Activity No. 8.1	
Sorting Algorithms PT2	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 9/27/2025**
Section: CPE21S4	**Date Submitted: 9/27/2025**
Name(s): Anastacio, Lester Arvid P.	**Instructor: Engr. Jimlord M. Quejado**

6. Output

A.1. Array

Code:

8_1_Algo.cpp sorting_algorithms.h

```cpp
#include <iostream>
#include <random>
#include <algorithm>
#include "sorting_algorithms.h"

const int SIZE = 100;

void printArray(const int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        std::cout << arr[i] << ((i + 1) % 10 == 0 ? "\n" : "\t");
    }
    std::cout << "\n";
}

int main() {
    int originalArray[SIZE];

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(0, 999);


    for (int i = 0; i < SIZE; ++i) {
        originalArray[i] = dist(gen);
    }

    std::cout << "Original Unsorted Array (Table 8-1):\n";
    printArray(originalArray, SIZE);

    int shellSorted[SIZE];
    std::copy(originalArray, originalArray + SIZE, shellSorted);
    shellSort(shellSorted, SIZE);
    std::cout << "\nShell Sorted Array:\n";
    printArray(shellSorted, SIZE);

    return 0;
}
```

Header:

8_1_Algo.cpp sorting_algorithms.h

```cpp
#ifndef SORTING_ALGORITHMS_H
#define SORTING_ALGORITHMS_H
#include <iostream>

void shellSort(int arr[], int size);
void mergeSort(int arr[], int size);
void quickSort(int arr[], int size);

#endif
```

Observation:

- The code is organized in a way that separates the sorting part of the algorithms from the core functionality via the declaration in a header file (sorting_algorithms.h), which is more modular and allows easier maintenance. Through the use of modern C++ random library, the main.cpp produces an array of 100 random integers without using C-style functions which were traditionally employed. Afterward, the program prints out the array in its unsorted state and visualizes a sorting process by taking the copy of the array and using the Shell Sort method from the header to sort it, thus keeping the original data for the subsequent sorting methods. Basically, this way of implementation is very neat and modular, thus it becomes straightforward to add more algorithms and use it as a teaching tool to compare different sorting methods on the same dataset.

A.2. Shell Sort
Code:

8_1_Algo.cpp sorting_algorithms.h

```
#ifndef SORTING_ALGORITHMS_H
#define SORTING_ALGORITHMS_H
#include <iostream>

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;
        }
    }
}
void mergeSort(int arr[], int size);
void quickSort(int arr[], int size);

#endif
```

Output:

```
D:\DATA ANA\8_1_Algo.exe

Original Unsorted Array (Table 8-1):
561     726     348     915     6       594     872     515     959     975
169     661     316     528     701     788     255     74      574     329
338     583     238     30      172     928     966     19      653     779
477     336     66      342     13      414     841     907     282     651
920     570     848     7       560     286     889     729     1       702
262     452     337     223     934     872     478     717     978     885
767     336     702     671     166     605     497     66      646     768
664     826     851     144     533     2       306     493     328     97
413     692     562     625     696     754     576     461     319     777
994     751     582     976     397     177     409     449     142     835


Shell Sorted Array:
1       2       6       7       13      19      30      66      66      74
97      142     144     166     169     172     177     223     238     255
262     282     286     306     316     319     328     329     336     336
337     338     342     348     397     409     413     414     449     452
461     477     478     493     497     515     528     533     560     561
562     570     574     576     582     583     594     605     625     646
651     653     661     664     671     692     696     701     702     702
717     726     729     751     754     767     768     777     779     788
826     835     841     848     851     872     872     885     889     907
915     920     928     934     959     966     975     976     978     994



--------------------------------
Process exited after 1.035 seconds with return value 0
Press any key to continue . . .
```

Observation:

- the function Shell Sort is that is added here is the method to initially sorts the elements, which are far from each other, by using a gap sequence that starts at half the size of the array and reduces by half each iteration. Basically, this process enables the elements to come closer to their correct positions at a much faster pace than if a standard insertion sort was used, thus, for large arrays, the performance will be significantly better. When the interval gets smaller, the algorithm makes the more detailed sorting passes until the whole array is sorted. In essence, this way the number of comparisons and shifts are less than what would have been done if a simple insertion sort had been used thus, on average, the performance for medium to large datasets is better.

