2). It performs better than Bubble Sort but is still inefficient for larger arrays. The running time increases as the array size grows.

Insertion Sort however has a time complexity of O(n^2) in the worst case. Like Bubble and Selection Sort, it becomes less efficient as the array size increases. It performs better than the other two in many cases due to its adaptive nature (good for partially sorted data).

The time complexity of Merge Sort is O(n log n), where 'n' is the number of elements in the array that is being sorted. Merge Sort is an efficient, comparison-based, divide-and-conquer sorting algorithm known for its stable and predictable time complexity. Thus it takes the least time to execute.

For small arrays, all three sorting algorithms may work reasonably quickly. However, as the array size increases, their running times increase significantly. For very large arrays (e.g., 1000000 elements),

these sorting algorithms become impractical for real-world applications except for Merge Sort and thus it is used more in such cases.

```
Array size: 100
Bubble Sort Time: 0 seconds
Selection Sort Time: 0 seconds
Insertion Sort Time: 0 seconds
Merge Sort Time: 0 seconds

Array size: 1000
Bubble Sort Time: 0.002 seconds
Selection Sort Time: 0.001 seconds
Insertion Sort Time: 0.001 seconds
Merge Sort Time: 0 seconds

Array size: 10000
Bubble Sort Time: 0.295 seconds
Selection Sort Time: 0.086 seconds
Insertion Sort Time: 0.061 seconds
Merge Sort Time: 0.001 seconds

Array size: 100000
Bubble Sort Time: 32.06 seconds
Selection Sort Time: 8.85 seconds
Insertion Sort Time: 6.644 seconds
Merge Sort Time: 0.011 seconds
```

Task 3:

Bubble Sort's running time increases in both ascending and descending order inputs.

Selection Sort's running time is also increasing between ascending and descending order inputs.

Insertion Sort's running time is relatively consistent between ascending and descending order inputs and barely needs to do any execution. Merge sort doesn’t take any time at all and barely needs execution.

The results are expected because Bubble Sort, Selection Sort, and Insertion Sort are less efficient and have consistent running times for ascending and descending order inputs. These simple sorting algorithms have time complexities that are not greatly influenced by the structure of the input data.

In general, the sorting algorithms like Bubble Sort, Selection Sort, and Insertion Sort perform similarly on both ascending and descending order inputs because they involve a large number of comparisons and swaps, making the order of the data less significant compared to more efficient sorting algorithms.

```
Array size: 100
Bubble Sort Ascending Time: 0 seconds
Bubble Sort Descending Time: 0 seconds
Selection Sort Ascending Time: 0 seconds
Selection Sort Descending Time: 0 seconds
Insertion Sort Ascending Time: 0 seconds
Insertion Sort Descending Time: 0 seconds
Merge Sort Ascending Time: 0 seconds
Merge Sort Descending Time: 0 seconds

Array size: 1000
Bubble Sort Ascending Time: 0.001 seconds
Bubble Sort Descending Time: 0.002 seconds
Selection Sort Ascending Time: 0.001 seconds
Selection Sort Descending Time: 0.001 seconds
Insertion Sort Ascending Time: 0 seconds
Insertion Sort Descending Time: 0 seconds
Merge Sort Ascending Time: 0 seconds
Merge Sort Descending Time: 0 seconds

Array size: 10000
Bubble Sort Ascending Time: 0.09 seconds
Bubble Sort Descending Time: 0.253 seconds
Selection Sort Ascending Time: 0.084 seconds
Selection Sort Descending Time: 0.087 seconds
Insertion Sort Ascending Time: 0 seconds
Insertion Sort Descending Time: 0 seconds
Merge Sort Ascending Time: 0.001 seconds
Merge Sort Descending Time: 0 seconds

Array size: 100000
Bubble Sort Ascending Time: 9.125 seconds
Bubble Sort Descending Time: 25.677 seconds
Selection Sort Ascending Time: 8.864 seconds
Selection Sort Descending Time: 8.736 seconds
Insertion Sort Ascending Time: 0 seconds
Insertion Sort Descending Time: 0.001 seconds
Merge Sort Ascending Time: 0.006 seconds
Merge Sort Descending Time: 0.007 seconds
```

