Activity No. 8	
SORTING ALGORITHMS: SHELL, MERGE, AND QUICK SORT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 21/10/2024
Section: CPE21S4	**Date Submitted:**
Name(s): Mamaril, Justin Kenneth I.	**Instructor:** Prof. Sayo

6. Output

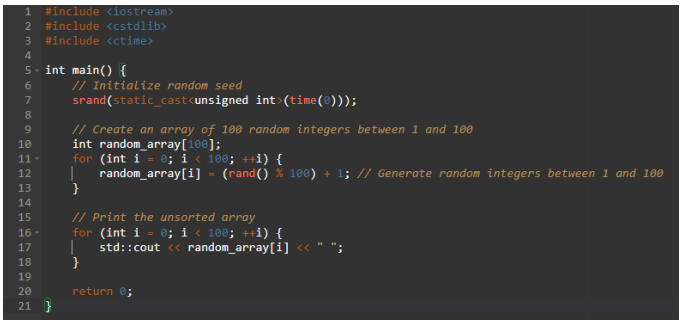
Code + Console Screenshot	
Observation	The program creates an array of 100 random integers between 1 and 100 and prints them out. It uses libraries to handle input/output, random number generation, and time management.

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

Code + Console Screenshot	

<table>
<tr>
<td></td>
<td>

```
#ifndef SORTING_ALGORITHM_H
#define SORTING_ALGORITHM_H

// Function to perform Shell Sort
void shellSort(int arr[], int size);

// Function to print the array
void printArray(int arr[], int size);

#endif // SORTING_ALGORITHM_H
```

main.cpp | SortingAlgorithm.h | SortingAlgorithm.cpp

```
#include <iostream>
#include "SortingAlgorithm.h"

// Function to perform Shell Sort
void shellSort(int arr[], int size) {
    // Start with a big gap, then reduce the gap
    for (int interval = size / 2; interval > 0; interval /= 2) {
        // Do a gapped insertion sort for this interval size
        for (int i = interval; i < size; i++) {
            // Save the value at the current position
            int temp = arr[i];
            int j;
            // Shift earlier gap-sorted elements up until the correct location for arr[i] is found
            for (j = i; j >= interval && arr[j - interval] > temp; j -= interval) {
                arr[j] = arr[j - interval];
            }
            // Put temp (the original arr[i]) in its correct location
            arr[j] = temp;
        }
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        std::cout << arr[i] << " ";
    std::cout << std::endl;
}
```

input

```
Unsorted array: 58 8 57 20 3 13 79 67 76 63 32 58 36 78 82 96 21 23 2 41 31 10 57 3 3 28
13 52 95 67 75 52 74 32 71 28 44 2 94 20 64 25 29 51 54 10 99 26 33 52 67 63 62 23 17 64
50 29 67 45 47 42 48 72 73 71 99 16 72 92 87 87 17 15 89 22 25 87 48 57 39 66 71 100 88 8
8 63 90 16 81 86 15 22 33 86 94 3 85 62 26
Sorted array using Shell Sort: 2 2 3 3 3 3 8 10 10 13 13 15 15 16 16 17 17 20 20 21 22 22
 23 23 25 25 26 26 28 28 29 29 31 32 32 33 33 36 39 41 42 44 45 47 48 48 50 51 52 52 52 5
4 57 57 57 58 58 62 62 63 63 63 64 64 66 67 67 67 67 71 71 71 72 72 73 74 75 76 78 79 81
82 85 86 86 87 87 87 88 88 89 90 92 94 94 95 96 99 99 100


...Program finished with exit code 0
Press ENTER to exit console.
```

</td>
</tr>
<tr>
<td>Observation</td>
<td>the main.cpp handles the generation of the random array, calls the sorting function, and prints both the unsorted and sorted arrays. The SortingAlgorithm.h contains the declarations for the sorting and printing functions. While the SortingAlgorithm.cpp Implements the Shell Sort algorithm and the function to print the array. In the end It generates an array of 100 random integers, prints the unsorted array, sorts it using Shell Sort, and then prints the sorted array. Shell Sort improves upon the efficiency of insertion sort by allowing the exchange of elements that are far apart, reducing the number of inversions in the array. It is particularly effective for medium-sized arrays and is simple to implement.</td>
</tr>
</table>

