Hands-on Activity 8.1	
Sorting Algorithms Pt2	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 09 - 27 - 25
Section: CPE21S4	**Date Submitted:** 09 - 27 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado

6. Output

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

Header:

```
#ifndef SORTING_PT2_H
#define SORTING_PT2_H

template<typename T>
void printArray(T arr[], size_t arrSize) {
   for (int i = 0; i < arrSize; i++) {
       std::cout << arr[i] << " ";
   }std::cout << '\n';
}
void randomNumberArray100(int arr[], size_t arrSize) {
   for (int i = 0; i < arrSize; i++) {
       arr[i] = rand() % 100;
   }
}

#endif //SORTING_PT2_H
```

Main:

```
#include <iostream>
#include "sorting_pt2.h"
const int maxSize = 100;

int main() {

   int arr[maxSize];
   randomNumberArray100(arr, maxSize);

   printArray(arr, maxSize);

   return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\HOA 8.1 Sorting Algorithms Pt2\cmake-build-debug\HOA_8_1_Sorting_Algorithm
41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36 91 4 2 53 92 82 21 16 18 95 47 26 71 38 69 12 67 99 35 94 3 11
 22 33 73 64 41 11 53 68 47 44 62 57 37 59 23 41 29 78 16 35 90 42 88 6 40 42 64 48 46 5 90 29 70 50 6 1 93 48 29 23 84
54 56 40 66 76 31 8 44 39 26 23 37 38 18 82 29 41

Process finished with exit code 0
```

Using the same random number generator in the previous hands-on-activities, I initialized the random array of numbers from 0-99. Since I didn't provide a seed, I will get the same set of random numbers every time I run the program. I added it in the sorting_pt2 header to keep my main driver function clean.

Table 8-2. Shell Sort Technique

Header:

```
template<typename T>
void printArray(T arr[], size_t arrSize) {
   for (int i = 0; i < arrSize; i++) {
       std::cout << arr[i] << " ";
   }std::cout << '\n';
}
void randomNumberArray100(int arr[], size_t arrSize) {
   for (int i = 0; i < arrSize; i++) {
       arr[i] = rand() % 100;
   }
}

// Shell Sort
template<typename T>
void shellSort(T arr[], size_t size) {
   for (size_t gap = size / 2; gap > 0; gap /= 2) {
       // Perform a gapped insertion sort
       for (size_t i = gap; i < size; i++) {
           T temp = arr[i];
           size_t j = i;

           // Shift earlier gap-sorted elements up until the correct location is
found
           while (j >= gap && arr[j - gap] > temp) {
               arr[j] = arr[j - gap];
               j -= gap;
           }

           // Put temp (the original arr[i]) in its correct location
           arr[j] = temp;
       }
   }
}
```

main:

```
#include <iostream>
```

```cpp
#include "sorting_pt2.h"
const int maxSize = 100;

void line() {
   std::cout <<
"---------------------------------------------------------------------\n";
}

int main() {

   int arr[maxSize];
   std::cout << "Random Array of 100 elements: \n";
   randomNumberArray100(arr, maxSize);
   printArray(arr, maxSize);

   line();

   std::cout << "Using Shell sort to sort arrays: \n";
   shellSort(arr, maxSize);
   printArray(arr, maxSize);



   return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\HOA 8.1 Sorting Algorithms Pt2\cmake-build-debug\HOA_8_1_Sorting_Algorithm
Random Array of 100 elements:
41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36 91 4 2 53 92 82 21 16 18 95 47 26 71 38 69 12 67 99 35 94 3 11
 22 33 73 64 41 11 53 68 47 44 62 57 37 59 23 41 29 78 16 35 90 42 88 6 40 42 64 48 46 5 90 29 70 50 6 1 93 48 29 23 84
54 56 40 66 76 31 8 44 39 26 23 37 38 18 82 29 41
------------------------------------------------------------
Using Shell sort to sort arrays:
0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29 29 31 33 34 35 35 36 37 37 38 38 39 40 4
0 41 41 41 41 42 42 42 44 44 45 46 47 47 48 48 50 53 53 54 56 57 58 59 61 62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 7
8 78 81 82 82 84 88 90 90 91 91 92 93 94 95 95 99

Process finished with exit code 0
```

The program begins on its first iteration by setting the gap to be half of the max size. After every iteration, the gap is halved until we reach 1. Now, within each iteration of this outer loop, the elements chosen are “gap” elements apart (so in this iteration, all elements “gap” apart are treated as a single subgroup to be sorted). The element at index i is the gap element which is stored in the temp variable. Then, within the while loop, while the j element is greater than the gap element and the element before it is greater than temp, arr[j] gets shifted to the right. Then, after that loop, the temp variable is inserted in its rightful place at arr[j]. To implement it in main, it takes two parameters: the array itself and the size of the array.

