Activity No. 8	
Sorting Algorithms	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 10/21/24**
Section: CPE21S4	**Date Submitted: 10/21/24**
Name(s): Dominic Joseph P. Virtucio	**Instructor: Ma'am Sayo**

6. Output

Code + Console Screenshot

C/C++

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main() {
    srand(time(0));
    int arr[100];

    for (int i = 0; i < 100; i++) {
        arr[i] = rand() % 100 + 1;
    }

    cout << "Random unsorted numbers: ";
    for (int i = 0; i < 100; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output Clear

```
/tmp/SZXGlHYeOG.o
Random unsorted numbers: 84 70 88 2 52 5 96 76 27 42 94 76 73 22 40 71 55 23
    81 41 55 15 37 38 65 25 50 1 42 62 94 77 84 33 31 35 89 26 62 15 20 56 42
    92 29 81 15 83 4 95 75 10 62 11 99 26 87 100 79 80 62 72 9 45 4 39 31 92
    64 93 58 83 100 99 27 28 80 41 11 35 87 37 44 48 48 42 26 86 41 4 66 54 27
    74 98 30 64 81 73 27
```

Observations	The code generates an array of 100 random integers between 1 and 100. The output shows the unsorted array of these random numbers.

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

Code + Console Screenshot

C/C++

```
#include <iostream>
#include <cstdlib>
```

```
#include <ctime>
using namespace std;

const int array_capacity = 100;

void sortUsingShell(int arr[], int length) {
    for (int interval = length / 2; interval > 0; interval /= 2)
{
        for (int index = interval; index < length; index++) {
            int current_value = arr[index];
            int position;
            for (position = index; position >= interval &&
arr[position - interval] > current_value; position -= interval)
{
                arr[position] = arr[position - interval];
            }
            arr[position] = current_value;
        }
    }
}

int main() {
    int random_numbers[array_capacity];
    srand(time(0));

    for (int x = 0; x < array_capacity; x++) {
        random_numbers[x] = rand() % 101;
    }

    cout << "INITIAL (UNSORTED) VALUES" << endl << endl;
    for (int k = 0; k < array_capacity; k++) {
        cout << random_numbers[k] << " ";
    }
    cout << endl;

    sortUsingShell(random_numbers, array_capacity);

    cout << "FINAL (SORTED) VALUES" << endl << endl;
    for (int k = 0; k < array_capacity; k++) {
        cout << random_numbers[k] << " ";
    }
    cout << endl;

    return 0;
}
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

const int array_capacity = 100;

void sortUsingShell(int arr[], int length) {
    for (int interval = length / 2; interval > 0; interval /= 2)
{
```

```
            for (int index = interval; index < length; index++) {
                int current_value = arr[index];
                int position;
                for (position = index; position >= interval &&
    arr[position - interval] > current_value; position -= interval)
    {
                    arr[position] = arr[position - interval];
                }
                arr[position] = current_value;
            }
        }
    }

    int main() {
        int random_numbers[array_capacity];
        srand(time(0));

        for (int x = 0; x < array_capacity; x++) {
            random_numbers[x] = rand() % 101;
        }

        cout << "INITIAL (UNSORTED) VALUES" << endl << endl;
        for (int k = 0; k < array_capacity; k++) {
            cout << random_numbers[k] << " ";
        }
        cout << endl;

        sortUsingShell(random_numbers, array_capacity);

        cout << "FINAL (SORTED) VALUES" << endl << endl;
        for (int k = 0; k < array_capacity; k++) {
            cout << random_numbers[k] << " ";
        }
        cout << endl;

        return 0;
    }
```

Output Clear

```
/tmp/uh5NIiAW6t.o
Random unsorted numbers: 97 17 74 52 22 15 95 39 16 41 58 48 23 51 75 65 26 1 61 79 46
    43 94 67 61 79 10 53 26 13 50 22 81 23 74 2 89 20 92 56 60 50 4 83 100 30 47 77 30
    60 56 27 54 1 93 14 80 2 66 5 66 15 79 46 89 4 99 29 23 42 85 83 91 40 17 43 69 15
    19 98 74 26 24 27 79 68 92 58 21 57 14 38 23 44 35 63 47 33 92 70

Sorted using Shell Sort: 1 1 2 2 4 4 5 10 13 14 14 15 15 15 16 17 17 19 20 21 22 22 23
    23 23 23 24 26 26 26 27 27 29 30 30 33 35 38 39 40 41 42 43 43 44 46 46 47 47 48
    50 50 51 52 53 54 56 56 57 58 58 60 60 61 61 63 65 66 66 67 68 69 70 74 74 74 75
    77 79 79 79 79 80 81 83 83 85 89 89 91 92 92 92 93 94 95 97 98 99 100
```

Observations	The code implements the Shell Sort algorithm, which sorts the array using a gap sequence. The algorithm works by repeatedly dividing the array into smaller sub-arrays with gaps, sorting each sub-array using insertion sort, and then decreasing the gap size until it reaches 1. The output shows the unsorted array followed by the sorted array

	using Shell Sort.