A.3. Merge Sort

Code:

8_1_Algo.cpp | sorting_algorithms.h

```cpp
#ifndef SORTING_ALGORITHMS_H
#define SORTING_ALGORITHMS_H
#include <iostream>

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

    int* leftArr = new int[n1];
    int* rightArr = new int[n2];

    for (int i = 0; i < n1; ++i)
        leftArr[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        rightArr[j] = arr[middle + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k++] = leftArr[i++];
        } else {
            arr[k++] = rightArr[j++];
        }
    }

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

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

    delete[] leftArr;
    delete[] rightArr;
}

void merge_sort(int arr[], int left, int right) {
    if (left >= right) return;

    int middle = (left + right) / 2;

    merge_sort(arr, left, middle);
    merge_sort(arr, middle + 1, right);
    merge(arr, left, middle, right);
}

void mergeSort(int arr[], int size) {
    merge_sort(arr, 0, size - 1);
}



#endif
```

Output:

```
D:\DATA ANA\8_1_Algo.exe

Original Unsorted Array (Table 8-1):
561     726     348     915     6       594     872     515     959     975
169     661     316     528     701     788     255     74      574     329
338     583     238     30      172     928     966     19      653     779
477     336     66      342     13      414     841     907     282     651
920     570     848     7       560     286     889     729     1       702
262     452     337     223     934     872     478     717     978     885
767     336     702     671     166     605     497     66      646     768
664     826     851     144     533     2       306     493     328     97
413     692     562     625     696     754     576     461     319     777
994     751     582     976     397     177     409     449     142     835


Merge Sorted Array:
1       2       6       7       13      19      30      66      66      74
97      142     144     166     169     172     177     223     238     255
262     282     286     306     316     319     328     329     336     336
337     338     342     348     397     409     413     414     449     452
461     477     478     493     497     515     528     533     560     561
562     570     574     576     582     583     594     605     625     646
651     653     661     664     671     692     696     701     702     702
717     726     729     751     754     767     768     777     779     788
826     835     841     848     851     872     872     885     889     907
915     920     928     934     959     966     975     976     978     994


--------------------------------
Process exited after 1.05 seconds with return value 0
Press any key to continue . . .
```

Observation:

- The merge sort function that is added here recursively divides the input array into smaller subarrays by calculating middle indices until each subarray contains a single element and the It merges these subarrays by creating temporary arrays to hold the left and right halves, and copies elements back into the original array in sorted order. Memory for temporary arrays is dynamically allocated and properly freed after merging and the recursion ensures that sorting happens on increasingly larger portions of the array, with the merge step combining these sorted segments efficiently. Overall, the code cleanly separates the divide or the recursive calls and conquer or merge phases, handling array boundaries and indexing precisely.

A.4. Quick Sort

Code:

8_1_Algo.cpp   sorting_algorithms.h

```cpp
#ifndef SORTING_ALGORITHMS_H
#define SORTING_ALGORITHMS_H
#include <iostream>

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

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

void quickSort(int arr[], int size) {
    quick_sort(arr, 0, size - 1);
}


#endif
```

Output:

```
D:\DATA ANA\8_1_Algo.exe

Original Unsorted Array (Table 8-1):
561     726     348     915     6       594     872     515     959     975
169     661     316     528     701     788     255     74      574     329
338     583     238     30      172     928     966     19      653     779
477     336     66      342     13      414     841     907     282     651
920     570     848     7       560     286     889     729     1       702
262     452     337     223     934     872     478     717     978     885
767     336     702     671     166     605     497     66      646     768
664     826     851     144     533     2       306     493     328     97
413     692     562     625     696     754     576     461     319     777
994     751     582     976     397     177     409     449     142     835


Quick Sorted Array:
1       2       6       7       13      19      30      66      66      74
97      142     144     166     169     172     177     223     238     255
262     282     286     306     316     319     328     329     336     336
337     338     342     348     397     409     413     414     449     452
461     477     478     493     497     515     528     533     560     561
562     570     574     576     582     583     594     605     625     646
651     653     661     664     671     692     696     701     702     702
717     726     729     751     754     767     768     777     779     788
826     835     841     848     851     872     872     885     889     907
915     920     928     934     959     966     975     976     978     994


--------------------------------
Process exited after 1.05 seconds with return value 0
Press any key to continue . . .
```

Observation:

- The functions here are the public interface that begins the sorting process recursively by calling it quick_sort, which sorts an array by partitioning it around the pivot value chosen as the last element, where partition rearranges the elements so that those less than or equal to the pivot are on the left side of the pivot and those greater are on the right, after quick_sort executes partition on the subarrays on the left and right of the pivot, continuing this process recursively until the entire array is sorted.

