ACTIVITY NO. 7

SORTING ALGORITHMS: BUBBLE, SELECTION, AND INSERTION SORT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 09/18/25**
Section: CPE21S4	**Date Submitted: 09/18/25**
Name: Cruz, Axl Waltz E.	**Instructor:** Engr. Jimlord Quejado

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.

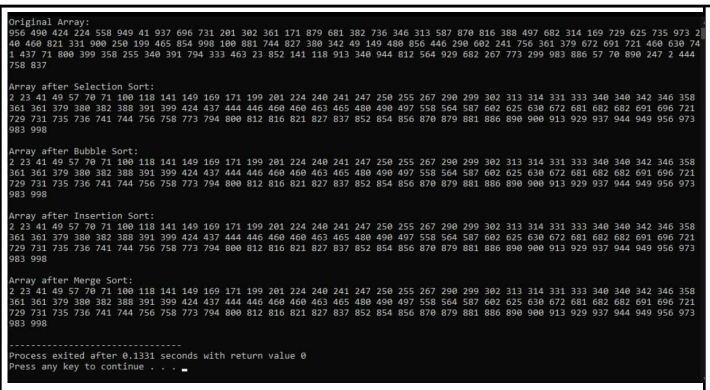
6. Output

Code + Console Screenshot	Table 7-1. Array of Values for Sort Algorithm Testing

```
#ifndef SORT_ALGORITHMS_H
#define SORT_ALGORITHMS_H

void copyArray(int source[], int destination[], int size) {
    for (int i = 0; i < size; i++) {
        destination[i] = source[i];
    }
}

void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}

void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void insertionSort(int arr[], int size) {
    for (int i = 1; i < size; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        arr[j + 1] = key;
    }
}
```

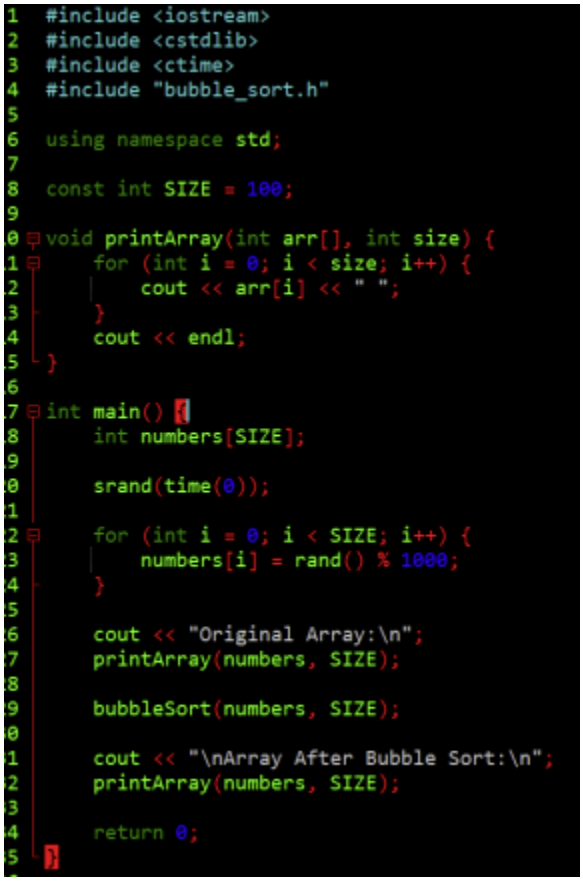
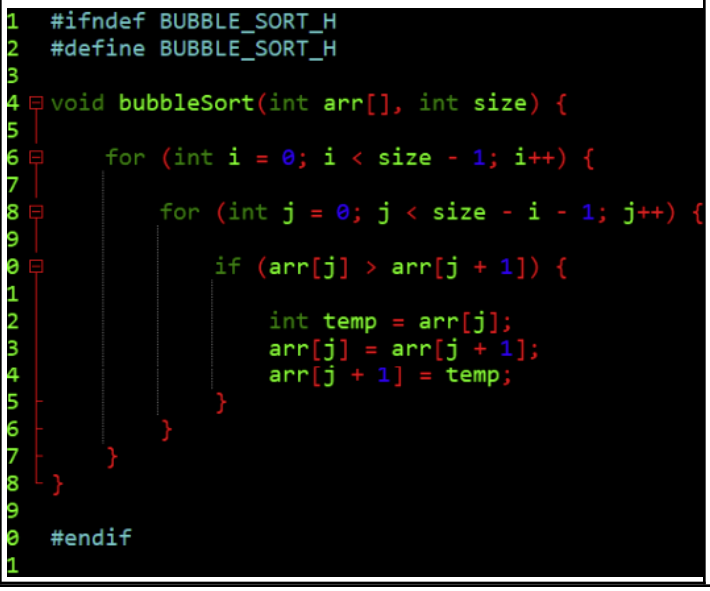
```
void merge(int arr[], int left, int middle, int right) {
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int L[100], R[100];

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];

    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];

    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++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int middle = (left + right) / 2;

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

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

#endif
```

