Assignment 8.1	
Using Sorting Algorithms 2	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 09 - 23 - 25
Section: CPE21S4	**Date Submitted:** 09 - 23 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado

6. Output

Objectives: To create a program in C++ using sorting algorithms.

1. To be familiarized with the sorting algorithms.
2. To be able to differentiate the types of sorting algorithms.
3. To be able to create a program with sorting algorithms.

Answer the following questions:

1. Explain the quick sort, shell sort and merge sort types of sorting algorithms.
    - Shell Sort
        - Shell sort improves upon the insertion sort by introducing a new element called the "gap." It begins by calculating the gap which is usually the array size divided by 2. Afterwards, the main array is now divided by whatever the gap is and subsequently inserted into their own subvectors. For example, for an array size 9 and a gap size of 4, every 4th element will be grouped together and so on. Then, the insertion sort will be implemented into each subvector, sorting them. Afterwards, the old gap will be divided by 2 again and the main vector will again be grouped into a gap amount of subvectors. Then, the insertion sort is implemented until we do a final insertion sort on the whole array where the gap size is 1. This will be the final insertion and afterwards, the array will finally be sorted.
    - Merge Sort
        - The merge sort is a recursive algorithm that uses a divide and conquer strategy by continuously splitting a vector in half. If the vector is empty or has one time, it will be considered sorted already. Otherwise, we recursively invoke a merge sort on both halves (left and right) and sort them both. After sorting both halves, we merge them then the larger vector is compared and sorted to the other larger vector. This process continues until, again, the subvector's size becomes one or none. The main merge sort

function, mergeSort(), takes three arguments: The left element(s), the right element(s), then the original array and sorts it accordingly.

- Quick Sort
    - The quick sort involves four variables: the i, j, pivot, and the temp variable. It works under three conditions:
        a. j++ if j > pivot
        b. i++ if j < pivot, then swap(i, j), then j++
        c. if j == pivot, i++ then swap(i, j)
    - It begins by initializing the pivot, usually being the last element at index arrSize-1. Then, i begins at index -1, and j will start at index 0. Per iteration, a comparison between j and the pivot and does whatever the three conditions are satisfied. Whenever the 3rd condition is satisfied, the swapped i will become the “split point” and divide the array by two (left and right hand side). Noticeably, all values to the left of the split point are less than it while all the values at the right are greater than it. Then, we repeat the quicksort algorithm by starting at the left and resetting the values and indexes of i and j , with our new pivot being the index [split point -1]. We continue doing the quicksort algorithm until we sort it. In this case, if we can no longer compare the element then it means it's already sorted. Lastly, we continue the process at the right side, the pivot being the last element, and the i being the index of split point and j being the index of [split point + 1]. We continue the quick sorting algorithm until we’re finally met with the sorted array.

2. Give simple sample programs in C++ that use the above sorting algorithms. Use a user input of 10 integer values in your example elements in an array to be sorted. Explain how the programs work.

Shell Sort

```cpp
#include <bits/stdc++.h>
#include <vector>

// Shell Sort
/* function to sort arr using shellSort */
int shellSort(int arr[], int n)
{
   // Start with a big gap, then reduce the gap
   for (int gap = n/2; gap > 0; gap /= 2)
   {
       // Do a gapped insertion sort for this gap size.
       // The first gap elements a[0..gap-1] are already in gapped order
       // keep adding one more element until the entire array is
       // gap sorted
       for (int i = gap; i < n; i += 1)
       {
           // add a[i] to the elements that have been gap sorted
           // save a[i] in temp and make a hole at position i
           int temp = arr[i];

           // shift earlier gap-sorted elements up until the correct
           // location for a[i] is found
           int j;
           for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
               arr[j] = arr[j - gap];

           //  put temp (the original a[i]) in its correct location
           arr[j] = temp;
       }
   }
   return 0;
}

int main() {
   int arr[] = {21, 36, 74, -2, 3}, i;
   int n = sizeof(arr)/sizeof(arr[0]);

   std::cout << "Array before sorting: \n";
   printArray(arr, n);

   shellSort(arr, n);

   std::cout << "\nArray after sorting: \n";
   printArray(arr, n);

   return 0;
}
```

