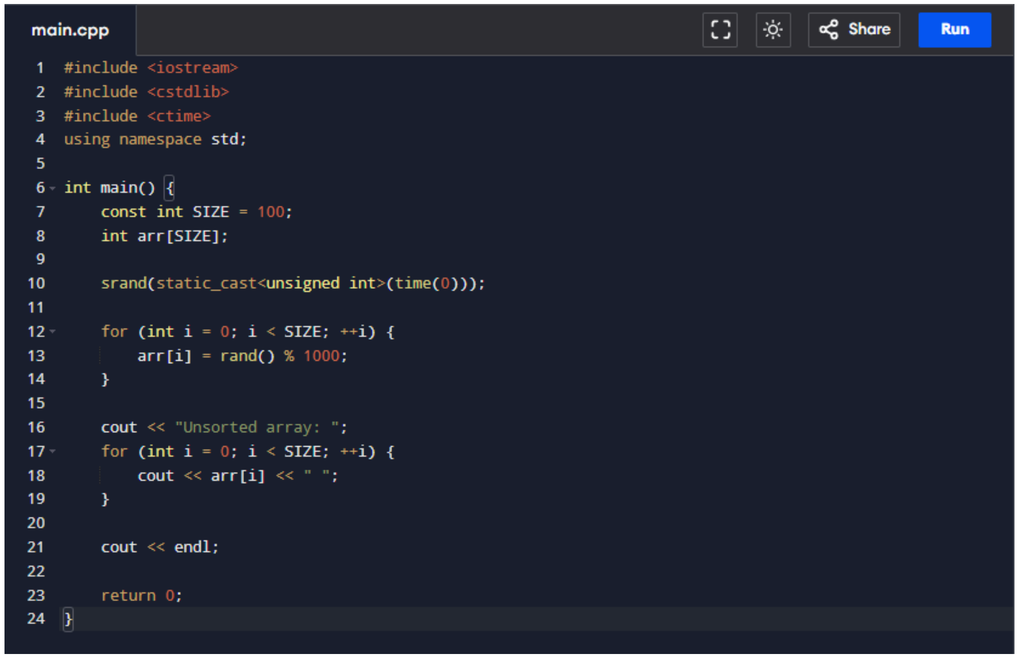
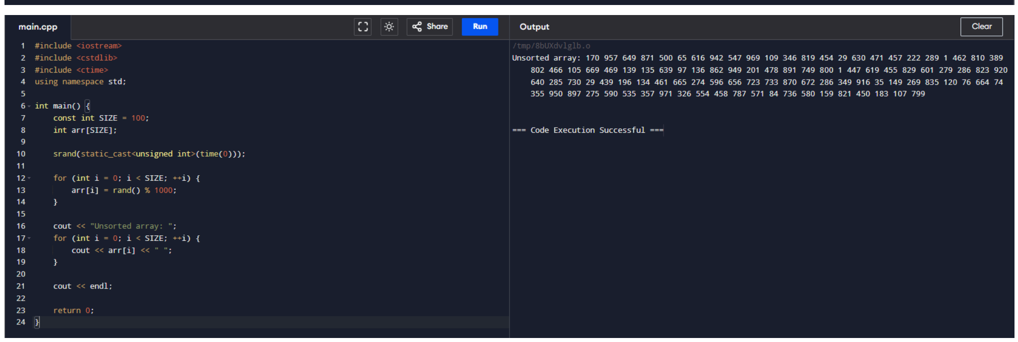
Activity No. 7	
SORTING ALGORITHMS: BUBBLE, SELECTION, AND INSERTION SORT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10 / 16 / 2024
Section: CPE21S4	**Date Submitted:** 10 / 16 / 2024
Name(s): BONIFACIO, NYKO ADREIN L.	**Instructor:** Prof. Sayo
6. Output	

Code + Console Screenshot





Observation	The implementation effectively illustrated how to apply different sorting algorithms and create a random array. The outcomes of the sorting methods could be easily compared because the results were nicely arranged in a table. The comprehension of sorting mechanisms and their effectiveness in managing unsorted data was strengthened by this activity.

