Activity No. 8	
SORTING ALGORITHMS: SHELL, MERGE, AND QUICK SORT	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/21/2024
Section: CPE21S1	**Date Submitted:** 10/21/2024
Name(s): Bonifacio, Redj Guillian F.	**Instructor:** Sayo, Maria Rizette

6. Output

Code + Console Screenshot

```cpp
int main() {
    const int SIZE = 100;
    std::vector<int> arr(SIZE);
    std::srand(std::time(nullptr));

    for (int& num : arr) num = std::rand() % 1000;
```

Observations

I noticed that after seeding using srand, the code successfully uses srand() to create 50 random integers, and the dataset varies each time the program runs. Before proceeding, the second loop verifies that the dataset has been created exactly as expected by correctly displaying these random values.

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

Code + Console Screenshot

```cpp
void shellSort(std::vector<int>& arr) {
    for (int gap = arr.size() / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < arr.size(); i++) {
            int temp = arr[i], j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];
            arr[j] = temp;
        }
    }
}
```

Observations

I noticed that Shell Sort effectively lowers the number of comparisons by letting elements jump over larger gaps at the beginning, which makes the sorting process faster. I find it easy to understand how it divides the array and gradually decreases the gaps, and I like how it makes insertion sort work better.

Table 8-2. Shell Sort Technique

Code + Console Screenshot

```cpp
void merge(std::vector<int>& arr, int left, int mid, int right) {
    std::vector<int> L(arr.begin() + left, arr.begin() + mid + 1);
    std::vector<int> R(arr.begin() + mid + 1, arr.begin() + right + 1);

    for (int i = left, j = 0, k = 0; i <= right; i++) {
        if (j >= L.size()) arr[i] = R[k++];
        else if (k >= R.size() || L[j] <= R[k]) arr[i] = L[j++];
        else arr[i] = R[k++];
    }
}
```

Observations	I find merge sort is effective because it splits the array into smaller pieces and then puts them back together in the right order. I prefer that as it keeps the order of equal elements t he same, which is helpful in situations where that matters

Table 8-3. Merge Sort Algorithm

Code + Console Screenshot	(see code below)

```cpp
int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high], i = low - 1;
    for (int j = low; j < high; j++)
        if (arr[j] < pivot) std::swap(arr[++i], arr[j]);
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(std::vector<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);
    }
}

```

Observations	Amongst all the sort algorithms, I find quick sort the most efficient because it sorts the array without needing extra memory. I also like that it usually performs well, but I have to remember that if I choose a bad pivot, it can slow down the sorting process

Table 8-4. Quick Sort Algorithm

7. Supplementary Activity

Problem 1:
Code

```cpp
#include <iostream>
#include <vector>

void insertionSort(std::vector<int>& arr) {
    for (int i = 1; i < arr.size(); i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

void selectionSort(std::vector<int>& arr) {
    for (int i = 0; i < arr.size() - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < arr.size(); j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        std::swap(arr[i], arr[minIndex]);
    }
}

int partition(std::vector<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) {
            std::swap(arr[++i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}
```

```cpp

void quickSort(std::vector<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);
    }
}

int main() {
    std::vector<int> arr = {10, 7, 8, 9, 1, 5};

    std::cout << "Original Array: ";
    for (int num : arr) std::cout << num << " ";
    std::cout << std::endl;

    int pivotIndex = partition(arr, 0, arr.size() - 1);
    std::cout << "Pivot Index: " << pivotIndex << " with value " << arr[pivotIndex] << std::endl;

    std::vector<int> leftSubList(arr.begin(), arr.begin() + pivotIndex);
    std::vector<int> rightSubList(arr.begin() + pivotIndex + 1, arr.end());

    insertionSort(leftSubList);
    std::cout << "Sorted Left Sub-list: ";
    for (int num : leftSubList) std::cout << num << " ";
    std::cout << std::endl;

    selectionSort(rightSubList);
    std::cout << "Sorted Right Sub-list: ";
    for (int num : rightSubList) std::cout << num << " ";
    std::cout << std::endl;

    return 0;
}
```

Output

```
/tmp/4EikKAukKj.o
Original Array: 10 7 8 9 1 5
Pivot Index: 1 with value 5
Sorted Left Sub-list: 1
Sorted Right Sub-list: 7 8 9 10


=== Code Execution Successful ===
```

Observation	The code efficiently demonstrates sorting algorithms by using quicksort for

	partitioning and then applying insertion and selection sorts on the resulting sublists.

Problem 2:

Code

```cpp
#include <iostream>
#include <vector>
#include <ctime>

void merge(std::vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    std::vector<int> L(n1), R(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++];
}

void mergeSort(std::vector<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);
    }
}

int quickPartition(std::vector<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) {
            std::swap(arr[++i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivotIndex = quickPartition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

int main() {
    std::vector<int> array = {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43, 19, 74};

    std::vector<int> quickArr = array;

    double start_s = clock();
    quickSort(quickArr, 0, quickArr.size() - 1);
    double stop_s = clock();
    std::cout << "Sorted Array using Quick Sort: ";
    for (int num : quickArr) std::cout << num << " ";
    std::cout << std::endl;
    std::cout << "Quick Sort time: " << (stop_s - start_s) / (CLOCKS_PER_SEC / 1000) << " milliseconds." << std::endl;

    std::vector<int> mergeArr = array;

    start_s = clock();
    mergeSort(mergeArr, 0, mergeArr.size() - 1);
    stop_s = clock();
    std::cout << "Sorted Array using Merge Sort: ";
    for (int num : mergeArr) std::cout << num << " ";
    std::cout << std::endl;
    std::cout << "Merge Sort time: " << (stop_s - start_s) / (CLOCKS_PER_SEC / 1000) << " milliseconds." << std::endl;

    return 0;
}
```

Output

```
/tmp/e3jyPv0F2b.o
Sorted Array using Quick Sort: 4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Quick Sort time: 0.003 milliseconds.
Sorted Array using Merge Sort: 4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Merge Sort time: 0.014 milliseconds.


=== Code Execution Successful ===
```

Observation	Generally, quick sort is faster and has more consistent results comparted to merge sort. Both Quick Sort and Merge Sort have an average time complexity of O(N log N) because they divide the array into smaller parts and sort them efficiently

8. Conclusion

In sorting algorithms, I find quick sort the fastest and best for saving memory, while I find merge sort easier because it keeps things in order when there are equal elements. Knowing these differences helps me pick the right

sorting method for different situations.
9. Assessment Rubric