Table 8-2. Shell Sort Technique

Code + Console Screenshot	

C/C++

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

const int array_capacity = 100;

void combine(int arr[], int start, int mid_point, int end) {
    int size_left = mid_point - start + 1;
    int size_right = end - mid_point;

    int* left_part = new int[size_left];
    int* right_part = new int[size_right];

    for (int a = 0; a < size_left; a++)
        left_part[a] = arr[start + a];
    for (int b = 0; b < size_right; b++)
        right_part[b] = arr[mid_point + 1 + b];

    int x = 0, y = 0, z = start;
    while (x < size_left && y < size_right) {
        if (left_part[x] <= right_part[y]) {
            arr[z] = left_part[x];
            x++;
        } else {
            arr[z] = right_part[y];
            y++;
        }
        z++;
    }

    while (x < size_left) {
        arr[z] = left_part[x];
        x++;
        z++;
    }

    while (y < size_right) {
        arr[z] = right_part[y];
        y++;
        z++;
    }

    delete[] left_part;
    delete[] right_part;
}

void recursiveMergeSort(int arr[], int start, int end) {
    if (start < end) {
```

```
        int mid_point = start + (end - start) / 2;

        recursiveMergeSort(arr, start, mid_point);
        recursiveMergeSort(arr, mid_point + 1, end);
        combine(arr, start, mid_point, end);
    }
}

int main() {
    int random_numbers[array_capacity];
    srand(static_cast<unsigned>(time(0)));

    for (int i = 0; i < array_capacity; i++) {
        random_numbers[i] = rand() % 101;
    }

    cout << "INITIAL (UNSORTED) VALUES" << endl;
    for (int k = 0; k < array_capacity; k++) {
        cout << random_numbers[k] << " ";
    }
    cout << endl << endl;

    recursiveMergeSort(random_numbers, 0, array_capacity - 1);

    cout << "FINAL (SORTED) VALUES" << endl;
    for (int k = 0; k < array_capacity; k++) {
        cout << random_numbers[k] << " ";
    }
    cout << endl;

    return 0;
}
```

Output Clear

```
/tmp/8lFVGlQNSy.o
INITIAL (UNSORTED) VALUES
38 70 97 46 43 0 88 51 43 8 53 13 81 41 46 12 42 97 38 82 20 89 100 1 89 12 77
    34 59 96 17 97 31 80 9 74 81 97 24 90 4 77 69 85 85 15 63 26 78 1 75 65 90
    40 66 45 19 8 45 78 3 28 75 35 8 84 75 55 46 100 44 50 42 79 0 93 94 64 86
    38 31 60 69 87 66 0 31 85 8 76 63 11 71 3 12 79 53 88 33 99

FINAL (SORTED) VALUES
0 0 0 1 1 3 3 4 8 8 8 8 9 11 12 12 12 13 15 17 19 20 24 26 28 31 31 31 33 34
    35 38 38 38 40 41 42 42 43 43 44 45 45 46 46 46 50 51 53 53 55 59 60 63 63
    64 65 66 66 69 69 70 71 74 75 75 75 76 77 77 78 78 79 79 80 81 81 82 84 85
    85 85 86 87 88 88 89 89 90 90 93 94 96 97 97 97 97 99 100 100
```

Observations	The code implements the Merge Sort algorithm, which recursively divides the array into sub-arrays, sorts them, and then merges them back together. The algorithm works by repeatedly dividing the array into two halves, sorting each half recursively, and then merging the sorted halves back together. The output shows the unsorted array followed by the sorted array using Merge Sort.

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot	

C/C++

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

const int array_capacity = 100;

void quickSortAlgorithm(int arr[], int start, int end) {
    int left_index = start, right_index = end;
    int pivot_value = arr[(start + end) / 2];

    while (left_index <= right_index) {
        while (arr[left_index] < pivot_value) left_index++;
        while (arr[right_index] > pivot_value) right_index--;
        if (left_index <= right_index) {
            swap(arr[left_index], arr[right_index]);
            left_index++;
            right_index--;
        }
    }

    if (start < right_index)
        quickSortAlgorithm(arr, start, right_index);
    if (left_index < end)
        quickSortAlgorithm(arr, left_index, end);
}

int main() {
    int random_values[array_capacity];
    srand(time(0));

    for (int i = 0; i < array_capacity; i++) {
        random_values[i] = rand() % 101;
    }

    cout << "INITIAL UNSORTED VALUES" << endl;
    for (int k = 0; k < array_capacity; k++) {
        cout << random_values[k] << " ";
    }
    cout << endl;
    cout << endl;


    quickSortAlgorithm(random_values, 0, array_capacity - 1);

    cout << "SORTED VALUES USING QUICK SORT" << endl;
    for (int k = 0; k < array_capacity; k++) {
        cout << random_values[k] << " ";
    }
    cout << endl;

    return 0;
}
```

