Activity No. 7.2	
Hands-on Activity 7.2 Sorting Algorithms	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 21/10/2024**
Section: CPE21S4	**Date Submitted: 21/10/2024**
Name(s): Kenn Jie Valleser	**Instructor: Engr. Ma. Rizette Sayo**
6. Output	

Code + Console Screenshot

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

using namespace std;

void CRandomValues(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % 1000 + 1;
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    const int size = 100;
    int randomArray[size];

    CRandomValues(randomArray, size);


    cout << "Unsorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << randomArray[i] << " ";
    }
    cout << endl;

    return 0;
}
```

```
Unsorted Array:
986 755 469 278 33 810 262 135 970 703 442 360 967 38 134 314 652 864 326 80 486 479 181 101 677 610 339 101 233 315 957
 719 620 991 987 521 28 613 790 101 36 603 401 994 571 653 833 94 213 88 272 491 329 446 150 151 643 891 681 537 811 112
 423 656 502 413 161 934 266 732 67 269 776 35 17 801 693 546 292 750 241 86 276 643 797 302 408 728 942 972 973 516 583
 174 852 409 126 190 287 892

--------------------------------
Process exited after 0.03985 seconds with return value 0
Press any key to continue . . .
```

Observation

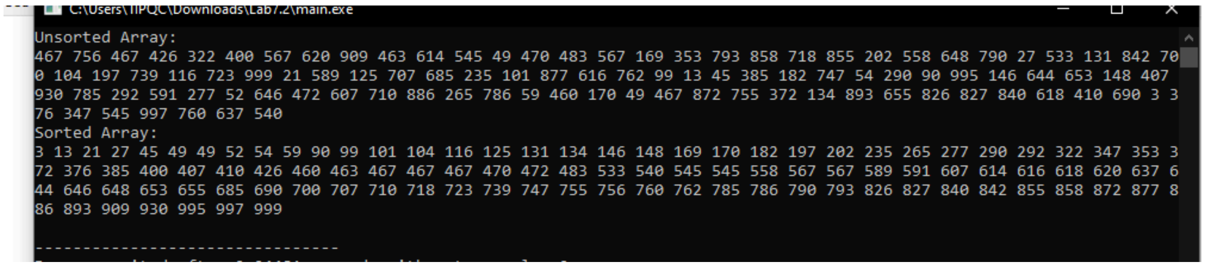
It uses ctime to be able to create numbers from 1 to 1000 and then when it is executed it displays random values

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

Code + Console Screenshot	(code below)

```
#include <iostream>
#include <ctime>
#include "Shell Sort.h"
using namespace std;

void CRandomValues(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % 1000 + 1;
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    const int size = 100;
    int randomArray[size];

    CRandomValues(randomArray, size);


    cout << "Unsorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << randomArray[i] << " ";
    }
    cout << endl;


            shellSort(randomArray, size);

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

=============================================================================
#ifndef SHELL_SORT_H
#define SHELL_SORT_H


void shellSort(int array[], int size) {

    for (int interval = size / 2; interval > 0; interval /= 2) {
        for (int i = interval; i < size; i++) {
            int temp = array[i];
            int j = i;
                   while (j >= interval && array[j - interval] > temp) {
                array[j] = array[j - interval];
                j -= interval;
```

```
            }
            array[j] = temp;
        }
    }
}

#endif
```



Observation	After applying Shell Sort, the output revealed a sorted array, meaning the algorithm's effectiveness in organizing the data efficiently.

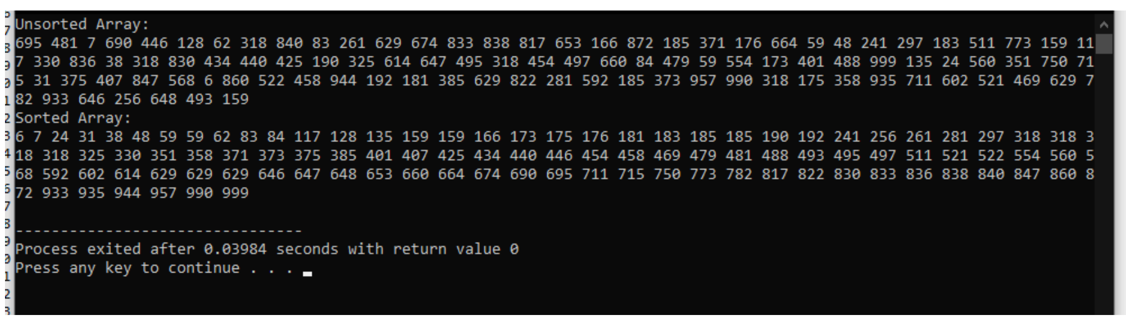
Table 8-2. Shell Sort Technique

Code + Console Screenshot

```
#include <iostream>
#include <ctime>
#include "Merge Sort.h"
using namespace std;

void CRandomValues(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % 1000 + 1;
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    const int size = 100;
    int randomArray[size];

