Activity No. 7.1	
Sorting Algorithms Pt1	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: Sept. 18 2025**
Section:CPE21S4	**Date Submitted: Sept. 18 2025**
Name(s): Crishen Luper S. Pulgado	**Instructor: Sir Jimlord Quejado**

6. Output

Table 7-1:

Code + Console Screenshot

main.cpp sorts.h

```cpp
#include <iostream>
#include <cstdlib>
#include "sorts.h"


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


    for (int i = 0; i < SIZE; i++) {
        arr[i] = std::rand() % 100 + 1;
    }

    std::cout << "Unsorted Array: " << std::endl;
    for (int i = 0; i < SIZE; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << "\n\n";
```

```
Unsorted Array:
42 68 35 1 70 25 79 59 63 65 6 46 82 28 62 92 96 43 28 37 92 5 3 54 93 83 22 17 19 96 48 27 72 39 70 13 68 100 36 95 4 1
2 23 34 74 65 42 12 54 69 48 45 63 58 38 60 24 42 30 79 17 36 91 43 89 7 41 43 65 49 47 6 91 30 71 51 7 2 94 49 30 24 85
 55 57 41 67 77 32 9 45 40 27 24 38 39 19 83 30 42
```

Observations

Here, I've made the preparation for the arrays by using the cstdlib to generate a random number as it executes. I initiate the array size as 10 and added a for loop which adds a random number element that ranges to 0-199 but I added 1 so it is 0 – 100 numbers. Then, I printed the numbers using a for loop.

Table 7-2:

Code + Console Screenshot

main.cpp | sorts.h

```cpp
#ifndef SORTS_H
#define SORTS_H
#include <algorithm>

template< typename T>
void bubbleSort(T arr[], size_t arrSize){

    for(size_t i = 0; i < arrSize; i++){

        for (size_t j = i + 1; j < arrSize; j++){

            if (arr[j] < arr[i]){
                std::swap(arr[j], arr[i]);
            }
        }
    }
}
#endif
```

```cpp
bubbleSort(arr, SIZE);
std::cout << "Sorted Array (Bubble):\n";
for (int i = 0; i < SIZE; i++){
    std::cout << arr[i] << " ";
}
std::cout << std::endl;
```

```
Sorted Array (Bubble):
1 2 3 4 5 6 6 7 7 9 12 12 13 17 17 19 19 22 23 24 24 24 25 27 27 28 28 30 30 30 30 32 34 35 36 36 37 38 38 39 39 40 41 4
1 42 42 42 42 43 43 43 45 45 46 47 48 48 49 49 51 54 54 55 57 58 59 60 62 63 63 65 65 65 67 68 68 69 70 70 71 72 74 77 7
9 79 82 83 83 85 89 91 91 92 92 93 94 95 96 96 100
```

Observations

Here, I created the sort.h file to put all the sorting function. In the function bubbleSort, it takes the elements and the array size. For the loop of the function, the outer loop controls the passes or iterations. With every pass, it is ensured that at least one element (the biggest/smallest, based on comparison) has been placed in the right position. The i is the current position in the array that we want to put the correct element info. The inner loop starts from i+1 and compares all elements after i with the element at i. It finds the smallest element and swaps it forward. The condition statement compares the current element arr[i] with the later element which is arr[j]. If the element later is smaller, then they are swapped. For example, the elements 5,2,7,8,1 in the random number. The first pass is the first index which is i=0 and it is compare with each later element. So, the 5, which is the current index we are comparing, is swap since 5 is greater than 2. The current i=0 is 2 since they are swapped. Then we compare again with 7, so no swap. Then, compare it to 8, no swap. Then compare it to 1, which it will be swap position and 1 will be the first index and 2 will be the last. So, the array will be like {1,5,7,8,2}. Next it will compare the next index which is 5 through the arrays and repeat the same process until it is sorted.