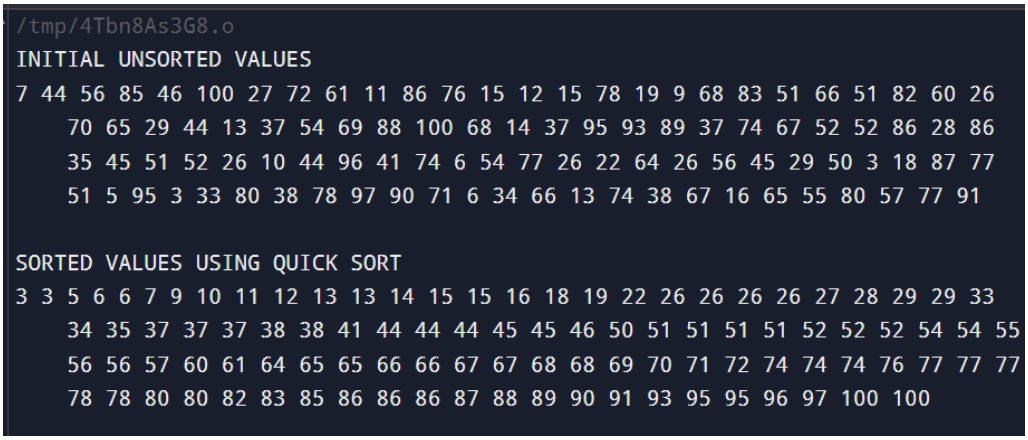
	
Observations	The code implements the Quick Sort algorithm, which uses a pivot element to partition the array and recursively sorts the sub-arrays. The algorithm works by selecting a pivot element, partitioning the array around the pivot element, and then recursively sorting the sub-arrays to the left and right of the pivot. The output shows the unsorted array followed by the sorted array using Quick Sort.

Table 8-4. Quick Sort Algorithm

7. Supplementary Activity

Problem 1:

C/C++

```
#include <iostream>
using namespace std;

int divide(int array[], int low, int high) {
    int pivot_element = array[high];
    int smaller_element_index = low - 1;

    for (int current_index = low; current_index < high; current_index++) {
        if (array[current_index] < pivot_element) {
            smaller_element_index++;
            swap(array[smaller_element_index], array[current_index]);
        }
    }
    swap(array[smaller_element_index + 1], array[high]);
    return smaller_element_index + 1;
}

void simpleInsertionSort(int array[], int low, int high) {
    for (int current = low + 1; current <= high; current++) {
        int key_value = array[current];
        int prev_index = current - 1;
        while (prev_index >= low && array[prev_index] > key_value) {
            array[prev_index + 1] = array[prev_index];
            prev_index--;
        }
        array[prev_index + 1] = key_value;
    }
}
```

```
void mergeArrays(int array[], int start, int mid_point, int end) {
    int left_size = mid_point - start + 1;
    int right_size = end - mid_point;

    int left_array[left_size], right_array[right_size];

    for (int i = 0; i < left_size; i++)
        left_array[i] = array[start + i];
    for (int j = 0; j < right_size; j++)
        right_array[j] = array[mid_point + 1 + j];

    int left_index = 0, right_index = 0, merged_index = start;

    while (left_index < left_size && right_index < right_size) {
        if (left_array[left_index] <= right_array[right_index]) {
            array[merged_index] = left_array[left_index];
            left_index++;
        } else {
            array[merged_index] = right_array[right_index];
            right_index++;
        }
        merged_index++;
    }

    while (left_index < left_size) {
        array[merged_index] = left_array[left_index];
        left_index++;
        merged_index++;
    }

    while (right_index < right_size) {
        array[merged_index] = right_array[right_index];
        right_index++;
        merged_index++;
    }
}

void customMergeSort(int array[], int start, int end) {
    if (start < end) {
        int mid_point = start + (end - start) / 2;

        customMergeSort(array, start, mid_point);
        customMergeSort(array, mid_point + 1, end);

        mergeArrays(array, start, mid_point, end);
    }
}

void hybridSort(int array[], int low, int high) {
    if (low < high) {
        int partition_index = divide(array, low, high);

        simpleInsertionSort(array, low, partition_index - 1);
        customMergeSort(array, partition_index + 1, high);
    }
}
```

