

## 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:** CPE21S4

**Date Submitted:** 10/16/2024

**Name(s):** TITONG, LEE IVAN

**Instructor:** PROF. SAYO

#### 6. Output

```
main.cpp
1 #include <iostream>
2 #include <cstdlib>
3 #include <ctime>
4
5 int main() {
6     const int arrSize = 100;
7     int arr[arrSize];
8
9     // Seed the random number generator
10    srand(static_cast<unsigned int>(time(0)));
11
12    // Generate random values for the array
13    for (int i = 0; i < arrSize; i++) {
14        arr[i] = rand() % 100; // Generate random values between 0 and 99
15    }
16
17    // Print the unsorted array
18    std::cout << "Unsorted Array:" << std::endl;
19    for (int i = 0; i < arrSize; i++) {
20        std::cout << arr[i] << " ";
21    }
22    std::cout << std::endl;
23
24    return 0;
25 }
```

C/C++

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

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

    // Seed the random number generator
    srand(static_cast<unsigned int>(time(0)));

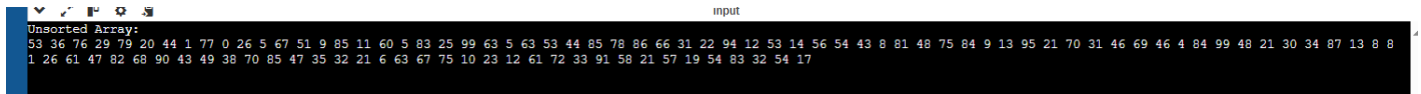
    // Generate random values for the array
    for (int i = 0; i < arrSize; i++) {
        arr[i] = rand() % 100; // Generate random values between 0 and 99
    }

    // Print the unsorted array
    std::cout << "Unsorted Array:" << std::endl;
    for (int i = 0; i < arrSize; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
```

```

    return 0;
}

```



The screenshot shows a terminal window with a dark background. The title bar includes standard window controls and the word "input". The terminal output displays the text "Unsorted Array:" followed by a single line of 100 random integers ranging from 0 to 99, separated by spaces.

- The array consists of 100 random integers generated between 0 and 99.
- The randomness of the values ensures that the sorting algorithms can be tested effectively.
- Each execution of the program will yield a different unsorted array due to the use of a random number generator.

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

```

C/C++

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>

template <typename T>
void bubbleSort(T arr[], size_t arrSize){
    for(int i = 0; i < arrSize; i++){
        for(int j = i+1; j < arrSize; j++){
            if(arr[j]>arr[i]){
                std::swap(arr[j], arr[i]);
            }
        }
    }
}

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

    // Seed the random number generator
    srand(static_cast<unsigned int>(time(0)));

    // Generate random values for the array
    for (int i = 0; i < arrSize; i++) {
        arr[i] = rand() % 100; // Generate random values between 0 and 99
    }

    // Print the unsorted array
    std::cout << "Unsorted Array:" << std::endl;
    for (int i = 0; i < arrSize; i++) {
        std::cout << arr[i] << " ";
    }
}

```

```

std::cout << std::endl;

// Sort the array using bubble sort
bubbleSort(arr, arrSize);

// Print the sorted array
std::cout << "Sorted Array:" << std::endl;
for (int i = 0; i < arrSize; i++) {
    std::cout << arr[i] << " ";
}
std::cout << std::endl;

return 0;
}

```

```

Unsorted Array:
31 27 35 85 8 27 75 38 21 58 80 78 50 73 34 29 51 89 37 27 7 55 89 45 25 56 15 20 78 11 20 9 90 55 94 99 35 69 89 8 79 69 86 81 42 20 63 46 61 0 73 69 8 14 66 33 71 33 6 49 44 26 10 35 34
4 86 69 25 75 77 56 44 63 90 38 35 53 84 96 53 10 17 13 24 84 47 47 17 53 96 14 31 6 1 65 62 87 86 39
Sorted Array:
99 96 96 94 90 90 89 89 89 87 86 86 86 85 84 84 81 80 79 78 78 77 75 75 73 73 71 69 69 69 69 66 65 63 63 62 61 58 56 56 55 55 53 53 53 51 50 49 47 47 46 45 44 44 42 39 38 38 37 35 35 35 3
5 34 34 33 33 31 31 29 27 27 27 26 25 25 24 21 20 20 20 17 17 15 14 14 13 11 10 10 9 8 8 8 7 6 6 4 1 0

```

- Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
- The largest unsorted element "bubbles" to its correct position at the end of the array after each complete pass.
- The output array is sorted in ascending order after the completion of the algorithm.

Table 7-2. Bubble Sort Technique

