Activity No. 8.1	
Sorting Algorithm Pt2	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: Sept. 26, 2025**
Section: CPE21S4	**Date Submitted: Sept. 27, 2025**
Name(s): Crishen Luper S. Pulgado	**Instructor:** Sir Jimlord Quejado

6. Output

Code + Console Screenshot

```
Z:\Crishen\cpp\mainSort.exe
Original Array:
42 68 35 1 70 25 79 59 63 65 6 46 82 28 62 92 96 43 28 37 92 5 3 54 93 83 22 17 19 96 48 27 72 39 70 13 68 100 36 95 4 1
2 23 34 74 65 42 12 54 69 48 45 63 58 38 60 24 42 30 79 17 36 91 43 89 7 41 43 65 49 47 6 91 30 71 51 7 2 94 49 30 24 85
 55 57 41 67 77 32 9 45 40 27 24 38 39 19 83 30 42

--------------------------------
Process exited after 0.0835 seconds with return value 0
Press any key to continue . . .
```

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "sort2.h"

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

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

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



    return 0;
}
```

Observations

Similar to the previous activities, I’ve added an array that has a size of 100 that generates a random numbers from 1 to 100.

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

Code + Console Screenshot

```
Z:\Crishen\cpp\mainSort.exe
42 7 2 1 38 25 24 42 30 57 6 36 77 28 9 4 12 23 24 37 39 5 3 30 42 48 22 17 19 49 30 24 59 39 65 13 46 82 32 62 7 40 27
28 38 47 6 12 30 69 51 45 35 58 70 48 27 72 55 70 17 67 91 36 89 45 41 43 34 49 65 19 83 54 71 83 68 63 94 96 60 79 85 6
3 79 41 68 100 43 95 92 96 43 65 74 92 42 91 54 93
After gap = 12:
17 7 2 1 7 23 24 24 27 5 3 12 30 28 9 4 12 25 24 28 30 39 6 13 42 41 22 17 19 40 27 37 34 47 6 19 42 48 32 36 35 45 30 4
2 38 49 55 30 42 54 51 45 38 49 41 43 39 57 65 36 46 67 54 62 43 58 70 48 43 65 65 63 77 69 68 83 68 63 92 96 59 72 74 7
0 79 82 71 93 89 95 94 96 60 79 85 92 83 91 91 100
After gap = 6:
17 7 2 1 3 12 24 24 9 4 6 13 24 28 22 5 6 19 27 28 27 17 7 23 30 37 30 36 12 25 30 41 32 39 19 30 41 42 34 45 35 36 42 4
3 38 47 38 40 42 48 39 49 43 45 42 48 43 57 55 49 46 54 51 62 65 58 70 67 54 65 65 63 77 69 59 72 68 63 79 82 60 79 74 7
0 83 91 68 83 85 92 92 96 71 93 89 95 94 96 91 100
After gap = 3:
1 3 2 4 6 9 5 6 12 17 7 13 17 7 19 24 12 22 24 19 23 27 24 25 30 28 27 30 28 30 36 35 30 39 37 32 41 38 34 42 41 36 42 4
2 38 42 43 39 45 43 40 46 48 43 47 48 45 49 54 49 57 55 51 62 65 54 65 65 58 70 67 59 72 68 60 77 69 63 79 74 63 79 82 6
8 83 85 70 83 89 71 92 91 91 93 96 92 94 96 95 100
After gap = 1:
1 2 3 4 5 6 6 7 7 9 12 12 13 17 17 19 19 22 23 24 24 24 25 27 27 28 28 30 30 30 30 32 34 35 36 36 37 38 38 39 39 40 41 4
1 42 42 42 42 43 43 43 45 45 46 47 48 48 49 49 51 54 54 55 57 58 59 60 62 63 63 65 65 65 67 68 68 69 70 70 71 72 74 77 7
9 79 82 83 83 85 89 91 91 92 92 93 94 95 96 96 100

