Activity No. 8	
Sorting Algorithms	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/21/2024
Section: CPE21S4	**Date Submitted:** 10/22/2024
Name(s): Edson Ray E. San Juan	**Instructor:** Prof. Maria Rizette Sayo
6. Output	

Code + Console Screenshot

```
// sortAlgo.h
#ifndef SORT_ALGOR_H
#define SORT_ALGOR_H

#include <iostream>
using namespace std;

// Function prototypes for sorting algorithms
void bubbleSort(int arr[], int size);
void selectionSort(int arr[], int size);
void insertionSort(int arr[], int size);
void mergeSort(int arr[], int left, int right);
void shellSort(int arr[], int size);
void quickSort(int arr[], int low, int high);
void displayArray(int arr[], int size);

// Bubble Sort implementation
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

// Selection Sort implementation
void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}
```

```
// Insertion Sort implementation
void insertionSort(int arr[], int size) {
    for (int i = 1; i < size; 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;
    }
}

// Merge Sort implementation
void merge(int arr[], int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int leftArr[n1], rightArr[n2];

    for (int i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArr[j] = arr[middle + 1 + j];
    }

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
```

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;

        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);

        merge(arr, left, middle, right);
    }
}

// Shell Sort implementation
void shellSort(int arr[], int size) {
    for (int interval = size / 2; interval > 0; interval /= 2) {
        for (int i = interval; i < size; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= interval && arr[j - interval] > temp; j
-= interval) {
                arr[j] = arr[j - interval];
            }
            arr[j] = temp;
        }
    }
}

// Quick Sort implementation
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low , high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to display the array
void displayArray(int arr[], int size) {  // Ensure this is
```

```
defined
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}


#endif  // SORT_ALGOR_H
```

```
// main.cpp
#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

void generateRandomArray(int arr[], int size) {
    // Seed the random number generator
    srand(time(0));

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

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " "; // Using cout from the standard
namespace
    }
    cout << endl;
}

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

    generateRandomArray(randomArray, size);
    cout << "Random Array: ";
    printArray(randomArray, size);

    return 0;
}
```

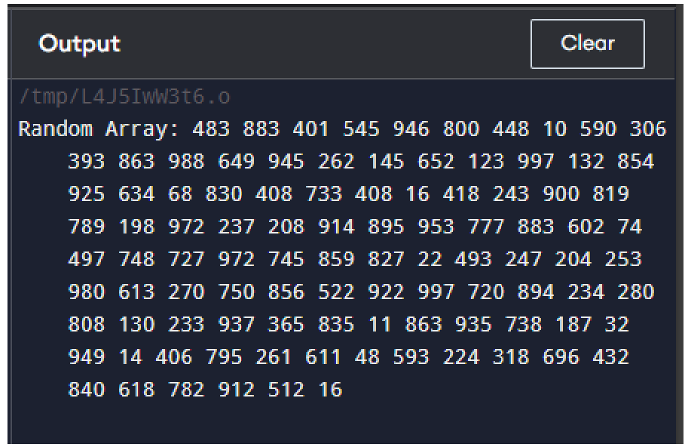
	 Output Clear /tmp/L4J5IwW3t6.o Random Array: 483 883 401 545 946 800 448 10 590 306 393 863 988 649 945 262 145 652 123 997 132 854 925 634 68 830 408 733 408 16 418 243 900 819 789 198 972 237 208 914 895 953 777 883 602 74 497 748 727 972 745 859 827 22 493 247 204 253 980 613 270 750 856 522 922 997 720 894 234 280 808 130 233 937 365 835 11 863 935 738 187 32 949 14 406 795 261 611 48 593 224 318 696 432 840 618 782 912 512 16
Observations	The generateRandomArray function uses srand(time(0)) to create different random numbers each run, generating values between 0 and 999 with rand() % 1000. The printArray function displays these values clearly, and can be formatted in a table for better readability. The code is modular for easier maintenance, and const int size = 100; allows for quick adjustments to the array size.

Table 8-1. Array of Values for Sort Algorithm Testing

Code + Console Screenshot	#include <iostream>
	#include <ctime>
	#include <cstdlib>
	
	using namespace std; // Correct placement of the using directive
	
	void generateRandomArray(int arr[], int size) {
	    srand(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
	    }
	}
	
	void printArray(int arr[], int size) {
	    for (int i = 0; i < size; i++) {
	        cout << arr[i] << " ";
	    }
	    cout << endl;
	}
	
	void shellSort(int array[], int size) {
	    int interval = size / 2; // Start with a large interval

	<pre> while (interval > 0) {
	        for (int i = interval; i < size; i++) {
	            int temp = array[i];
	            int j = i;
	
	            // Insertion sort for the interval
	            while (j >= interval && array[j - interval] > temp) {
	                array[j] = array[j - interval];
	                j -= interval;
	            }
	            array[j] = temp; // Insert the element at the correct
	position
	        }
	        interval /= 2; // Reduce the interval
	    }
	}
	
	int main() {
	    const int size = 100;
	    int randomArray[size];
	
	    generateRandomArray(randomArray, size);
	    cout << "Random Array: ";
	    printArray(randomArray, size);
	
	    shellSort(randomArray, size);
	    cout << "\nSorted Array: ";
	    printArray(randomArray, size);
	
	    return 0;
	}</pre>