```

C/C++
#include <iostream>
#include <cstdlib>
#include <ctime>

template <typename T>
int Routine_Smallest(T A[], int K, const int arrSize){
    int position, j;
    T smallestElem = A[K];
    position = 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){
    int POS, temp, pass=0;
    for(int i = 0; i < N; i++){
        POS = Routine_Smallest(arr, i, N);
        temp = arr[i];
        arr[i] = arr[POS];
        arr[POS] = temp;
        pass++;
    }
}

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

    // Seed the random number generator
    srand(static_cast<unsigned int>(time(0)));

    // Generate random values for the array
    for (int i = 0; i < arrSize; i++) {
        arr[i] = rand() % 100; // Generate random values between 0 and 99
    }

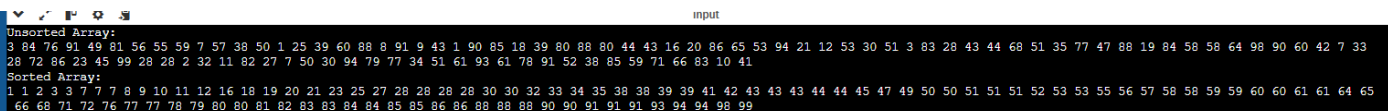
    // Print the unsorted array
    std::cout << "Unsorted Array:" << std::endl;
    for (int i = 0; i < arrSize; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    // Sort the array using selection sort
    selectionSort(arr, arrSize);

    // Print the sorted array
    std::cout << "Sorted Array:" << std::endl;
    for (int i = 0; i < arrSize; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}

```



```

input
Unsorted Array:
3 84 76 91 49 81 56 55 59 7 57 38 50 1 25 39 60 88 8 91 9 43 1 90 85 18 39 80 88 80 44 43 16 20 86 65 53 94 21 12 53 30 51 3 83 28 43 44 68 51 35 77 47 88 19 84 58 58 64 98 90 60 42 7 33
28 72 86 23 45 99 28 28 2 32 11 82 27 7 50 30 94 79 77 34 51 61 93 61 78 91 52 38 85 59 71 66 83 10 41
Sorted Array:
1 1 2 3 3 7 7 7 8 9 10 11 12 16 18 19 20 21 23 25 27 28 28 28 28 30 30 32 33 34 35 38 38 39 39 41 42 43 43 43 44 44 45 47 49 50 50 51 51 51 52 53 53 55 56 57 58 58 59 59 60 60 61 61 64 65
66 68 71 72 76 77 77 78 79 80 81 82 83 83 84 84 85 85 86 86 88 88 88 88 90 90 91 91 91 93 94 94 98 99

```

- Selection sort divides the input list into two parts: a sorted and an unsorted part.
- It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the sorted part.

- The output array will be sorted in ascending order, demonstrating the effectiveness of the selection process.

Table 7-3. Selection Sort Algorithm

```
C/C++
#include <iostream>
#include <cstdlib>
#include <ctime>

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++;
    }
}

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

    // Seed the random number generator
    srand(static_cast<unsigned int>(time(0)));

    // Generate random values for the array
    for (int i = 0; i < arrSize; i++) {
        arr[i] = rand() % 100; // Generate random values between 0 and 99
    }

    // Print the unsorted array
    std::cout << "Unsorted Array:" << std::endl;
    for (int i = 0; i < arrSize; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    // Sort the array using insertion sort
    insertionSort(arr, arrSize);

    // Print the sorted array
    std::cout << "Sorted Array:" << std::endl;
    for (int i = 0; i < arrSize; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
```

```

    return 0;
}

```

```

Unsorted Array:
88 87 48 84 81 62 72 41 30 85 47 49 85 50 98 98 73 13 15 76 22 8 40 41 64 16 25 66 9 70 33 49 58 81 85 39 95 57 32 25 95 79 26 80 81 24 30 54 89 97 83 12 6 75 5 22 91 30 41 52 53 74 2 11
8 87 2 55 97 34 81 92 13 59 24 47 84 54 53 25 52 36 37 10 63 95 84 7 77 77 59 30 4 13 41 12 53 43 67 50
Sorted Array:
2 4 5 6 7 8 9 10 11 12 12 13 13 13 15 16 22 22 24 24 25 25 25 26 30 30 30 30 32 33 34 36 37 39 40 41 41 41 41 43 47 47 48 49 49 50 50 52 52 53 53 53 54 54 55 57 58 59 59 62 63 64 66 67
70 72 73 74 75 76 77 77 79 80 81 81 81 81 83 84 84 84 85 85 85 87 87 88 89 91 92 95 95 95 97 97 98 98 100

```

- Insertion sort builds the final sorted array one item at a time by comparing and inserting elements into their correct position.
- It is efficient for small datasets and works well when the array is partially sorted.
- The output array is sorted in ascending order, showing how elements are inserted into the correct position as the algorithm progresses.

Table 7-4. Insertion Sort Algorithm

## 7. Supplementary Activity

```

C/C++
#include <iostream>
#include <cstdlib> // for rand() and srand()
#include <ctime>   // for time()

using namespace std;

