Activity No. 7.1	
Sorting Algorithms Pt1	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 9/18/2025**
Section: CPE21S4	**Date Submitted: 9/18/2025**
Name(s): Anastacio, Lester Arvid P.	**Instructor: Engr. Jimlord M. Quejado**

6. Output

Table 7-1:

Full Code:

sorting_algorithms.h | Main1.cpp

```cpp
#include <iostream>
#include <cstdlib> // this is what we will be using for random values
#include <ctime> // this is for the function time()
#include <algorithm> // for the std::copy for it to be able to copy the rand values of elements within the array
#include "sorting_algorithms.h" // the header file

const int SIZE = 100;

// This is the function to print out the array
template <typename T>
void printArray(const T arr[], int size){
    for (int i = 0; i < size; i++){
        std::cout << arr[i] << " ";
        if ((i+1) % 20 == 0) std::cout << std::endl; // this will print out 20 elements in each line
    }
    std::cout << std::endl;
}

int main(){
    // For seeding the random number generator
    std::srand(static_cast<unsigned int>(std::time(nullptr)));

    // This function creates and fill the array with random values which ranges from 0 to 999
    int org_Array[SIZE];
    for (int i = 0; i < SIZE; i++){
        org_Array[i] = std::rand() % 1000;
    }

    // This will print out the unsorted array for the comparison to the sorted one
    std::cout << "Unsorted Array:\n";
    printArray(org_Array, SIZE);


    // Create copies for each sorting algorithm
    int bubbleArray[SIZE];
    int selectionArray[SIZE];
    int insertionArray[SIZE];

    //this will copy the array generated randomly
    std::copy(org_Array, org_Array + SIZE, bubbleArray);
    std::copy(org_Array, org_Array + SIZE, selectionArray);
    std::copy(org_Array, org_Array + SIZE, insertionArray);

    // Sort using Bubble Sort
    bubbleSort(bubbleArray, SIZE);
    std::cout << "\nBubble Sorted Array:\n";
    printArray(bubbleArray, SIZE);

    // Sort using Selection Sort
    selectionSort(selectionArray, SIZE);
    std::cout << "\nSelection Sorted Array:\n";
    printArray(selectionArray, SIZE);

    // Sort using Insertion Sort
    insertionSort(insertionArray, SIZE);
    std::cout << "\nInsertion Sorted Array:\n";
    printArray(insertionArray, SIZE);

    return 0;
}
```

Parts by Part:

```cpp
#include <iostream>
#include <cstdlib> // this is what we will be using for random values
#include <ctime> // this is for the function time()
#include <algorithm> // for the std::copy for it to be able to copy the rand values of elements within the array
#include "sorting_algorithms.h" // the header file
```

- This is the libraries that I used to create the main program, this contains the means to create the random generation of numbers and an easier way to be able to copy arrays.

```cpp
const int SIZE = 100;
```

- this line here defines the constant integer names SIZE to be set to 100, in which represent the number of elements that can be inputted within the array.

```cpp
// This is the function to print out the array
template <typename T>
void printArray(const T arr[], int size){
    for (int i = 0; i < size; i++){
        std::cout << arr[i] << " ";
        if ((i+1) % 20 == 0) std::cout << std::endl; // this will print out 20 elements in each line
    }
    std::cout << std::endl;
}
```

- This part here is the function in which prints the elements of any array that it passes, it loops through the array and outputs each element by spaces and in every 20 elements this will print a newline for a much better readability.

```cpp
18
19  int main(){
20      // For seeding the random number generator
21      std::srand(static_cast<unsigned int>(std::time(nullptr)));
22
```

- This function here is where the random number generator is seeded using the current system time, std::time(nullptr), this function ensures the sequence of random numbers differs on each run in which prevents the program to generate the same set of random values.

```cpp
    // This function creates and fill the array with random values which ranges from 0 to 999
    int org_Array[SIZE];
    for (int i = 0; i < SIZE; i++){
        org_Array[i] = std::rand() % 1000;
    }

    // This will print out the unsorted array for the comparison to the sorted one
    std::cout << "Unsorted Array:\n";
    printArray(org_Array, SIZE);
```

