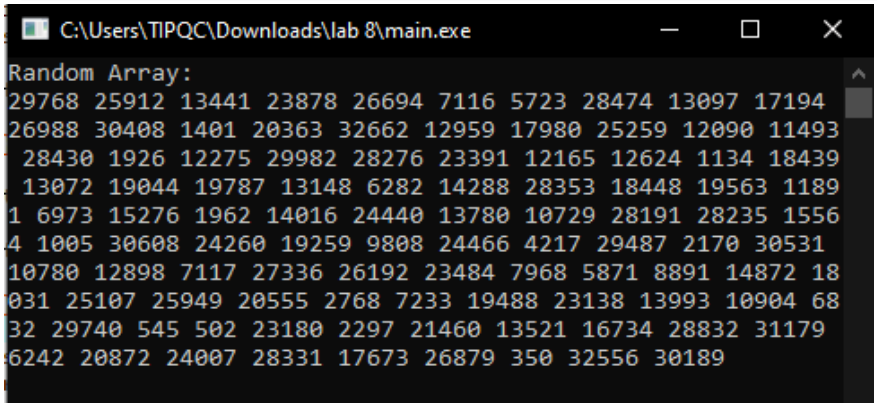
Activity No. 8	
SORTING ALGORITHMS: SHELL, MERGE, AND QUICK SORT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/21/24
Section: CPE21S4	**Date Submitted:** 10/23/24
Name(s): Magistrado, Aira Pauleen M.	**Instructor:** Maria Rizette Sayo

6. Output

Code + Console Screenshot

```
C/C++
#include <iostream>
#include <cstdlib>
#include <ctime>

const int max_size = 100;

int main() {
    int dataset[max_size];
    srand(time(0));

    for (int i = 0; i < max_size; i++) {
        dataset[i] = rand();
    }

   std::cout << "Random Array:" << std::endl;

    for (int i = 0; i < max_size; i++) {
        std::cout << dataset[i] << " ";
    }

    std::cout << std::endl;
    return 0;
}
```



Observations	The code generates a random array of 100 integers and outputs them to the console. The random array of 100 integers generated with srand(time(0)) ensures that the integers are unique each time the program is run.

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

Code + Console Screenshot	

C/C++

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

const int max_size = 100;

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;
        }
    }
}

int main() {
    int dataset[max_size];
    srand(time(0));

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

    std::cout <<
"--------------------------------------------------------------------------
------------------------------" << std::endl;
    std::cout << "Random Array:" << std::endl;
    for (int i = 0; i < max_size; i++) {
        std::cout << dataset[i] << " ";
    }
    std::cout << std::endl;

    shellSort(dataset, max_size);

    std::cout <<
"--------------------------------------------------------------------------
------------------------------" << std::endl;
    std::cout << "Sorted Array (Shell Sort):" << std::endl;
    for (int i = 0; i < max_size; i++) {
        std::cout << dataset[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