    CRandomValues(randomArray, size);


    cout << "Unsorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << randomArray[i] << " ";
    }
    cout << endl;


            mergeSort(randomArray, 0, size - 1);

    cout << "Sorted Array:\n";
```

```
    for (int i = 0; i < size; ++i) {
        cout << randomArray[i] << " "; // Display all elements in a single line
    }
    cout << endl;

    return 0;
}
================================================================================
#ifndef MERGE_SORT_H
#define MERGE_SORT_H


void merge(int array[], int left, int middle, int right) {
    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] = array[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = array[middle + 1 + j];


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

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            array[k] = L[i];
            i++;
        } else {
            array[k] = R[j];
            j++;
        }
        k++;
    }


    while (i < n1) {
        array[k] = L[i];
        i++;
        k++;
    }


    while (j < n2) {
        array[k] = R[j];
```

```
        j++;
        k++;
    }


    delete[] L;
    delete[] R;
}


void mergeSort(int array[], int left, int right) {
    if (left < right) {

        int middle = left + (right - left) / 2;


        mergeSort(array, left, middle);
        mergeSort(array, middle + 1, right);


        merge(array, left, middle, right);
    }
}

#endif
```



Observation	The sorted array after using Merge Sort confirmed the algorithm's ability to consistently arrange elements in ascending order.

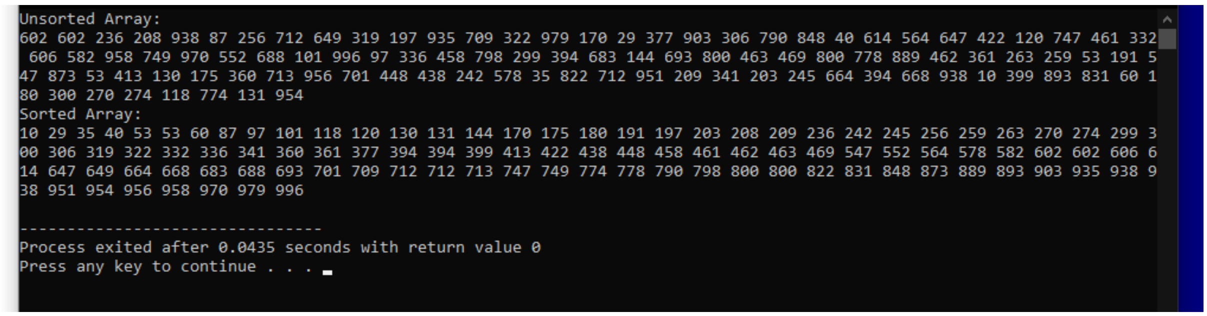
Table 8-3. Merge Sort Algorithm

Code + Console Screenshot

```
#include <iostream>
#include <ctime>
#include "Quick Sort.h"
using namespace std;

void CRandomValues(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % 1000 + 1;
    }
```

```
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    const int size = 100;
    int randomArray[size];

    CRandomValues(randomArray, size);


    cout << "Unsorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << randomArray[i] << " ";
    }
    cout << endl;


            quickSort(randomArray, 0, size - 1);

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

    return 0;
}
============================================================================
#ifndef QUICK_SORT_H
#define QUICK_SORT_H


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

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

    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {
            i++;
            swap(array[i], array[j]);
        }
    }
    swap(array[i + 1], array[high]);
    return i + 1;
}
```

<table>
<tr><td></td><td>
<pre>
void quickSort(int array[], int low, int high) {
    if (low < high) {
        int pivot = partition(array, low, high);
        quickSort(array, low, pivot - 1);
        quickSort(array, pivot + 1, high);
    }
}

#endif
</pre>


</td></tr>
<tr><td>Observation</td><td>Sorting with Quick Sort, the array was successfully ordered, highlighting the algorithm's performance in handling diverse datasets efficiently.</td></tr>
</table>

Table 8-4. Quick Sort Algorithm

7. Supplementary Activity

Problem 1:

Yes, we can sort the left and right sublists from the partition method in Quick Sort using other sorting algorithms. For example, if we have the array [3, 6, 8, 10, 1, 2, 1] and use 1 as the pivot, the array might be partitioned into [1] as the left sublist and [3, 2, 6, 8, 10] as the right sublist. We could then sort the right sublist using Merge Sort, which involves dividing it into smaller parts, sorting those, and then merging them back together. This way, even though Quick Sort is used initially, we can still utilize another sorting algorithm to organize the sublists effectively.

Problem 2:

For the array {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19, 74\}, both Merge Sort and Quick Sort would give the best time performance, each with a complexity of O(N • log N). Merge Sort works by dividing the array into smaller parts and then merging them back in a sorted order, while Quick Sort selects a pivot to partition the array and sorts the sublists. Both methods use a divide-and-conquer approach, which allows them to be efficient even for larger datasets.

8. Conclusion

In this activity, I learned about Shell Sort, Merge Sort, and Quick Sort, and how each algorithm has unique strengths and use cases. Analyzing the procedures helped me understand that Shell Sort is efficient for smaller or nearly sorted datasets, while Merge Sort consistently performs well on larger arrays due to its stable O(N • log N) complexity. Quick Sort, although faster in practice, can vary in efficiency based on the choice of pivot. Overall, I feel I did well in grasping these sorting algorithms, but I aim to improve my ability to compare their efficiencies in different scenarios and practice their implementations more frequently.

9. Assessment Rubric