Activity No. 7.1

Hands-on Activity 7.1
Sorting Algorithms

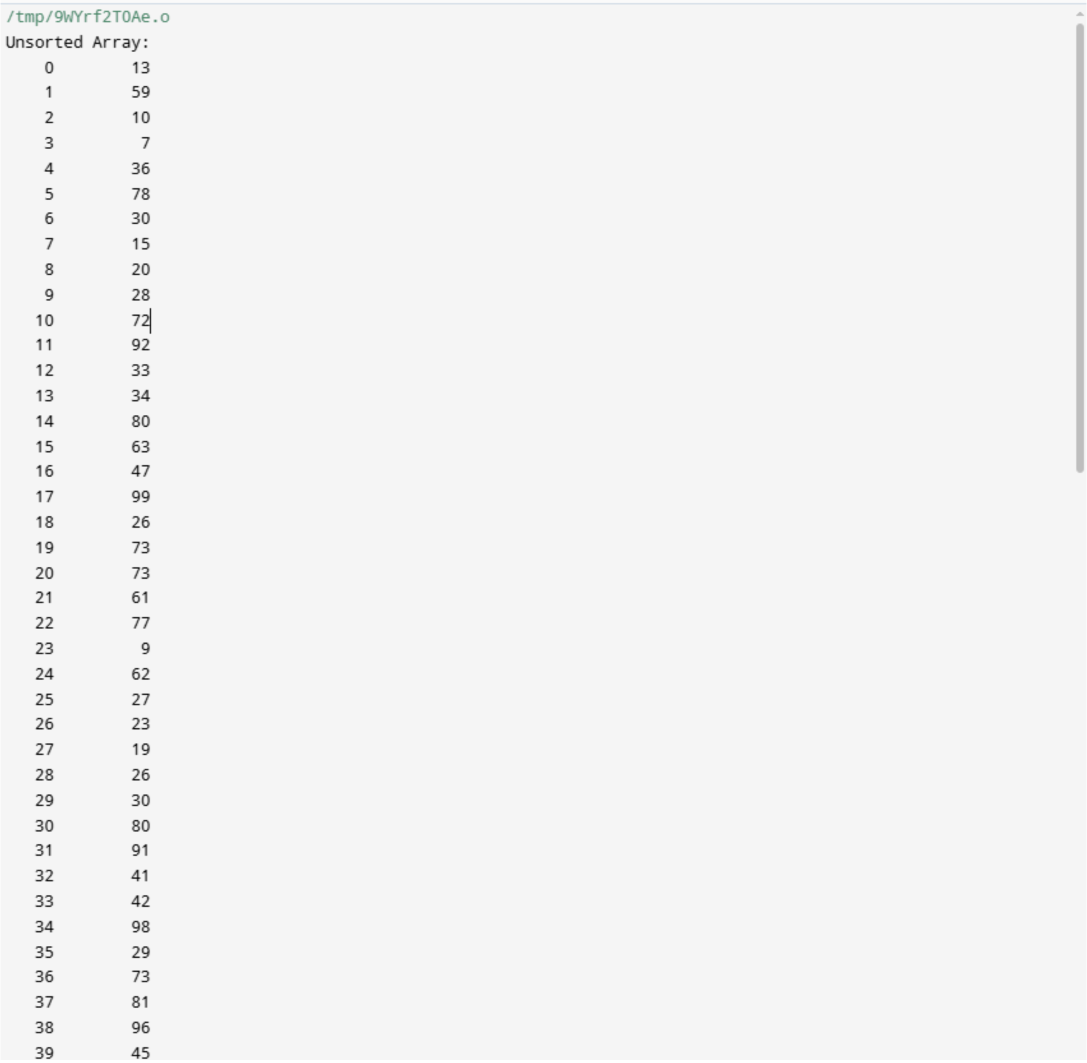
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/16/24**
Section: CPE21S4	**Date Submitted:10/16/24**
Name(s): Solis, Paul Vincent M.	**Instructor: Prof Sayo**