Table 8-3. Merge Sort Algorithm

Header:

```
// Merge Sort
// Merge two sorted subarrays: arr[left..mid] and arr[mid+1..right]
template<typename T>
void merge(T arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temporary arrays
    T* L = new T[n1];
    T* R = new T[n2];

    // Copy data
    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];

    // Merge back
    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++];
        }
    }

    // Copy
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    delete[] L;
    delete[] R;
}

// Recursive merge sort
template<typename T>
void merge_sort(T arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);

        // Merge the sorted halves
```

```
        merge(arr, left, mid, right);
    }
}
```

Main:

```
int main() {

    int arr[maxSize];
    std::cout << "Random Array of 100 elements: \n";
    randomNumberArray100(arr, maxSize);
    printArray(arr, maxSize);

    line();

    std::cout << "Merge Sort: \n";
    merge_sort(arr, 0, maxSize - 1);
    printArray(arr, maxSize);


    return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\HOA 8.1 Sorting Algorithms Pt2\cmake-build-debug\HOA_8_1_Sorting_Algorithm
Random Array of 100 elements:
41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36 91 4 2 53 92 82 21 16 18 95 47 26 71 38 69 12 67 99 35 94 3 11
 22 33 73 64 41 11 53 68 47 44 62 57 37 59 23 41 29 78 16 35 90 42 88 6 40 42 64 48 46 5 90 29 70 50 6 1 93 48 29 23 84
54 56 40 66 76 31 8 44 39 26 23 37 38 18 82 29 41
----------------------------------------------------------------
Merge Sort:
0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29 29 31 33 34 35 35 36 37 37 38 38 39 40 4
0 41 41 41 41 42 42 42 44 44 45 46 47 47 48 48 50 53 53 54 56 57 58 59 61 62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 7
8 78 81 82 82 84 88 90 90 91 91 92 93 94 95 95 99

Process finished with exit code 0
```

The algorithm contains two functions: the merge and merge_sort function. Merge is the backbone of the program because this is where the actual sorting occurs. It takes four parameters: the array itself, the index of the left element, the middle element, and the index of the right element (or element at the end of the array).
int n1 = mid - left + 1;
int n2 = right - mid;

These are used to calculate the size of the two sub arrays, n1 being the size of the left array and the n2 as the size of the right array. Then:

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

We allocate memory for the temporary arrays for the sorting. Afterwards, with the temporary arrays and their sizes declared, we can copy the data using two for loops:

```
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];
```

Then, three variables are declared: i and j which track positions in the left and right arrays, then the k which is the index in the original array for when we start merging them. Then we do this loop:

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

While i (index for left) is less than size n1 (this ensures only elements within this sub array are included) and while j < n2 (same logic), we test two conditions: if L[i] is less than or equal to R[j], then in the original array, its position will be to the left. Else, the element being tested will be to the right of the array, again, keeping the index k in check. This is also where the arrays are finally merged, completely sorted. Then lastly:

```
while (i < n1) arr[k++] = L[i++];
while (j < n2) arr[k++] = R[j++];
```

These loops, where only one actually executes, will copy any remaining elements from the L or R subarrays. Once all this is done, we delete L and R to deallocate memory.

Then in merge_sort(), we make sure that the left is less than the right, otherwise if left >= right, the array is already sorted. If not, we proceed by first calculating the midpoint. Then, the function recursively calls itself by merge sorting the two halves again and again until it is sorted. Afterwards, the two halves are merged together at the end which results in the final completed sorted array.

Table 8-4. Quick Sort Algorithm

Header:

NOTE: I didn't use the utility library, instead, I defined a basic swap function also within the header file:

```
template<typename T>
void swap(T &a, T &b) {
   T temp = a;
   a = b;
   b = temp;
   swaps++; // count every swap
}
```

```
// QuickSort
// Partition function to place pivot in correct position
template<typename T>
int partition(T arr[], int low, int high) {
   T pivot = arr[high]; // choose last element as pivot
   int i = low - 1;

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

           // increment j if:
           // element > pivot

           // increment i if:
           // element < pivot; swap i and j; j++

           // if j == pivot; i++; swap j to pivot
       }
   }

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

// Recursive quicksort
template<typename T>
void quicksort(T arr[], int low, int high) {
   if (low < high) {
       int pivot = partition(arr, low, high);
       quicksort(arr, low, pivot - 1);
       quicksort(arr, pivot + 1, high);
   }
}
```

main:

```cpp
int main() {

    int arr[maxSize];
    std::cout << "Random Array of 100 elements: \n";
    randomNumberArray100(arr, maxSize);
    printArray(arr, maxSize);

    line();

    std::cout << "QuickSort: \n";
    quicksort(arr, 0, maxSize - 1);
    printArray(arr, maxSize);

    return 0;
}
```