```
Output                                                    Clear
/tmp/wdELzfx9Jf.o
Random Array: 394 357 927 540 558 590 771 539 321
    246 684 811 910 62 629 749 936 668 883 173 74
    566 233 584 31 667 143 170 10 986 523 756 343
    802 649 253 393 420 792 714 18 829 877 928 891
    858 30 179 526 913 353 952 480 586 536 863 253
    32 385 263 18 908 371 713 711 20 967 104 440 111
    170 459 940 399 387 184 257 769 363 135 683 716
    88 515 654 976 378 259 360 763 522 378 671 246
    92 734 266 411 190 707

Sorted Array: 10 18 18 20 30 31 32 62 74 88 92 104
    111 135 143 170 170 173 179 184 190 233 246 246
    253 253 257 259 263 266 321 343 353 357 360 363
    371 378 378 385 387 393 394 399 411 420 440 459
    480 515 522 523 526 536 539 540 558 566 584 586
    590 629 649 654 667 668 671 683 684 707 711 713
    714 716 734 749 756 763 769 771 792 802 811 829
    858 863 877 883 891 908 910 913 927 928 936 940
    952 967 976 986
```

Observations	The generateRandomArray function creates a random array each time by seeding with the current time and generates numbers from 0 to 999. The printArray function displays the values clearly, The Shell Sort is faster than simpler algorithms like bubble or insertion sort, and its gap reduction method (dividing by 2) can be optimized further.

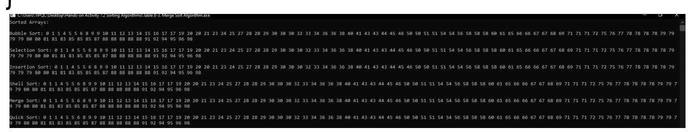
Table 8-2. Shell Sort Technique

Code + Console Screenshot

```
// main.cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "sortAlgo.h"
using namespace std;

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

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

    // Generate random array
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 100;  // Random values between 0
```

```
and 99
    }

    // Display the generated random array
    cout << "Random Array:" << endl;
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl << endl;

    // Apply sorting algorithms
    int arrBubble[size];
    int arrSelection[size];
    int arrInsertion[size];
    int arrShell[size];
    int arrMerge[size];
    int arrQuick[size];

    for (int i = 0; i < size; i++) {
        arrBubble[i] = arr[i];
        arrSelection[i] = arr[i];
        arrInsertion[i] = arr[i];
        arrShell[i] = arr[i];
        arrMerge[i] = arr[i];
        arrQuick[i] = arr[i];
    }

    bubbleSort(arrBubble, size);
    selectionSort(arrSelection, size);
    insertionSort(arrInsertion, size);
    shellSort(arrShell, size);
    mergeSort(arrMerge, 0, size - 1);
    quickSort(arrQuick, 0, size - 1);

    // Display the sorted arrays
    cout << "Sorted Arrays:" << endl;
    cout << "\nBubble Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrBubble[i] << " ";
    }
    cout << endl;

    cout << "\nSelection Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrSelection[i] << " ";
    }
    cout << endl;

    cout << "\nInsertion Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrInsertion[i] << " ";
```

```
    }
    cout << endl;

    cout << "\nShell Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrShell[i] << " ";
    }
    cout << endl;

    cout << "\nMerge Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrMerge[i] << " ";
    }
    cout << endl;

    cout << "\nQuick Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrQuick[i] << " ";
    }
    cout << endl;

