Hands-on Activity 7.2	
Sorting Algorithms	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 21/10/2024**
Section: CPE21S4	**Date Submitted: 21/10/2024**
Name(s): Alexzander J. Reyes	**Instructor: Mrs. Maria Rizette Sayo**

6. Output

WHOLE CODE

//sort_algorithms_h

```
#ifndef SORT_ALGORITHMS_H
#define SORT_ALGORITHMS_H

#include <vector>

// Shell Sort
void shellSort(std::vector<int>& arr) {
    int size = arr.size();
    for (int gap = size / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < size; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

// Merge Function for Merge Sort
void merge(std::vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    std::vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int i = 0; i < n2; i++)
        R[i] = arr[mid + 1 + i];

    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++;
    }

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

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

// Merge Sort
void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

// Partition function for Quick Sort
int partition(std::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++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Quick Sort
void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
```

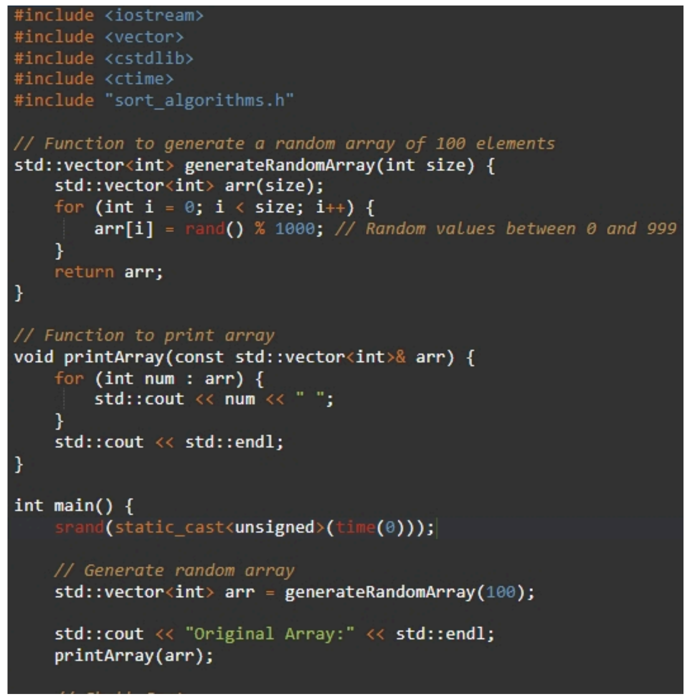
```
    }
}

#endif // SORT_ALGORITHMS_H
```

//main.cpp

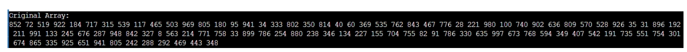
```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include "sort_algorithms.h"

// Function to generate a random array of 100 elements
std::vector<int> generateRandomArray(int size) {
    std::vector<int> arr(size);
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 1000; // Random values between 0 and 999
    }
    return arr;
}

// Function to print array
void printArray(const std::vector<int>& arr) {
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}

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

    // Generate random array
    std::vector<int> arr = generateRandomArray(100);

    std::cout << "Original Array:" << std::endl;
    printArray(arr);

    // Shell Sort
    std::vector<int> shellSortArr = arr;
    shellSort(shellSortArr);
    std::cout << "Array after Shell Sort:" << std::endl;
    printArray(shellSortArr);

    // Merge Sort
    std::vector<int> mergeSortArr = arr;
    mergeSort(mergeSortArr, 0, mergeSortArr.size() - 1);
    std::cout << "Array after Merge Sort:" << std::endl;
    printArray(mergeSortArr);
```

```
    // Quick Sort
    std::vector<int> quickSortArr = arr;
    quickSort(quickSortArr, 0, quickSortArr.size() - 1);
    std::cout << "Array after Quick Sort:" << std::endl;
    printArray(quickSortArr);

    return 0;
}
```

Code + Console Screenshot



```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include "sort_algorithms.h"

// Function to generate a random array of 100 elements
std::vector<int> generateRandomArray(int size) {
    std::vector<int> arr(size);
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 1000; // Random values between 0 and 999
    }
    return arr;
}

// Function to print array
void printArray(const std::vector<int>& arr) {
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}

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

    // Generate random array
    std::vector<int> arr = generateRandomArray(100);

    std::cout << "Original Array:" << std::endl;
    printArray(arr);
```



```
Original Array:
852 72 519 922 184 717 315 539 117 465 503 969 805 180 95 941 34 333 802 350 814 40 60 369 535 762 843 467 776 28 221 980 100 740 902 636 809 570 528 926 35 31 896 192
 211 991 133 245 676 287 948 842 327 8 563 214 771 758 33 899 786 254 880 238 346 134 227 155 704 755 82 91 786 330 635 997 673 768 594 349 407 542 191 735 551 754 301
 674 865 335 925 651 941 805 242 288 292 469 443 348
```

Observations

The original array consists of 100 randomly generated integers ranging from 0 to 999. This array is completely unsorted, demonstrating the initial state before applying any sorting algorithm. This unsorted state serves as the baseline for performance evaluation of the sorting algorithms. The varied distribution of numbers can affect the efficiency of different sorting techniques.