Output

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\HOA 8.1 Sorting Algorithms Pt2\cmake-build-debug\HOA_8_1_Sorting_Algorithm
Random Array of 100 elements:
41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36 91 4 2 53 92 82 21 16 18 95 47 26 71 38 69 12 67 99 35 94 3 11
 22 33 73 64 41 11 53 68 47 44 62 57 37 59 23 41 29 78 16 35 90 42 88 6 40 42 64 48 46 5 90 29 70 50 6 1 93 48 29 23 84
54 56 40 66 76 31 8 44 39 26 23 37 38 18 82 29 41
------------------------------------------------------------
QuickSort:
0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29 29 31 33 34 35 35 36 37 37 38 38 39 40 4
0 41 41 41 41 42 42 42 44 44 45 46 47 47 48 48 50 53 53 54 56 57 58 59 61 62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 7
8 78 81 82 82 84 88 90 90 91 91 92 93 94 95 95 99

Process finished with exit code 0
```

The algorithm consists of two parts: the partition function which keeps track of the position of the pivot point, and the actual quick sorting function.

Beginning with the partition function, it begins by defining the pivot point to start at the last element. It then initializes the position for i and j: i is the element at index -1, and the j is the index of the element to be compared and tested with pivot.
Three conditions are tested:

1. j++ if: element > pivot
2. i++ if: element < pivot; swap i and j; j++
3. if j == pivot; i++; swap j to pivot

This moves the pivot point closer to the middle then ensures that the elements to the left of the pivot are less than it while all elements to the right are greater. This is called partitioning, where two partitions are separated by the pivot point.

Finally, with the position of the pivot point set, we then go to the actual quickSort function to be called. The base case is (low < high) where if the subarray has only one or zero elements, it is already sorted. Then, we declare the position of the pivot point using the partition function created beforehand. It is recursive because it then calls itself twice: to

quickSort the left partition, then to quicksort the right partition, with the mid point being the pivot point to separate the two partitions. Afterwards, the array will then be finally sorted.

NOTE:
The codes above are basic implementations of the algorithms that don't keep track of the comparisons and swaps. In the supplementary activity, I added counters for comparisons and swaps to keep track of them for comparing the algorithms:

To keep the count of comparisons and swaps as is across function calls, I declared them as static. Additionally, I defined them using size_t in the testing since I declared the size of the array using sizeof(), and this prevents mismatch errors if ever. Then, I modified the sorting algorithms to have size++ and comparisons++.

```
static size_t comparisons = 0;
static size_t swaps = 0;

void resetCounters() {
   comparisons = 0;
   swaps = 0;
}
void printCounters() {
   std::cout << "Comparisons: " << comparisons << '\n';
   std::cout << "Swaps: " << swaps << '\n';
}
template<typename T>
void swap(T &a, T &b) {
   T temp = a;
   a = b;
   b = temp;
   swaps++;
}
```

```
// Shell Sort
template<typename T>
void shellSort(T arr[], size_t size) {
   // Start with a big gap, then reduce it
   for (size_t gap = size / 2; gap > 0; gap /= 2) {
       // Perform a gapped insertion sort
       for (size_t i = gap; i < size; i++) {
           T temp = arr[i];
           size_t j = i;

           // Shift earlier gap-sorted elements up until the correct
location is found
           while (j >= gap) {
               comparisons++;
               if (arr[j - gap] > temp) {
                   arr[j] = arr[j - gap];
                   swaps++;
```

```
                    j -= gap;
                } else {
                    break;
                }
            }

            // Put temp (the original arr[i]) in its correct location
            arr[j] = temp;
        }
    }
}
```

```
// Merge Sort
// Merge two sorted subarrays: arr[left..mid] and arr[mid+1..right]
template<typename T>
void merge(T arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temporary arrays
    T* L = new T[n1];
    T* R = new T[n2];

    // Copy data
    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];

    // Merge back
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        comparisons++;
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
            swaps++;
        } else {
            arr[k++] = R[j++];
            swaps++;
        }
    }

    // Copy
    while (i < n1) {
        arr[k++] = L[i++];
        swaps++;
    }
    while (j < n2) {
        arr[k++] = R[j++];
```

```
        swaps++;
    }

    delete[] L;
    delete[] R;
}

// Recursive merge sort
template<typename T>
void merge_sort(T arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
```

```
// QuickSort
// Partition function to place pivot in correct position
template<typename T>
int partition(T arr[], int low, int high) {
    T pivot = arr[high]; // choose last element as pivot
    int i = low - 1;

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

            // increment j if:
            // element > pivot

            // increment i if:
            // element < pivot; swap i and j; j++

            // if j == pivot; i++; swap j to pivot
        }
    }

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

```cpp
// Recursive quicksort
template<typename T>
void quicksort(T arr[], int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
        quicksort(arr, low, pivot - 1);
        quicksort(arr, pivot + 1, high);
    }
}
```

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.