7. Supplementary Activity

Problem 1: Can we sort the left sublist and right sublist from the partition method in quick sort using other sorting algorithms? Demonstrate an example.

- Yes, it is possible to handle the unsorted parts on the left and right of the list after the partition phase of quicksort by applying different algorithms. After dividing the array into two sub-lists which is the left and right, these sub-lists can be independently sorted by any algorithms as merge sort, insertion sort, bubble sort and so on. The partitioning in quicksort is just a process of separating the array, and the sublists can be sorted with any other algorithm after the separation.

  Example:

Mergesorting_partition.cpp

```cpp
#include <iostream>

// Merge function for merge sort
void merge(int arr[], int left, int mid, int right, int temp[]) {
    int i = left, j = mid + 1, k = left;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    for (int x = left; x <= right; x++)
        arr[x] = temp[x];
}

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

// Partition function (quicksort step)
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++;
            std::swap(arr[i], arr[j]);
        }
```

```cpp
        }
    }

    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

int main() {
    int arr[] = {5, 3, 8, 4, 2, 7, 1, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    // Step 1: Partition the array
    int pivotIndex = partition(arr, 0, n - 1);
    int pivot = arr[pivotIndex];

    std::cout << "After partitioning (pivot = " << pivot << "): ";
    for (int i = 0; i < n; i++) std::cout << arr[i] << " ";
    std::cout << std::endl;

    // Step 2: Sort left and right parts using merge sort
    int temp[100]; // Temporary array for merge sort

    // Sort left subarray
    mergeSort(arr, 0, pivotIndex - 1, temp);

    // Sort right subarray
    mergeSort(arr, pivotIndex + 1, n - 1, temp);

    // Final output
    std::cout << "Final sorted array: ";
    for (int i = 0; i < n; i++) std::cout << arr[i] << " ";
    std::cout << std::endl;

    return 0;
}
```

Output:

```
D:\DATA ANA\Mergesorting_

Original array: 5 3 8 4 2 7 1 6
After partitioning (pivot = 6): 5 3 4 2 1 6 8 7
Final sorted array: 1 2 3 4 5 6 7 8

--------------------------------
Process exited after 1.028 seconds with return value 0
Press any key to continue . . .
```

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?

- Both quick sort and merge sort are good when it comes to time in the basis of the array given and each with an average time complexity of O(N log N) in which is a plus for data of such a size and usually quick sort is seen to be more efficient in real-world scenarios as it is an in-place sort, i.e. it does not take any additional memory, but its worst-case time complexity however can become $O(N^2)$ if the pivot is wrongly selected while merge sort leads the way in terms of time complexity with O(N log N) for both best and worst cases, but a considerable amount of memory is still required for the temporary arrays allocated during the merging phase and lastly the main reason for both algorithms to have the complexity of O(N log N) is that they both break the array into even smaller subarrays recursively and quick sorting selects a pivot and splits around it, while merge sorting breaks it down even and then combines it. Therefore, even though quick sort could be accelerated because it occupies less memory, merge sort is more reliable and consistent, thus, the choice of a better one when memory is not an issue or when a stable sorting is needed.

8. Conclusion

- This activity has exposed us to various sorting methods besides those more commonly known. We have been able to look into and understand C++ programs that implement quick sorting, shell sorting, and merge sorting algorithms. By these programs, we learned the operations of each algorithm and recognized their different ways of sorting data. We also segregated the sorting algorithms types by visualizing their methods—divide and conquer he used for Quick Sort and Merge Sort, and gap-based insertion for Shell Sort. Last but not least, through code writing and code execution, we fulfilled the goal of producing the working implementations of sorting algorithms.

9. Assessment Rubric