Table 8-2. Shell Sort Technique

Code + Console Screenshot

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "SortingAlgorithm.h"

// Function to generate an array of random integers
void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = (rand() % 100) + 1; // Random values between 1 and 100
    }
}

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

    // Seed for random number generation
    srand(static_cast<unsigned int>(time(0)));

    // Generate random array
    generateRandomArray(arr, SIZE);

    // Print the original unsorted array
    std::cout << "Unsorted array: ";
    printArray(arr, SIZE);

    // Sort using Shell Sort
    shellSort(arr, SIZE);
    std::cout << "Sorted array using Shell Sort: ";
    printArray(arr, SIZE);

    return 0;
}
```

main.cpp | SortingAlgorithm.h | SortingAlgorithm.cpp

```cpp
#ifndef SORTING_ALGORITHM_H
#define SORTING_ALGORITHM_H

// Function to perform Merge Sort
void mergeSort(int arr[], int left, int right);

// Function to print the array
void printArray(int arr[], int size);

#endif // SORTING_ALGORITHM_H
```

main.cpp | SortingAlgorithm.h | SortingAlgorithm.cpp

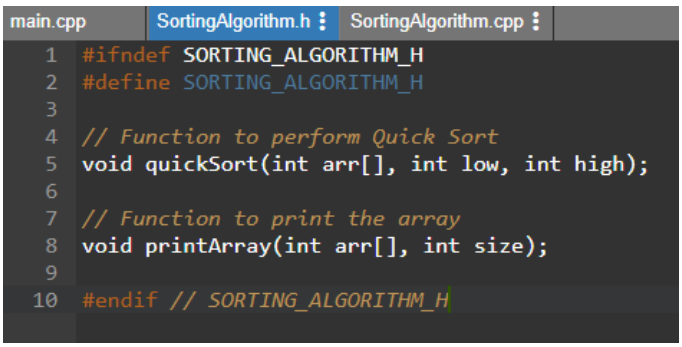
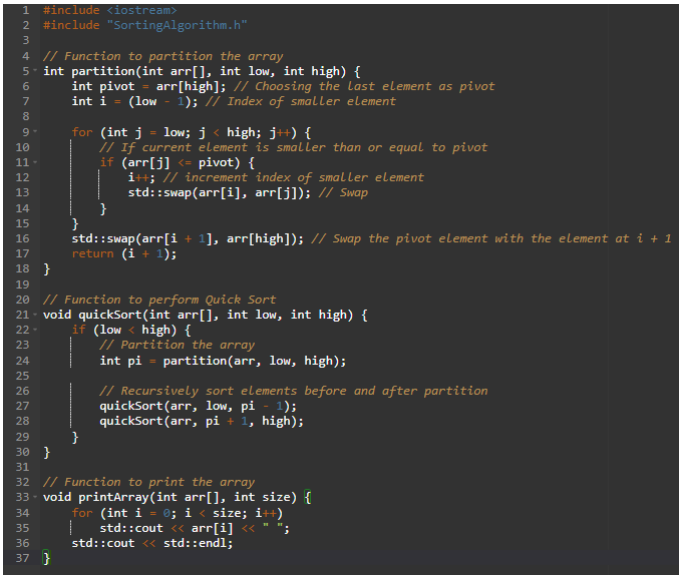
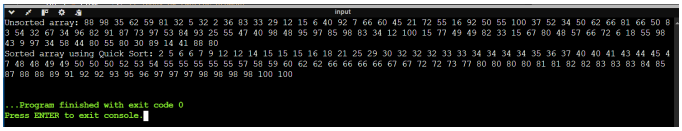
```cpp
#include <iostream>
#include "SortingAlgorithm.h"

// Function to merge two subarrays
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1; // Size of left subarray
    int n2 = right - mid;    // Size of right subarray

    // Create temporary arrays
    int* L = new int[n1];
    int* R = new int[n2];

    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temporary arrays back into arr[left..right]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements of L[] if any
    while (i < n1) {
        arr[k] = L[i];
```