    return 0;
}
```



Observations	The code generates a random array of integers, sorts it using six different sorting algorithms (Bubble Sort, Selection Sort, Insertion Sort, Shell Sort, Merge Sort, and Quick Sort), and displays both the original and sorted arrays. Each sorting algorithm operates on a copy of the original array to showcase the sorted results separately.

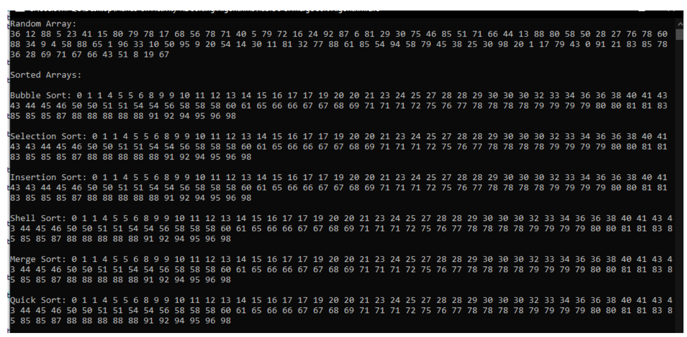
Table 8-3. Merge Sort Algorithm

Code + Console Screenshot

```
// main.cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "sortAlgo.h"
using namespace std;

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

    const int size = 100;
    int arr[size];
```

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

    cout << "Random Array:" << endl;
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl << endl;

    // Apply sorting algorithms
    int arrBubble[size];
    int arrSelection[size];
    int arrInsertion[size];
    int arrShell[size];
    int arrMerge[size];
    int arrQuick[size];

    for (int i = 0; i < size; i++) {
        arrBubble[i] = arr[i];
        arrSelection[i] = arr[i];
        arrInsertion[i] = arr[i];
        arrShell[i] = arr[i];
        arrMerge[i] = arr[i];
        arrQuick[i] = arr[i];
    }

    bubbleSort(arrBubble, size);
    selectionSort(arrSelection, size);
    insertionSort(arrInsertion, size);
    shellSort(arrShell, size);
    mergeSort(arrMerge, 0, size - 1);
    quickSort(arrQuick, 0, size - 1);

    // Display the sorted arrays
    cout << "\nSorted Arrays:" << endl;
    cout << "Bubble Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrBubble[i] << " ";
    }
    cout << endl;

    cout << "\nSelection Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrSelection[i] << " ";
    }
    cout << endl;

    cout << "\nInsertion Sort: ";
```

```
    for (int i = 0; i < size; i++) {
        cout << arrInsertion[i] << " ";
    }
    cout << endl;

    cout << "\nShell Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrShell[i] << " ";
    }
    cout << endl;

    cout << "\nMerge Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrMerge[i] << " ";
    }
    cout << endl;

    cout << "\nQuick Sort: ";
    for (int i = 0; i < size; i++) {
        cout << arrQuick[i] << " ";
    }
    cout << endl;

    return 0;
}
```



Observations	The code generates a random array of integers, applies six different sorting algorithms (Bubble Sort, Selection Sort, Insertion Sort, Shell Sort, Merge Sort, and Quick Sort), and prints both the original and sorted arrays for comparison. Each algorithm operates on a separate copy of the array, ensuring that the original random values remain unchanged throughout the sorting process.

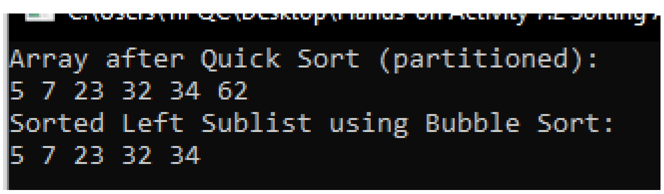
Table 8-4. Quick Sort Algorithm

7. Supplementary Activity

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

Problem 1: Can we sort the left sub list and right sub list from the partition method in quick sort using other sorting algorithms? Demonstrate an example.

```
// main.cpp
#include <iostream>
#include <cstdlib>
#include "sortAlgo.h"
using namespace std;

int main() {
    int arr[] = {34, 7, 23, 32, 5, 62};
    int size = sizeof(arr) / sizeof(arr[0]);

    // Step 1: Perform Quick Sort to partition the array
    quickSort(arr, 0, size - 1);

    // Display the array after Quick Sort
    cout << "Array after Quick Sort (partitioned):" << endl;
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Step 2: Sort the left sublist using Bubble Sort
    int leftSublistSize = 5; // Size of the left sublist
    int leftSublist[5] = {34, 7, 23, 32, 5}; // Left sublist

    bubbleSort(leftSublist, leftSublistSize);

    // Display the sorted left sublist
    cout << "Sorted Left Sublist using Bubble Sort:" << endl;
    for (int i = 0; i < leftSublistSize; i++) {
        cout << leftSublist[i] << " ";
    }
    cout << endl;

    return 0;
}
```