Final Sorted Array (Shell Sort):
1 2 3 4 5 6 6 7 7 9 12 12 13 17 17 19 19 22 23 24 24 24 25 27 27 28 28 30 30 30 30 32 34 35 36 36 37 38 38 39 39 40 41 4
1 42 42 42 42 43 43 43 45 45 46 47 48 48 49 49 51 54 54 55 57 58 59 60 62 63 63 65 65 65 67 68 68 69 70 70 71 72 74 77 7
9 79 82 83 83 85 89 91 91 92 92 93 94 95 96 96 100

Total swaps performed: 457

--------------------------------
Process exited after 0.1528 seconds with return value 0
Press any key to continue . . .
```

```cpp
#include <iostream>
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
int shellSort(int arr[], int size) {
    int swaps = 0;

    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];
                swaps++;
            }
            arr[j] = temp;
        }

        std::cout << "After gap = " << gap << ": " << std::endl;
        printArray(arr, size);
    }

    return swaps;
}

#endif
```

Observations

Here, it starts with the function printArray() which loops through the array and prints each element separated by spaces. The shellSort() function sorts the array using the shell sort algorithm and the swaps counts the number of swaps made into the sorting technique. The first loop divides into halves the array for the gap. Each array reduces the gap by half until it becomes 1. When it is gap = 1, it will start the normal insertion sort. The inner loop is the gapped insertion sort where it picks an element array and store it into the temp. The, it compares it with the elements that are gap apart. It shifts larger elements forward to make spaces. It inserts into temp into its correct position. After the finishing sorting with the current gap, it will print the current state of the array that is sorted.

Table 8-2. Shell sort technique

Code + Console Screenshot

```
=== Merge Sort ===
Final Sorted Array (Merge Sort):
1 2 3 4 5 6 6 7 7 9 12 12 13 17 17 19 19 22 23 24 24 24 25 27 27 28 28 30 30 30 30 32 34 35 36 36 37 38
1 42 42 42 42 43 43 43 45 45 46 47 48 48 49 49 51 54 54 55 57 58 59 60 62 63 63 65 65 65 67 68 68 69 70
9 79 82 83 83 85 89 91 91 92 92 93 94 95 96 96 100
Total divisions: 99
Total conquers (merges): 99
```

```cpp
int divisions = 0;
int conquers = 0;
void merge(int arr[], int left, int middle, int right) {
    conquers++;

    int n1 = middle - left + 1;
    int n2 = right - middle;

    int* L = new int[n1];
    int* R = new int[n2];

    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[middle + 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++;
    }

    delete[] L;
    delete[] R;
}

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

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

Observations

Here in merge sorting, it initializes the division and conquers to count the number of it as it sorts. Inside the merge function, the conquers++ means merging the two sorted halves into one. Next, it finds the sizes

	of two halves which counts the size of the left and right subarray. The L[] holds the left half and R[] holds the right half. These two are temporary arrays. The first loop fills the left array with elements from left until to middle. While the second loop fills the right array with elements from middle+1, since we've counted the middle to the left, to the right-side array. It merges the two-halves by comparing the left array and right array, places the smaller one into the original array, and moves the index I or J forward. It copies the rest of the other half into the array if one half is finished. Moreover, it deletes the temporary array left and right. The mergeSort() function checks if the array has only 1 element, greater than left and equal to right, it does not happen anything since it is already sorted. The divisions++ increases as it splits into halves the array so that it will count the number of divisions. After that, it finds the middle by dividing the array. Then, it just recursively sorts out the left and right halves. After both is sorted, it merges and combines them back in order. Both the division and conquer is 99 since you divide them until the array is a single element and at the same time you merge all the divided array so it has the same result.