- In this function is where the array is created and filled in, the for loop fills in the array with random integers that ranges from 0 to 999 using the modulo operator on the rand() function, this is where the array serves as the initial unsorted data set is found and below that said function is where it will print out the contents of the org_Array.

```
// Create copies for each sorting algorithm
int bubbleArray[SIZE];
int selectionArray[SIZE];
int insertionArray[SIZE];

//this will copy the array generated randomly
std::copy(org_Array, org_Array + SIZE, bubbleArray);
std::copy(org_Array, org_Array + SIZE, selectionArray);
std::copy(org_Array, org_Array + SIZE, insertionArray);
```

- In this section, is where the arrays of the three which are the bubbleArray, selectionArray and insertionArray is to be filled by copying the contents of the org_Array using the function std::copy() of the algorithm library, this ensures each sorting function to work on the same unsorted data independently to prevent each other from affecting one another.

```
// Sort using Bubble Sort
bubbleSort(bubbleArray, SIZE);
std::cout << "\nBubble Sorted Array:\n";
printArray(bubbleArray, SIZE);

// Sort using Selection Sort
selectionSort(selectionArray, SIZE);
std::cout << "\nSelection Sorted Array:\n";
printArray(selectionArray, SIZE);

// Sort using Insertion Sort
insertionSort(insertionArray, SIZE);
std::cout << "\nInsertion Sorted Array:\n";
printArray(insertionArray, SIZE);
```

- And lastly, these are the functions to prints out the array that is sorted out by the three sorting algorithms within the header.

Output:

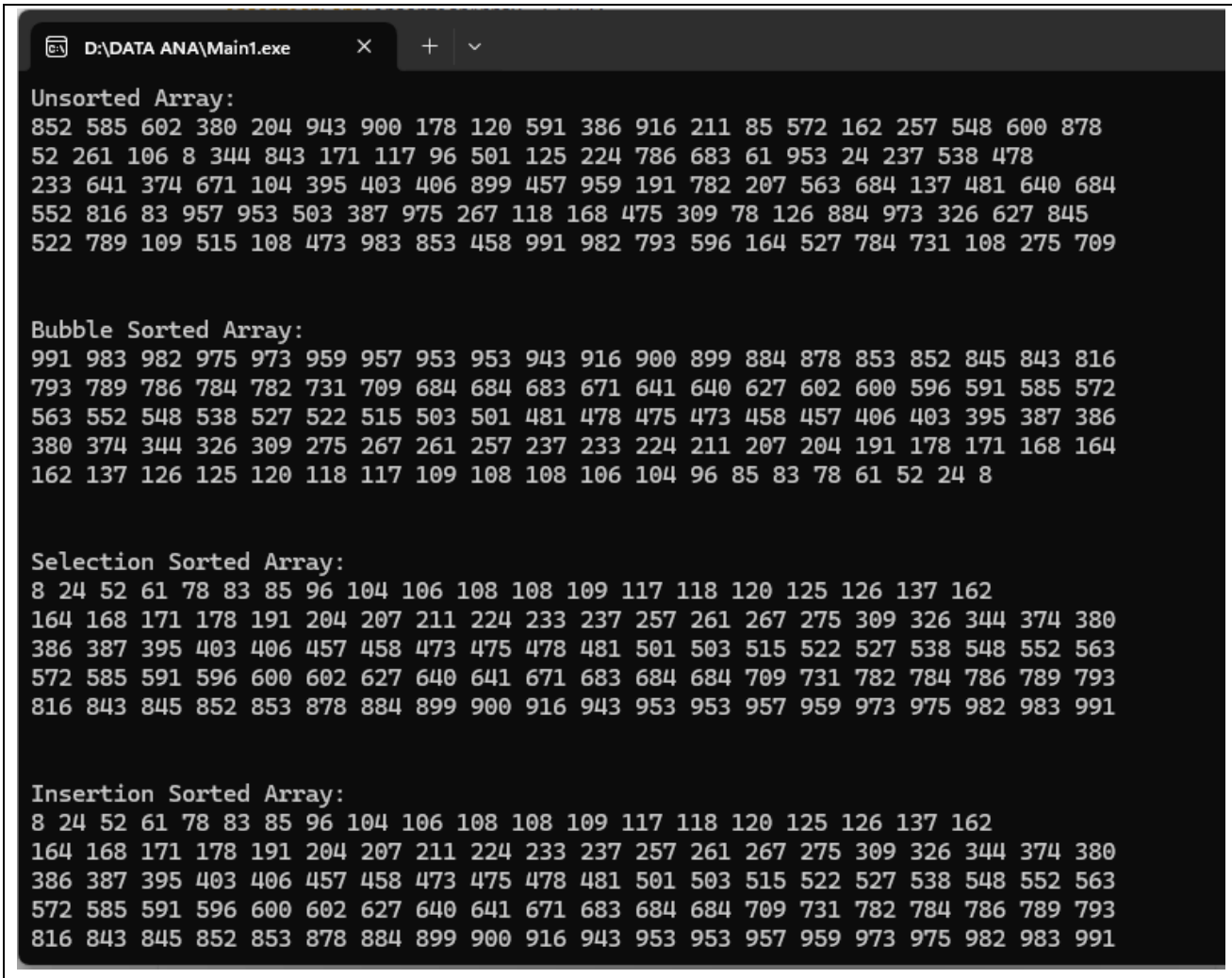

Table 7-2 – Bubble Sorting

Code:

```
#ifndef SORTING_ALGORITHMS_H
#define SORTING_ALGORITHMS_H
#include <iostream>

//Bubble Sorting
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 - 1; i++) {
        // Step 2: For j = i + 1 to N - 1 repeat
        for (int j = i + 1; j < arrSize; j++) {
            // Step 3: if arr[j] > arr[i], swap arr[j] and arr[i]
            if (arr[j] > arr[i]) {
                std::swap(arr[j], arr[i]);
            }
        }
        // [End of Inner for loop]
    }
    // [End of Outer for loop]
    // Step 4: Exit
}
```

Explanation:

- This is the template function of the bubblesort in which sorts the array in descending order, the way how this specific code works however is that it takes two parameters which is the array of arr[] of the type T and the

arrSize in which represents the total number of elements within the array, Inside the function, is the outer for loops runs from the I = 0 up to the arrSize -2, this means it makes N – 1 total passes over the array and with each pass is to intend to help find the correct position for one element by comparing it with all the elements that come after it and within this outer loop, there is a nested for loop that starts at j = I + 1 and runs up to the last index of the array and the inner loop is responsible for comparing the current element at the index I with every other elements that follows it in the array and during each comparison, if the element at the index j is greater than the element in the array the function will swap the two values.

Output:

```
Unsorted Array:
852 585 602 380 204 943 900 178 120 591 386 916 211 85 572 162 257 548 600 878
52 261 106 8 344 843 171 117 96 501 125 224 786 683 61 953 24 237 538 478
233 641 374 671 104 395 403 406 899 457 959 191 782 207 563 684 137 481 640 684
552 816 83 957 953 503 387 975 267 118 168 475 309 78 126 884 973 326 627 845
522 789 109 515 108 473 983 853 458 991 982 793 596 164 527 784 731 108 275 709


Bubble Sorted Array:
991 983 982 975 973 959 957 953 953 943 916 900 899 884 878 853 852 845 843 816
793 789 786 784 782 731 709 684 684 683 671 641 640 627 602 600 596 591 585 572
563 552 548 538 527 522 515 503 501 481 478 475 473 458 457 406 403 395 387 386
380 374 344 326 309 275 267 261 257 237 233 224 211 207 204 191 178 171 168 164
162 137 126 125 120 118 117 109 108 108 106 104 96 85 83 78 61 52 24 8
```

Table 7-3 – Selection Sorting

Code:

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

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

    // Step 1: Repeat Steps 2 and 3 for i = 0 to N-1
    for (int i = 0; i < N - 1; i++) {
        // Step 2: Call routine smallest(arr, i, N, POS)
        POS = Routine_Smallest(arr, i, N);
        // Step 3: Swap arr[i] with arr[POS]
        temp = arr[i];
        arr[i] = arr[POS];
        arr[POS] = temp;
        // Count pass
        pass++;
    }
    // [End of loop]
    // Step 4: EXIT
}

