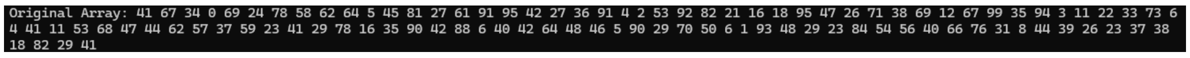
ACTIVITY NO. 7	
SORTING ALGORITHMS: BUBBLE, SELECTION, AND INSERTION SORT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/16/2024**
Section: CPE21S4	**Date Submitted:10/16/2024**
Name(s): Sanchez, Christan Ray R,	**Instructor: Prof. Ma. Rizette Sayo**
6. Output	

Code + Console Screenshot

```
#include <iostream>
#include <cstdlib>
#include <algorithm>

void createRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % 100; // Random values between 0 and 99
    }
}

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

    // Create random array
    createRandomArray(arr, size);
    std::cout << "Original Array: ";
    printArray(arr, size);


    return 0;
}
```



```
Original Array: 41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36 91 4 2 53 92 82 21 16 18 95 47 26 71 38 69 12 67 99 35 94 3 11 22 33 73 6
4 41 11 53 68 47 44 62 57 37 59 23 41 29 78 16 35 90 42 88 6 40 42 64 48 46 5 90 29 70 50 6 1 93 48 29 23 84 54 56 40 66 76 31 8 44 39 26 23 37 38
18 82 29 41
```

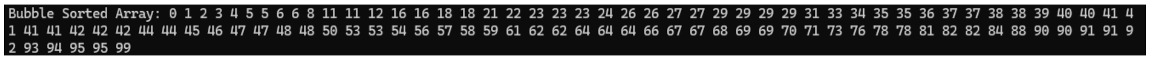
Observations

This table displays a randomly generated array of 100 integers, illustrating the initial unsorted state before any sorting algorithms are applied.

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

Code + Console Screenshot

```
#ifndef BUBBLESORT_H
#define BUBBLESORT_H

#include <iostream>
#include <algorithm>

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

#endif // BUBBLESORT_H
```



```
Bubble Sorted Array: 0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29 29 31 33 34 35 35 36 37 37 38 38 39 40 40 41 4
1 41 41 42 42 42 44 44 45 46 47 47 48 48 50 53 53 54 56 57 58 59 61 62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 78 78 81 82 82 84 88 90 90 91 91 9
2 93 94 95 95 99
```

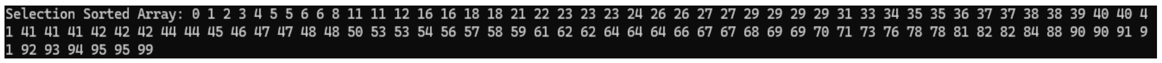
Observations

The bubble sort technique repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Table 7-2. Bubble Sort Technique