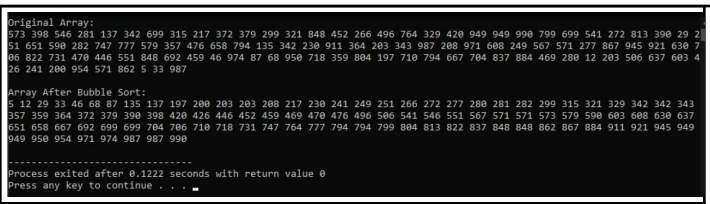
```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "sort_algorithms.h"

using namespace std;

const int SIZE = 100;

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int original[SIZE];
    int arr[SIZE];

    srand(time(0));

    for (int i = 0; i < SIZE; i++) {
        original[i] = rand() % 1000;
    }

    cout << "Original Array:\n";
    printArray(original, SIZE);

    copyArray(original, arr, SIZE);
    selectionSort(arr, SIZE);
    cout << "\nArray after Selection Sort:\n";
    printArray(arr, SIZE);

    copyArray(original, arr, SIZE);
    bubbleSort(arr, SIZE);
    cout << "\nArray after Bubble Sort:\n";
    printArray(arr, SIZE);

    copyArray(original, arr, SIZE);
    insertionSort(arr, SIZE);
    cout << "\nArray after Insertion Sort:\n";
    printArray(arr, SIZE);

    copyArray(original, arr, SIZE);
    mergeSort(arr, 0, SIZE - 1);
    cout << "\nArray after Merge Sort:\n";
    printArray(arr, SIZE);

    return 0;
}
```



Observations	My observation on this code is compares four sorting algorithms using the same set of random values. It shows how each sort affects the array, making it easy to see differences in sorting results. The structure is clear and easy to follow for learning purposes.

Table 7-2. Bubble Sort Technique

Code + Console Screenshot	 

	
Observations	My observation in this code is uses bubble sort to arrange 100 random numbers in ascending order. It prints the original and sorted arrays, making it easy to understand how the sorting changed the data. While bubble sort is simple and not the fastest, it works well here for demonstrating how sorting algorithms function.