6. Output

Code + Console Screenshot

C/C++

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

using namespace std;

int main() {
  srand(time(0)); // seed for random number generation
  int arr[100];

  cout << "Unsorted Array:" << endl;

  for(int i = 0; i < 100; i++) {
     arr[i] = rand() % 100; // generate random number between 0 and 99
     cout << setw(5) << i << setw(10) << arr[i] << endl;
  }

  return 0;
}
```



```
/tmp/9WYrf2TOAe.o
Unsorted Array:
    0      13
    1      59
    2      10
    3       7
    4      36
    5      78
    6      30
    7      15
    8      20
    9      28
   10      72
   11      92
   12      33
   13      34
   14      80
   15      63
   16      47
   17      99
   18      26
   19      73
   20      73
   21      61
   22      77
   23       9
   24      62
   25      27
   26      23
   27      19
   28      26
   29      30
   30      80
   31      91
   32      41
   33      42
   34      98
   35      29
   36      73
   37      81
   38      96
   39      45
```

Observations	A given array or list of elements can be rearranged using sorting algorithms based on a comparison operator applied to the elements. Counting sort, Radix sort, Bucket sort, Bubble sort, Selection sort, Insertion sort, Merge sort, Quick sort, Heap sort, and more are some of the sorting algorithms available. Each has advantages and disadvantages of its own. These algorithms fall into three categories: non-adaptive sorting, stable and unstable sorting, and internal and external sorting.

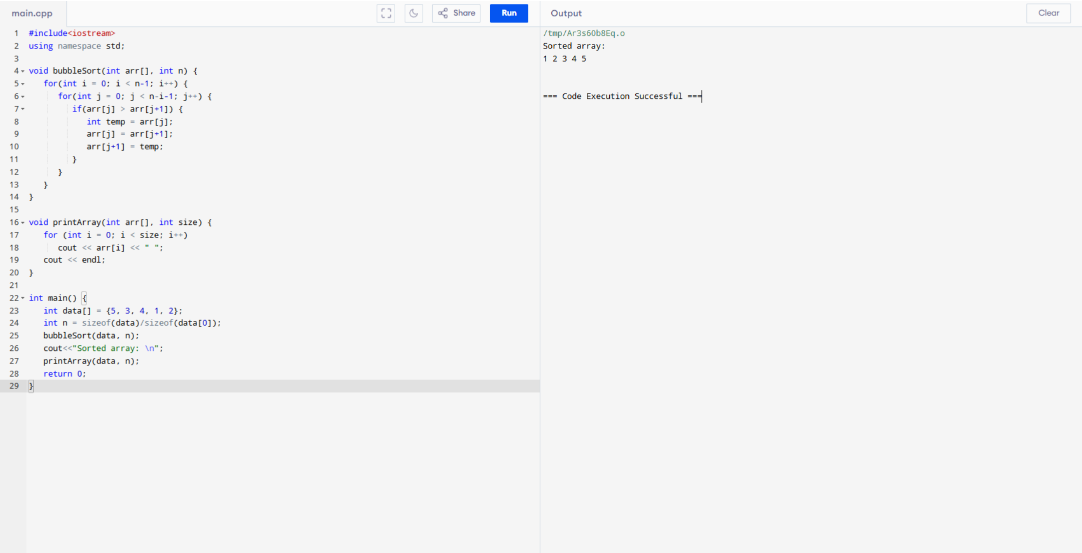
Table 7-1. Array of Values for Sort Algorithm Testing

Code + Console Screenshot

C/C++

```
#include<iostream>
using namespace std;

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

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

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

int main() {
   int data[] = {5, 3, 4, 1, 2};
   int n = sizeof(data)/sizeof(data[0]);
   bubbleSort(data, n);
   cout<<"Sorted array: \n";
   printArray(data, n);
   return 0;
}
```



```
#include<iostream>
using namespace std;

