Activity No. < 8>	
SORTING ALGORITHMS: SHELL, MERGE, AND QUICK SORT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 9/27/25**
Section: CPE21S4	**Date Submitted: 9/27/25**
Name(s): Quioyo Angelo	**Instructor: Engr. Jimlord Quejado**

6. Output:

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

Main CPP:

table 8.1.cpp | sorts_8.1.h

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include "sorts_81.h"
using namespace std;

const int SIZE = 100;

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

void copyArray(int src[], int dest[], int n) {
    for (int i=0; i<n; i++) {
        dest[i] = src[i];
    }
}

int main() {
    srand(time(0));

    int arr[SIZE];
    for (int i=0; i<SIZE; i++) {
        arr[i] = rand() % 1000;
    }

    cout << "Original Array (unsorted):\n";
    printArray(arr, SIZE);
    cout << "==================================================\n";

    int arrShell[SIZE], arrMerge[SIZE], arrQuick[SIZE];
    copyArray(arr, arrShell, SIZE);
    copyArray(arr, arrMerge, SIZE);
    copyArray(arr, arrQuick, SIZE);

    shellSort(arrShell, SIZE);
    mergeSort(arrMerge, 0, SIZE-1);
    quickSort(arrQuick, 0, SIZE-1);

    cout << "Array after Shell Sort:\n";
    printArray(arrShell, SIZE);
    cout << "--------------------------------------------------\n";

    cout << "Array after Merge Sort:\n";
    printArray(arrMerge, SIZE);
    cout << "--------------------------------------------------\n";

    cout << "Array after Quick Sort:\n";
    printArray(arrQuick, SIZE);
    cout << "--------------------------------------------------\n";

    cout << "\nTable 8-1: Sorting Results\n";
    cout << "--------------------------------------------------\n";
    cout << left << setw(15) << "Index"
         << setw(15) << "ShellSort"
         << setw(15) << "MergeSort"
         << setw(15) << "QuickSort" << endl;
    cout << "--------------------------------------------------\n";

    for (int i=0; i<SIZE; i++) {
        cout << left << setw(15) << i
             << setw(15) << arrShell[i]
             << setw(15) << arrMerge[i]
             << setw(15) << arrQuick[i] << endl;
    }
    cout << "--------------------------------------------------\n";

    return 0;
}
```

Header File:

table 8.1.cpp | sorts_8.1.h

```cpp
#ifndef SORTS_H
#define SORTS_H

#include <iostream>
using namespace std;

// Shell Sort
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 = i;
            while (j >= gap && arr[j-gap] > temp) {
                arr[j] = arr[j-gap];
                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 = 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) {
        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);
    }
}
```

```cpp
// Quick Sort
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 pi = partition(arr, low, high);

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

#endif
```

Output:

```
Original Array:
62 41 98 44 89 20 96 79 70 70 74 28 73 4 48 72 8 8 61 4

Sorted using Shell Sort:
4 4 8 8 20 28 41 44 48 61 62 70 70 72 73 74 79 89 96 98

Sorted using Merge Sort:
4 4 8 8 20 28 41 44 48 61 62 70 70 72 73 74 79 89 96 98

Sorted using Quick Sort:
4 4 8 8 20 28 41 44 48 61 62 70 70 72 73 74 79 89 96 98

--------------------------------
Process exited after 0.1038 seconds with return value 0
Press any key to continue . . .
```

Explanation:

In this program, I used the three different sorting algorithms: Shell Sort, Merge Sort, and Quick Sort. The program first generates 20 random numbers and then makes copies of the array so each sorting method can work on the same data. I put all the sorting functions inside my header file (sorts.h) to make the main code cleaner. After running, the output shows that all three algorithms sort the numbers correctly, just in different ways.

Table 8-2. Shell Sort Technique

Main CPP:

shell_header.cpp | shell_header.h

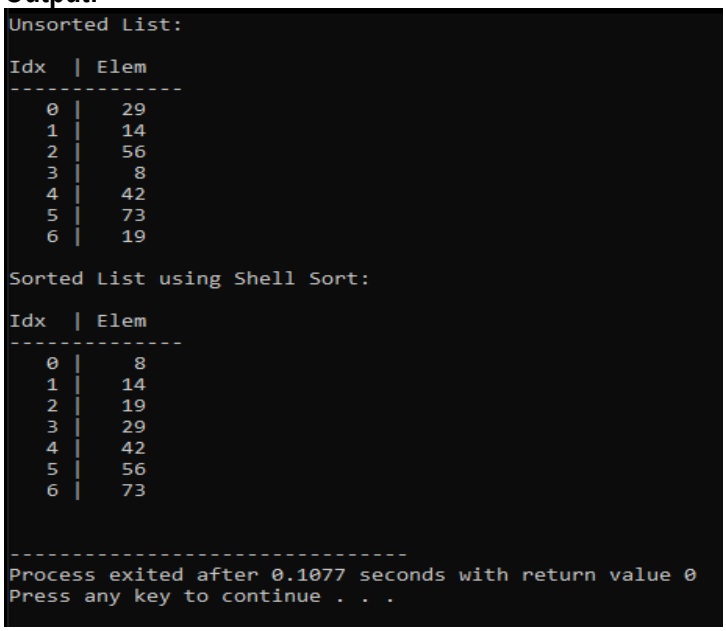
```cpp
#include "shell_header.h"

int main() {
    int numbers[] = {29, 14, 56, 8, 42, 73, 19};
    int n = sizeof(numbers) / sizeof(numbers[0]);

    cout << "Unsorted List:\n";
    showArray(numbers, n);

    doShellSort(numbers, n);

    cout << "Sorted List using Shell Sort:\n";
    showArray(numbers, n);

    return 0;
}

```

Header File:

shell_header.cpp | shell_header.h

```cpp
#ifndef SHELL_HEADER_H
#define SHELL_HEADER_H

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

void doShellSort(int nums[], int length) {
    for (int step = length / 2; step > 0; step /= 2) {
        for (int i = step; i < length; i++) {
            int current = nums[i];
            int j = i;
            // shifting elements
            while (j >= step && nums[j - step] > current) {
                nums[j] = nums[j - step];
                j -= step;
            }
            nums[j] = current;
        }
    }
}

void showArray(int nums[], int length) {
    cout << "\nIdx  | Elem\n";
    cout << "--------------\n";
    for (int i = 0; i < length; i++) {
        cout << setw(4) << i << " | " << setw(4) << nums[i] << endl;
    }
    cout << endl;
}

#endif

```

Output:



Explanation:

In this program I sorts the numbers using the Shell Sort method. This starts with a big gap between numbers, then makes the gap smaller until it is 1.The doShellSort function does the sorting by moving numbers to the right place. The showArray function shows the list in a small table with index and element.
In the end, the program prints the list before and after sorting.

Table 8-3. Shell Sort Technique

Main CPP:

msert.cpp | msort.h

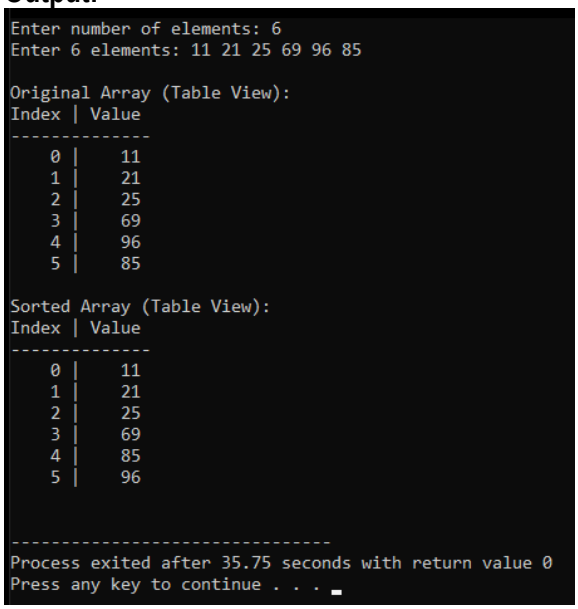
```cpp
#include "msort.h"