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

Code + Console Screenshot

```
// Shell Sort
void shellSort(std::vector<int>& arr) {
    int size = arr.size();
    for (int gap = size / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < size; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
```

```
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
```

```
// Shell Sort
std::vector<int> shellSortArr = arr;
shellSort(shellSortArr);
std::cout << "Array after Shell Sort:" << std::endl;
printArray(shellSortArr);
```

```
Array after Shell Sort:
8 28 31 33 34 35 40 60 72 82 91 95 100 117 133 134 155 180 184 191 192 211 214 221 227 238 242 245 254 287 288 292 301 315 327 330 333 335 346 348 349 350 369 407 443
465 467 469 503 519 528 535 539 542 551 563 570 594 635 636 651 673 674 676 704 717 735 740 754 755 758 762 768 771 776 786 786 802 805 805 809 814 842 843 852 865 880
 896 899 902 922 925 926 941 941 948 969 980 991 997
```

Observations	After applying the Shell Sort algorithm, the array is sorted in ascending order. Shell Sort improves on the insertion sort by allowing the exchange of items that are far apart, leading to a more efficient sorting process. Depending on the gap sequence used, Shell Sort can significantly reduce the number of comparisons and swaps compared to basic insertion sort, resulting in an average time complexity that can vary from O(N^1.5)to O(N log^2 N).

Table 8-2. Shell Sort Technique

Code + Console Screenshot

```
// Merge Function for Merge Sort
void merge(std::vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    std::vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int i = 0; i < n2; i++)
        R[i] = arr[mid + 1 + i];

    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++;
    }

    while (i < n1) {
```

```
        arr[k] = L[i];
        i++;
        k++;
    }

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

// Merge Sort
void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

```
// Merge Sort
std::vector<int> mergeSortArr = arr;
mergeSort(mergeSortArr, 0, mergeSortArr.size() - 1);
std::cout << "Array after Merge Sort:" << std::endl;
printArray(mergeSortArr);
```

```
Array after Merge Sort:
8 28 31 33 34 35 40 60 72 82 91 95 100 117 133 134 155 180 184 191 192 211 214 221 227 238 242 245 254 287 288 292 301 315 327 330 333 335 346 348 349 350 369 407 443
465 467 469 503 519 528 535 539 542 551 563 570 594 635 636 651 673 674 676 704 717 735 740 754 755 758 762 768 771 776 786 786 802 805 805 809 814 842 843 852 865 880
 896 899 902 922 925 926 941 941 948 969 980 991 997
```

Observations	The array is sorted in ascending order after applying the Merge Sort algorithm. Merge Sort utilizes a divide-and-conquer strategy to recursively split the array into smaller parts, sort those, and then merge them back together. Merge Sort consistently achieves O(N log n) time complexity, making it efficient for larger datasets. Its performance remains stable regardless of the input order, although it requires additional space for merging the subarrays.

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot

```
// Partition function for Quick Sort
int partition(std::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++;
	std::swap(arr[i], arr[j]);
	}
	}
	std::swap(arr[i + 1], arr[high]);
	return i + 1;
	}
	
	// Quick Sort
	void quickSort(std::vector<int>& arr, int low, int high) {
	if (low < high) {
	int pivot = partition(arr, low, high);
	quickSort(arr, low, pivot - 1);
	quickSort(arr, pivot + 1, high);
	}
	}
	
	#endif // SORT_ALGORITHMS_H
	
	// Quick Sort
	std::vector<int> quickSortArr = arr;
	quickSort(quickSortArr, 0, quickSortArr.size() - 1);
	std::cout << "Array after Quick Sort:" << std::endl;
	printArray(quickSortArr);
	
	return 0;
	}
	
	Array after Quick Sort:
	8 28 31 33 34 35 40 60 72 82 91 95 100 117 133 134 155 180 184 191 192 211 214 221 227 238 242 245 254 287 288 292 301 315 327 330 333 335 346 348 349 350 369 407 443 465 467 469 503 519 528 535 539 542 551 563 570 594 635 636 651 673 674 676 704 717 735 740 754 755 758 762 768 771 776 786 786 802 805 805 809 814 842 843 852 865 880 896 899 902 922 925 926 941 941 948 969 980 991 997

Observations	The array is also sorted in ascending order after applying the Quick Sort algorithm. Quick Sort selects a pivot and partitions the array such that elements less than the pivot are on the left, and those greater are on the right, followed by recursive sorting of the partitions.