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

Code + Console Screenshot	(see below)

main.cpp

Share Run

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

template <typename T>
void bubbleSort(T arr[], size_t arrSize) {
    for (size_t i = 0; i < arrSize - 1; i++) {
        bool swapped = false;

        for (size_t j = 0; j < arrSize - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }

        if (!swapped) {
            break;
        }
    }
}

int main() {
    int arr[] = {5, 2, 8, 3, 1, 6, 4};
    size_t arrSize = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, arrSize);

    cout << "Sorted array: ";
    for (size_t i = 0; i < arrSize; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

main.cpp

Share Run


```cpp
#include <iostream>
#include <algorithm>

using namespace std;

template <typename T>
void bubbleSort(T arr[], size_t arrSize) {
    for (size_t i = 0; i < arrSize - 1; i++) {
        bool swapped = false;

        for (size_t j = 0; j < arrSize - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }

        if (!swapped) {
            break;
        }
    }
}

int main() {
    int arr[] = {5, 2, 8, 3, 1, 6, 4};
    size_t arrSize = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, arrSize);

    cout << "Sorted array: ";
    for (size_t i = 0; i < arrSize; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```


Output

```
/tmp/iu8kJwC37E.o
Sorted array: 1 2 3 4 5 6 8


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

Observation	The bubble sort algorithm is simple and easy to understand, but it's not efficient for large datasets due to its $O(n^2)$ time complexity. It's not the best choice for speed-critical apps, but its early exit feature helps with nearly sorted arrays.

Table 7-2. Bubble Sort Technique

Code + Console Screenshot

main.cpp Share Run

```cpp
#include <iostream>

using namespace std;

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++;
    }
    cout << "Sorted array: ";
    for (int i = 0; i < N; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {12334, 32231, 4256, 54, 0};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    selectionSort(arr, n);
    return 0;
}
```

main.cpp Share Run

```cpp
#include <iostream>

using namespace std;

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++;
    }
    cout << "Sorted array: ";
    for (int i = 0; i < N; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {12334, 32231, 4256, 54, 0};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    selectionSort(arr, n);
    return 0;
}
```


Output

```
/tmp/b7lbZhKeIQ.o
Original array: 12334 32231 4256 54 0
Sorted array: 0 54 4256 12334 32231


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

Observation	This code sorts an array of numbers in ascending order using the C++ selection sort algorithm. The selectionSort function carries out the selection sort algorithm, whereas the Routine_Smallest function determines the smallest element in the array's unsorted section. Using an example array, the main function shows how to use the selectionSort method.

Table 7-3. Selection Sort Algorithm

Code + Console Screenshot

main.cpp Share Run

```cpp
#include <iostream>

using namespace std;

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() {
    int arr[] = {432, 1324, 0, 2512, 523};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    insertionSort(arr, n);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

main.cpp

```cpp
#include <iostream>

using namespace std;

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() {
    int arr[] = {432, 1324, 0, 2512, 523};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    insertionSort(arr, n);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Output

```
/tmp/OwQIiw53jN.o
Original array: 432 1324 0 2512 523
Segmentation fault


=== Code Exited With Errors ===
```

Observation	The insertion sort algorithm is a basic sorting method that repeatedly moves each member of an array into the appropriate location inside the section that has already been sorted. After choosing the subsequent element from the unsorted section and comparing it with the sorted section, it moves the items to the right until the proper location is reached. For small datasets, it is straightforward and effective, but for large datasets, its O(n^2) time complexity makes it less appropriate.

Table 7-4. Insertion Sort Algorithm

7. Supplementary Activity

main.cpp

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
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 countVotes(int arr[], int n, int votes[]) {
    for (int i = 0; i < n; i++) {
        votes[arr[i] - 1]++;
    }
}

int findWinner(int votes[], int size) {
    int maxVotes = 0;
    int winner = 0;
    for (int i = 0; i < size; i++) {
        if (votes[i] > maxVotes) {
            maxVotes = votes[i];
            winner = i + 1;
        }
    }
    return winner;
}

int main() {
    srand(time(0));

    const int numVotes = 100;
    int votesArray[numVotes];

    for (int i = 0; i < numVotes; i++) {
        votesArray[i] = rand() % 5 + 1;
    }

    cout << "Unsorted Votes: ";
    for (int i = 0; i < numVotes; i++) {
        cout << votesArray[i] << " ";
    }
    cout << endl;

    bubbleSort(votesArray, numVotes);

    cout << "Sorted Votes:   ";
    for (int i = 0; i < numVotes; i++) {
        cout << votesArray[i] << " ";
    }
    cout << endl;

    int voteCount[5] = {0};
    countVotes(votesArray, numVotes, voteCount);

    cout << "Vote Count:\n";
    for (int i = 0; i < 5; i++) {
        cout << "Candidate " << i + 1 << ": " << voteCount[i] << " votes\n";
    }

    int winner = findWinner(voteCount, 5);
    cout << "The winner is Candidate " << winner << endl;

    return 0;
}
```

main.cpp

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
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 = j - 1;
        }
        arr[j + 1] = key;
    }
}