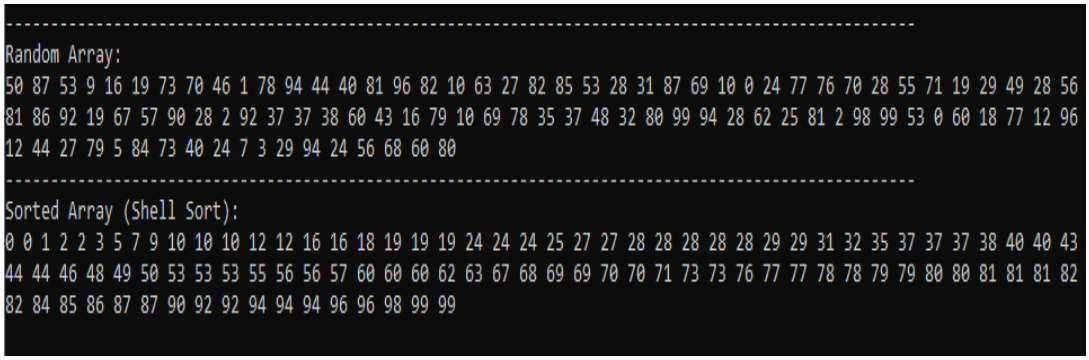
	
Observations	The code displays the original 100 integers that are generated unsorted and then sorted. It is sorted using the shell sort technique, which sorts it in ascending order. Compared to other sorting algorithms, shell sort speeds up the sorting process by reducing the number of steps required to sort an array. It achieves this by comparing elements that are far apart and gradually closing the gap between them.

Table 8-2. Shell Sort Technique

Code + Console Screenshot

C/C++

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

const int max_size = 100;

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

    int L[n1], R[n2]; //it is split to L (left) and R (right)
    //n1 is the number of elements in the first half
    //n2 is the number of elements in the second half

    for (int i = 0; i < n1; i++) L[i] = arr[left + i]; //copies each
element from the first half of arr to L
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j]; //copies each
element from the second half of arr to R

    int i = 0, j = 0, k = left;
    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++];
}
```

```cpp
void mergeSort(int arr[], int left, int right) {
    if (left < right) { ////If left is not less than right, the array is
already sorted
        int mid = left + (right - left) / 2; //Calculates the middle
point of the current subarray.
        mergeSort(arr, left, mid); //sorts 1st half
        mergeSort(arr, mid + 1, right); //sorts 2nd half
        merge(arr, left, mid, right); //combines the two
    }
}

int main() {
    int dataset[max_size];
    srand(time(0));

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

    std::cout <<
"--------------------------------------------------------------------------
------------------------------" << std::endl;
    std::cout << "Random Array:" << std::endl;
    for (int i = 0; i < max_size; i++) {
        std::cout << dataset[i] << " ";
    }
    std::cout << std::endl;

    mergeSort(dataset, 0, max_size - 1);

    std::cout <<
"--------------------------------------------------------------------------
------------------------------" << std::endl;
    std::cout << "Sorted Array (Merge Sort):" << std::endl;
    for (int i = 0; i < max_size; i++) {
        std::cout << dataset[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

```
-----------------------------------------------------------------------------------------------
Random Array:
65 27 94 84 52 31 2 20 69 52 24 57 59 63 2 35 77 48 41 72 66 39 31 61 31 51 47 29 21 95 80 58 92 46 79 81 90 83 41 92 76
 5 61 63 16 15 93 72 94 36 38 50 88 60 44 51 64 86 64 12 67 5 66 46 52 84 62 3 5 19 66 7 14 95 88 63 36 65 44 25 56 54 2
9 41 15 57 56 91 0 17 24 59 4 36 32 66 37 56 86 45
-----------------------------------------------------------------------------------------------
Sorted Array (Merge Sort):
0 2 2 3 4 5 5 5 7 12 14 15 15 16 17 19 20 21 24 24 25 27 29 29 31 31 31 32 35 36 36 36 37 38 39 41 41 41 44 44 45 46 46
47 48 50 51 51 52 52 52 54 56 56 56 57 57 58 59 59 60 61 61 62 63 63 63 64 64 65 65 66 66 66 66 67 69 72 72 76 77 79 80
81 83 84 84 86 86 88 88 90 91 92 92 93 94 94 95 95
```

Observations	The code displays the original 100 integers that are generated unsorted and then sorted. It is sorted using the merge sort technique, which sorts it in ascending order. It also shows the divide and conquer strategy, in which the array is repeatedly divided into smaller sub-arrays, which are then sorted and merged back into a single sorted array.