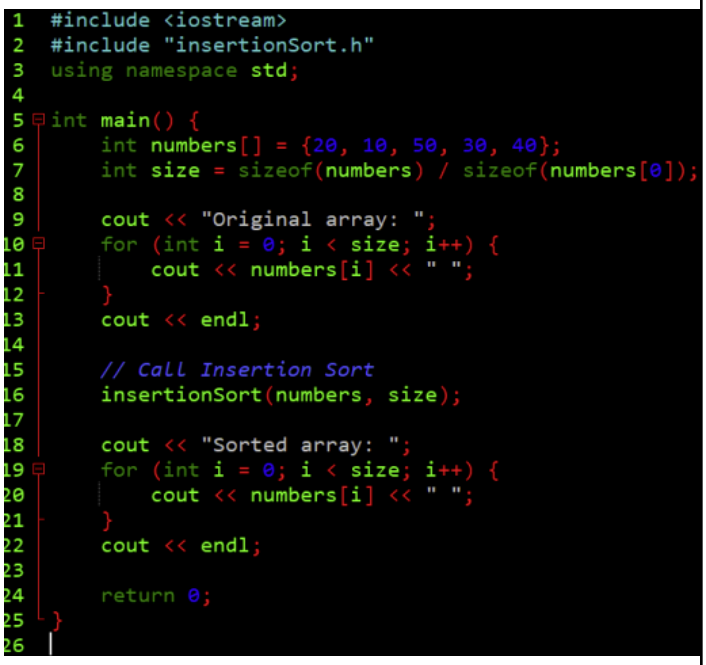
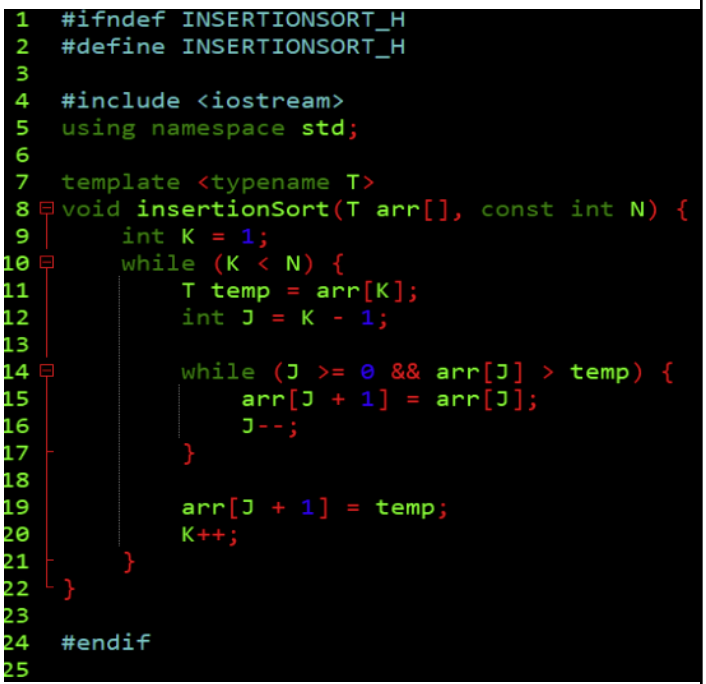
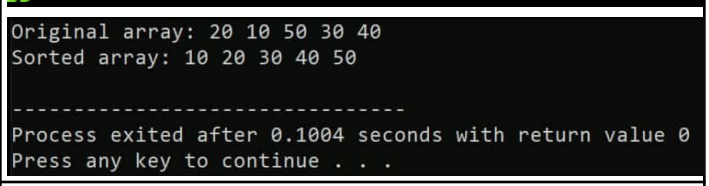
Code + Console Screenshot	Table 7-3. Selection Sort Algorithm

```cpp
#include <iostream>
#include "selectionSort.h"
using namespace std;

int main() {
    int numbers[] = {20, 10, 50, 30, 40};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    cout << "Original array: ";
    for(int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    selectionSort(numbers, size);

    cout << "Sorted array: ";
    for(int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    return 0;
}
```

```cpp
#ifndef SELECTIONSORT_H
#define SELECTIONSORT_H

#include <iostream>
using namespace std;

// Routine to find the smallest element position
template <typename T>
int Routine_Smallest(T A[], int K, const int arrSize) {
    int position = K;
    T smallestElem = A[K];

    for(int J = K + 1; J < arrSize; J++) {
        if(A[J] < smallestElem) {
            smallestElem = A[J];
            position = J;
        }
    }
    return position;
}

// Selection Sort
template <typename T>
void selectionSort(T arr[], const int N) {
    int POS, pass = 0;

    for(int i = 0; i < N; i++) {
        POS = Routine_Smallest(arr, i, N);

        // Swap arr[i] and arr[POS]
        T temp = arr[i];
        arr[i] = arr[POS];
        arr[POS] = temp;

        pass++;
    }
}

#endif
```

```
10 20 30 40 50
--------------------------------
Process exited after 0.09899 seconds with return value 0
Press any key to continue . . .
```

Observations	My observation in this part is the program uses a header file that defines a routine to find the smallest element and a selection sort template function. It prints both the original and sorted array, allowing you to clearly see the effect of the sort. This example demonstrates clean modular programming by separating the sorting logic into its own header file, making the code easier to maintain and reuse.