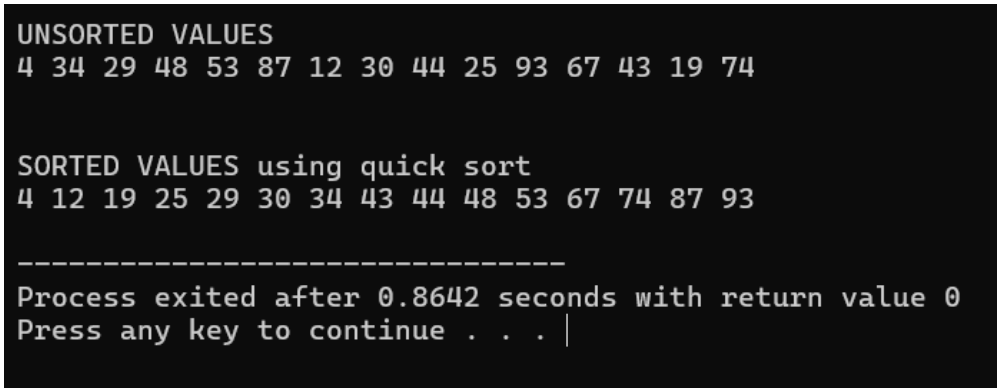
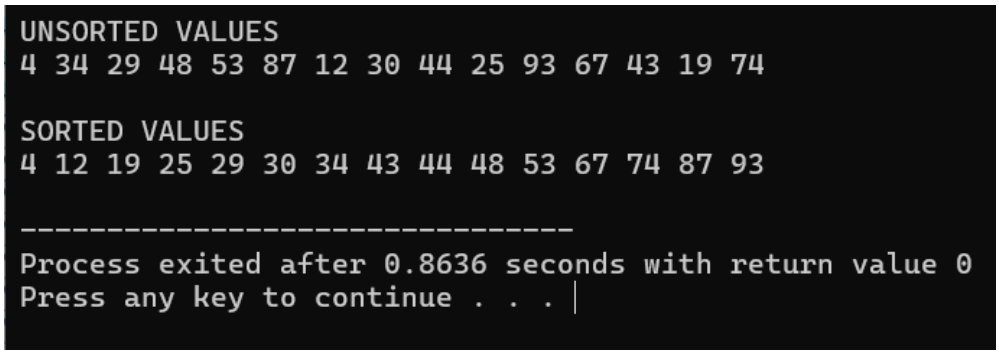
```cpp
void displayArray(int array[], int length) {
    for (int i = 0; i < length; i++)
        cout << array[i] << " ";
    cout << endl;
}

int main() {
    int values[] = {12, 4, 7, 9, 1, 15, 8, 11, 13, 14, 46, 16, 17, 19, 20, 21, 22, 23, 24,
25, 26};
    int size = sizeof(values) / sizeof(values[0]);

    cout << "Unsorted Values: ";
    displayArray(values, size);

    hybridSort(values, 0, size - 1);

    cout << "Sorted Values: ";
    displayArray(values, size);

    return 0;
}
```

Output Clear

```
/tmp/FG8ChdDjkB.o
Unsorted Values: 12 4 7 9 1 15 8 11 13 14 46 16 17 19 20 21 22 23 24 25 26
Sorted Values: 1 4 7 8 9 11 12 13 14 15 16 17 19 20 21 22 23 24 25 26 46
```

PROBLEM 2

Using shell sort	UNSORTED VALUES 4 34 29 48 53 87 12 30 44 25 93 67 43 19 74 SORTED VALUES 4 12 19 25 29 30 34 43 44 48 53 67 74 87 93 -------------------------------- Process exited after 0.9038 seconds with return value 0 Press any key to continue . . .

using merge sort	
using quick sort	

q: Which sorting algorithm provides the fastest time performance? Why do merge sort and quicksort have a time complexity of O(N•log N)?