```
    }

    // Copy remaining elements of R[] if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    // Free allocated memory
    delete[] L;
    delete[] R;
}

// Function to perform Merge Sort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2; // Find the middle point

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        std::cout << arr[i] << " ";
    std::cout << std::endl;
}
```

```
input
16 96 99 75 70 66 6 13 38 81 11 20 13 57 48 26 13 26 60 48 99 32 50 70 77 23 3
67 10 20 12 77 15 63 51 85 80 57 97 17 37 59 89 1 67 88 78 80 13 38 79 11 69 2
9 81 97 3 83 63 64 2 27 40 17 89 42 53 20 98 1 89 87 59 77 39 26 64 17
Sorted array using Merge Sort: 1 1 2 3 3 3 5 6 10 11 11 12 13 13 13 13 15 16 17
17 17 20 20 20 20 20 23 26 26 26 27 27 28 29 31 32 32 35 36 37 38 38 39 39 40
42 46 46 46 47 48 48 50 51 53 57 57 59 59 60 62 63 63 64 64 66 66 67 67 69 70 7
0 73 73 75 76 77 77 77 78 79 80 80 81 81 81 83 84 85 87 88 89 89 89 96 97 97 98
99 99
```

Observation	The header file declares the functions for Merge Sort and printing the array. While the implementation file contains the definitions for the Merge Sort algorithm and the function to print the array. Lastly the main file that generates the random array, sorts it using Merge Sort, and prints the results. SortingAlgorithm.h contains the declarations for the Merge Sort and printing functions. SortingAlgorithm.cpp implements the Merge Sort algorithm and the function to print the array. And main.cpp handles the generation of the random array, calls the sorting function, and prints both the unsorted and sorted arrays. Merge Sort is a stable and efficient sorting algorithm that consistently performs well with a time complexity of O(n log n) across all cases. It is particularly useful for sorting linked lists and large datasets where stability is a concern.

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot	   
Observation	SortingAlgorithm.h contains the declarations for the Quick Sort and printing functions. While SortingAlgorithm.cpp implements the Quick Sort

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "SortingAlgorithm.h"

// Function to generate an array of random integers
void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = (rand() % 100) + 1; // Random values between 1 and 100
    }
}

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

    // Seed for random number generation
    srand(static_cast<unsigned int>(time(0)));

    // Generate random array
    generateRandomArray(arr, SIZE);

    // Print the original unsorted array
    std::cout << "Unsorted array: ";
    printArray(arr, SIZE);

    // Sort using Shell Sort
    shellSort(arr, SIZE);
    std::cout << "Sorted array using Shell Sort: ";
    printArray(arr, SIZE);

    return 0;
}
```

main.cpp | SortingAlgorithm.h | SortingAlgorithm.cpp

```cpp
#ifndef SORTING_ALGORITHM_H
#define SORTING_ALGORITHM_H

// Function to perform Quick Sort
void quickSort(int arr[], int low, int high);

// Function to print the array
void printArray(int arr[], int size);

#endif // SORTING_ALGORITHM_H
```

```cpp
#include <iostream>
#include "SortingAlgorithm.h"

// Function to partition the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j < high; j++) {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot) {
            i++; // increment index of smaller element
            std::swap(arr[i], arr[j]); // Swap
        }
    }
    std::swap(arr[i + 1], arr[high]); // Swap the pivot element with the element at i + 1
    return (i + 1);
}

// Function to perform Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        std::cout << arr[i] << " ";
    std::cout << std::endl;
}
```

```
Unsorted array: 88 98 35 62 59 81 32 5 32 2 36 83 33 29 12 15 6 40 92 7 66 60 45 21 72 55 16 92 50 55 100 37 52 34 50 62 66 81 66 50 8 -
9 54 32 67 34 96 82 91 87 73 97 53 84 93 25 55 47 40 98 48 95 97 85 98 83 34 12 100 15 77 49 49 82 33 15 67 80 48 57 66 72 6 18 55 98
43 9 97 34 58 44 80 55 80 30 89 14 41 88 80
Sorted array using Quick Sort: 2 5 6 6 7 9 12 12 14 15 15 15 16 18 21 25 29 30 32 32 32 33 33 34 34 34 34 35 36 37 40 40 41 43 44 45 4
7 48 48 49 49 50 50 50 52 53 54 55 55 55 55 55 57 58 59 60 62 62 66 66 66 66 67 67 72 72 73 77 80 80 80 80 81 81 82 82 83 83 84 85
87 88 88 89 91 92 92 93 95 96 97 97 97 98 98 98 98 100 100


...Program finished with exit code 0
Press ENTER to exit console.
```