// Routine smallest (Array, Current Position, Size of Array)
```

```
// Routine smallest (Array, Current Position, Size of Array)
template <typename T>
int Routine_Smallest(T arr[], int K, const int arrSize) {
    int position, j;
    // Step 1: Initialize smallestElem = arr[K]
    T smallestElem = arr[K];
    // Step 2: Initialize position = K
    position = K;
    // Step 3: For j = K+1 to N-1, repeat
    for (int j = K + 1; j < arrSize; j++) {
        if (arr[j] < smallestElem) {
            smallestElem = arr[j];
            position = j;
        }
    }
    // Step 4: Return position
    return position;
}
```

Explanation:

- The selectionSort function sorts an array in ascending order using the selection sort algorithm. It works by finding the smallest unsorted element of the array, and then swapping it with the element in the current position. The function loops to each position in the array, and for each position, the helper function Routine_Smallest is called to obtain the index of the smallest element in the unsorted part and then the Routine_Smallest function outlines an approach by assuming the first element of the subarray is the smallest and then goes through the array to locate the smallest and return its index and Constructing the sorted portion of the array to then placing it in the appropriate position for each of the passes is built by these functions. The algorithm is quite simple and out of the memory provided to it, it can sort the elements. Although it can be considered simple, it is rather inefficient with large amounts in the data set.

Output:

```
Unsorted Array:
852 585 602 380 204 943 900 178 120 591 386 916 211 85 572 162 257 548 600 878
52 261 106 8 344 843 171 117 96 501 125 224 786 683 61 953 24 237 538 478
233 641 374 671 104 395 403 406 899 457 959 191 782 207 563 684 137 481 640 684
552 816 83 957 953 503 387 975 267 118 168 475 309 78 126 884 973 326 627 845
522 789 109 515 108 473 983 853 458 991 982 793 596 164 527 784 731 108 275 709


Selection Sorted Array:
8 24 52 61 78 83 85 96 104 106 108 108 109 117 118 120 125 126 137 162
164 168 171 178 191 204 207 211 224 233 237 257 261 267 275 309 326 344 374 380
386 387 395 403 406 457 458 473 475 478 481 501 503 515 522 527 538 548 552 563
572 585 591 596 600 602 627 640 641 671 683 684 684 709 731 782 784 786 789 793
816 843 845 852 853 878 884 899 900 916 943 953 953 957 959 973 975 982 983 991
```

Table 7-4 – Insertion Sorting

Code:

```cpp
// Insertion Sort

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

    // Step 1: Repeat Steps 2 to 5 for K = 1 to N-1
    while (K < N) {
        // Step 2: Set temp = arr[K]
        temp = arr[K];
        // Step 3: Set J = K - 1
        J = K - 1;
        // Step 4: Repeat while temp < arr[J], shifting elements to the right
        while (J >= 0 && temp < arr[J]) {
            // Set arr[J + 1] = arr[J]
            arr[J + 1] = arr[J];
            // Set J = J - 1
            J--;
        }
        // Step 5: Set arr[J + 1] = temp
        arr[J + 1] = temp;
        // [End of inner loop]
        // [End of loop]
        K++;
    }

    // Step 6: Exit
}
```

Explanation:

- From index 1, the function insertionSort is a Templated function that sorts a provided array in ascending order by building a sorted section one element at a time, starting from index one, and comparing it p. The variable temp will be set at the current element and the function determines the position by shifting the larger elements to the right, nested with a while condition. The element if greater than temp gets shifted one position to the right to make space, and when the appropriate position is identified, places the stored element in that position. This is a nested condition and the same is done with the rest of the elements. The array is said to be completely sorted when every single element present has been sorted. This algorithm works in place and the sorting works efficiently with small or almost sorted datasets, however, longer, random datasets may take time.

Output:

```
Unsorted Array:
852 585 602 380 204 943 900 178 120 591 386 916 211 85 572 162 257 548 600 878
52 261 106 8 344 843 171 117 96 501 125 224 786 683 61 953 24 237 538 478
233 641 374 671 104 395 403 406 899 457 959 191 782 207 563 684 137 481 640 684
552 816 83 957 953 503 387 975 267 118 168 475 309 78 126 884 973 326 627 845
522 789 109 515 108 473 983 853 458 991 982 793 596 164 527 784 731 108 275 709