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot

```cpp
int quickComparisons = 0;

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

    for (int j = low; j < high; j++) {
        quickComparisons++;
        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 pivotIndex = partition(arr, low, high);

        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}
```

```cpp
//quick sort
std::cout << "\n=== Quick Sort ===" << std::endl;
quickComparisons = 0;
quickSort(arr, 0, SIZE - 1);
std::cout << "Final Sorted Array (Quick Sort):" << std::endl;
printArray(arr, SIZE);
std::cout << "Total comparisons (Quick Sort): " << quickComparisons << std::endl;
```

```
=== Quick Sort ===
Final Sorted Array (Quick Sort):
1 2 3 4 5 6 6 7 7 9 12 12 13 17 17 19 19 22 23 24 24 24 25 27 27 28 28 30 30 30 30 32 34 35 36 36 37 38 38 39 39 40 41 4
1 42 42 42 42 43 43 43 45 45 46 47 48 48 49 49 51 54 54 55 57 58 59 60 62 63 63 65 65 65 67 68 68 69 70 70 71 72 74 77 7
9 79 82 83 83 85 89 91 91 92 92 93 94 95 96 96 100
Total comparisons (Quick Sort): 3323
```

Observations

In quick sort, the partition function is where the arrays divide the lower value to the pivot is on the left, higher value than the pivot is on the right, and the picking of the pivot which is the last element. First, it chooses the last element on the subarray as the pivot. The index i represents as the boundary of the elements smaller than the pivot. For the for loop, it checks each element except the pivot. The quickComparison that is initialize before the function is increment every time we compare the arr[j] with pivot. If the arr[j] is greater than the pivot, it belong to the left side then it inreases i boundary.

	Arr[i] and arr[j] swaps to move the elements into the left partition. After the loop ends, all elements smaller than the pivot are placed on the left and larger are on the right. It swaps the pivot into its correct position. In the quick sort function, it has a condition where if the array segment has only 1 element, it is already sorted. Now, it will rearranges the subarray and places pivot in its correct position. It will now call the quick sort for recursion to sort the left half of the array and the right half. Each call divides the array around a pivot and conquers each half recursively.

Table 8-4. Quick sort Algorithm

7. Supplementary Activity

Problem 1:

```
Original array:
29 10 14 37 13 25 5 30 18 7

Array after sorting with Partition + (Merge Sort on left, Insertion Sort on right):
5 7 10 13 14 18 25 29 30 37
```

```cpp
#ifndef SORTING_H
#define SORTING_H

#include <iostream>

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

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

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

    int *L = new int[n1];
    int *R = new int[n2];

    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) {
```

```cpp
    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++];

    delete[] L;
    delete[] R;
}

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);
    }
}

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 quickPartitionWithOtherSorts(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);

        mergeSort(arr, low, pivotIndex - 1);
```

```cpp

        insertionSort(arr, pivotIndex + 1, high);
    }
}

#endif
```

```cpp
#include <iostream>
#include "sorting.h"

int main() {
    const int size = 10;
    int arr[size] = {29, 10, 14, 37, 13, 25, 5, 30, 18, 7};

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

    quickPartitionWithOtherSorts(arr, 0, size - 1);

    std::cout << "\nArray after sorting with Partition + (Merge Sort on left, Insertion Sort on right):\n";
    for (int i = 0; i < size; i++) std::cout << arr[i] << " ";
    std::cout << "\n";

    return 0;
}
```