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

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

int main() {
   int data[] = {5, 3, 4, 1, 2};
   int n = sizeof(data)/sizeof(data[0]);
   bubbleSort(data, n);
   cout<<"Sorted array: \n";
   printArray(data, n);
   return 0;
}
```

```
/tmp/Ar3s60b8Eq.o
Sorted array:
1 2 3 4 5


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

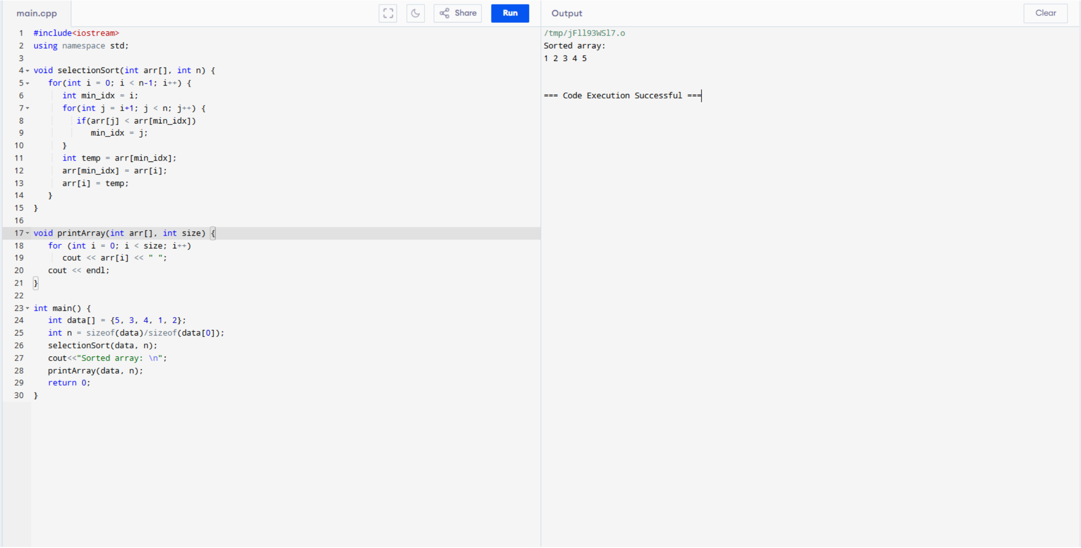
Observations	A straightforward sorting method called the Bubble Sort algorithm iteratively steps over the list, compares nearby components, and swaps them if they are out of order. The largest element "bubbling" to the end of the list with each iteration of this procedure continues until the list is sorted. With an O(n^2) time complexity, the Bubble Sort method is less effective for large datasets.