Yes, we can utilize a hybrid approach by combining multiple sorting algorithms into one. This can optimize performance by either performing less comparisons or less swapping, depending on the array.

I created a hybrid quick sort function that uses shell sorting to sort the left and right sub lists of the partition method:

```cpp
template<typename T>
void hybridQuickShellSort(T arr[], int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
        shellSort(arr + low, pivot - low);
        shellSort(arr + pivot + 1, high - pivot);
    }
}
```

In standard quickSorting:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\HOA 8.1 Sorting Algorithms Pt2\cmake-build-debug\HOA_8_1_Sorting_Algorithm
Random Array of 100 elements:
41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36 91 4 2 53 92 82 21 16 18 95 47 26 71 38 69 12 67 99 35 94 3 11
 22 33 73 64 41 11 53 68 47 44 62 57 37 59 23 41 29 78 16 35 90 42 88 6 40 42 64 48 46 5 90 29 70 50 6 1 93 48 29 23 84
54 56 40 66 76 31 8 44 39 26 23 37 38 18 82 29 41
----------------------------------------------------------------
Standard Quicksorting:
0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29 29 31 33 34 35 35 36 37 37 38 38 39 40 4
0 41 41 41 41 42 42 42 44 44 45 46 47 47 48 48 50 53 53 54 56 57 58 59 61 62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 7
8 78 81 82 82 84 88 90 90 91 91 92 93 94 95 95 99
Comparisons: 592
Swaps: 413

Process finished with exit code 0
```

In hybrid quickShellSorting:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\HOA 8.1 Sorting Algorithms Pt2\cmake-build-debug\HOA_8_1_Sorting_Algorithm
Random Array of 100 elements:
41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36 91 4 2 53 92 82 21 16 18 95 47 26 71 38 69 12 67 99 35 94 3 11
 22 33 73 64 41 11 53 68 47 44 62 57 37 59 23 41 29 78 16 35 90 42 88 6 40 42 64 48 46 5 90 29 70 50 6 1 93 48 29 23 84
54 56 40 66 76 31 8 44 39 26 23 37 38 18 82 29 41
------------------------------------------------------------
Hybrid Quicksorting with shellsort:
0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29 29 31 33 34 35 35 36 37 37 38 38 39 40 4
0 41 41 41 41 42 42 42 44 44 45 46 47 47 48 48 50 53 53 54 56 57 58 59 61 62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 7
8 78 81 82 82 84 88 90 90 91 91 92 93 94 95 95 99
Comparisons: 721
Swaps: 304

Process finished with exit code 0
```

	Quick Sort	Hybrid QuickShellSort
Comparisons	592	721
Swaps	413	304

When comparing both outcomes, the hybrid sorting performs more comparisons but much less swapping. Less swapping means that it uses less memory. The increased comparisons, however, is due to the duplicate elements in the array itself, like 11 and 18.

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?

