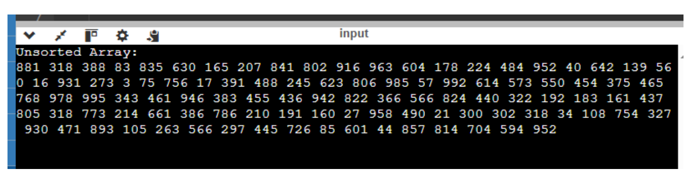
Activity No. 8	
Sorting Algorithms	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** Oct 21, 2024
Section:CPE21S4	**Date Submitted:** Oct 23, 2024
Name(s): Danica T. Guariño	**Instructor: Ma'am Rizette Sayo**
6. Output	

Code + Console Screenshot

C/C++

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

 int main() {

 const int size = 100;
    int arr[size];
    std::srand(std::time(0));

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


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



Observations	In this program, I just did the same process I did last activity wherein I called 100 random array numbers and all of it is still unsorted.

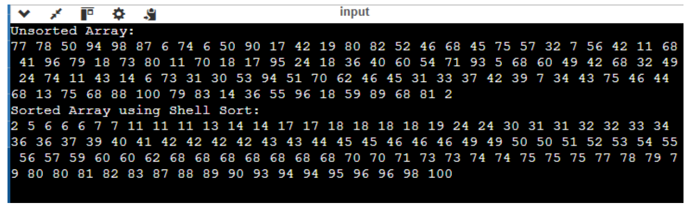
Table 8-1. Array of Values for Sort Algorithm Testing

Code + Console Screenshot	

C/C++

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

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() {
    const int size = 100;
    int arr[size];
    std::srand(std::time(0)); // Seed
for random number generation

    for (int i = 0; i < size; ++i) {
        arr[i] = std::rand() % 100 + 1;
// Random values between 1 and 1000
    }

    // Display the unsorted array
    cout << "Unsorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << std::endl;

    // Sort the array using Shell Sort
    shellSort(arr, size);

    // Display the sorted array
    cout << "Sorted Array using Shell
Sort:\n";
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << std::endl;

    return 0;
}
```

<table>
<tr>
<td></td>
<td>



</td>
</tr>
<tr>
<td>Observations</td>
<td>I used the same codes in my main but in this program, the unsorted numbers are now sorted and I used Shell Sort in sorting.</td>
</tr>
</table>

Table 8-2. Shell Sort Technique

<table>
<tr>
<td>Code + Console Screenshot</td>
<td>

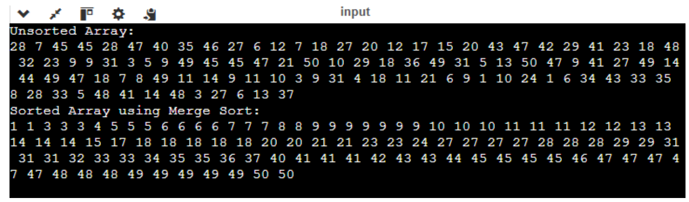
C/C++

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

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

    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];
```

</td>
</tr>
</table>

```
        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)
        return;

    int middle = left + (right - left) /
2;
    mergeSort(arr, left, middle);
    mergeSort(arr, middle + 1, right);
    merge(arr, left, middle, right);
}

int main() {
    const int size = 100;
    int arr[size];
    std::srand(std::time(0)); // Seed
for random number generation

    for (int i = 0; i < size; ++i) {
        arr[i] = std::rand() % 50 + 1;
// Random values between 1 and 1000
    }

    // Display the unsorted array
    cout << "Unsorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << std::endl;

    // Sort the array using Merge Sort
    mergeSort(arr, 0, size - 1);

    // Display the sorted array
    cout << "Sorted Array using Merge
Sort:\n";
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << std::endl;
```

```
    return 0;
}
```



Observations	I used the same codes in my main but in this program, the unsorted array numbers are now sorted and I used Merge Sort in sorting.