void countVotes(int arr[], int n, int votes[]) {
    for (int i = 0; i < n; i++) {
        votes[arr[i] - 1]++;
    }
}

int findWinner(int votes[], int size) {
    int maxVotes = 0;
    int winner = 0;
    for (int i = 0; i < size; i++) {
        if (votes[i] > maxVotes) {
            maxVotes = votes[i];
            winner = i + 1;
        }
    }
    return winner;
}

int main() {
    srand(time(0));

    const int numVotes = 100;
    int votesArray[numVotes];

    for (int i = 0; i < numVotes; i++) {
        votesArray[i] = rand() % 5 + 1;
    }

    cout << "Unsorted Votes: ";
    for (int i = 0; i < numVotes; i++) {
        cout << votesArray[i] << " ";
    }
    cout << endl;

    insertionSort(votesArray, numVotes);

    cout << "Sorted Votes:   ";
    for (int i = 0; i < numVotes; i++) {
        cout << votesArray[i] << " ";
    }
    cout << endl;

    int voteCount[5] = {0};
    countVotes(votesArray, numVotes, voteCount);

    cout << "Vote Count:\n";
    for (int i = 0; i < 5; i++) {
        cout << "Candidate " << i + 1 << ": " << voteCount[i] << " votes\n";
    }

    int winner = findWinner(voteCount, 5);
    cout << "The winner is Candidate " << winner << endl;

    return 0;
}
```

Output Console Showing Sorted Array	Manual Count	Count Result of Algorithm
Unsorted Votes: 4 2 3 5 4 1 5 5 3 5 1 3 1 2 1 1 5 2 1 4 5 1 5 5 5 2 2 4 5 3 2 1 1 4 5 5 2 1 1 4 2 1 4 4 4 4 4 5 3 2 3 2 4 2 1 3 5 4 4 2 4 5 2 4 5 3 3 3 5 3 4 1 5 2 5 3 2 3 2 4 1 1 3 5 3 5 4 4 1 2 5 4 3 3 2 5 5 2 2 1 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 2 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 5 5 5 5	Vote Count: Candidate 1: 18 votes Candidate 2: 20 votes Candidate 3: 17 votes Candidate 4: 21 votes Candidate 5: 24 votes	The winner is Candidate 5
Unsorted Votes: 5 2 3 3 4 5 3 1 5 3 4 4 2 3 2 4 3 2 5 5 5 4 3 1 5 1 3 4 3 2 1 2 3 3 4 1 4 4 3 5 1 1 1 4 1 2 3 3 5 2 4 1 3 3 1 2 1 3 3 3 4 5 5 3 2 5 4 3 3 1 2 1 4 4 4 4 5 3 3 1 5 2 2 4 4 2 2 1 2 4 4 5 5 5 5 2 4 3 1 4 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 3 3 3 3 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 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5	Vote Count: Candidate 1: 17 votes Candidate 2: 17 votes Candidate 3: 25 votes Candidate 4: 23 votes Candidate 5: 18 votes	The winner is Candidate 3
Unsorted Votes: 4 2 1 4 2 4 1 1 1 4 1 4 2 5 1 2 2 2 3 4 1 2 4 5 1 2 5 5 5 4 1 5 2 3 5 4 1 5 4 4 3 1 2 1 5 4 4 3 1 4 2 1 2 2 5 4 3 2 4 4 5 1 3 3 5 2 3 3 3 3 1 2 4 4 5 5 4 5 5 4 3 1 1 5 4 3 5 1 4 5 5 5 5 4 2 2 3 5 4 5 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 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 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 Count: Candidate 1: 19 votes Candidate 2: 18 votes Candidate 3: 14 votes Candidate 4: 25 votes Candidate 5: 24 votes	The winner is Candidate 4

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

- Yes, because the algorithm used straightforward sorting and counting techniques to get accurate, dependable results, it was successful in solving the challenge. It was a workable method for vote counting because it was effective for the scale of the problem and the reasoning was simple to apply.

8. Conclusion

In conclusion, the fundamental algorithms of bubble, selection, and insertion sort serve as examples of important ideas in efficiency and sorting. Despite being straightforward, bubble sort's $O(n^2)$ time complexity makes it ineffective for large datasets. Selection sort provides a simpler selection procedure but likewise has $O(n^2)$ performance. Insertion sort is useful for smaller lists since it works better with partially sorted material. When combined, these algorithms offer a vital starting point for comprehending more complex sorting strategies, highlighting the need for programming that strikes a compromise between simplicity and effectiveness.

9. Assessment Rubric