ACTIVITY NO. 7

SORTING ALGORITHMS: BUBBLE, SELECTION, AND INSERTION SORT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:**
Section:	**Date Submitted:**
Name:	**Instructor:** Engr. Roman M. Richard

1. Objective(s)

Create C++ code to sort through data using sorting algorithms

2. Intended Learning Outcomes (ILOs)

After this activity, the student should be able to:

- Create C++ code for implementation of selection, bubble, insertion and merge sort.
- Solve given data sorting problems using appropriate basic sorting algorithms

3. Discussion

Sorting is the process of placing elements from a collection in order. For example, a list of words could be sorted alphabetically or by length. A list of cities could be sorted by population, by area, or by zip code. We have already seen several algorithms that were able to benefit from having a sorted list (recall the final anagram example and the binary search).

4. Materials and Equipment

Personal Computer with C++ IDE
Recommended IDE:

- CLion (must use TIP email to download)
- DevC++ (use the embarcadero fork or configure to C++17)

5. Procedure

ILO A: Create C++ code for implementation of selection, bubble, insertion and merge sort.

Preparation Task:

- Create an array of elements with random values. The array must contain 100 elements that are not sorted. Use the created array for each sorting algorithm below. Show the output of the preparation task on table 7-1 in section 6.
- Create a header file for the implementation of the different sorting algorithms.
- Import this header file into your main.cpp.

A.1. Bubble Sort Technique

Using the bubble sort technique, sorting is done in passes or iteration. Thus at the end of each iteration, the heaviest element is placed at its proper place in the list. In other words, the largest element in the list bubbles up.

General Algorithm

```
template <typename T>
void bubbleSort(T arr[], size_t arrSize){
    //Step 1: For i = 0 to N-1 repeat Step 2
    for(int i = 0; i < arrSize; i++){
        //Step 2: For J = i + 1 to N - I repeat
        for(int j = i+1; j < arrSize; j++){
```

```
                //Step 3: if A[J] > A[i]
                if(arr[j]>arr[i]){
                    //Swap A[J] and A[i]
                    std::swap(arr[j], arr[i]);
                }
                //[End of Inner for loop]
            }
            //[End if Outer for loop]
        }
        //Step 4: Exit
    }
```

Include a screenshot of your code and output console along with your observations in table 7-2.

A.2. Selection Sort Algorithm

Selection sort is quite a straightforward sorting technique as the technique only involves finding the smallest element in every pass and placing it in the correct position. Selection sort works efficiently when the list to be sorted is of small size but its performance is affected badly as the list to be sorted grows in size.
Hence we can say that selection sort is not advisable for larger lists of data.

```
template <typename T> int Routine_Smallest(T A[], int K, const int arrSize);
```

```
//General Algorithm
//Selection Sort (Array, Size of Array)
template <typename T>
void selectionSort(T arr[], const int N){
    int POS, temp, pass=0;
    //Step 1: Repeat Steps 2 and 3 for K = 1 to N-1
    for(int i = 0; i < N; i++){
        //Step 2: Call routine smallest(A, K, N,POS)
        POS = Routine_Smallest(arr, i, N);
        temp = arr[i];
        //Step 3: Swap A[K] with A [POS]
        arr[i] = arr[POS];
        arr[POS] = temp;
        //Count
        pass++;
    }
    //[End of loop]
    //Step 4: EXIT
}
```

```
//Routine smallest (Array, Current Position, Size of Array)
template <typename T>
int Routine_Smallest(T A[], int K, const int arrSize){
    int position, j;
    //Step 1: [initialize] set smallestElem = A[K]
    T smallestElem = A[K];
    //Step 2: [initialize] set POS = K
    position = K;
    //Step 3: for J = K+1 to N -1,repeat
    for(int J=K+1; J < arrSize; J++){
        if(A[J] < smallestElem){
            smallestElem = A[J];
```

```
                position = J;
            }
        }
        //Step 4: return POS
        return position;
    }
```

Include a screenshot of your code and output console along with your observations in table 7-3.

A.3. Insertion Sort Algorithm

In the insertion sort technique, we start from the second element and compare it with the first element and put it in a proper place. Then we perform this process for the subsequent elements. We compare each element with all its previous elements and put or insert the element in its proper position. Insertion sort technique is more feasible for arrays with a smaller number of elements. It is also useful for sorting linked lists. Linked lists have a pointer to the next element (in case of a singly linked list) and a pointer to the previous element as well (in case of a doubly linked list). Hence it becomes easier to implement insertion sort for a linked list.

General Algorithm

```
//General Algorithm
//Insertion Sort
template <typename T>
void insertionSort(T arr[], const int N){
    int K = 0, J, temp;
    //Step 1: Repeat Steps 2 to 5 for K = 1 to N-1
    while(K < N){
        //Step 2: set temp = A[K]
        temp = arr[K];
        //Step 3: set J = K - 1
        J = K-1;
        //Step 4: Repeat while temp <=A[J]
        while(temp <= arr[J]){
            //set A[J + 1] = A[J]
            arr[J+1] = arr[J];
            //set J = J - 1
            J--;
            //[end of inner loop]
        }
        //Step 5: set A[J + 1] = temp
        arr[J+1] = temp;
        //[end of loop]
        K++;
    }
    //Step 6: exit
}
```

Include a screenshot of your code and output console along with your observations in table 7-4.

6. Output

Code + Console Screenshot	
Observations	

Table 7-1. Array of Values for Sort Algorithm Testing

Code + Console Screenshot	
Observations	

Table 7-2. Bubble Sort Technique

Code + Console Screenshot	
Observations	

Table 7-3. Selection Sort Algorithm

Code + Console Screenshot	
Observations	

Table 7-4. Insertion Sort Algorithm

7. Supplementary Activity

ILO B: Solve given data sorting problems using appropriate basic sorting algorithms

Candidate 1	Bo Dalton Capistrano
Candidate 2	Cornelius Raymon Agustín
Candidate 3	Deja Jayla Bañaga
Candidate 4	Lalla Brielle Yabut
Candidate 5	Franklin Relano Castro

List of Candidates

Problem: Generate an array A[0…100] of unsorted elements, wherein the values in the array are indicative of a vote to a candidate. This means that the values in your array must only range from 1 to 5. Using sorting and searching techniques, develop an algorithm that will count the votes and indicate the winning candidate.

NOTE: The sorting techniques you have the option of using in this activity can be either bubble, selection, or insertion sort. Justify why you chose to use this sorting algorithm.

Deliverables:

- Pseudocode of Algorithm
- Screenshot of Algorithm Code
- Output Testing

Output Console Showing Sorted Array	Manual Count	Count Result of Algorithm

Question: Was your developed vote counting algorithm effective? Why or why not?

8. Conclusion

Provide the following:

- Summary of lessons learned
- Analysis of the procedure
- Analysis of the supplementary activity
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?

9. Assessment Rubric