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot

C/C++

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

const int max_size = 100;

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    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);
    }
}

int main() {
    int dataset[max_size];
    srand(time(0));

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

    std::cout <<
"--------------------------------------------------------------------------
-----------------------------" << std::endl;
    std::cout << "Random Array:" << std::endl;
    for (int i = 0; i < max_size; i++) {
        std::cout << dataset[i] << " ";
    }
    std::cout << std::endl;

    quickSort(dataset, 0, max_size - 1);
```

```
    std::cout <<
"--------------------------------------------------------------------------------
------------------------------" << std::endl;
    std::cout << "Sorted Array (Quick Sort):" << std::endl;
    for (int i = 0; i < max_size; i++) {
        std::cout << dataset[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

```
--------------------------------------------------------------------------------------------
Random Array:
44 49 51 64 30 42 57 77 61 58 28 73 42 46 96 50 61 41 87 18 65 81 29 7 22 58 50 32 52 13 80 95 85 48 97 19 27 52 91 16 1
3 6 95 29 16 34 63 82 80 97 83 10 7 22 41 10 60 79 0 66 53 77 46 71 4 46 14 74 78 81 25 48 83 79 18 67 37 4 95 4 0 47 82
 9 23 83 98 65 85 50 75 70 48 11 63 50 98 7 5 17
--------------------------------------------------------------------------------------------
Sorted Array (Quick Sort):
0 0 4 4 4 5 6 7 7 7 9 10 10 11 13 13 14 16 16 17 18 18 19 22 22 23 25 27 28 29 29 30 32 34 37 41 41 42 42 44 46 46 46 47
 48 48 48 49 50 50 50 50 51 52 52 53 57 58 58 60 61 61 63 63 64 65 65 66 67 70 71 73 74 75 77 77 78 79 79 80 80 81 81 82
 82 83 83 83 85 85 87 91 95 95 95 96 97 97 98 98
```

Observations	The code displays the original 100 integers that are generated unsorted and then sorted. It is sorted using the quick sort technique, which sorts it in ascending order. It shows the partitioning step, in which the array is divided into two sub-arrays around the pivot element and then sorted repeatedly.

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.

C/C++

```
#include <iostream>
#include <algorithm>

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 - 1; j++) {
        if (arr[j] < pivot) {
            i++;
```

```
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void insertionSort(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;
    }
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int p = partition(arr, low, high);

        insertionSort(arr, low, p - 1);

        quickSort(arr, p + 1, high);
    }
}

int main() {
    int arr[] = {10, 45, 20, 15, 30, 5, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    quickSort(arr, 0, n - 1);

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

    return 0;
}
```

```
Original array: 10 45 20 15 30 5 3
Sorted array: 3 5 10 15 20 30 45
```

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?**

I think that merge sort gives the fastest time performance for this algorithm. It is the fastest due to its consistent time complexity of O(N log N), making it ideal for large datasets. Quick sort takes O(N log N) time on average, it is efficient in most cases but it can slow down to $O(N^2)$ like when the array is already sorted and the pivot is poorly chosen. Merge sort and quick sort both work by breaking the array into smaller pieces until each piece is just one element. This makes sorting easier. They cut the array in half each time, which makes the sorting process faster. At each step, they do simple tasks like merging or sorting, which take a fixed amount of time.

8. Conclusion

In conclusion, I learned three important sorting algorithms: Shell Sort, Merge Sort, and Quick Sort. I learned that shell sorting speeds up the sorting process by reducing the number of steps required to sort an array. It accomplishes this by comparing elements that are distant and gradually closing the gap between them. Merge Sort uses a divide-and-conquer strategy where it splits an array into smaller pieces, sorts them, and then puts them back together. Quick Sort also uses a divide-and-conquer strategy picks a middle point, arranges the array around it, and sorts each part. The supplementary activity helped me understand more about Merge Sort and Quick Sort. They were designed to be fast and efficient, even for large datasets, with a time complexity of O(N log N). I think I did well in this activity. I successfully implemented the three sorting algorithms and learned a lot about how they are used. My areas for improvement include learning more about sorting algorithms and how they are applied in real-world scenarios.

9. Assessment Rubric