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

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

    cout << "\nOriginal Array (Table View):\n";
    printTable(arr, n);

    mergeSort(arr, 0, n - 1);

    cout << "Sorted Array (Table View):\n";
    printTable(arr, n);

    return 0;
}

```

Header File:

msert.cpp | msort.h

```cpp
#ifndef MSORT_H
#define MSORT_H

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

void doMerge(int arr[], int start, int mid, int e
    int n1 = mid - start + 1;
    int n2 = end - mid;

    int left[n1], right[n2];

    for (int i = 0; i < n1; i++)
        left[i] = arr[start + i];
    for (int j = 0; j < n2; j++)
        right[j] = arr[mid + 1 + j];

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

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

    while (i < n1) {
        arr[k] = left[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = right[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int start, int end) {
    if (start < end) {
        int mid = (start + end) / 2;
        mergeSort(arr, start, mid);
        mergeSort(arr, mid + 1, end);
        doMerge(arr, start, mid, end);
    }
}

void printTable(int arr[], int n) {
    cout << "Index | Value\n";
    cout << "--------------\n";
    for (int i = 0; i < n; i++) {
        cout << setw(5) << i << " | " << setw(5) << arr[i] << endl;
    }
    cout << endl;
}

#endif

```

Output:



Explanation:

This program I used the merge sort algorithm to sort an array of numbers.
The mergeSort function divides the array into smaller parts, and the doMerge function combines the array back in order. The program also uses printTable to display the array in a table format with index and value.
The user can clearly see the array before sorting and after sorting, making the process easy to follow.

Table 8-4. Quick Sort Algorithm

Main CPP:

qs.cpp qs.h

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

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

    while (true) {
        while (i <= high && arr[i] <= pivot) i++;
        while (j >= low && arr[j] > pivot) j--;

        if (i < j) {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        } else {
            break;
        }
    }

    int temp = arr[low];
    arr[low] = arr[j];
    arr[j] = temp;

    return j;
}

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 n;
    cout << "Enter size: ";
    cin >> n;

```

```cpp
int main() {
    int n;
    cout << "Enter size: ";
    cin >> n;

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

    quickSort(arr, 0, n - 1);

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

    return 0;
}

```

Output:

```
Enter size: 5
Input elements: 12 22 18 19 69
Result: 12 18 19 22 69

--------------------------------
Process exited after 13.88 seconds with return value 0
Press any key to continue . . .
```

Explanation:

The program takes numbers from the user and sorts them using Quick Sort. It picks the first element as the pivot and then moves smaller numbers to the left and bigger numbers to the right. The process keeps repeating on the left and right parts of the array until everything is sorted. In the end, the program prints out the sorted array.

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.

- **In the quick sort algorithm you divide an array into two sections based on a pivot number -- those elements less than the pivot will go on the left, and those greater than the pivot on the right. With 12, 7, 15, 20, 5, 10 as our array items, where 12 is the pivot, we create a left side with 7, 5, 10 and a right side with 15, 20. Here, we could apply insertion sort to the left and bubble sort for the right portion of the structure. When we re-combine the two arrays we have then established an ordered array as: {5, 7, 10, 12, 15, 20}.**

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?

- **The array provided should be sorted very quickly in the quick sort or in the merge sort. Both of these algorithm-based techniques are much faster than bubble sort, selection sort, or insertion sort. The reason for their speed is that they divide the array into smaller portions. They solve these sections little by little. The total number of steps in this case becomes around N log N, not N^2. Hence, quick sort and merge sort are good enough for bigger arrays.**

8. Conclusion:

- These activity showed Shell Sort, Merge Sort and Quick Sort correctly sorted the data in ascending order. Again, the process of approaching the sorting problem for the three algorithms differed, while at the same time, the goal remains to sort data items. While again Shell sort soon show its simplicity and speed. Merge Sort when decomposing the problem into small problems just made it easier to solve. Just another reaffirmation of the same observations, is the flexibility of the Quick Sort algorithm - there as many ways is it writable you still have at least one correct result - all of the time. Once again this show that the approached to the sorting provided is more subjective than objective and the possibilities is are considered immediately - especially with larger arrays.

9. Assessment Rubric