Activity No. 7

SORTING ALGORITHMS: BUBBLE, SELECTION, AND INSERTION SORT

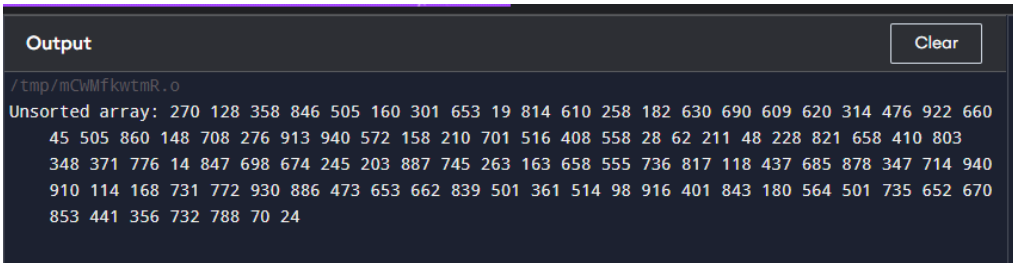
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/16/2024
Section: CPE21S4	**Date Submitted:** 10/16/2024
Name(s): Edson Ray E. San Juan	**Instructor:** Prof. Maria Rizette Sayo

6. Output

Code + Console Screenshot

Code:
```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

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

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

    // Fill the array with random values
    for (int i = 0; i < SIZE; ++i) {
        arr[i] = rand() % 1000;  // Generate random numbers between 0 and 999
    }

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

    cout << endl;

    return 0;
}
```
—---

Output:



```
/tmp/mCWMfkwtmR.o
Unsorted array: 270 128 358 846 505 160 301 653 19 814 610 258 182 630 690 609 620 314 476 922 660
    45 505 860 148 708 276 913 940 572 158 210 701 516 408 558 28 62 211 48 228 821 658 410 803
    348 371 776 14 847 698 674 245 203 887 745 263 163 658 555 736 817 118 437 685 878 347 714 940
    910 114 168 731 772 930 886 473 653 662 839 501 361 514 98 916 401 843 180 564 501 735 652 670
    853 441 356 732 788 70 24
```

Observations

The C++ code creates an array of 100 random numbers between 0 and 999 using

	rand(). The srand(time(0)) ensures the numbers change every time the program runs. The numbers are not sorted and are printed in the order they're generated.

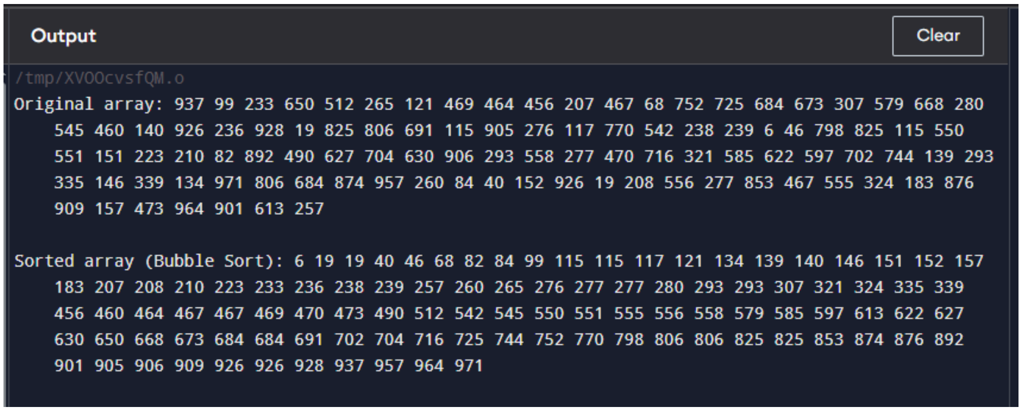
Table 7-1. Array of Values for Sort Algorithm Testing

Code + Console Screenshot

Code:

```
// SortingAlgorithms.h
#ifndef SORTINGALGORITHMS_H
#define SORTINGALGORITHMS_H

class SortingAlgorithms {
public:
    // Bubble Sort
    static void bubbleSort(int arr[], int size);

    // Insertion Sort
    static void insertionSort(int arr[], int size);

    // Selection Sort
    static void selectionSort(int arr[], int size);

    // Merge Sort
    static void mergeSort(int arr[], int left, int right);

    // Quick Sort
    static void quickSort(int arr[], int left, int right);

private:
    // Helper function for Merge Sort to merge two subarrays
    static void merge(int arr[], int left, int mid, int right);

    // Helper function for Quick Sort to partition the array
    static int partition(int arr[], int left, int right);
};

#endif // SORTINGALGORITHMS_H
—-------------------------
// main.cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "SortingAlgorithms.h"  // Import the header file

using namespace std;

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]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

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

    srand(static_cast<unsigned int>(time(0)));

    for (int i = 0; i < SIZE; ++i) {
        arr[i] = rand() % 1000;  // Generate random numbers between 0 and 999
    }

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

    bubbleSort(arr, SIZE);

    cout << "\nSorted array (Bubble Sort): ";
    for (int i = 0; i < SIZE; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```
—------------------------

Output:



```
/tmp/XVOOcvsfQM.o
Original array: 937 99 233 650 512 265 121 469 464 456 207 467 68 752 725 684 673 307 579 668 280
    545 460 140 926 236 928 19 825 806 691 115 905 276 117 770 542 238 239 6 46 798 825 115 550
    551 151 223 210 82 892 490 627 704 630 906 293 558 277 470 716 321 585 622 597 702 744 139 293
    335 146 339 134 971 806 684 874 957 260 84 40 152 926 19 208 556 277 853 467 555 324 183 876
    909 157 473 964 901 613 257

Sorted array (Bubble Sort): 6 19 19 40 46 68 82 84 99 115 115 117 121 134 139 140 146 151 152 157
    183 207 208 210 223 233 236 238 239 257 260 265 276 277 277 280 293 293 307 321 324 335 339
    456 460 464 467 467 469 470 473 490 512 542 545 550 551 555 556 558 579 585 597 613 622 627
    630 650 668 673 684 684 691 702 704 716 725 744 752 770 798 806 806 825 825 853 874 876 892
    901 905 906 909 926 926 928 937 957 964 971
```

Observations

The SortingAlgorithms.h header file is organized and uses templates for flexibility, with function declarations and include guards to prevent multiple inclusions.

	In the .cpp file, sorting algorithms like Bubble Sort are clearly implemented and use standard library functions for efficiency. The main function generates random numbers to test the sorting and prints both the original and sorted arrays.

Table 7-2. Bubble Sort Technique

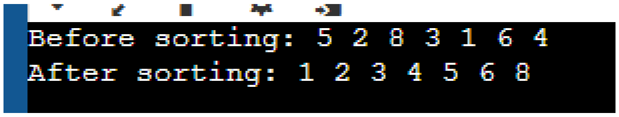
Code + Console Screenshot

Code:

```
// SortingAlgorithms.h
#ifndef SORTINGALGORITHMS_H
#define SORTINGALGORITHMS_H

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

template <typename T>
int Routine_Smallest(T A[], int K, const int arrSize);

#endif // SORTINGALGORITHMS_H
—-----------------
// selectionSort.cpp
#include "SortingAlgorithms.h"

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++;
    }
}

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;
}

// Explicitly instantiate the template functions for int type
```

```
template void selectionSort<int>(int arr[], const int N);
template int Routine_Smallest<int>(int A[], int K, const int arrSize);
—-----------------------------
// main.cpp
#include <iostream>
#include "SortingAlgorithms.h"

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

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

   selectionSort(arr, N);

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

   return 0;
}
—--------------------------------
```

Output:



```
Before sorting: 5 2 8 3 1 6 4
After sorting: 1 2 3 4 5 6 8
```

Observations

The SortingAlgorithms.h file declares the selectionSort and Routine_Smallest functions as templates, allowing them to work with various data types, but it does not include their implementations. The sortingAlgorithms.cpp file provides these implementations, utilizes templates, and includes the header for function declarations. The main.cpp file uses the selectionSort function to sort an integer array.

Table 7-3. Selection Sort Algorithm

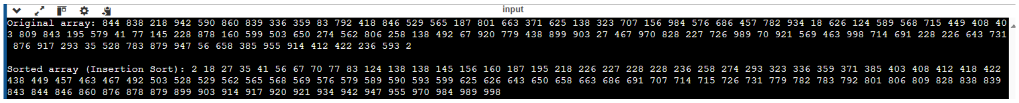
Code + Console Screenshot

Code:

```
// SortingAlgorithms.h
#ifndef SORTINGALGORITHMS_H
#define SORTINGALGORITHMS_H

// Function declarations for sorting algorithms
template <typename T>
```

```
void insertionSort(T arr[], const int N);