- Based on the time consumed by various sorting algorithms, "quicksort" is the fastest, requiring only 0.01905 seconds, which is faster than shell sort and merge sort. Quicksort's average time complexity is O(N•log N), making it efficient in most circumstances. It works by selecting a "pivot" value and then splitting the array into two groups: one with elements less than the pivot and one with larger elements, repeating the process repeatedly.
- Merge sort has a temporal complexity of O(N•log N). It operates by continually dividing the array in half, sorting each half, and combining the sorted halves. Both techniques have O(N•log N) time complexity because they split the problem into smaller subarrays, resulting in logarithmic recursion.

8. Conclusion

After completing the hands-on activity, I learnt about other sorting algorithms such as shell sort, merge sort, and rapid sort. I was able to construct and test these sorting algorithms, determining which of the three was the fastest. In addition, I obtained a better knowledge of the time complexity of merge sort and quicksort, both of which have a time complexity of O(nlogn), and I now understand why.

9. Assessment Rubric

Problem 2 codes	
Shell Sort	

C/C++

```
#include <iostream>
using namespace std;

void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; 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;
        }
    }
}

int main() {
    int arr[15] = {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19,
74};

    cout << "Unsorted array:" << endl;
    for (int i = 0; i < 15; i++) {
        cout << arr[i] << " ";
    }
    cout << endl << endl;

    shellSort(arr, 15);

    cout << "Sorted array:" << endl;
    for (int i = 0; i < 15; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Merge Sort

C/C++

```
#include <iostream>
using namespace std;

void combine(int arr[], int start, int mid, int end) {
    int left_size = mid - start + 1;
    int right_size = end - mid;

    int* left_arr = new int[left_size];
    int* right_arr = new int[right_size];

    for (int i = 0; i < left_size; i++)
        left_arr[i] = arr[start + i];
    for (int j = 0; j < right_size; j++)
```

```
        right_arr[j] = arr[mid + 1 + j];

    int left_index = 0, right_index = 0, merged_index = start;

    while (left_index < left_size && right_index < right_size) {
        if (left_arr[left_index] <= right_arr[right_index]) {
            arr[merged_index] = left_arr[left_index];
            left_index++;
        } else {
            arr[merged_index] = right_arr[right_index];
            right_index++;
        }
        merged_index++;
    }

    while (left_index < left_size) {
        arr[merged_index] = left_arr[left_index];
        left_index++;
        merged_index++;
    }

    while (right_index < right_size) {
        arr[merged_index] = right_arr[right_index];
        right_index++;
        merged_index++;
    }

    delete[] left_arr;
    delete[] right_arr;
}

void mergeSort(int arr[], int start, int end) {
    if (start < end) {
        int mid = start + (end - start) / 2;

        mergeSort(arr, start, mid);
        mergeSort(arr, mid + 1, end);
        combine(arr, start, mid, end);
    }
}

int main() {
    int nums[15] = {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19,
74};

    cout << "Unsorted values:" << endl;
    for (int i = 0; i < 15; i++) {
        cout << nums[i] << " ";
    }

    cout << endl << endl;

    mergeSort(nums, 0, 14);

    cout << "Sorted values:" << endl;
    for (int i = 0; i < 15; i++) {
```

	`` cout << nums[i] << " "; } cout << endl; return 0; } ```
Quick Sort	C/C++ ``` #include <iostream> using namespace std; void quickSort(int arr[], int left, int right) { int i = left, j = right; int pivot = arr[(left + right) / 2]; while (i <= j) { while (arr[i] < pivot) i++; while (arr[j] > pivot) j--; if (i <= j) { swap(arr[i], arr[j]); i++; j--; } } if (left < j) quickSort(arr, left, j); if (i < right) quickSort(arr, i, right); } int main() { int numbers[15] = {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19, 74}; cout << "Unsorted values:" << endl; for (int index = 0; index < 15; index++) { cout << numbers[index] << " "; } cout << endl; cout << endl; quickSort(numbers, 0, 14); cout << "Sorted values using quick sort:" << endl; for (int index = 0; index < 15; index++) { cout << numbers[index] << " "; } cout << endl; ```

	`` return 0; } ```