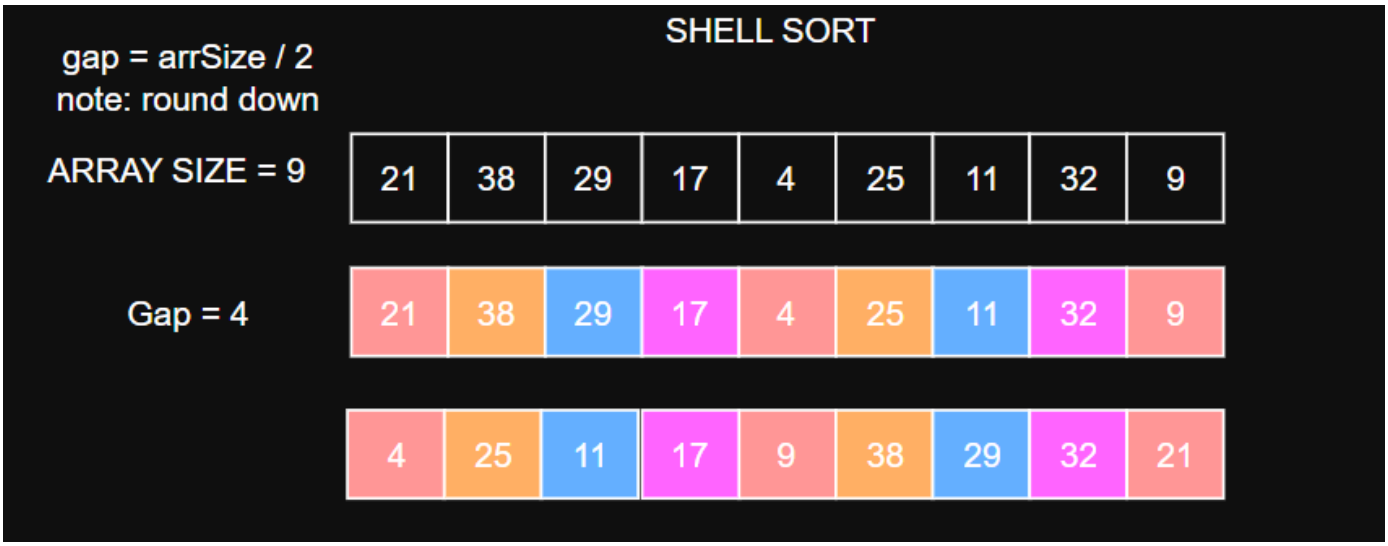
Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\untitled\cmake-build-debug\untitled.exe"
Array before sorting:
21 36 74 -2 3

Array after sorting:
-2 3 21 36 74

Process finished with exit code 0
```

Shell sort begins by taking two parameters: the array and the size. The outer loop and main iteration condition is the gap in this line: for (int gap = n/2; gap > 0; gap /= 2). The logic begins where the gap is the size divided by 2, and while the gap is positive, it keeps dividing it by 2 as its increment logic. Then, for the inner loop: for (int i = gap; i < n; i += 1), it implements the insertion sort logic. The i is equal to the gap, which means for all elements at the gap point, it will be compared and sorted by the sorting algorithm within.



In this array, all of the elements at gap = 4, so all elements at gap position apart will have the insertion algorithm implemented. This continues until the gap becomes 1, where the final insertion sort happens in the almost sorted array.

```
int temp = arr[i];
int j;
for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
   arr[j] = arr[j - gap];
```

This part of the algorithm shifts all the elements to the right that are greater than the key then inserts the element (which is stored in the temp variable) once the key is greater than the element to the left of it.

Merge Sort

```cpp
#include <bits/stdc++.h>
#include <vector>

// Merge Sort
void merge(std::vector<int>& vec, int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary vectors
    std::vector<int> leftVec(n1), rightVec(n2);

    // Copy data to temporary vectors
    for (i = 0; i < n1; i++)
        leftVec[i] = vec[left + i];
    for (j = 0; j < n2; j++)
        rightVec[j] = vec[mid + 1 + j];