Table 8-3. Merge Sort Algorithm

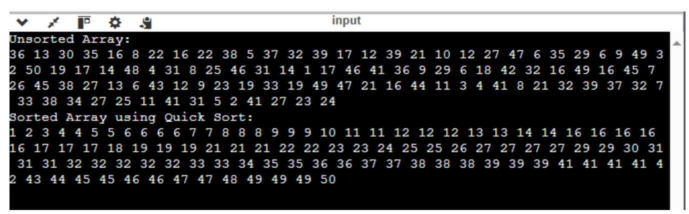
Code + Console Screenshot

C/C++

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
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++;
            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 pi = partition(arr, low,
high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    const int size = 100;
```

<table>
<tr>
<td></td>
<td>

```
    int arr[size];
    std::srand(std::time(0)); // Seed
for random number generation

    for (int i = 0; i < size; ++i) {
        arr[i] = std::rand() % 50 + 1;
// Random values between 1 and 1000
    }

    // Display the unsorted array
    cout << "Unsorted Array:\n";
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << std::endl;

    // Sort the array using Quick Sort
    quickSort(arr, 0, size - 1);

    // Display the sorted array
    cout << "Sorted Array using Quick
Sort:\n";
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << std::endl;

    return 0;
}
```



```
Unsorted Array:
36 13 30 35 16 8 22 16 22 38 5 37 32 39 17 12 39 21 10 12 27 47 6 35 29 6 9 49 3
2 50 19 17 14 48 4 31 8 25 46 31 14 1 17 46 41 36 9 29 6 18 42 32 16 49 16 45 7
26 45 38 27 13 6 43 12 9 23 19 33 19 49 47 21 16 44 11 3 4 41 8 21 32 39 37 32 7
 33 38 34 27 25 11 41 31 5 2 41 27 23 24
Sorted Array using Quick Sort:
1 2 3 4 4 5 5 6 6 6 6 7 7 8 8 8 9 9 9 10 11 11 12 12 12 13 13 14 14 16 16 16 16
16 17 17 17 18 19 19 19 21 21 21 22 22 23 23 24 25 25 26 27 27 27 27 29 29 30 31
 31 31 32 32 32 32 32 33 33 34 35 35 36 36 37 37 38 38 38 39 39 39 41 41 41 41 4
2 43 44 45 45 46 46 47 47 48 49 49 49 50
```

</td>
</tr>
<tr>
<td>Observations</td>
<td>I used the same codes in my main but in this program, the unsorted numbers are now sorted and I used Quick Sort in sorting.</td>
</tr>
</table>

Table 8-4. Quick Sort Algorithm

7. Supplementary Activity

ILO B: Solve given data sorting problems using appropriate basic sorting algorithms

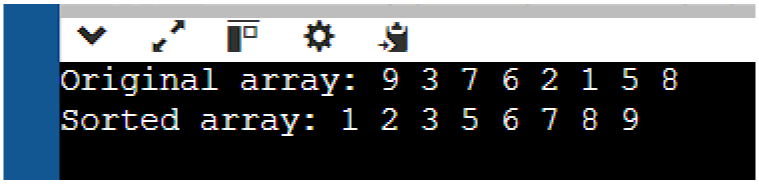
Problem 1: Can we sort the left sub list and right sub list from the partition method in quick sort using other sorting algorithms? Demonstrate an example.

C/C++

```cpp
#include <iostream>


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 - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}


void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

```cpp
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 quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);


        mergeSort(arr, low, pi - 1);

        bubbleSort(arr + pi + 1, high - pi);
    }
}

int main() {
    int arr[] = {9, 3, 7, 6, 2, 1, 5, 8};
    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;

    quickSort(arr, 0, n - 1);

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

    return 0;
}
```

Output:



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?

Quick Sort is the quickest sorting method in this scenario. Even though Merge Sort is an effective sorting algorithm with unique properties and a time complexity, it is always stable and will perform consistently in all cases. Quick Sort also averages. However, it fails to perform in the worst-case scenario due to bad pivot selection. Although Quick Sort is faster and takes up less space than Merge Sort, it is not stable. Merge Sort is utilized because it is stable and appropriate for huge data sets. Both algorithms use the divide-and-conquer strategy to achieve their objectives.

8. Conclusion

In conclusion, It was quite challenging in the beginning but mostly only during the initial process. By working with these algorithms, I obtained a better knowledge of their core concepts and operations. It allowed me to distinguish between the three sorting algorithms: Shell Sort, Merge Sort, and Quick Sort. This activity improved my algorithmic understanding while also improving my problem-solving abilities. It emphasized the need of selecting the appropriate sorting algorithm based on specific criteria such as dataset size, stability requirements, and memory limits.

9. Assessment Rubric