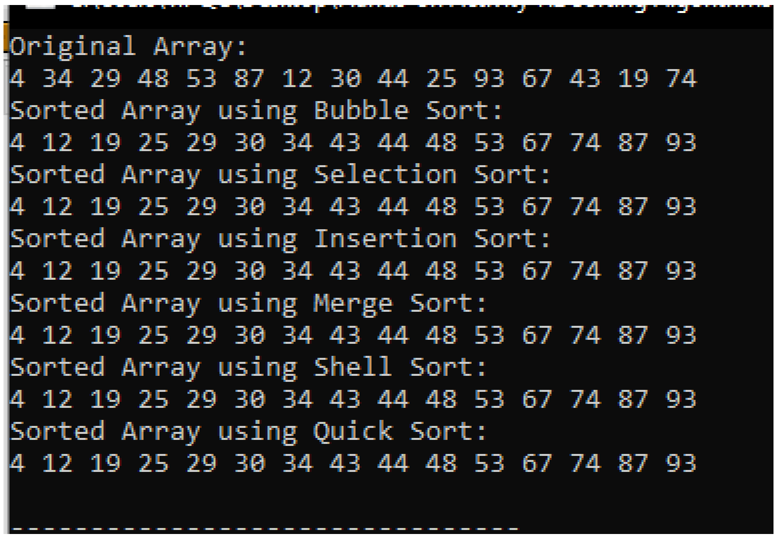


Problem 2: Suppose we have an array which consists of {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19, 74}. What sorting algorithm will give you the fastest time performance? Why can merge sort and quick sort have O(N • log N) for their time complexity?

```
// main.cpp
#include <iostream>
#include "sortAlgo_prob2-supp.h"
using namespace std;

int main() {
    int arr[] = {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19, 74};
```

```
int size = sizeof(arr) / sizeof(arr[0]);

// Display original array
cout << "Original Array:" << endl;
displayArray(arr, size);

// Sort using Bubble Sort
int bubbleSortedArray[15];
for (int i = 0; i < size; i++) {
    bubbleSortedArray[i] = arr[i];
}
bubbleSort(bubbleSortedArray, size);
cout << "Sorted Array using Bubble Sort:" << endl;
displayArray(bubbleSortedArray, size);

// Sort using Selection Sort
int selectionSortedArray[15];
for (int i = 0; i < size; i++) {
    selectionSortedArray[i] = arr[i];
}
selectionSort(selectionSortedArray, size);
cout << "Sorted Array using Selection Sort:" << endl;
displayArray(selectionSortedArray, size);

// Sort using Insertion Sort
int insertionSortedArray[15];
for (int i = 0; i < size; i++) {
    insertionSortedArray[i] = arr[i];
}
insertionSort(insertionSortedArray, size);
cout << "Sorted Array using Insertion Sort:" << endl;
displayArray(insertionSortedArray, size);

// Sort using Merge Sort
int mergeSortedArray[15];
for (int i = 0; i < size; i++) {
    mergeSortedArray[i] = arr[i];
}
mergeSort(mergeSortedArray, 0, size - 1);
cout << "Sorted Array using Merge Sort:" << endl;
displayArray(mergeSortedArray, size);

// Sort using Shell Sort
int shellSortedArray[15];
for (int i = 0; i < size; i++) {
    shellSortedArray[i] = arr[i];
}
shellSort(shellSortedArray, size);
cout << "Sorted Array using Shell Sort:" << endl;
displayArray(shellSortedArray, size);
```

```
    // Sort using Quick Sort
    int quickSortedArray[15];
    for (int i = 0; i < size; i++) {
        quickSortedArray[i] = arr[i];
    }
    quickSort(quickSortedArray, 0, size - 1);
    cout << "Sorted Array using Quick Sort:" << endl;
    displayArray(quickSortedArray, size);

    return 0;
}
```



```
Original Array:
4 34 29 48 53 87 12 30 44 25 93 67 43 19 74
Sorted Array using Bubble Sort:
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Sorted Array using Selection Sort:
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Sorted Array using Insertion Sort:
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Sorted Array using Merge Sort:
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Sorted Array using Shell Sort:
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Sorted Array using Quick Sort:
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93

--------------------------------
```

8. Conclusion

Provide the following:

• Summary of lessons learned

In this experiment, I learned how different sorting algorithms like Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Shell Sort, and Quick Sort work.

• Analysis of the procedure

Implementing and testing the sorting algorithms allowed me to see how each one works on a random array.

• Analysis of the supplementary activity

The supplementary task of combining Quick Sort with other algorithms showed how hybrid approaches can optimize sorting.

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

I did well in implementing the sorting algorithms. I can improve by focusing on code optimization and exploring advanced techniques.

9. Assessment Rubric