Hands-on Activity 7.1	
Sorting Algorithms	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/16/2024
Section: CPE21S1	**Date Submitted:** 10/17/2024
Name(s): Bonifacio, Redj Guillian F.	**Instructor:** Sayo, Maria Rizette

6. Output

Code + Console Screenshot

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

int main() {
    const int maximum = 100;
    int numbers[maximum];

    srand(time(0));

    for (int i = 0; i < maximum; i++) {
        numbers[i] = rand() % 100;
    }

    for (int j = 0; j < maximum; j++) {
        std::cout << numbers[j] << " ";
    }

    std::cout << std::endl;

    return 0;
}

```

```
/tmp/2x59Yv3OxW.o
79 19 86 79 3 89 72 74 36 8 93 51 59 36 62 30 44 53 25 94 90 75 5 0 81 33 52 19 82 40 83
    61 59 21 92 15 62 65 41 99 73 34 50 32 71 65 15 67 70 92 61 60 19 19 13 52 4 65 24 86
    57 59 0 68 80 92 83 43 57 25 42 31 11 44 15 82 61 82 50 31 26 63 44 97 82 57 2 86 74
    26 25 31 85 25 51 17 17 35 60 75


=== Code Execution Successful ===
```

Observations	This code creates an array of 100 random numbers, ranging from 0 to 99. It starts by setting up the necessary libraries and uses the current time to ensure the numbers are different each time you run it. Then, it fills the array with random values and prints them out

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

Code + Console Screenshot

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

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;
}

template <typename T>
void selectionSort(T arr[], const int N) {
    for (int i = 0; i < N - 1; i++) {
        int POS = Routine_Smallest(arr, i, N);
        std::swap(arr[i], arr[POS]);
    }
}

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

int main() {
    const int arraySize = 100;
    int arr[arraySize];
    srand(time(0));

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

    std::cout << "Original Array:\n";
    printArray(arr, arraySize);

    selectionSort(arr, arraySize);

    std::cout << "Sorted Array:\n";
    printArray(arr, arraySize);

    return 0;
}
```

	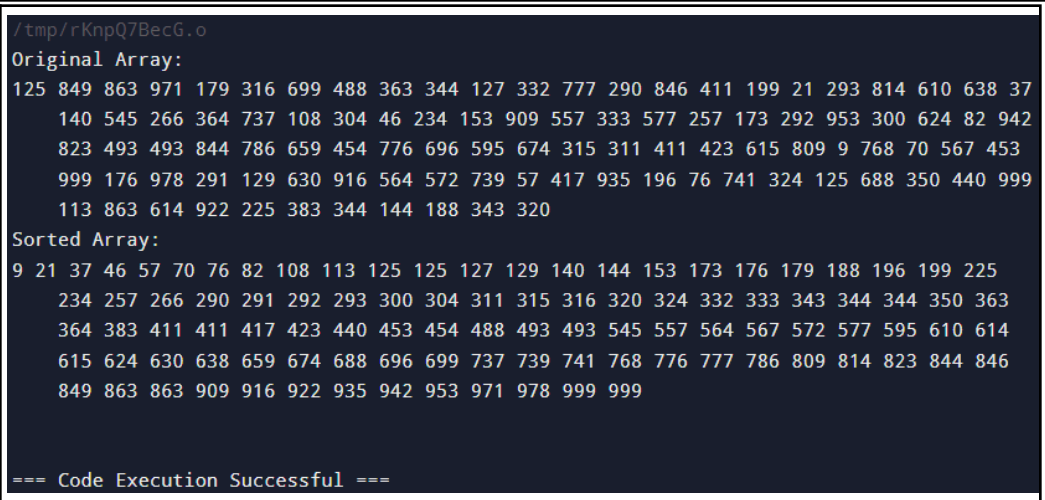
Observations	This program creates an array of 100 random integers, prints the original unsorted array, then sorts it using a selection sort algorithm template. After sorting, it displays the sorted array, illustrating the use of templates to handle sorting for different data types.