Task 1 Code:

```cpp
#include <iostream>
using namespace std;

class sorting{
```

```
public:
    // Function to implement bubble sort. Used swap() in all
    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]);
                }
            }
        }
    }

    // Function to implement 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]);
        }
    }

    // Function to implement bubble 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;
        }
    }

    // function to use in merge sort
    void merge(int arr[], int left, int mid, int right) {
        int n1 = mid - left + 1;
```

```
        int n2 = right - mid;

        int L[n1], R[n2];

        for (int i = 0; i < n1; i++) {
            L[i] = arr[left + i];
        }
        for (int i = 0; i < n2; i++) {
            R[i] = arr[mid + 1 + i];
        }

        int i = 0, j = 0, k = left;

        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    // Calling the merge function recursively
    void mergeSort(int arr[], int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;

            mergeSort(arr, left, mid);
```

```
                mergeSort(arr, mid + 1, right);

                merge(arr, left, mid, right);
            }
        }
};

int main() {
    sorting Sorts;
    // Calling bubble sort and displaying output
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    Sorts.bubbleSort(arr, n);

    cout<< "BUBBLE SORT: "<<endl;
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    cout<<endl<<endl;

    // Calling Insertion sort and displaying output
    int arr2[] = {12, 11, 13, 5, 6};
    n = sizeof(arr2) / sizeof(arr2[0]);

    Sorts.insertionSort(arr2, n);

    cout<< "INSERTION SORT: "<<endl;
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr2[i] << " ";
    }

    // Calling Selection sort and displaying output
    int arr3[] = {64, 25, 12, 22, 11};
    n = sizeof(arr3) / sizeof(arr3[0]);

    Sorts.selectionSort(arr3, n);

    cout<<endl<<endl;
    cout<< "SELECTION SORT: "<<endl;
```

```
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr3[i] << " ";
    }

    // Calling bubble sort and displaying output
    int arr4[] = {64, 34, 25, 12, 22, 11, 90, 23, 55, 12, 55};
    n = sizeof(arr4) / sizeof(arr4[0]);

    cout<<endl<<endl;

    Sorts.mergeSort(arr4, 0, n - 1);

    cout<< "MERGE SORT: "<<endl;
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr4[i] << " ";
    }

    cout<<endl<<endl;

    return 0;
}
```

Task 2 Code:

```
#include <iostream>
#include <ctime>
#include <vector>
#include <algorithm>

using namespace std;

class SortTiming{
    public:
    // Function to generate an array of random numbers in the range 1 to 100
        vector<int> generateRandomArray(int size) {
            vector<int> arr(size);
            for (int i = 0; i < size; i++) {
```

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

    // Function to implement bubble sort. Used swap() in all
    void bubbleSort(vector<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]);
                }
            }
        }
    }

    // Function to implement selection sort
    void selectionSort(vector<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]);
        }
    }

    // Function to implement bubble sort
    void insertionSort(vector<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;
        }
    }
```

```cpp
// function to use in merge sort
void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int i = 0; i < n2; i++) {
        R[i] = arr[mid + 1 + i];
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Calling the merge function recursively
void mergeSort(vector<int>& arr, int left, int right) {
```

```
            if (left < right) {
                int mid = left + (right - left) / 2;

                mergeSort(arr, left, mid);
                mergeSort(arr, mid + 1, right);

                merge(arr, left, mid, right);
            }
        }
};

int main() {
    SortTiming srt;
    srand(static_cast<unsigned>(time(0))); // Seed the random number generator

    // Array sizes
    vector<int> sizes = {100, 1000, 10000, 100000, 1000000};

    for (int size : sizes) {
        vector<int> arr = srt.generateRandomArray(size);

        // Measure time for Bubble Sort
        clock_t start_time = clock();
        srt.bubbleSort(arr, size);
        clock_t end_time = clock();
        double bubbleSortTime = static_cast<double>(end_time - start_time) /
CLOCKS_PER_SEC;

        // Restore the original unsorted array
        random_shuffle(arr.begin(), arr.end());

        // Measure time for Selection Sort
        start_time = clock();
        srt.selectionSort(arr,  size);
        end_time = clock();
        double selectionSortTime = static_cast<double>(end_time - start_time) /
CLOCKS_PER_SEC;

        // Restore the original unsorted array
        random_shuffle(arr.begin(), arr.end());

        // Measure time for Insertion Sort
        start_time = clock();
```

```
        srt.insertionSort(arr, size);
        end_time = clock();
        double insertionSortTime = static_cast<double>(end_time - start_time) /
CLOCKS_PER_SEC;

        // Restore the original unsorted array
        random_shuffle(arr.begin(), arr.end());

        // Measure time for Merge Sort
        start_time = clock();
        srt.mergeSort(arr, 0, size-1);
        end_time = clock();
        double mergeSortTime = static_cast<double>(end_time - start_time) /
CLOCKS_PER_SEC;

        cout << "Array size: " << size << endl;
        cout << "Bubble Sort Time: " << bubbleSortTime << " seconds" << endl;
        cout << "Selection Sort Time: " << selectionSortTime << " seconds" <<
endl;
        cout << "Insertion Sort Time: " << insertionSortTime << " seconds" <<
endl;
        cout << "Merge Sort Time: " << mergeSortTime << " seconds" << endl;
        cout << endl;
    }


    return 0;
}
```

Task 3 Code:

```
#include <iostream>
#include <ctime>
#include <vector>
#include <algorithm>

using namespace std;

class SortTiming{
    public:
    // Function to generate an array of random numbers in the range 1 to 100
        vector<int> generateRandomArray(int size) {
            vector<int> arr(size);
```

```
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 100 + 1;
    }
    return arr;
}

// Function to implement bubble sort. Used swap() in all
void bubbleSort(vector<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]);
            }
        }
    }
}

// Function to implement selection sort
void selectionSort(vector<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]);
    }
}

// Function to implement bubble sort
void insertionSort(vector<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;
    }
```

```
}

// function to use in merge sort
void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int i = 0; i < n2; i++) {
        R[i] = arr[mid + 1 + i];
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Calling the merge function recursively
```

```
        void mergeSort(vector<int>& arr, int left, int right) {
            if (left < right) {
                int mid = left + (right - left) / 2;

                mergeSort(arr, left, mid);
                mergeSort(arr, mid + 1, right);

                merge(arr, left, mid, right);
            }
        }
};

int main() {
    SortTiming srt;
    srand(static_cast<unsigned>(time(0))); // Seed the random number generator

    // Array sizes
    vector<int> sizes = {100, 1000, 10000, 100000, 1000000};

    for (int size : sizes) {
        vector<int> arrAsc = srt.generateRandomArray(size);
        vector<int> arrDesc = arrAsc;

        clock_t start_time = clock();
        sort(arrAsc.begin(), arrAsc.end());
        clock_t end_time = clock();
        double stdSortAscendingTime = static_cast<double>(end_time -
start_time) / CLOCKS_PER_SEC;

        // Measure time for sort (descending order)
        start_time = clock();
        sort(arrDesc.rbegin(), arrDesc.rend());
        end_time = clock();
        double stdSortDescendingTime = static_cast<double>(end_time -
start_time) / CLOCKS_PER_SEC;

        start_time = clock();
        srt.bubbleSort(arrAsc, size);
        end_time = clock();
        double bubbleSortAscendingTime = static_cast<double>(end_time -
start_time) / CLOCKS_PER_SEC;

        // Measure time for Bubble Sort
```

```
        start_time = clock();
        srt.bubbleSort(arrDesc, size);
        end_time = clock();
        double bubbleSortDescendingTime = static_cast<double>(end_time -
start_time) / CLOCKS_PER_SEC;

        // Measure time for Selection Sort
        start_time = clock();
        srt.selectionSort(arrAsc, size);
        end_time = clock();
        double selectionSortAscendingTime = static_cast<double>(end_time -
start_time) / CLOCKS_PER_SEC;

        // Measure time for Selection Sort
        start_time = clock();
        srt.selectionSort(arrDesc, size);
        end_time = clock();
        double selectionSortDescendingTime = static_cast<double>(end_time -
start_time) / CLOCKS_PER_SEC;

        // Measure time for Insertion Sort
        start_time = clock();
        srt.insertionSort(arrAsc, size);
        end_time = clock();
        double insertionSortAscendingTime = static_cast<double>(end_time -
start_time) / CLOCKS_PER_SEC;

        // Measure time for Insertion Sort
        start_time = clock();
        srt.insertionSort(arrDesc, size);
        end_time = clock();
        double insertionSortDescendingTime = static_cast<double>(end_time -
start_time) / CLOCKS_PER_SEC;

        // Measure time for Merge Sort
        start_time = clock();
        srt.mergeSort(arrAsc, 0, size-1);
        end_time = clock();
        double mergeSortAscendingTime = static_cast<double>(end_time -
start_time) / CLOCKS_PER_SEC;

        // Measure time for Merge Sort
        start_time = clock();
```

```
        srt.mergeSort(arrDesc, 0, size-1);
        end_time = clock();
        double mergeSortDescendingTime = static_cast<double>(end_time -
start_time) / CLOCKS_PER_SEC;


        cout << "Array size: " << size << endl;
        cout << "Bubble Sort Ascending Time: " << bubbleSortAscendingTime << "
seconds" << endl;
        cout << "Bubble Sort Descending Time: " << bubbleSortDescendingTime <<
" seconds" << endl;
        cout << "Selection Sort Ascending Time: " << selectionSortAscendingTime
<< " seconds" << endl;
        cout << "Selection Sort Descending Time: " <<
selectionSortDescendingTime << " seconds" << endl;
        cout << "Insertion Sort Ascending Time: " << insertionSortAscendingTime
<< " seconds" << endl;
        cout << "Insertion Sort Descending Time: " <<
insertionSortDescendingTime << " seconds" << endl;
        cout << "Merge Sort Ascending Time: " << mergeSortAscendingTime << "
seconds" << endl;
        cout << "Merge Sort Descending Time: " << mergeSortDescendingTime << "
seconds" << endl;
        cout<<endl;
    }


    return 0;
}
```

Deliverables

Compile a single word document by filling in the solution part and submit this Word file on LMS. 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.