Table 7-2. Bubble Sort Technique

Code + Console Screenshot

```C/C++
#include<iostream>
using namespace std;
```

```
void selectionSort(int arr[], int n) {
   for(int i = 0; i < n-1; i++) {
      int min_idx = i;
      for(int j = i+1; j < n; j++) {
         if(arr[j] < arr[min_idx])
            min_idx = j;
      }
      int temp = arr[min_idx];
      arr[min_idx] = arr[i];
      arr[i] = temp;
   }
}

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

int main() {
   int data[] = {5, 3, 4, 1, 2};
   int n = sizeof(data)/sizeof(data[0]);
   selectionSort(data, n);
   cout<<"Sorted array: \n";
   printArray(data, n);
   return 0;
}
```



```
#include<iostream>
using namespace std;

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

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

int main() {
   int data[] = {5, 3, 4, 1, 2};
   int n = sizeof(data)/sizeof(data[0]);
   selectionSort(data, n);
   cout<<"Sorted array: \n";
   printArray(data, n);
   return 0;
}
```

```
/tmp/jFll93WSl7.o
Sorted array:
1 2 3 4 5


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

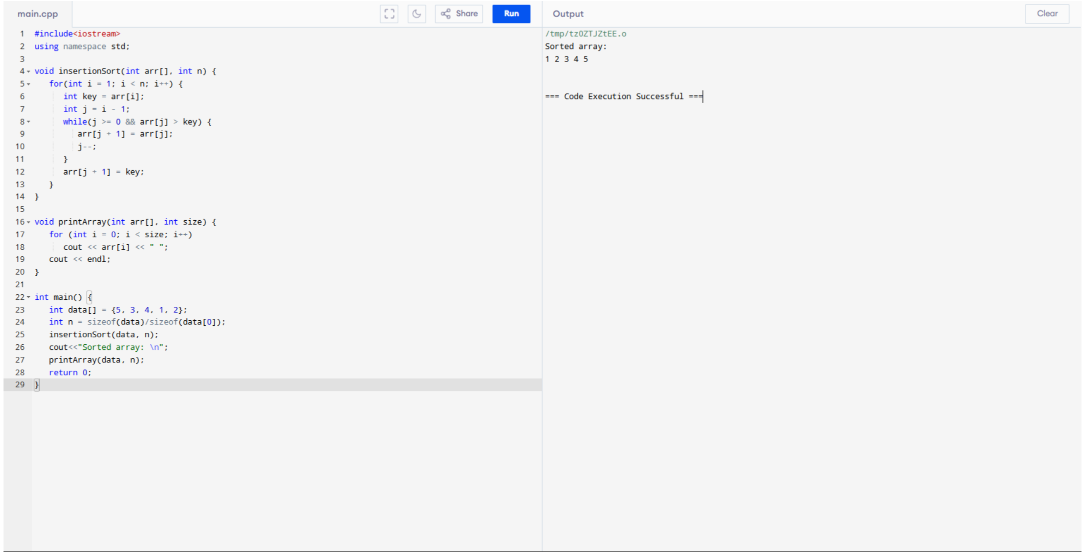
Observations	By continually locating the minimum element in the unsorted region and pushing it to the beginning of the unsorted region, the Selection Sort algorithm is a straightforward sorting method that splits the list into sorted and unsorted regions. Every time the list is sorted, this process is repeated, with the smallest entry being "selected" and relocated to its proper position. For large datasets, the Selection Sort algorithm is less effective due to its O(n^2) time complexity.

Table 7-3. Selection Sort Algorithm

Code + Console Screenshot

C/C++

```cpp
#include<iostream>
using namespace std;

void insertionSort(int arr[], int n) {
   for(int i = 1; i < n; i++) {
      int key = arr[i];
      int j = i - 1;
      while(j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j--;
      }
      arr[j + 1] = key;
   }
}

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

int main() {
   int data[] = {5, 3, 4, 1, 2};
   int n = sizeof(data)/sizeof(data[0]);
   insertionSort(data, n);
   cout<<"Sorted array: \n";
   printArray(data, n);
   return 0;
}
```



```cpp
#include<iostream>
using namespace std;

void insertionSort(int arr[], int n) {
   for(int i = 1; i < n; i++) {
      int key = arr[i];
      int j = i - 1;
      while(j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j--;
      }
      arr[j + 1] = key;
   }
}

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

int main() {
   int data[] = {5, 3, 4, 1, 2};
   int n = sizeof(data)/sizeof(data[0]);
   insertionSort(data, n);
   cout<<"Sorted array: \n";
   printArray(data, n);
   return 0;
}
```