Table 7-3:

Code + Console Screenshot

```cpp
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;
}
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 passes
        pass++;
    }
    //Step 4: EXIT
}

    selectionSort(arr, SIZE);
    std::cout << "Sorted array (Selection):\n";
    for (int i = 0; i < SIZE; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << "\n";

    return 0;
```

```
Sorted array (Selection):
1 2 3 4 5 6 6 7 7 9 12 12 13 17 17 19 19 22 23 24 24 24 25 27 27 28 28 30 30 30 30 32 34 35 36 36 37 38 38 39 39 40 41 4
1 42 42 42 42 43 43 43 45 45 46 47 48 48 49 49 51 54 54 55 57 58 59 60 62 63 63 65 65 65 67 68 68 69 70 70 71 72 74 77 7
9 79 82 83 83 85 89 91 91 92 92 93 94 95 96 96 100
```

Observations

I observed that the Routine_smallest finds the smallest element from position K and starts assuming that the smallest is at A[K]. It has a loop that starts from the element right after K which is the J = K+1. It goes all the way to the end of the array which is arrSize-1. If a smaller element has found, it updates the smallestElem and its position. Meanwhile, the selectionSort has a loop that foes from the first element, which is i=0, to the last element, which is the i = N-1. It assumes that arr[i] is the current position where the next smallest element should be placed. It then calls the routine_smallest to find the position of the smallest element in the unsorted array. Then, it swaps the element at i with the element at POS. Thus, the routine_smallest makes it the inner loop and the selection sort makes it the outer loop. So, for example is that we have in the random number in an array has [8,2,12,14,6]. We start at i=0 which is the first position then we call the routine smallest to find the smallest number

	in the entire array which makes it 2 the smallest. Then, the 8 and 2 position's will swap and will have an order like this [2,8,12,14,6]. Then, it will start at second positon i=1 and calls again the routine_smallest and scans the 8,12,14,6. After finding the smallest, it will swap the 6 and 8 postion so it will be [2,6,12,14,8]. This will be repeated until it is sorted.

Table 7-4:

Code + Console Screenshot

```
template <typename T>
void insertionSort(T arr[], const int N){
    int K = 0, J, temp;
    while(K < N){
        temp = arr[K];
        J = K-1;

        while(temp <= arr[J]){
            arr[J+1] = arr[J];
            J--;
        }

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

```
    insertionSort(arr,SIZE);
    std::cout << "Sorted array (insertion sort):\n";
    for (int i=0; i<SIZE;i++){
        std::cout << arr[i] <<" ";
    }
    std::cout << "\n";

    return 0;
}
```

```
Sorted array (insertion sort):
1 2 3 4 5 6 6 7 7 9 12 12 13 17 17 19 19 22 23 24 24 24 25 27 27 28 28 30 30 30 30 32 34 35 36 36 37 38
1 42 42 42 42 43 43 43 45 45 46 47 48 48 49 49 51 54 54 55 57 58 59 60 62 63 63 65 65 65 67 68 68 69 70
9 79 82 83 83 85 89 91 91 92 92 93 94 95 96 96 100
```

Observations

I observe that the function insertionSort has a while loop which starts from K=0 and goes up to N-1. At each step, it picks up the current element and inserts it into the sorted part of the array which is the left side. It will temporarily store the value of the element we want to insert in the correct place. So, for example is there is [6,2,12,1] when K=1, we take the 2 and insert it into [6]; when K=2, we take the 12 and insert into [2,6]. The inner loop will run if J is greater than or equal to 0, and the element at arr[J] one step to the right. After each shift, decrease J to check the next element on the left. So, for example is we insert 1 in [2,6,12], it compares 1 starting from right to left. Once the inner loop finishes, at -1 meaning temp is the smallest so far so it will insert at the front. At some index where arr[j] <= temp meaning we found the right position.

Supplementary:
START

Declare an array of integers
Declare variables: n (size), i, j, temp
Declare counters: swaps = 0, comparisons = 0

// ----- Input -----
Ask user for number of elements (n)
For i from 0 to n-1:

```
    Input array[i]

For i from 0 to n-2:
    For j from 0 to n-i-2:
        comparisons = comparisons + 1
        If array[j] > array[j+1]:
            Swap array[j] and array[j+1]
            swaps = swaps + 1

Display the sorted array
Display total number of comparisons and swaps

Ask user for search target (key)
Set found = false
For i from 0 to n-1:
    If array[i] == key:
        found = true
        Display "Element found at index i"
        Break

If found == false:
    Display "Element not found"