    // Merge the temporary vectors back into vec[left..right]
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (leftVec[i] <= rightVec[j]) {
            vec[k] = leftVec[i];
            i++;
        } else {
            vec[k] = rightVec[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of leftVec[], if any
    while (i < n1) {
        vec[k] = leftVec[i];
        i++;
        k++;
    }

    // Copy the remaining elements of rightVec[], if any
    while (j < n2) {
        vec[k] = rightVec[j];
        j++;
        k++;
    }
}

// The subarray to be sorted is in the index range [left..right]
void mergeSort(std::vector<int>& vec, int left, int right) {
    if (left < right) {

        // Calculate the midpoint
        int mid = left + (right - left) / 2;

        // Sort first and second halves
```

```
        mergeSort(vec, left, mid);
        mergeSort(vec, mid + 1, right);

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

int main() {

    std::vector<int> vec = {12, 11, 13, 5, 6, 7};
    int n = vec.size();

    std::cout << "Unsorted: \n";
    for (auto i: vec)
        std::cout << i << " ";
    std::cout << '\n';

    // Sorting vec using mergesort
    mergeSort(vec, 0, n - 1);

    std::cout << "Sorted: \n";
    for (auto i: vec)
        std::cout << i << " ";
    std::cout << '\n';

    return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\untitled\cmake-build-debug\untitled.exe"
Unsorted:
12 11 13 5 6 7
Sorted:
5 6 7 11 12 13

Process finished with exit code 0
```

Merge sort begins with the merge function, where it creates two temporary vectors to store the left and right half. It splits the entire vector into two halves using the temporary vectors by copying the data using two for loops:

```
for (i = 0; i < n1; i++)
      leftVec[i] = vec[left + i];
   for (j = 0; j < n2; j++)
      rightVec[j] = vec[mid + 1 + j];
```

Merging the two vectors is implemented using a while loop and merges the left vectors and the right vectors. While merging, it already compares the two elements hence the returned merged vector will be sorted already

```
i = 0;
j = 0;
k = left;
while (i < n1 && j < n2) {
    if (leftVec[i] <= rightVec[j]) {
        vec[k] = leftVec[i];
        i++;
    } else {
        vec[k] = rightVec[j];
        j++;
    }
    k++;
}
```

For the actual mergeSort() function,

```
if (left < right) {

    // Calculate the midpoint
    int mid = left + (right - left) / 2;

    // Sort first and second halves
    mergeSort(vec, left, mid);
    mergeSort(vec, mid + 1, right);

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

If left is less than right, so once the merging is done, it will then mergesort both halves of the original vector. It is recursive, so there are no loops involved. It continuously invokes the mergeSort function to keep sorting the left and right halves of the temporary vectors until the final two vectors are properly sorted and the left are all < than right. In this case, the merging finally happens where the two halves are recombined into the completed and sorted vector.

Quick Sort

```
// C++ Program to demonstrate how to implement the quick
// sort algorithm
#include <iostream>
#include <vector>

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

int partition(std::vector<int> &vec, int low, int high) {

    // Selecting last element as the pivot
    int pivot = vec[high];

    // Index of element just before the last element
    // It is used for swapping
    int i = (low - 1);
```

```
    for (int j = low; j <= high - 1; j++) {

        // If current element is smaller than or
        // equal to pivot
        if (vec[j] <= pivot) {
            i++;
            swap(vec[i], vec[j]);
        }
    }

    // Put pivot to its position
    swap(vec[i + 1], vec[high]);

    // Return the point of partition
    return (i + 1);
}

void quickSort(std::vector<int> &vec, int low, int high) {

    // Base case: This part will be executed till the starting
    // index low is lesser than the ending index high
    if (low < high) {

        // pi is Partitioning Index, arr[p] is now at
        // right place
        int pi = partition(vec, low, high);

        // Separately sort elements before and after the
        // Partition Index pi
        quickSort(vec, low, pi - 1);
        quickSort(vec, pi + 1, high);
    }
}

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

    std::cout << "Unsorted: \n";
    for (int i = 0; i < n; i++) {
        std::cout << vec[i] << " ";
    }std::cout << '\n';

    // Calling quicksort for the vector vec
    quickSort(vec, 0, n - 1);

    std::cout << "Unsorted using quicksort: \n";
    for (auto i : vec) {
        std::cout << i << " ";
    }
    return 0;
}
```

