Hands-on Activity 8.1	
Sorting Algorithms Pt2	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 09/27/25**
Section: CPE 010-CPE21S4	**Date Submitted:09/27/25**
Name(s): Kerwin Jan B. Catungal	**Instructor: Engr. Jimlord Quejado**

6. Output

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

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

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

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

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

    cout << "Original Array:\n";
    printArray(arr, 20);

    int arr1[100], arr2[100], arr3[100];
    for (int i = 0; i < 100; i++) {
        arr1[i] = arr[i];
        arr2[i] = arr[i];
        arr3[i] = arr[i];
    }

    shellSort(arr1, 100);
    cout << "\nSorted using Shell Sort:\n";
    printArray(arr1, 20);

    mergeSort(arr2, 0, 99);
    cout << "\nSorted using Merge Sort:\n";
    printArray(arr2, 20);

    quickSort(arr3, 0, 99);
    cout << "\nSorted using Quick Sort:\n";
    printArray(arr3, 20);

    return 0;
}

```

Header File:

```c
#ifndef SORTING_H
#define SORTING_H

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

// Merge Sort
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[100], R[100];
    int i, j, k;

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

    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++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

```c
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

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

#endif
```

C:\Users\Olaco\Downloads\today\main.exe

```
Original Array:
18 39 8 80 10 22 18 11 71 77 63 96 10 3 32 62 3 72 93 20

Sorted using Shell Sort:
0 0 1 2 3 3 4 4 7 8 10 10 11 11 12 13 16 17 17 18

Sorted using Merge Sort:
0 0 1 2 3 3 4 4 7 8 10 10 11 11 12 13 16 17 17 18

Sorted using Quick Sort:
0 0 1 2 3 3 4 4 7 8 10 10 11 11 12 13 16 17 17 18

--------------------------------
Process exited after 0.2747 seconds with return value 0
Press any key to continue . . .
```

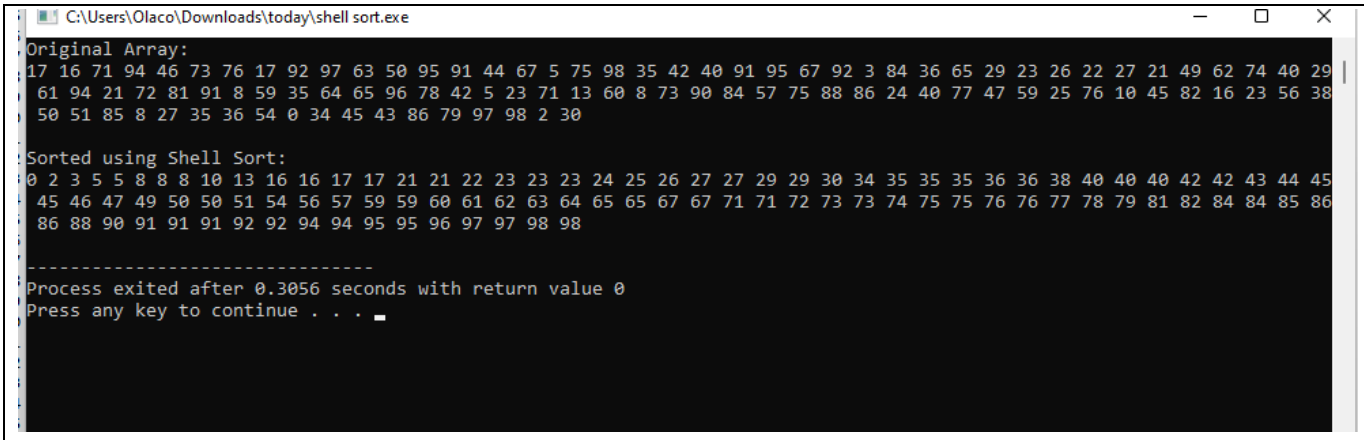
Here I made an array consisting of 100 random numbers and displayed it. and I duplicated this array three times, allowing me to sequentially apply Shell Sort, Merge Sort, and Quick Sort. Each sorting method was unique, but all successfully organized the numbers sequentially from the lowest to the highest.I verified that all three methods produced an identical output and I assembled an array of 100 random numbers which I displayed as the original output

Table 8-2. Shell Sort Technique:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "sorting.h"

using namespace std;

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


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

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


    shellSort(arr, SIZE);

    cout << "Sorted using Shell Sort:\n";
    for (int i = 0; i < SIZE; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Header File:

```
#ifndef SORTING_H
#define SORTING_H

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

#endif
```



So here I started by creating an array with 100 random numbers and showed them as the original output. I added the shellSort function in a header file and used it in my main code. The Shell Sort works by comparing numbers that are far apart first, then slowly making the gaps smaller. After that the numbers get arranged closer and closer until the array is sorted and it displayed the sorted array from smallest to biggest.