	algorithm and the function to print the array. Lastly, the main.cpp Handles the generation of the random array, calls the sorting function, and prints both the unsorted and sorted arrays. Quick Sort is a fast and efficient sorting algorithm that works by dividing the array into smaller parts, sorting those parts, and then combining them back together. It is widely used due to its average-case efficiency and simplicity.

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.

- Yes, we can sort the left and right sublists from the partition method in Quick Sort using other sorting algorithms. This can be useful if we want to optimize performance for smaller sublists or if we want to use a more stable sorting algorithm.

For example in this source code:

```
#include <iostream>
#include <vector>

using namespace std;

// Function to perform Insertion Sort
void insertionSort(vector<int>& arr, int low, int high) {
    for (int i = low + 1; i <= high; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= low && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

// Function to partition the array
int partition(vector<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;
}

// Function to perform Quick Sort with Insertion Sort for sublists
void quickSortWithInsertion(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        // Sort left sublist using Insertion Sort
        insertionSort(arr, low, pi - 1);
        // Sort right sublist using Insertion Sort
        insertionSort(arr, pi + 1, high);
    }
}

// Main function to demonstrate the sorting
int main() {
    vector<int> arr = {8, 7, 6, 1, 0, 9, 2};
    quickSortWithInsertion(arr, 0, arr.size() - 1);

    // Output the sorted array
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl; // Output: 0 1 2 6 7 8 9
    return 0;
}
```

Explanation of the code:

Insertion Sort Function:

- It sorts a portion of the array from index low to high using the insertion sort algorithm.

Partition Function:

- It selects the last element as the pivot and rearranges the elements in the array such that elements less than the pivot are on the left and those greater are on the right.

Quick Sort with Insertion Sort:

- This function recursively partitions the array and sorts the resulting left and right sublists using insertion sort.

Main Function:

- Initializes an array and calls the quickSortWithInsertion function to sort it.
- Finally, it prints the sorted array.

In Conclusion, Quick Sort is used for partitioning, while Insertion Sort is applied to the smaller sublists for potentially better performance.

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?

- For the array {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19, 74}, This is a general cases with a moderate-size array, so i think both Quick Sort and Merge Sort are excellent choices. Quick Sort has an average case time complexity and uses less memory. However, Merge Sort is always (O(N \log N)), they are also stable and maintain the relative order of equal elements, but merge sort requires additional memory for merging.

- Merge Sort and Quick Sort have (O(N \log N)) Time complexity through the following mechanisms. In Quick Sort, the array is divided into two subarrays based on a pivot element and each subarray is then sorted independently. Also, Each level of recursion processes (N) elements, and the depth of the recursion tree is Always (O(\log N)) on average. While in Merge Sort, the array is split into two halves recursively until each subarray contains a single element. It also merge two sorted halves requires (O(N)) time, also the same with Quick Sort

8. Conclusion

In conclusion, testing 100 random integers array with Shell Sort, Merge Sort, and Quick Sort algorithms revealed that all three algorithms efficiently sorted the array. However, the performance varied, with Quick Sort showing the fastest average-case time complexity of O(N log N), closely followed by Merge Sort. Shell Sort, although simpler to implement, demonstrated a time complexity of O(N log N) in the best case but was outperformed by the other two algorithms. Merge Sort proved to be the most stable, maintaining the relative order of equal elements, while Quick Sort's performance was highly dependent on the pivot selection. Overall, the choice of algorithm depends on the specific requirements, such as stability, memory constraints, and average-case performance. Ultimately, Quick Sort and Merge Sort are suitable choices for large datasets, while Shell Sort is better suited for smaller arrays or educational purposes.

9. Assessment Rubric