

## Assignment 8.1

### Using Sorting Algorithms 2

**Course Code:** CPE010

**Program:** Computer Engineering

**Course Title:** Data Structures and Algorithms

**Date Performed:** 9/23/25

**Section:** CPE21S4

**Date Submitted:** 9/23/25

**Name(s):** Avila, Vince Gabriel V.

**Instructor:** Engr. Jimlord Quejado

#### 6. Output

##### 1. Explain the quick sort, shell sort and merge sort types of sorting algorithms.

###### Quick Sort

Quick Sort is a method of sorting where numbers are selected to be ordered around one number, the pivot that is smaller on its left and greater on its right. It carries on like this with the smaller groups until all is well. It generally runs very quickly, but may run slowly if the pivot is poorly chosen. It does not require additional space in which to operate. But it could rearrange the ordering of equal numbers.

---

###### Shell Sort

Shell Sort is a generalized version of insertion sort. It begins by comparing numbers that are widely separated; then progressively compares numbers that are closer together. This helps place the right numbers in the correct order more quickly. It is good for medium sized lists and it does not require additional memory. But it's not always the fastest or the most accurate.

---

###### Merge Sort

With Merge Sort it takes the list, breaks it into smallest pieces and sorts them individually, then reassembles in correct order. The time it will take to sort the list is the same, even if the list was only half sorted. It uses additional space to do the merging but it is very reliable and preserves the order of the same elements. It makes sense when accurate and stable is the goal.

##### 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.

**Quick Sort:**

main.cpp

```
1 // This program is to demonstrate how to implement the quick
2 // sort algorithm
3 #include <bits/stdc++.h>
4 using namespace std;
5
6 int partition(vector<int> &vec, int low, int high) {
7
8     // Selecting last element as the pivot
9     int pivot = vec[high];
10
11     // Index of element just before the last element
12     // It is used for swapping
13     int i = (low - 1);
14
15     for (int j = low; j <= high - 1; j++) {
16
17         // If current element is smaller than or
18         // equal to pivot
19         if (vec[j] <= pivot) {
20             i++;
21             swap(vec[i], vec[j]);
22         }
23     }
24
25     // Put pivot to its position
26     swap(vec[i + 1], vec[high]);
27
28     // Return the point of partition
29     return (i + 1);
30 }
31
32 void quickSort(vector<int> &vec, int low, int high) {
33
34     // Base case: This part will be executed till the starting
35     // index low is lesser than the ending index high
36     if (low < high) {
37
38         // pi is Partitioning Index, arr[p] is now at
39         // right place
40         int pi = partition(vec, low, high);
41
42         // Separately sort elements before and after the
43         // Partition Index pi
44         quickSort(vec, low, pi - 1);
45         quickSort(vec, pi + 1, high);
46     }
```

```
46 }
47 }
48
49 int main() {
50     vector<int> vec = {10, 7, 8, 9, 1, 5};
51     int n = vec.size();
52
53     // Calling quicksort for the vector vec
54     quickSort(vec, 0, n - 1);
55
56     for (auto i : vec) {
57         cout << i << " ";
58     }
59     return 0;
60 }
```

## Output

10 27 38 43

=== Code Execution Successful ===

## Analysis:

This C++ code sorts a list of numbers in ascending order using the Quick Sort algorithm. It grabs the last number in the list as a pivot, then organizes the whole list so all the numbers less than that are to its left and bigger ones to right. This process is called partitioning. Then it does the same thing again for the left and right parts of the list using recursion so that ultimately all is sorted. Quick Sort is typically very fast and will not require extra memory to perform. Finally, the program prints the sorted numbers.