Code + Console Screenshot	Table 7-4. Insertion Sort Algorithm   
Observations	I observe here was the program demonstrates sorting an array

```cpp
#include <iostream>
#include "insertionSort.h"
using namespace std;

int main() {
    int numbers[] = {20, 10, 50, 30, 40};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    cout << "Original array: ";
    for (int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    // Call Insertion Sort
    insertionSort(numbers, size);

    cout << "Sorted array: ";
    for (int i = 0; i < size; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    return 0;
}
```

```cpp
#ifndef INSERTIONSORT_H
#define INSERTIONSORT_H

#include <iostream>
using namespace std;

template <typename T>
void insertionSort(T arr[], const int N) {
    int K = 1;
    while (K < N) {
        T temp = arr[K];
        int J = K - 1;

        while (J >= 0 && arr[J] > temp) {
            arr[J + 1] = arr[J];
            J--;
        }

        arr[J + 1] = temp;
        K++;
    }
}

#endif
```

```
Original array: 20 10 50 30 40
Sorted array: 10 20 30 40 50

--------------------------------
Process exited after 0.1004 seconds with return value 0
Press any key to continue . . .
```

	using insertion sort, with the algorithm defined in a separate header file. It displays the original unsorted array, performs the sort, and then shows the sorted output. This version highlights a different algorithm while still keeping the modular structure, making the code easy to organize and understand.

7. Supplementary Activity

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

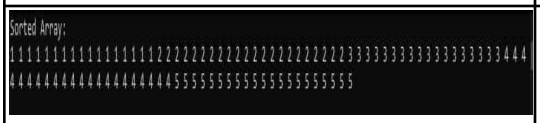
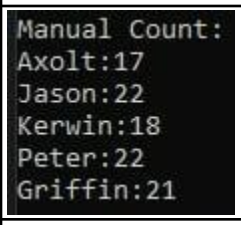
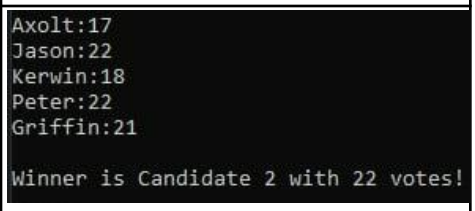
List of Candidates

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
		

- The code uses selection sort to arrange votes and counts how many each candidate received. It clearly identifies the winner based on the highest vote count. The logic is simple and works well for small datasets like this.

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

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

- Yes, the developed vote counting algorithm was effective. It correctly generated random votes, sorted them using a simple algorithm (insertion sort), and accurately counted each candidate's total using conditional checks. Sorting the array first made it easier to manually verify the results, and the logic was clear, efficient for the data size, and easy to understand and debug.

8. Conclusion

Provide the following:

- Summary of lessons learned
  - I learned how to apply basic sorting algorithms such as Bubble Sort, Selection Sort, and Insertion Sort in C++. These algorithms helped me understand how data can be arranged for easier analysis and comparison. I also learned how sorting can support other tasks like searching, counting, and organizing data clearly.
- Analysis of the procedure
  - The activity involved generating random data, applying multiple sorting techniques, and displaying results in a clear format. Using a header file made the code more organized and reusable across different parts of the program. Each sorting method showed different behaviors, which helped me compare their strengths and weaknesses.
- Analysis of the supplementary activity
  - The vote-counting task showed how sorting could assist in analyzing grouped data more easily. By sorting the votes first, I was able to count efficiently using simple conditional logic. The activity proved that even basic algorithms can solve real-world problems when applied properly.
- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?
  - I think I did well in completing the activity by writing clear, working code and understanding the logic behind each step. One area I can improve is optimizing the code and reducing repetition in certain sections. Overall, the activity helped me build confidence in using C++ for basic algorithm implementation and problem-solving.

9. Assessment Rubric