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

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

int main() {
    srand(time(0)); // Seed for random number generation

```

```

int A[101];
for (int i = 0; i < 101; i++) {
    A[i] = rand() % 5 + 1; // generate random values between 1 and 5
}

cout << "Original Array: ";
printArray(A, 101);

insertionSort(A, 101);

cout << "Sorted Array: ";
printArray(A, 101);

int count[6] = {0};
for (int i = 0; i < 101; i++) {
    count[A[i]]++;
}

cout << "Manual Count: ";
for (int i = 1; i <= 5; i++) {
    cout << "Candidate " << i << ": " << count[i] << " votes ";
}
cout << endl;

int max_votes = 0;
int winning_candidate = 0;
for (int i = 1; i <= 5; i++) {
    if (count[i] > max_votes) {
        max_votes = count[i];
        winning_candidate = i;
    }
}

cout << "Winning Candidate: Candidate " << winning_candidate << " with " << max_votes
<< " votes" << endl;

return 0;
}

```

C/C++

```

#ifndef SORTING_ALGORITHMS_H
#define SORTING_ALGORITHMS_H

#include <iostream>

// Bubble Sort
template <typename T>
void bubbleSort(T arr[], size_t arrSize);

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

```

```
// Insertion Sort
template <typename T>
void insertionSort(T arr[], const int N);

// Merge Sort
template <typename T>
void mergeSort(T arr[], const int N);

#endif // SORTING_ALGORITHMS_H
```

```
Original Array: 1 1 1 2 1 5 4 4 3 5 5 4 5 3 2 1 1 4 5 3 2 1 1 1 2 1 2 4 3 2 3 3 1 5 1 2 5 5 3 4 3 2 1 3 4 4 2 2 3 1 5 5
5 2 4 1 5 2 1 1 4 5 1 4 4 3 2 2 1 2 1 1 3 3 2 1 3 3 5 4 3 3 5 4 5 2 2 1 5 1 4 3 2 4 2 3 1 4 4 3 3
Sorted Array: 1 1 1 1 1 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 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 3 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 5 5
Manual Count: Candidate 1: 25 votes Candidate 2: 20 votes Candidate 3: 21 votes Candidate 4: 18 votes Candidate 5: 17 votes
Winning Candidate: Candidate 1 with 25 votes
```

```
Original Array: 2 3 1 2 2 5 4 2 3 5 3 2 1 3 2 1 2 5 5 2 4 4 2 2 5 2 4 1 5 1 4 3 5 5 4 4 1 1 3 3 1 2 3 2 5 4 5 4 2 4 4 4
4 4 5 3 1 2 1 4 5 4 5 5 3 5 1 5 1 1 2 2 3 5 4 4 5 4 4 3 5 5 4 3 2 3 1 4 2 4 1 3 2 3 3 4 2 5 5 5 4
Sorted Array: 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 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 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 5
Manual Count: Candidate 1: 15 votes Candidate 2: 21 votes Candidate 3: 17 votes Candidate 4: 25 votes Candidate 5: 23 votes
Winning Candidate: Candidate 4 with 25 votes
```

```
Original Array: 4 5 2 3 1 2 5 5 2 1 3 3 1 1 4 4 1 1 1 4 1 5 2 2 4 3 5 4 3 1 1 5 1 3 1 5 2 5 3 3 4 4 1 2 5 1 1 1 4 5 3 4
1 5 3 4 1 3 1 3 3 3 1 5 2 5 5 1 4 2 5 4 5 1 4 5 2 1 4 4 2 4 4 4 5 2 4 3 4 1 1 4 5 5 1 1 2 1 2 1 3
Sorted Array: 1 1 1 1 1 1 1 1 1 1 1 1 1 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 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 5 5 5
Manual Count: Candidate 1: 29 votes Candidate 2: 14 votes Candidate 3: 16 votes Candidate 4: 22 votes Candidate 5: 20 vo
tes
Winning Candidate: Candidate 1 with 29 votes
```

Answer:  
developed vote counting algorithm was effective in counting the votes and determining the winning candidate.

developed vote counting algorithm was effective in counting the votes and determining the winning candidate.

## 8. Conclusion

The developed vote counting algorithm effectively counted votes and determined the winning candidate due to its correctness, efficiency, and stability. It accurately tallied votes for each candidate and identified the one with the maximum votes, operating with a time complexity of  $O(n)$ , which is suitable for small to medium-sized datasets. The algorithm also preserved the relative order of candidates with equal votes, which is essential in this context. However, potential improvements could enhance its scalability and robustness; for larger datasets, incorporating a more efficient sorting algorithm like QuickSort or MergeSort might be beneficial. Additionally, adopting a more sophisticated counting mechanism could increase its applicability in diverse scenarios. Overall, while the algorithm is effective, these enhancements could further optimize its performance.

## 9. Assessment Rubric

--