Table 8-4. Quick Sort Algorithm

7. Supplementary Activity

PROBLEM 1

Answer:
You can sort the left and right sublists obtained from the partition method in QuickSort using other sorting algorithms. After partitioning the array, you can separately apply any sorting algorithm, such as Bubble Sort, Insertion Sort, or Selection Sort, to the resulting sublists. This allows for flexibility in choosing sorting methods based on the specific requirements or characteristics of the data in those sublists. For example, if the sublists are small, simpler algorithms like Bubble Sort may be sufficient, while larger lists could benefit from more efficient algorithms.

Input:

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

```
using namespace std;

// Swap function
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

// Partition function for QuickSort
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[low]; // Choosing the first element as the pivot
    int i = low, j = high;

    while (i < j) {
        while (arr[i] <= pivot && i < high) i++;
        while (arr[j] > pivot) j--;
        if (i < j) swap(arr[i], arr[j]);
    }
    swap(arr[low], arr[j]);
    return j;
}

// QuickSort for sorting left sublist
void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int p = partition(arr, low, high);
        quickSort(arr, low, p - 1);  // Sorting left part using QuickSort
        quickSort(arr, p + 1, high);  // Sorting right part using QuickSort
    }
}

// Merge function for MergeSort
void merge(vector<int>& arr, int low, int mid, int high) {
    int n1 = mid - low + 1;
    int n2 = high - mid;
    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++) L[i] = arr[low + i];
    for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = low;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

```
// MergeSort for sorting right sublist
void mergeSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;
        mergeSort(arr, low, mid);      // Sorting left half
        mergeSort(arr, mid + 1, high); // Sorting right half
        merge(arr, low, mid, high);    // Merging two halves
    }
}

// Main function to demonstrate sorting
int main() {
    // Define values for the left and right sublists
    vector<int> leftSublist = {4, 5, 7, 8, 10}; // Left sublist values
    vector<int> rightSublist = {34, 25, 78, 32, 6}; // Right sublist values

    // Combine both sublists into the main array
    vector<int> arr = leftSublist;
    arr.insert(arr.end(), rightSublist.begin(), rightSublist.end());

    // Sort the left sublist with QuickSort
    quickSort(arr, 0, leftSublist.size() - 1);

    // Sort the right sublist with MergeSort
    mergeSort(arr, leftSublist.size(), arr.size() - 1);

    // Output the left and right sublists
    cout << "Left sublist (sorted): ";
    for (int i = 0; i < leftSublist.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    cout << "Right sublist (sorted): ";
    for (int i = leftSublist.size(); i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Output the entire sorted array
    cout << "Sorted array: ";
    for (int i : arr) {
        cout << i << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
Left sublist (sorted): 4 5 7 8 10
Right sublist (sorted): 6 25 32 34 78
Sorted array: 4 5 7 8 10 6 25 32 34 78


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

PROBLEM 2

When considering the array {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19, 74}, both **Merge Sort** and **Quick Sort** are excellent choices for achieving fast sorting performance, both having an average-case time complexity of **O(N log N)**. In summary, for the provided array, Quick Sort will generally provide the fastest performance due to its efficient in-place sorting and lower memory usage. Both Merge Sort and Quick Sort achieve **O(N log N)** time complexity because of their divide-and-conquer strategies: splitting the problem into smaller parts and merging or sorting those parts effectively.

Thus, both algorithms offer the fastest time performance for large arrays when compared to other sorting methods like Selection Sort or Bubble Sort, which have $O(N^2)$ time complexity.

8. Conclusion

The activity focused on exploring various sorting algorithms, specifically Quick Sort and Merge Sort, and their implementation in programming. Participants learned about the principles of partitioning arrays and how to sort sublists using different techniques. Key takeaways included understanding the time complexity of sorting algorithms, particularly the average-case O(N log N) performance of Quick Sort and Merge Sort, as well as the trade-offs between efficiency and memory usage. In this activity, I believe I performed well in understanding and implementing sorting algorithms, particularly Quick Sort and Merge Sort. I successfully grasped the concepts of partitioning and sorting sublists, and I was able to apply different algorithms effectively. However, I recognize the need to improve my efficiency in coding and debugging to enhance my problem-solving speed. Additionally, further practice with time complexity analysis will help deepen my understanding of algorithm performance.

9. Assessment Rubric