#endif // SORTINGALGORITHMS_H
—-------------------------------
// insertionSort.cpp
#include "SortingAlgorithms.h"

template <typename T>
void insertionSort(T arr[], const int N) {
    int K = 1;
    while (K < N) {
        T temp = arr[K];
        int J = K - 1;

        while (J >= 0 && temp < arr[J]) {
            arr[J + 1] = arr[J];
            J--;
        }
        arr[J + 1] = temp;
        K++;
    }
}

// Explicit instantiation for int type
template void insertionSort<int>(int arr[], const int N);
—--------------------------------------
//main.cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "SortingAlgorithms.h"  // Import the header file

using namespace std;

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

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

    for (int i = 0; i < SIZE; ++i) {
        arr[i] = rand() % 1000;  // Generate random numbers between 0 and 999
    }

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

    insertionSort(arr, SIZE);  // Sort the array using Insertion Sort
    cout << "\nSorted array (Insertion Sort): ";
    for (int i = 0; i < SIZE; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
—------------------------
```

Output:



```
Original array: 844 838 218 942 590 860 839 336 359 83 792 418 846 529 565 187 801 663 371 625 138 323 707 156 984 576 686 457 782 934 18 626 124 589 568 715 449 408 40
3 809 843 195 579 41 77 145 228 878 160 599 503 650 274 562 806 258 138 492 67 920 779 438 899 903 27 467 970 828 227 726 989 70 921 569 463 998 714 691 228 226 643 731
 876 917 293 35 528 783 879 947 56 658 385 955 914 412 422 236 593 2

Sorted array (Insertion Sort): 2 18 27 35 41 56 67 70 77 83 124 138 138 145 156 160 187 195 218 226 227 228 228 236 258 274 293 323 336 359 371 385 403 408 412 418 422
438 449 457 463 467 492 503 528 529 562 565 568 569 576 579 589 590 593 599 625 626 643 650 658 663 686 691 707 714 715 726 731 779 782 783 792 801 806 809 828 838 839
843 844 846 860 876 878 879 899 903 914 917 920 921 934 942 947 955 970 984 989 998
```

Observations	The SortingAlgorithms.h file declares the insertionSort template function and prevents multiple inclusions with include guards. The insertionSort.cpp file implements this function and explicitly creates an integer version. In main.cpp, an array of random integers is generated, and both the unsorted and sorted arrays are displayed after sorting. The code is organized into separate files for clarity and uses srand(time(0)) for generating different random numbers each time it runs.

Table 7-4. Insertion Sort Algorithm

7. Supplementary Activity

ILO B: Solve given data sorting problems using appropriate basic sorting algorithms

Candidate 1	Bo Dalton Capistrano
Candidate 2	Cornelius Raymon Agustín
Candidate 3	Deja Jayla Bañaga
Candidate 4	Lalla Brielle Yabut
Candidate 5	Franklin Relano Castro

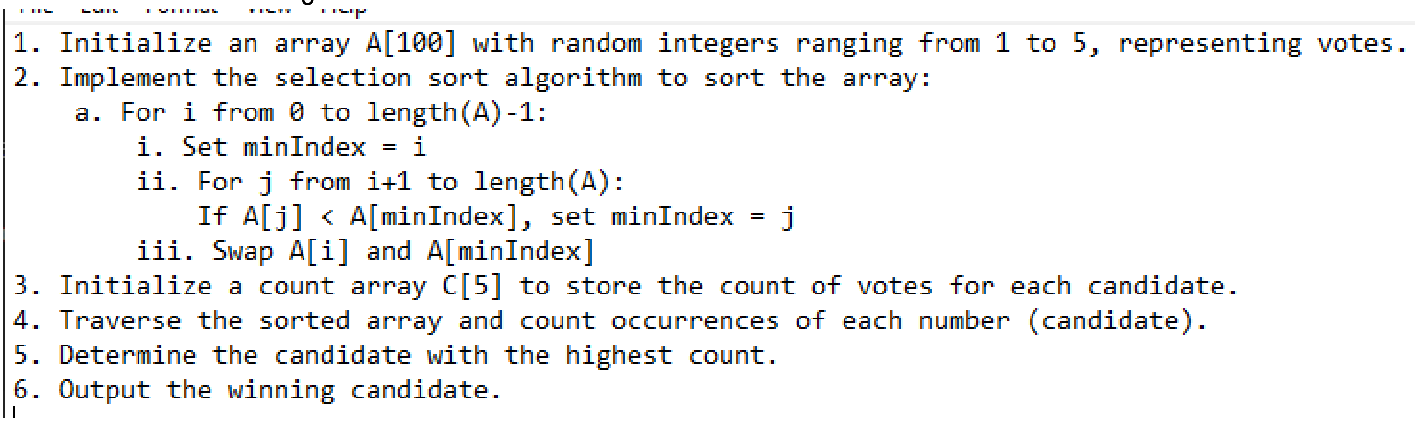
List of Candidates

Problem: Generate an array A[0…100] of unsorted elements, wherein the values in the array are indicative of a vote to a candidate. This means that the values in your array must only range from 1 to 5. Using sorting and searching techniques, develop an algorithm that will count the votes and indicate the winning candidate.

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.

Deliverables:

- Pseudocode of Algorithm

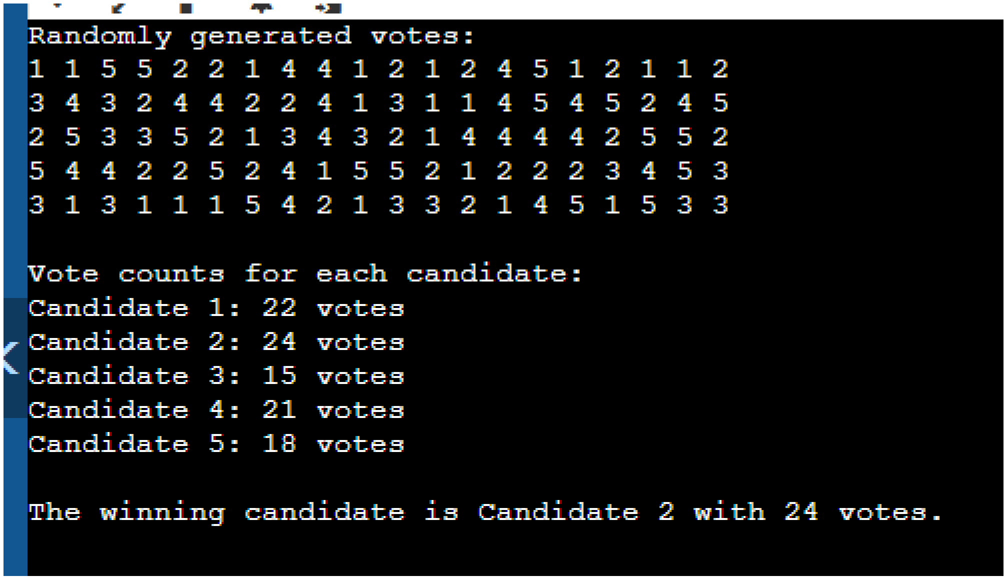