Main:

```cpp
#include <iostream>
#include "sorting_pt2.h"

const int maxSize = 100;

void line() {
   std::cout <<
"------------------------------------------------------------\n";
}
int main() {
```

```
    resetCounters();

    int original[] = {4, 34, 29, 48, 53, 87, 12, 30, 44, 25, 93, 67, 43,
19, 74};
    int size = sizeof(original) / sizeof(original[0]);

    // Copying array for testing
    int arrShell[maxSize], arrMerge[maxSize], arrQuick[maxSize],
arrQuickShell[maxSize];
    for (int i = 0; i < size; i++) {
        arrShell[i] = original[i];
        arrMerge[i] = original[i];
        arrQuick[i] = original[i];
        arrQuickShell[i] = original[i];
    }

    std::cout << "Original Array:\n";
    printArray(original, size);
    line();

    // Shell Sort
    std::cout << "Shell Sort:\n";
    resetCounters();
    shellSort(arrShell, size);
    printArray(arrShell, size);
    printCounters();
    line();

    // Merge Sort
    std::cout << "Merge Sort:\n";
    resetCounters();
    merge_sort(arrMerge, 0, size - 1);
    printArray(arrMerge, size);
    printCounters();
    line();

    // Quicksort
    std::cout << "Quicksort:\n";
    resetCounters();
    quicksort(arrQuick, 0, size - 1);
    printArray(arrQuick, size);
    printCounters();
    line();

    // QuickShellSort
    std::cout << "Quick Shell Sort Hybrid: \n";
    resetCounters();
    hybridQuickShellSort(arrQuickShell, 0, size - 1);
    printArray(arrQuickShell, size);
    printCounters();
```

```
    line();

    return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\HOA 8.1 Sorting Algori
Original Array:
4 34 29 48 53 87 12 30 44 25 93 67 43 19 74
------------------------------------------------------------------
Shell Sort:
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Comparisons: 47
Swaps: 17
------------------------------------------------------------------
Merge Sort:
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Comparisons: 42
Swaps: 59
------------------------------------------------------------------
Quicksort:
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Comparisons: 47
Swaps: 33
------------------------------------------------------------------
Quick Shell Sort Hybrid:
4 12 19 25 29 30 34 43 44 48 53 67 74 87 93
Comparisons: 63
Swaps: 38
------------------------------------------------------------------

Process finished with exit code 0
```

Algorithm	Comparisons	Swaps

Shell Sort	47	17
Merge Sort	42	59
Quicksort	47	33
QuickShellSort Hybrid	63	38

Based on the comparison and swap counts, shell sort performed the least amount of swapping with the second highest amount of comparisons. In this summary, shell sort performed the fastest due to the least amount of swaps performed, meaning it is less memory intensive than the other algorithms. Quicksort is the most balanced with its comparisons to swap ratio.

Why can merge sort and quick sort have O(N • log N) for their time complexity?

They have O(N * logN) for their time complexity because of their divide and conquer strategy. In merge sort, the array is always sorted in half then each half is sorted and merged. This always gives the same time complexity which makes it consistent in terms of performance. Meanwhile, in quick sort, the array is arranged using the partition method where all elements to the left are less than the pivot and all elements to the right are always greater. Afterwards, each side is sorted recursively which still falls under the divide and conquer strategy, (divide into two sides with pivot in the middle, then sort the left and right partitions).

8. Conclusion

In conclusion, we dove into more complex sorting algorithms. If I were to compare, the previous basic comparisons are done in a linear fashion, where the comparisons happen within the original array and all elements are compared to each other. This can be slow and memory intensive, especially with its worst cases. For example, in my previous activity, sorting 100 elements can perform hundreds to thousands of swaps and comparisons. However, with these sets of sorting algorithms, we are able to perform less swaps and comparisons through various strategies, generally by dividing the array into subarrays to be compared. The shell sort, an improvement upon insertion sort, divides an array into several sub arrays, where the elements included are a “gap” number apart. Then, an insertion sort algorithm is applied to these sub arrays all at the same time. This significantly reduces the number of swaps and comparisons. Meanwhile, the merge sort and quicksort algorithms perform a divide and conquer strategy, where the elements are separated from left and right, each sorted, then combined to provide the final sorted array. Merge sort uses constant division of the elements into two to sort the elements until they are sorted and combined together. Meanwhile, the quicksort algorithm divides the elements into two using a pivot value by partitioning the elements, where elements to the left are less than the pivot while the elements to the right are greater than the pivot. To actually sort the elements, it uses a base case of left < right, where if the sub array only has one element, then it is already sorted. Unlike merge sort, partitioning only rearranges the array through partitioning then it is sorted instead of splitting then merging it.

For the supplementary activity, after testing, I learned that combining multiple sorting algorithms can actually be more efficient. In the quickSort algorithm, I can sort the partitions using either shell sort or merge sort and it will still work just as fine, sometimes even better. This is proof that no sorting algorithm is necessarily better or worse than the other, it is a case-by-case basis. Sometimes, the best algorithm is one that combines multiple algorithms to perform the job well.

All in all, I think I understand the gist of these new sorting algorithms. I am able to understand and explain how these algorithms handle the comparisons, shifting, and sorting. In fact, I was able to compare the shell sort algorithm with the basic sorting algorithms with electrical circuits—the basic sorting involves a series circuit where each element is

accessed and compared in a linear fashion, meanwhile the algorithms used in this activity are similar to a parallel circuit, where the sorting is done through subarrays which results in a faster and more efficient algorithm. However, I am still struggling to properly visualize and reproduce the algorithms as actual code without references, so I have more practice to go. Overall, I like this activity as it showcases the different applications and implementations of sorting algorithms.

9. Assessment Rubric