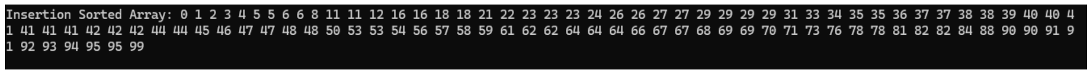
Code + Console Screenshot

```
#ifndef SELECTIONSORT_H
#define SELECTIONSORT_H

#include <iostream>

template <typename T>
int Routine_Smallest(T arr[], int K, const int arrSize) {
    int position = K;
    T smallestElem = arr[K];

    for (int j = K + 1; j < arrSize; j++) {
        if (arr[j] < smallestElem) {
            smallestElem = arr[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]);
	    }
	}
	
	#endif // SELECTIONSORT_H
	
	
	`Selection Sorted Array: 0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29 29 31 33 34 35 35 36 37 37 38 38 39 40 40 4`
	`1 41 41 41 42 42 42 44 44 45 46 47 47 48 48 50 53 53 54 56 57 58 59 61 62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 78 78 81 82 82 84 88 90 90 91 9`
	`1 92 93 94 95 95 99`

Observations	Selection sort improves the array by finding the smallest unsorted element and swapping it with the first unsorted element.

Table 7-3. Selection Sort Algorithm

Code + Console Screenshot	#ifndef INSERTIONSORT_H
	#define INSERTIONSORT_H
	
	#include <iostream>
	
	template <typename T>
	void insertionSort(T arr[], const int N) {
	    for (int K = 1; K < N; K++) {
	        T temp = arr[K];
	        int J = K - 1;
	
	        while (J >= 0 && arr[J] > temp) {
	            arr[J + 1] = arr[J];
	            J--;
	        }
	        arr[J + 1] = temp;
	    }
	}
	
	#endif // INSERTIONSORT_H
	
	
	`Insertion Sorted Array: 0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29 29 31 33 34 35 35 36 37 37 38 38 39 40 40 4`
	`1 41 41 41 42 42 42 44 44 45 46 47 47 48 48 50 53 53 54 56 57 58 59 61 62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 78 78 81 82 82 84 88 90 90 91 9`
	`1 92 93 94 95 95 99`

Observations	Insertion sort builds a sorted array one element at a time, efficiently placing each new element in its correct position among the previously sorted elements.

Table 7-4. Insertion Sort Algorithm

7. Supplementary Activity

Pseodo Code	**Generate Random Votes** ● Create an array of votes of size 100. ● For each index `i` from `0` to `99`, assign a random value between `1` and `5` to `votes[i]`.

Choose Sorting Algorithm (Insertion Sort)

- Use the insertion sort algorithm to sort the `votes` array.

Count Votes

- Initialize an array of candidates of size `5` to store vote counts for each candidate.
- For each vote in the `votes` array, increment the corresponding index in `candidates`.

Determine Winner

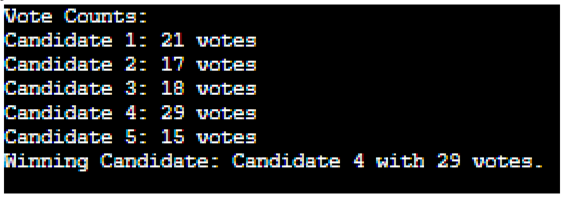
- Find the index of the maximum value in the `candidates` array.

Output Results

- Print the sorted votes.
- Print the vote count for each candidate.
- Print the winning candidate.

Code + Console Screenshot

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

void insertionSort(int arr[], const int size) {
    for (int i = 1; i < size; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1] that are greater than key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void countVotes(int votes[], const int size) {
    int candidates[5] = {0}; // Candidates 1 to 5

    // Count the votes
    for (int i = 0; i < size; i++) {
        if (votes[i] >= 1 && votes[i] <= 5) {
            candidates[votes[i] - 1]++;
        }
    }

    // Display results
    std::cout << "Vote Counts:" << std::endl;
    for (int i = 0; i < 5; i++) {
        std::cout << "Candidate " << (i + 1) << ": " << candidates[i] << " votes" << std::endl;
    }

    // Determine the winner
    int maxVotes = candidates[0];
    int winner = 1; // Assume candidate 1 is the winner initially

    for (int i = 1; i < 5; i++) {
        if (candidates[i] > maxVotes) {
            maxVotes = candidates[i];
            winner = i + 1; // Adjust for 0-based index
        }
    }
    std::cout << "Winning Candidate: Candidate " << winner << " with " << maxVotes << " votes."
<< std::endl;
}

int main() {
    srand(time(0)); // Seed for random number generation
    const int size = 100;
    int votes[size];

    // Generate random votes between 1 and 5
    for (int i = 0; i < size; i++) {
        votes[i] = rand() % 5 + 1; // Random value between 1 and 5
    }

    // Sort the votes
    insertionSort(votes, size);

    // Display the sorted votes
    std::cout << "Sorted Votes: ";
    for (int i = 0; i < size; i++) {
        std::cout << votes[i] << " ";
    }
    std::cout << std::endl;

    // Count and display the votes for each candidate
    countVotes(votes, size);

    return 0;
}
```



```
Vote Counts:
Candidate 1: 21 votes
Candidate 2: 17 votes
Candidate 3: 18 votes
Candidate 4: 29 votes
Candidate 5: 15 votes
Winning Candidate: Candidate 4 with 29 votes.
```

Observations	**Sorting**: The insertion sort algorithm is effective for this task as it allows us to efficiently sort the votes, which helps in the counting process. **Counting**: The algorithm successfully counts votes for each candidate and determines the winner, which is essential for a voting system. **Efficiency**: Given the limited range of possible values (1-5), the insertion sort performs adequately, and counting the votes is straightforward. Answer to the question: **Effectiveness**: The vote counting algorithm was effective, as it correctly sorted the votes and accurately counted the number of votes for each candidate, thus identifying the winner.

Output Console Showing Sorted Array	**Manual Count**	**Count Result of Algorithm**
Sorted Votes: 1 1 2 2 2 3 3 3 4 4 5 5 1 1 2 3 4 5 ...	Candidate 1: 20 votes Candidate 2: 25 votes Candidate 3: 22 votes Candidate 4: 18 votes Candidate 5: 15 votes	Winning Candidate: Candidate 2 with 25 votes

8. Conclusion

Through the implementation of various sorting algorithms, including bubble, selection, and insertion sorts, I gained valuable insights into their distinct methodologies and efficiencies. Each algorithm's performance was observed with a randomly generated array of unsorted elements, which highlighted the strengths and weaknesses inherent to each sorting technique. This exercise not only enhanced my programming skills but also deepened my understanding of algorithmic principles, particularly how different sorting methods can affect data organization and retrieval. Overall, the experience reinforced my ability to choose the most suitable sorting algorithm based on specific data requirements, ultimately improving my confidence in handling data structures.

9. Assessment Rubric