```
1. Initialize an array A[100] with random integers ranging from 1 to 5, representing votes.
2. Implement the selection sort algorithm to sort the array:
    a. For i from 0 to length(A)-1:
        i. Set minIndex = i
        ii. For j from i+1 to length(A):
            If A[j] < A[minIndex], set minIndex = j
        iii. Swap A[i] and A[minIndex]
3. Initialize a count array C[5] to store the count of votes for each candidate.
4. Traverse the sorted array and count occurrences of each number (candidate).
5. Determine the candidate with the highest count.
6. Output the winning candidate.
```

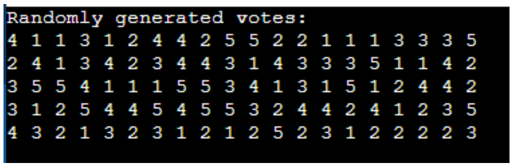
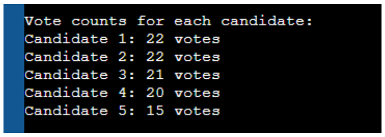
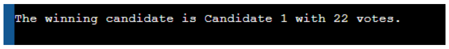
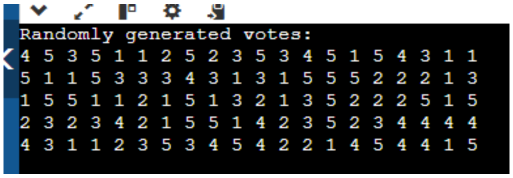
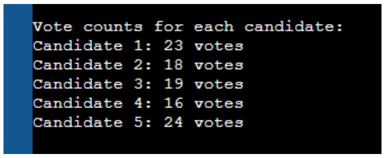
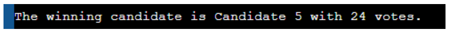
- Screenshot of Algorithm Code