Table 7-2. Bubble Sort Technique

Code + Console Screenshot

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int routineSmallest(int arr[], int k, int arr_size){
    int position, j;
    int smallestElem = arr[k];
    position = k;

    for(int j = k + 1; j < arr_size; j++){
        if(arr[j] < smallestElem){
            smallestElem = arr[j];
            position = j;
        }
    }
    return position;
}

void selectionSort(int arr[], int arr_size){
    int pos, temp, pass = 0;
    for(int i = 0; i < arr_size; i++){
        pos = routineSmallest(arr, i, arr_size);
        temp = arr[i];
        arr[i] = arr[pos];
        arr[pos] = temp;
        pass++;
    }
}

void display(int arr[], int arr_size){
    for(int t = 0; t < arr_size; t++){
        cout << arr[t] << " ";
    }
    cout << endl;
}

int main(){
    const int maximum = 100;
    int numbers[maximum];
    srand(time(0));
    for(int i = 0; i < maximum; i++){
        numbers[i] = rand() % 100;
    }

    cout << "UNSORTED ARRAY" << endl;
    display(numbers, maximum);

    cout << "\nSORTED ARRAY (Selection Sort)" << endl;
    selectionSort(numbers, maximum);
    display(numbers, maximum);

    return 0;
}

```

	/tmp/Rj1mIlgwcU.o UNSORTED ARRAY 28 66 81 20 21 69 67 58 78 63 56 76 39 57 94 84 31 8 3 59 72 45 83 99 99 22 17 54 26 77 99 6 43 80 26 16 1 93 74 79 9 31 8 0 40 2 37 23 11 40 82 35 37 17 34 88 39 3 43 65 80 42 71 75 74 49 44 76 43 18 7 4 1 15 56 42 18 93 17 81 33 52 16 71 69 2 59 9 5 54 26 38 48 50 65 23 99 9 51 94 SORTED ARRAY (Selection Sort) 0 1 1 2 2 3 3 4 5 6 7 8 8 9 9 9 11 15 16 16 17 17 17 18 18 20 21 22 23 23 26 26 26 28 31 31 33 34 35 37 37 38 39 39 40 40 42 42 43 43 43 44 45 48 49 50 51 52 54 54 56 56 57 58 59 59 63 65 65 66 67 69 69 71 71 72 74 74 75 76 76 77 78 79 80 80 81 81 82 83 84 88 93 93 94 94 99 99 99 99 === Code Execution Successful ===
Observations	This program generates 100 random numbers, displays them in their unsorted form, and then sorts them using the selection sort algorithm. After sorting, it shows the sorted array, demonstrating how selection sort rearranges the values in ascending order.

Table 7-3. Selection Sort Algorithm

Code + Console Screenshot	

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

void display(int arr[], int arr_size) {
    for (int t = 0; t < arr_size; t++) {
        cout << arr[t] << " ";
    }
    cout << endl;
}

void insertionSort(int arr[], int arr_size) {
    int num_2, key, num_1;
    for (num_2 = 1; num_2 < arr_size; num_2++) {
        key = arr[num_2];
        num_1 = num_2 - 1;

        while (num_1 >= 0 && arr[num_1] > key) {
            arr[num_1 + 1] = arr[num_1];
            num_1 = num_1 - 1;
        }
        arr[num_1 + 1] = key;
    }
}

int main() {
    const int maximum = 100;
    int numbers[maximum];
    srand(time(0));

    for (int i = 0; i < maximum; i++) {
        numbers[i] = rand() % 100;
    }

    cout << "UNSORTED ARRAY" << endl;
    display(numbers, maximum);

    cout << "\nSORTED ARRAY (Insertion Sort)" << endl;
    insertionSort(numbers, maximum);
    display(numbers, maximum);

    return 0;
}
```

```
/tmp/2FsMUVTTQ6.o
UNSORTED ARRAY
65 33 37 23 21 44 50 57 14 19 95 25 48 26 43 27 7 52 9 97 15 13 10 75 70 2 92 0 49 37 49 66 22 38
    89 43 34 39 52 49 59 47 26 59 25 21 39 84 73 48 33 41 61 95 68 84 98 60 84 47 49 86 66 71 76 55
    66 63 47 18 12 6 18 90 17 43 11 56 80 37 5 65 30 18 61 98 2 59 10 39 58 59 77 24 82 5 80 48 68
    27

SORTED ARRAY (Insertion Sort)
0 2 2 5 5 6 7 9 10 10 11 12 13 14 15 17 18 18 18 19 21 21 22 23 24 25 25 26 26 27 27 30 33 33 34 37
    37 37 38 39 39 39 41 43 43 43 44 47 47 47 48 48 48 49 49 49 49 50 52 52 55 56 57 58 59 59 59 59
    60 61 61 63 65 65 66 66 66 68 68 70 71 73 75 76 77 80 80 82 84 84 84 86 89 90 92 95 95 97 98 98



=== Code Execution Successful ===
```