// ----- End -----
END
```

```cpp
#ifndef VOTING_H
#define VOTING_H
#include <cstddef>
#include <utility>
#include <iostream>
template <typename T>
void bubbleSort(T arr[], size_t arrSize) {
    for (size_t i = 0; i < arrSize; i++) {
        for (size_t j = 0; j < arrSize - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}
void countVotes(const int arr[], int size, int counts[], int candidateCount) {

    for (int i = 0; i < candidateCount; i++) {
        counts[i] = 0;
    }

    for (int i = 0; i < size; i++) {
        if (arr[i] >= 1 && arr[i] <= candidateCount) {
            counts[arr[i] - 1]++;
        }
    }
}
int findWinner(const int counts[], int candidateCount) {
    int maxIndex = 0;
    for (int i = 1; i < candidateCount; i++) {
        if (counts[i] > counts[maxIndex]) {
            maxIndex = i;
        }
    }
    return maxIndex;
}
```

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "voting.h"
using namespace std;

int main() {
    const int SIZE = 100;
    const int CANDIDATES = 5;
    int votes[SIZE];

    for (int i = 0; i < SIZE; i++) {
        votes[i] = rand() % CANDIDATES + 1;
    }

    cout << "Unsorted Votes:\n";
    for (int i = 0; i < SIZE; i++) {
        cout << votes[i] << " ";
    }
    cout << "\n\n";

    bubbleSort(votes, SIZE);

    cout << "Sorted Votes:\n";
    for (int i = 0; i < SIZE; i++) {
        cout << votes[i] << " ";
    }
    cout << "\n\n";

    int counts[CANDIDATES];
    countVotes(votes, SIZE, counts, CANDIDATES);

    cout << "Vote Counts (Result of the Algorithm):\n";
    for (int i = 0; i < CANDIDATES; i++) {
        cout << "Candidate " << (i + 1) << ": " << counts[i] << " votes\n";
    }
```

```
Unsorted Votes:
2 3 5 1 5 5 4 4 3 5 1 1 2 3 2 2 1 3 3 2 2 5 3 4 3 3 2 2 4 1 3 2 2 4 5 3 3 5 1 5 4 2 3 4 4 5 2 2 4 4 3 5 3 3 3 5 4 2 5 4
2 1 1 3 4 2 1 3 5 4 2 1 1 5 1 1 2 2 4 4 5 4 5 5 2 1 2 2 2 4 5 5 2 4 3 4 4 3 5 2

Sorted Votes:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

Vote Counts (Result of the Algorithm):
Candidate 1: 14 votes
Candidate 2: 25 votes
Candidate 3: 20 votes
Candidate 4: 21 votes
Candidate 5: 20 votes

Winner: Candidate 2

--------------------------------
Process exited after 0.1061 seconds with return value 0
Press any key to continue . . .
```

List of candidates	Manual Count	Algorithm count
Bo Dalton Capistrano	14	14
Cornelius Raymon Agustín	25	25
Deja Jayla Bañaga	20	20
Lalla Brielle Yabut	21	21
Franklin Relano Castro	20	20

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

My chosen algorithm is bubble sort which is effective for this scenario. It is because that the elements or data are small, it only swaps when elements are out of order, the list of elements is almost sorted so it will do few swaps since we only have 1,2,3,4,5 only. If using other sorting algorithms such as selection sort, it does unnecessary swaps which one per pass even if it is already sorted. Insertion is not also a good choice since it does more shifting operations than bubble sort. Therefore, bubble swap is effective for this scenario since it allows few operations and efficient for small data. For the searching technique, it is more applicable to choose linear search even if the elements are sorted out already since there is only 100 votes and we just want to know if a number exist or how many times does a number appears. Binary search can only find the one occurrence quickly so it would require to go left and right to count duplicates. Unlike, linear searching counts the elements all in one simple pass.

7. Supplementary Activity

8. Conclusion

To conclude, there are different sorting techniques that are suitable for a certain scenario. Bubble sorting is sorted in passes or iteration and it will sort out first in the last index. Selection sorts by finding the smallest element and placing it in the right position by comparing it. Insertion is sorted by sorting the next and previous element and putting it in the right position or in other words it will sort out the previous elements. The procedures use for loops and while loops and it is mostly done by a nested loop. Some would an outer loop for the current position to compare and the inner loop to find the smallest number. The supplementary is very hard since you have to consider which is the best sorting technique for the scenario and bubble sort is the best for it since operates fewer than the other sorting techniques. In this activity, I think I did well for this since I somehow understand the concept of each sorting techniques but I still need to improve my implementation since I’ve really had a hard time for it.

9. Assessment Rubric