```
#include <iostream>
#include <cstdlib>  // For rand() and srand()
#include <ctime>    // For time()

using namespace std;

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

void displayVoteResults(int arr[], int n) {
    int count[5] = {0};
    for (int i = 0; i < n; i++) {
        count[arr[i] - 1]++;
    }

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

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

    cout << "\nThe winning candidate is Candidate " << winner << " with " << maxVotes << " votes." << endl;
}

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

    const int size = 100;
    int votes[size];

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

    cout << "Randomly generated votes:" << endl;
    for (int i = 0; i < size; i++) {
        cout << votes[i] << " ";
        if ((i + 1) % 20 == 0)
            cout << endl;
    }
    cout << endl;

    selectionSort(votes, size);
    displayVoteResults(votes, size);

    return 0;
}
```

- Output Testing



```
Randomly generated votes:
1 1 5 5 2 2 1 4 4 1 2 1 2 4 5 1 2 1 1 2
3 4 3 2 4 4 2 2 4 1 3 1 1 4 5 4 5 2 4 5
2 5 3 3 5 2 1 3 4 3 2 1 4 4 4 4 2 5 5 2
5 4 4 2 2 5 2 4 1 5 5 2 1 2 2 2 3 4 5 3
3 1 3 1 1 1 5 4 2 1 3 3 2 1 4 5 1 5 3 3

Vote counts for each candidate:
Candidate 1: 22 votes
Candidate 2: 24 votes
Candidate 3: 15 votes
Candidate 4: 21 votes
Candidate 5: 18 votes

The winning candidate is Candidate 2 with 24 votes.
```

Output Console Showing Sorted Array	Manual Count	Count Result of Algorithm
 Randomly generated votes: 4 1 1 3 1 2 4 4 2 5 5 2 2 1 1 1 3 3 3 5 2 4 1 3 4 2 3 4 4 3 1 4 3 3 3 5 1 1 4 2 3 5 5 4 1 1 1 5 5 3 4 1 3 1 5 1 2 4 4 2 3 1 2 5 4 4 5 4 5 5 3 2 4 4 2 4 1 2 3 5 4 3 2 1 3 2 3 1 2 1 2 5 2 3 1 2 2 2 2 3	 Vote counts for each candidate: Candidate 1: 22 votes Candidate 2: 22 votes Candidate 3: 21 votes Candidate 4: 20 votes Candidate 5: 15 votes	 The winning candidate is Candidate 1 with 22 votes.
 Randomly generated votes: 4 5 3 5 1 1 2 5 2 3 5 3 4 5 1 5 4 3 1 1 5 1 1 5 3 3 3 4 3 1 3 1 5 5 5 2 2 2 1 3 1 5 5 1 1 2 1 5 1 3 2 1 3 5 2 2 2 5 1 5 2 3 2 3 4 2 1 5 5 1 4 2 3 5 2 3 4 4 4 4 4 3 1 1 2 3 5 3 4 5 4 2 2 1 4 5 4 4 1 5	 Vote counts for each candidate: Candidate 1: 23 votes Candidate 2: 18 votes Candidate 3: 19 votes Candidate 4: 16 votes Candidate 5: 24 votes	 The winning candidate is Candidate 5 with 24 votes.

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

The vote-counting algorithm is effective because it correctly counts votes for each candidate and identifies the winner. It handles 100 votes efficiently, and the output is clear, showing the vote totals and the winning candidate. However, the sorting method used (selection sort) is slow for large datasets, and the algorithm doesn't handle ties, which could be improved.

8. Conclusion

Provide the following:

- Summary of lessons learned

Sorting algorithms are essential for organizing data in many fields, including data analysis, e-commerce, and everyday tasks like managing emails and music playlists. Understanding these algorithms helps us process information more efficiently.

- Analysis of the procedure

We studied various sorting algorithms and their effectiveness in different situations. For example, Quick Sort is great for large datasets, while Bubble Sort is simpler but works well for smaller ones. Each algorithm has its own strengths based on the data type and size.

- Analysis of the supplementary activity

The supplementary activities showed how sorting algorithms are used in real life, like organizing medical records or e-commerce data. This helped me see the practical importance of these algorithms beyond just theory.

- Concluding statement / Feedback: How well did you think you did in this activity? What are your areas for improvement?

I feel I did well in this activity by understanding both the theory and real-world applications of sorting algorithms. However, I want to improve my knowledge of more complex algorithms and their use in different technologies. My goal is to better match sorting methods to specific problems.

9. Assessment Rubric