Table 8-3. Merge Sort Technique:

```
#include <iostream>
#include "sorting.h"
using namespace std;

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    int arr[100];
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

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

    mergeSort(arr, 0, n - 1);

    cout << "Sorted using Merge Sort: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Header File:

```
#ifndef SORTING_H
#define SORTING_H


void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[50], R[50];

    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];

    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++;
    }
}

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

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

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

#endif
```

```
Enter number of elements: 6
Enter 6 elements: 2
4
5
36
6
78
Original Array: 2 4 5 36 6 78
Sorted using Merge Sort: 2 4 5 6 36 78

--------------------------------
Process exited after 19.34 seconds with return value 0
Press any key to continue . . .
```

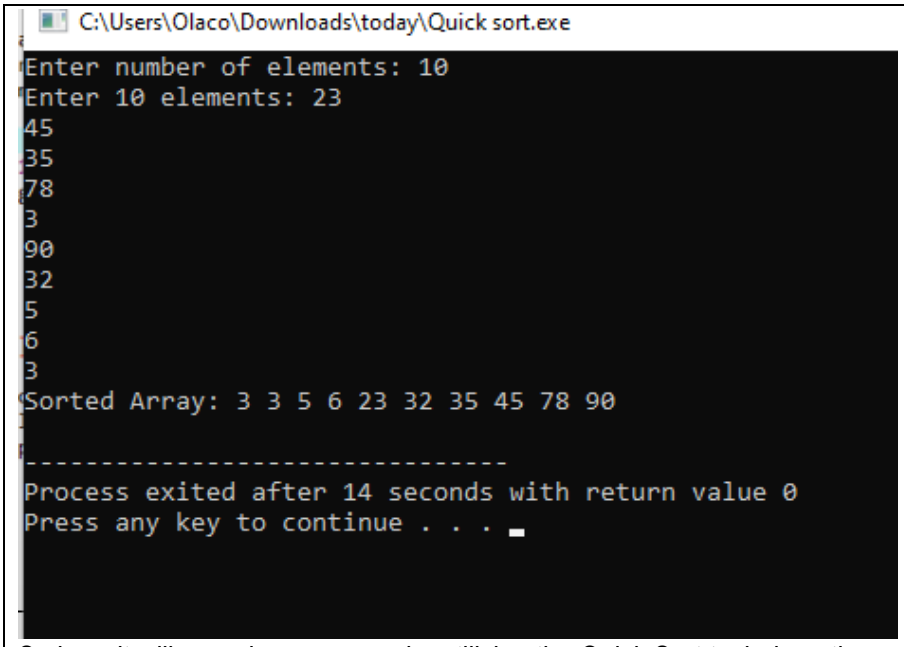
So here it will ask the user to input the number of elements and the respective elements before saving them to an array. Before proceeding to the sort operation, the program shows the user the original array to show them the order of the elements. The program sorts the array using the merge sort algorithm which al splits the array into smaller parts, sorts the smaller parts, and merges the parts back together into the original array in order. The program then shows the user the array after sorting. The merge function focuses on combining two parts and the mergeSort function is responsible for splitting the parts of the array.

Table 8-4. Quick Sort Algorithm:

```cpp
#include <iostream>
#include "quick sort.h"
using namespace std;

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    int arr[100];
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    quickSort(arr, 0, n - 1);

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

    return 0;
}
```

Header File:

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

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++;

            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}

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



So here it will organizes an array by utilizing the Quick Sort technique the program allows the user to specify the elements and the total number of elements. In the partition function the last element is connected to a pivot and the function organizes the arrays clearing the smaller array is to the left and the bigger arrays to the right. Then in the quickSort function, the same procedure is applied to the left and right sections of the array recursively until the whole array is organized. The sorted array is the last output that the program offers.

7. Supplementary Activity

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.

Quick Sort works by selecting a pivot and partitioning the array based on that pivot. A value smaller than the pivot will be on the left side, and anything larger will be on the right. If we use 12 as the pivot for the array [12, 7, 15, 20, 5, 10], everything smaller than 12 will go to the left array, which is [7, 5, 10], and everything larger than 12 will go to the right, which is [15, 20]. In this example, we can now use insertion sort to sort the left array, [7, 5, 10], and once that's sorted, we can then use bubble sort to sort the right array, [15, 20]. Then, combining the two sorted arrays will give us the complete sorted array: [5, 7, 10, 12, 15, 20]. This example illustrates that Quick Sort can also be more efficient when we combine sorting with other algorithms.

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?

When compared to bubble sort, selection sort, or insertion sort, Quick Sort and Merge Sort are much quicker at sorting an array. This is because both Quick Sort and Merge Sort breakdown the array into smaller parts, and sort each of those parts, a few at a time. At the end, the total number of operations required to sort the array is roughly equal to N log N, instead of N squared. This is why they are an effective way at quickly sorting large lists. Quick sorting with either of these methods can save you time, especially as the amount of data you need to sort increases.

8. Conclusion:

After doing this activity I understant that both Merge Sort and Quick Sort sorted the data in ascending order accurately. Each algorithm had a different approach to resolving the sorting issue, but the end objective was the same: arranging the data in order. Merge Sort was more straightforward in problem solving by dividing the issue into smaller sections. On the other hand, Quick Sort exhibited more versatility by employing pivots and recursive sorting. From a more subjective perspective, these programs showed that sorting a given array may have more than one direction to move in particularly when the array is large.

9. Assessment Rubric