```
/tmp/tzOZTJZtEE.o
Sorted array:
1 2 3 4 5


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

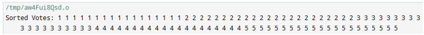
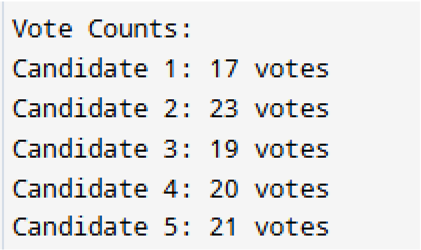
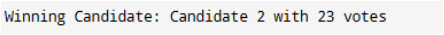
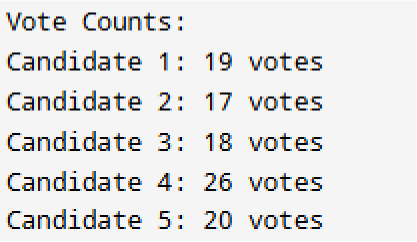
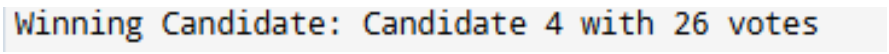
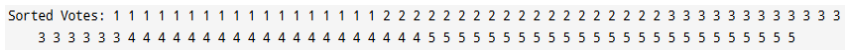
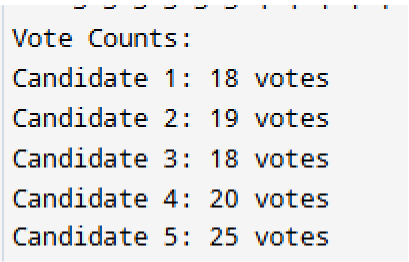
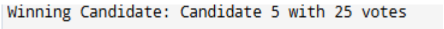
Observations

The Insertion Sort algorithm is a straightforward sorting method that operates by going over the array element by element and inserting each one into the appropriately designated spot in the portion of the array that has already been sorted. Each element is "inserted" into its proper location, and this procedure is continued until the array is sorted as a whole. For

	large datasets, the Insertion Sort method is less effective due to its O(n^2) time complexity.

Table 7-4. Insertion Sort Algorithm

7. Supplementary Activity

Output Console Showing Sorted Array	manual count	Count result of algorithm
 /tmp/aw4Fui8Qsd.o Sorted Votes: 1 1 1 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 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 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5	 Vote Counts: Candidate 1: 17 votes Candidate 2: 23 votes Candidate 3: 19 votes Candidate 4: 20 votes Candidate 5: 21 votes	 Winning Candidate: Candidate 2 with 23 votes
 Sorted Votes: 1 1 1 1 1 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 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 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: Candidate 1: 19 votes Candidate 2: 17 votes Candidate 3: 18 votes Candidate 4: 26 votes Candidate 5: 20 votes	 Winning Candidate: Candidate 4 with 26 votes
 Sorted Votes: 1 1 1 1 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 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 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5	 Vote Counts: Candidate 1: 18 votes Candidate 2: 19 votes Candidate 3: 18 votes Candidate 4: 20 votes Candidate 5: 25 votes	 Winning Candidate: Candidate 5 with 25 votes

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

- The implemented vote counting algorithm successfully tallied votes and identified the victor. Its straightforwardness and dependability made it a suitable option for this minor issue. Yet, its effectiveness is restricted due to the utilization of bubble sort, which is unsuitable for handling vast amounts of data. If the number of votes were to grow significantly, a more effective sorting algorithm would be needed. The algorithm's use of random number generation restricts its practicality in real-world situations. In general, the algorithm proved successful for this particular issue, but its constraints would require attention in a bigger deployment.

8. Conclusion

Sorting algorithms are crucial in a variety of real-life situations, with a wide range of applications and extensive use. Sorting algorithms are utilized in data analysis, online search engines, database management, machine learning, and social media for efficient organization and management of data. They are utilized in e-commerce, GPS guidance, and medical evaluation to enhance effectiveness and precision. Utilizing sorting algorithms in these areas offers multiple advantages, such as enhanced arrangement, higher productivity, and decreased mistakes. Furthermore, sorting algorithms play a crucial role in daily activities, like organizing emails, music playlists, and online learning platforms. In general, sorting algorithms are crucial instruments that greatly influence our everyday routines. They allow us to quickly and accurately process and analyze vast quantities of data, making them an essential part of today's technology. By comprehending and implementing sorting algorithms, we can enhance the efficiency and effectiveness of different systems and processes.

9. Assessment Rubric