Insertion Sorted Array:
8 24 52 61 78 83 85 96 104 106 108 108 109 117 118 120 125 126 137 162
164 168 171 178 191 204 207 211 224 233 237 257 261 267 275 309 326 344 374 380
386 387 395 403 406 457 458 473 475 478 481 501 503 515 522 527 538 548 552 563
572 585 591 596 600 602 627 640 641 671 683 684 684 709 731 782 784 786 789 793
816 843 845 852 853 878 884 899 900 916 943 953 953 957 959 973 975 982 983 991
```

7. Supplementary Activity
Full Code:

Main:

```cpp
#include <iostream>
#include <cstdlib>  // For rand()
#include <ctime>    // For time()
#include "sorting_algorithms.h"

int main() {
    const int SIZE = 100;
    int votes[SIZE];

    // Candidate names for aesthetic purposes
    const char* candidateNames[5] = {
        "Bo Dalton Capistrano",
        "Cornelius Raymon Agustin",
        "Deja Jayla Bañaga",
        "Lalla Brielle Yabut",
        "Franklin Relano Castro"
    };

    // Seed random number generator
    std::srand(static_cast<unsigned int>(std::time(nullptr)));

    // Generate random votes from 1 to 5
    for (int i = 0; i < SIZE; i++) {
        votes[i] = (std::rand() % 5) + 1;  // Values from 1 to 5
    }

    // Sort votes using insertion sort
    insertionSort(votes, SIZE);

    // Count votes for each candidate (1 to 5)
    int voteCounts[5] = {0};

    for (int i = 0; i < SIZE; i++) {
        voteCounts[votes[i] - 1]++;
    }

    // Display vote counts with candidate names
    std::cout << "Vote counts per candidate:\n";
    for (int i = 0; i < 5; i++) {
        std::cout << "Candidate " << (i + 1) << " - " << candidateNames[i]
                  << ": " << voteCounts[i] << " votes\n";
    }

    // Find winning candidate
    int maxVotes = voteCounts[0];
    int winner = 1;
    for (int i = 1; i < 5; i++) {
        if (voteCounts[i] > maxVotes) {
            maxVotes = voteCounts[i];
            winner = i + 1;
        }
    }

    std::cout << "\nThe winning candidate is Candidate " << winner
              << " - " << candidateNames[winner - 1]
              << " with " << maxVotes << " votes.\n";

    return 0;
}
```

Explanation Parts by Parts:

```
int main() {
    const int SIZE = 100;
    int votes[SIZE];

    // Candidate names for aesthetic purposes
    const char* candidateNames[5] = {
        "Bo Dalton Capistrano",
        "Cornelius Raymon Agustin",
        "Deja Jayla Bañaga",
        "Lalla Brielle Yabut",
        "Franklin Relano Castro"
    };
```

- This section here is where it declares a constant SIZE to be set to 100 to represent the number of votes and the int vote[SIZE] is created to store the votes in a array and below it is an array of candidateNames which is added just to make the output look nicer.

```
    // Seed random number generator
    std::srand(static_cast<unsigned int>(std::time(nullptr)));

    // Generate random votes from 1 to 5
    for (int i = 0; i < SIZE; i++) {
        votes[i] = (std::rand() % 5) + 1;  // Values from 1 to 5
    }
```

- In this section, is where the random number generator is seeded with the current time just to ensure that it will always bring out a random values on each new program run and below it is a loop which fills in the votes array with that said random integers between the ranges of 1 and 5 to represent the votes for each candidate.

```
    // Count votes for each candidate (1 to 5)
    int voteCounts[5] = {0};

    for (int i = 0; i < SIZE; i++) {
        voteCounts[votes[i] - 1]++;
    }
```

- This section here is where an integer array named voteCounts with the size of 5 is initialized with zeros to keep tracj of the total votes of each candidate receives and below it is a for loop that iterates through the sorted votes array, incrementing the count for the corresponding candidate basing on the votes value adjusted by -1.