Observations	This program generates 100 random numbers, shows them unsorted, then uses insertion sort to organize them. After sorting, it prints the sorted numbers, making it easy to see the change from random to sorted.

Table 7-4. Insertion Sort Algorithm

7. Supplementary Activity

Pseudocode of Algorithm
1. Start the program. 2. Set up the random number generator. 3. Create an array of votes for 100 votes. 4. Generate random votes between 1 and 5 for each candidate in the votes array. 5. Create an array count to keep track of votes for 5 candidates (all starting at zero). 6. Count the votes for each candidate by looping through votes and updating the count array. 7. Determine the candidate with the most votes by comparing the counts. 8. Display the total votes for each candidate. 9. Announce the winner. 10. End the program.

Screenshot of Algorithm Code

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

using namespace std;

int main() {
    srand(time(0));
    vector<int> votes(100);

    for (int& vote : votes) {
        vote = rand() % 5 + 1;
    }

    int counts[5] = {0};
    for (int vote : votes) {
        counts[vote - 1]++;
    }

    int winner = 0;
    for (int i = 1; i < 5; ++i) {
        if (counts[i] > counts[winner]) {
            winner = i;
        }
    }

    cout << "Vote counts for each candidate:\n";
    for (int i = 0; i < 5; ++i) {
        cout << "Candidate " << (i + 1) << ": " << counts[i] << "\n";
    }

    cout << "The winning candidate is: Candidate " << (winner + 1) << endl;

    return 0;
}

```

Output Testing

```
/tmp/K9TkKomMNV.o
Vote counts for each candidate:
Candidate 1: 24
Candidate 2: 14
Candidate 3: 24
Candidate 4: 17
Candidate 5: 21
The winning candidate is: Candidate 1


=== Code Execution Successful ===
```

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

- I chose bubble sort because it is simple and easy to implement, which helps illustrate basic sorting concepts. While not the most efficient for large datasets, its clarity makes it manageable for our small array of votes. It is a great tool for understanding sorting principles before tackling more complex algorithms

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

- The developed vote counting algorithm is effective because it is simple and easy to understand, with linear time complexity for counting votes. It efficiently handles the given 100 votes and can scale up without significant changes

8. Conclusion

This activity taught me about three sorting techniques: bubble sort, selection sort, and insertion sort. Selection sort finds the smallest or largest element from the unsorted section, while insertion sort builds a sorted section by placing each new element in its correct position. Bubble sort compares and swaps adjacent elements. I applied these methods in a supplementary task to sort votes and determine the winner by counting the highest instances. Overall, I learned that each sorting algorithm has its strengths and weaknesses, emphasizing the need to choose the right one for different situations.

9. Assessment Rubric