Output:

```
"D:\College\Year 2 CPE\Y2 Sem 1\CPE 010\Midterm\untitled\cmake-build-debug\untitled.exe"
Unsorted:
10 7 8 9 1 5
Unsorted using quicksort:
1 5 7 8 9 10
Process finished with exit code 0
```

The sorting algorithm begins with the partition function that declares the pivot which begins at the last element: int pivot = vec[high]. Then, i is initialized to be the low -1 so before the first element. Then it begins with this conditional for loop:

```
for (int j = low; j <= high - 1; j++) {

   // If current element is smaller than or
   // equal to pivot
   if (vec[j] <= pivot) {
       i++;
       swap(vec[i], vec[j]);
   }
}
```

While j is less than or equal to high, and if j is less than or equal to the pivot, then we increment i and swap i and j. Otherwise, we just increment j. Then, swap(vec[i + 1], vec[high]); which puts the pivot to its position. For the actual quickSort() function, similar to the merge sort, if the lower half is less than the right half, we call the pi to be the position of the partition or splitting point: int pi = partition(vec, low, high);. We then recursively call itself to sort both halves:

```
quickSort(vec, low, pi - 1);
quickSort(vec, pi + 1, high);
```

Once it's done, the final vector will finally be sorted.

References:

GeeksforGeeks. (2025d, September 23). *Merge sort*. GeeksforGeeks.

https://www.geeksforgeeks.org/dsa/merge-sort/

GeeksforGeeks. (2025e, August 8). *Quick sort*. GeeksforGeeks.

https://www.geeksforgeeks.org/dsa/quick-sort-algorithm/

GeeksforGeeks. (2025d, July 23). *Shell Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/shell-sort/

Shell Sort (With code in Python, C++, Java and C). (n.d.). https://www.programiz.com/dsa/shell-sort

W3Schools.com. (n.d.). https://www.w3schools.com/dsa/dsa_algo_mergesort.php

Source codes retrieved and analyzed from:

GeeksforGeeks. (2025a, July 23). *C++ program for merge sort*. GeeksforGeeks.

https://www.geeksforgeeks.org/cpp/cpp-program-for-merge-sort/

GeeksforGeeks. (2025, July 23). *C++ program for quick sort*. GeeksforGeeks.