```
43
44      // Find winning candidate
45      int maxVotes = voteCounts[0];
46      int winner = 1;
47      for (int i = 1; i < 5; i++) {
48          if (voteCounts[i] > maxVotes) {
49              maxVotes = voteCounts[i];
50              winner = i + 1;
51          }
52      }
53
54      std::cout << "\nThe winning candidate is Candidate " << winner
55                << " - " << candidateNames[winner - 1]
56                << " with " << maxVotes << " votes.\n";
57
```

- In this section, this is where it finds the candidate with the highest vote count, it first initializes maxVotes with the votes of the first candidate and winner with candidate 1 and then it loops through the remaining candidates updating the maxVotes and winner whenever a candidate has a higher vote count and then finally the std::cout

prints out the winner among the candidates, first is his candidate number, then his name and with the amount of votes the candidate achieved.

Reason:

- To do the task, I used the insertion sorting algorithm it is because the array has only a few values of 1 to 5 in which I think the insertion sorting is good at since it performs well on arrays that are already almost sorted or have many repeated elements, it is also much more simpler and efficient since the array only has 100 elements which is easier to implement compared to bubble and selection sorting, and I also think that it is a lot more faster than the other two sorting algorithms since the bubble and selection does perform more comparisons and swaps than insertion sort.

Header Code:

```
#ifndef INSERTION_SORTING_H
#define INSERTION_SORTING_H

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

    // Repeat Steps 2 to 5 for K = 1 to N-1
    while (K < N) {
        temp = arr[K];   // Store current element
        J = K - 1;       // Start comparing with previous element

        // Shift elements that are greater than temp to the right
        while (J >= 0 && temp < arr[J]) {
            arr[J + 1] = arr[J];
            J--;
        }

        // Insert temp into the correct position
        arr[J + 1] = temp;
        K++;
    }
}

#endif

```

Pseudocode:

```
START

DECLARE SIZE = 100
DECLARE votes[SIZE] as integer array
DECLARE candidateNames[5] as string array:
    [ "Bo Dalton Capistrano",
      "Cornelius Raymon Agustin",
      "Deja Jayla Bañaga",
      "Lalla Brielle Yabut",
      "Franklin Relano Castro" ]

INITIALIZE random seed using current time

// Generate 100 random votes between 1 and 5
FOR i from 0 to SIZE - 1 DO
    votes[i] ← RANDOM number from 1 to 5
END FOR

// Sort the array using insertion sort
CALL insertionSort(votes, SIZE)

// Count how many votes each candidate received
DECLARE voteCounts[5] and initialize all to 0

FOR i from 0 to SIZE - 1 DO
    vote ← votes[i]
    voteCounts[vote - 1] ← voteCounts[vote - 1] + 1
END FOR

// Display vote count per candidate
PRINT "Vote counts per candidate:"
FOR i from 0 to 4 DO
    PRINT "Candidate", i + 1, "-", candidateNames[i], ":", voteCounts[i], "votes"
END FOR

// Determine the winning candidate
DECLARE maxVotes ← voteCounts[0]
DECLARE winner ← 1

FOR i from 1 to 4 DO
    IF voteCounts[i] > maxVotes THEN
        maxVotes ← voteCounts[i]
        winner ← i + 1
    END IF
END FOR

// Display the winner
PRINT "The winning candidate is Candidate", winner, "-", candidateNames[winner - 1], "with", maxVotes, "votes."

END
```

Output:

```
D:\DATA ANA\activity7.1.exe
Vote counts per candidate:
Candidate 1 - Bo Dalton Capistrano: 23 votes
Candidate 2 - Cornelius Raymon Agustin: 26 votes
Candidate 3 - Deja Jayla Ba±aga: 19 votes
Candidate 4 - Lalla Brielle Yabut: 15 votes
Candidate 5 - Franklin Relano Castro: 17 votes

The winning candidate is Candidate 2 - Cornelius Raymon Agustin with 26 votes.

--------------------------------
Process exited after 1.021 seconds with return value 0
Press any key to continue . . .
```

8. Conclusion

- In this activity, we developed a program that simulates the voting system for five candidates, the learning process of this is quite an interesting journey since I did went through some helpful sites just to be able to understand and create this task with the use of insertion sort algorithm, it also doesn't prove that challenging since most of the changes involved is in the main.cpp rather than in the header whereas the algorithm are, all I did are just some simple modifications of the procedure code.

9. Assessment Rubric