## Shell Sort:

```
main.cpp
1 // C++ implementation of Shell Sort
2 #include <iostream>
3 using namespace std;
4
5 /* function to sort arr using shellSort */
6 int shellSort(int arr[], int n)
7 {
8     // Start with a big gap, then reduce the gap
9     for (int gap = n/2; gap > 0; gap /= 2)
10     {
11         // Do a gapped insertion sort for this gap size.
12         // The first gap elements a[0..gap-1] are already in gapped order
13         // keep adding one more element until the entire array is
14         // gap sorted
15         for (int i = gap; i < n; i += 1)
16         {
17             // add a[i] to the elements that have been gap sorted
18             // save a[i] in temp and make a hole at position i
19             int temp = arr[i];
20
21             // shift earlier gap-sorted elements up until the correct
22             // location for a[i] is found
23             int j;
24             for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
25                 arr[j] = arr[j - gap];
26
27             // put temp (the original a[i]) in its correct location
28             arr[j] = temp;
29         }
30     }
31     return 0;
32 }
```

### Output

Array before sorting:

12 34 54 2 3

Array after sorting:

2 3 12 34 54

=== Code Execution Successful ===

## Analysis:

This Shell Sort numbers program begins by comparing the second and sixth numbers which are 1 and continues on. It gradually narrows the search space of numbers it will compare until it tests adjacent numbers. This assists in getting numbers to their correct position faster than normal sort. The example below will print the list both before and after sorting. And at last, print out all the sorted numbers. Shell Sort is much faster than simple sort like insertion sort especially for large input data.

## Merge Sort:

```

main.cpp
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 // Merges two subarrays of arr[].
6 // First subarray is arr[left..mid]
7 // Second subarray is arr[mid+1..right]
8 void merge(vector<int>& arr, int left,
9            int mid, int right){
10
11     int n1 = mid - left + 1;
12     int n2 = right - mid;
13
14     // Create temp vectors
15     vector<int> L(n1), R(n2);
16
17     // Copy data to temp vectors L[] and R[]
18     for (int i = 0; i < n1; i++)
19         L[i] = arr[left + i];
20     for (int j = 0; j < n2; j++)
21         R[j] = arr[mid + 1 + j];
22
23     int i = 0, j = 0;
24     int k = left;
25
26     // Merge the temp vectors back
27     // into arr[left..right]
28     while (i < n1 && j < n2) {
29         if (L[i] <= R[j]) {
30             arr[k] = L[i];
31             i++;
32         }
33         else {
34             arr[k] = R[j];
35             j++;
36         }
37         k++;
38     }
39
40     // Copy the remaining elements of L[],
41     // if there are any
42     while (i < n1) {
43         arr[k] = L[i];
44         i++;
45         k++;
46     }
47
48     // Copy the remaining elements of R[],
49     // if there are any
50     while (j < n2) {
51         arr[k] = R[j];
52         j++;
53         k++;
54     }
55 }
56
57 // begin is for left index and end is right index
58 // of the sub-array of arr to be sorted
59 void mergeSort(vector<int>& arr, int left, int right){
60
61     if (left >= right)
62         return;
63
64     int mid = left + (right - left) / 2;
65     mergeSort(arr, left, mid);
66     mergeSort(arr, mid + 1, right);
67     merge(arr, left, mid, right);
68 }
69
70 // Driver code
71 int main(){
72
73     vector<int> arr = {38, 27, 43, 10};
74     int n = arr.size();
75
76     mergeSort(arr, 0, n - 1);
77     for (int i = 0; i < arr.size(); i++)
78         cout << arr[i] << " ";
79     cout << endl;
80
81     return 0;
82 }

```

## Output

10 27 38 43

=== Code Execution Successful ===

## Analysis:

This code is an implementation of Merge Sort algorithm in Kotlin, where the list is divided into smaller parts. It continues to split until all the parts contain only a single number. Then, it reassembles those parts in order with a merge step. The merge operation merges two ordered sequences into a single ordered sequence. At the end of the program, it prints the sorted numbers. Merge Sort isn't subject to surprises, and it does not depend on the order of records in memory or even on RAM size since splitting and merging in Merge Sort is always done with caution.

## References:

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>

GeeksforGeeks. (2025c, September 23). *Merge sort*. GeeksforGeeks. <https://www.geeksforgeeks.org/dsa/merge-sort>

<b>8. Conclusion</b>
<p>These programs show three ways of sorting numbers: Quick Sort, Shell Sort, and Merge Sort. Quick Sort picks a number that is the middle one, and then sorts the list very fast. Shell Sort compares the numbers far apart from each other, which makes sorting faster than simple sorts. Merge Sort breaks the list into small parts then puts the parts back together in order. Each way works in different ways but they all help to put the numbers in the right order.</p>
<b>9. Assessment Rubric</b>