Explanation:
It is possible to implement other sorting algorithms within the partition method that is used in the quick sort method. We can just partition it first then implement other sorting functions either on the right or left side of the array. Here in the demonstrated code, the swap function is used to exchange two values of two variables. So, we implemented the partition function, merge function, and shell sort algorithms. To implement these two sorting algorithms in the partition method, we make another function that is combined with the other two and input call the merge function to sort the left and shell function for the right side of the subarray.
Problem 2:

```
Z:\Crishen\cpp\mainSort2.exe
Original Array:
4 34 29 48 53 87 12 30 44 25 93 67 43 19 74

Sorted Array (Quick Sort):
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Quick Sort Comparisons: 47
Quick Sort Swaps: 33

Sorted Array (Merge Sort):
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Merge Sort Comparisons: 42

Sorted Array (Shell Sort):
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Shell Sort Comparisons: 17
Shell Sort Swaps: 17
```

```cpp
#ifndef FASTSORT_H
#define FASTSORT_H

#include <iostream>
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
//SHELL SORT
int shellComparisons = 0, shellSwaps = 0;

void shellSort(int arr[], int 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) {
                shellComparisons++;
                arr[j] = arr[j - gap];
                shellSwaps++;
            }
            arr[j] = temp;
        }
    }
}
//MERGE SORT
int mergeComparisons = 0;
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *L = new int[n1];
    int *R = new int[n2];
```

```

    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) {
        mergeComparisons++;
        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++];
    }
    delete[] L;
    delete[] R;
}

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 quickComparisons = 0, quickSwaps = 0;
```

```cpp
int quickComparisons = 0, quickSwaps = 0;

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
    quickSwaps++;
}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

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

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

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

        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}
#endif
```

```cpp
#include <iostream>
#include "fastsort.h"

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

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

    int quickArr[SIZE];
    for (int i = 0; i < SIZE; i++) {
        quickArr[i] = arr[i];
    }

    quickSort(quickArr, 0, SIZE - 1);

    std::cout << "\nSorted Array (Quick Sort):" << std::endl;
    printArray(quickArr, SIZE);
    std::cout << "Quick Sort Comparisons: " << quickComparisons << std::endl;
    std::cout << "Quick Sort Swaps: " << quickSwaps << std::endl;

    int mergeArr[SIZE];
    for (int i = 0; i < SIZE; i++) {
        mergeArr[i] = arr[i];
    }

    mergeSort(mergeArr, 0, SIZE - 1);

    std::cout << "\nSorted Array (Merge Sort):" << std::endl;
    printArray(mergeArr, SIZE);
    std::cout << "Merge Sort Comparisons: " << mergeComparisons << std::endl;

    int shellArr[SIZE];
    for (int i = 0; i < SIZE; i++) {
        shellArr[i] = arr[i];
    }
```
```cpp
    }

    shellSort(shellArr, SIZE);

    std::cout << "\nSorted Array (Shell Sort):" << std::endl;
    printArray(shellArr, SIZE);
    std::cout << "Shell Sort Comparisons: " << shellComparisons << std::endl;
    std::cout << "Shell Sort Swaps: " << shellSwaps << std::endl;

    return 0;
}

```

Explanation:
Based on the given array, the fastest to sort is the shell sort. It is because the shell sort reduces the number of swaps due to using of gaps and applying an insertion sort to it, which avoids the overhead of recursive function calls. Hence, it is more efficient than the other sort since it is only a small size of array. If the array has larger size, then quick sort has the advantage since due to the partitioning method. While the merge sort has higher memory overhead since it needs temporary arrays. Both the merge and quick sort have O(N x log N) complexity because both divides the array into two halves. For merge, at each level of recursion, it visits all N elements in merging. While the quick sort, it scans through all N elements once in partitioning.

8. Conclusion

In conclusion, the shell sort is the upgraded version of insertion sort since it operates the insertion sort with gaps and groups the numbers to sort out and insertion sort is operated so that it has less to swaps and compare. The merge sort divides the array into two halves and it is repeated until the array has only one size. Then, it sorts the subarrays and merges it back into the original size. The quick sort also divides the array by choosing a pivot and placing it into the right position. When it is placed, it divides the array into two halves which the left side has smaller value than the array in the pivot and larger on the right. The process is repeated until it is sorted. For the procedure, using a loop is necessary to switch the places of index and moves the index into the next position. For the quick and merge sort, it needs to divide the arrays into halves so we have to initialize a left and right arrays so that it would separate it. The supplementary has similar structure to the procedure but it only modifies a little. It also adds new function so that we could combine the sorting algorithm and implement the two halves with different sorting function. Sorting algorithms has different suitable size for an array to have the most efficient time of sorting. In this activity, I think I’ve understanded well the concept of the sorting algorithm and its process. However, I think I need to improve in the implementation of coding since I’ve had a hard time implementing it.

9. Assessment Rubric