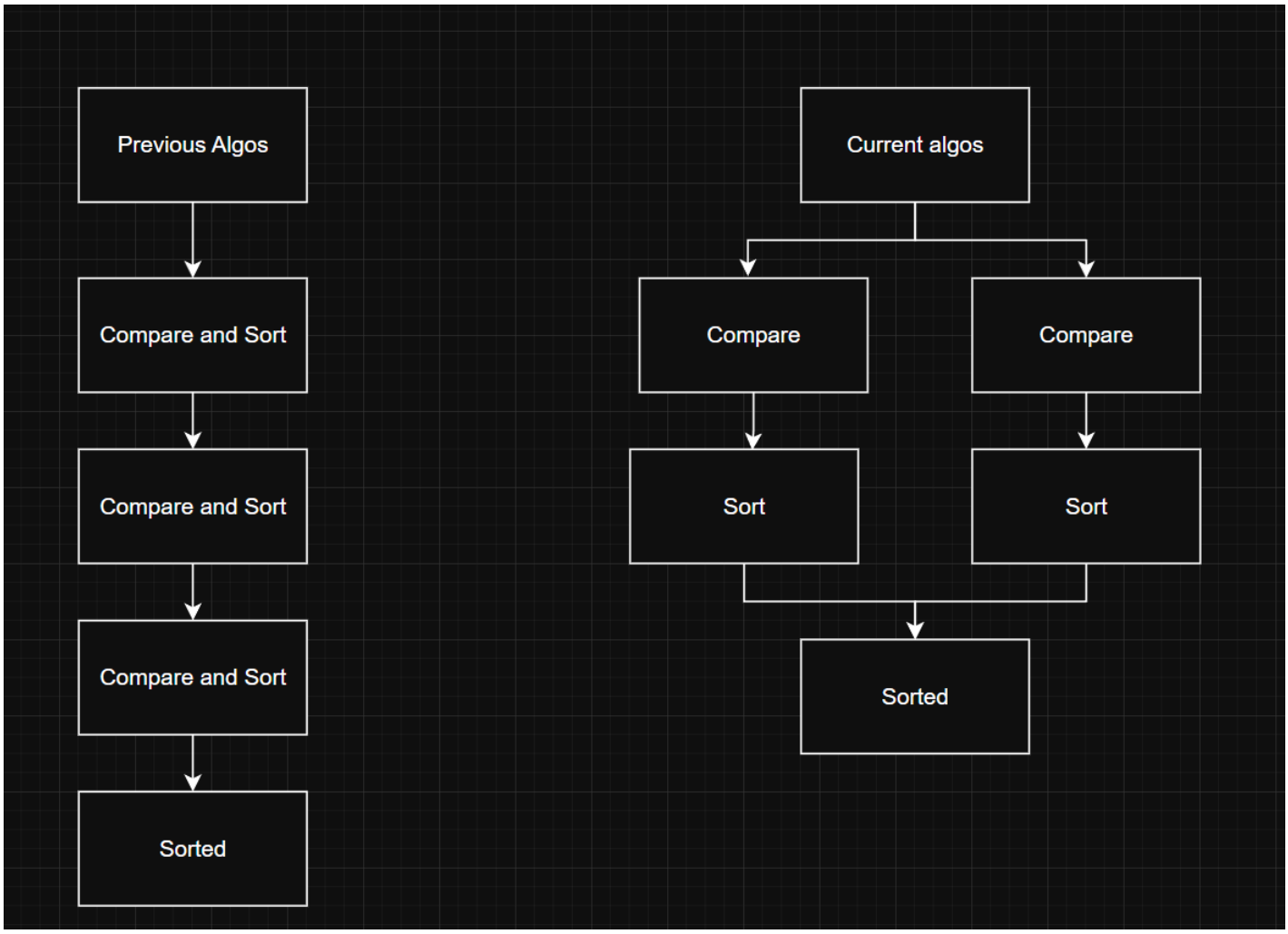
https://www.geeksforgeeks.org/cpp/cpp-program-for-quicksort/

GeeksforGeeks. (2025b, July 23). *Shell Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/shell-sort/

7. Conclusion

This activity explores the more efficient and optimized sorting algorithms, namely: the shell sort, merge short, and quicksort. First, the shell sort improves upon the insertion sort by performing the insertion sort at multiple subgroups of a vector at the same time which is achieved using the gap variable. All elements that are gap positions away from each other will be grouped together then the insertion sort will be implemented simultaneously. The gap continuously gets smaller because it's halved every iteration until the final vector is insertion sorted. Second, the merge short utilizes a divide and conquer strategy by subdividing vectors first then each merge sorts them accordingly. It begins by storing the left and right halves of the original vector into two temporary vectors then continuously splits them and merges, which consequently merges them. This occurs until we go back to the left half and right half of the original vector and they are merged, resulting in a fully sorted vector. Lastly, the quickSort utilizes partitions and pivots. The partition is the process of finding the pivot point or the split point to be able to sort the array where all elements to the left are less than the pivot and all elements to the right are greater than it. It begins by declaring the pivot to be the last element and incrementing i and j and swapping them with the pivot. The i begins at index -1 and j starts as the first element and it's also the element to always be compared to the pivot. Once the partition finishes, we finally get the array with the pivot point at the middle. We first quickSort the left half then the right half, which gives us our final sorted array. All of these algorithms are

much faster than the basic ones due to the comparisons being divided and executed at the same time. If I were to compare it, the previous algorithms always sort the entire array in a linear or series fashion, but these algorithms split the line into a specific number of subdivisions where the steps are now in parallel. This reduces the amount of comparisons which make it faster and generally safer and more efficient.



8. Assessment Rubric