# Sorting Algorithms Assignment Interview Notes

## Sorting Algorithms Overview

### General Overview

The purpose of sorting algorithms is to sort elements for a specific data structure in a specified order. In the context of this assignment, the sorting algorithms are sorting values in ascending order. One can modify these algorithms such that they would sort the elements in descending order. Sorting algorithms, with correct implementation, could sort elements such as strings in alphabetical order. Overall, sorting algorithms sort elements for a given data structure in a specified order.

### Different Types of Sorting Algorithms

There are multiple different sorting algorithms that are used in this course. For this assignment, the sorting algorithms that were used in this assignment were:

- Bubble Sort
- Merge Sort
- Myserty Sort - Selection Sort
- Quick Sort

In total, the sorting algorithms that we have gone over in this course are the following:

- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Radix Sort
- Selection Sort
- Shell Sort

In the context of the aforementioned sorting algorithms, the best and worst case time complexity of each algorithm is:

| **Algorithm** | **Best Case** $\Omega(n)$ | **Average Case** $\Theta(n)$ | **Worst Case** $\mathcal{O}(n)$ |
|---|---|---|---|
| Bubble Sort | $\Omega(n)$ | $\Theta(n^2)$ | $\mathcal{O}(n^2)$ |
| Insertion Sort | $\Omega(n)$ | $\Theta(n^2)$ | $\mathcal{O}(n^2)$ |
| Merge Sort | $\Omega(n\log(n))$ | $\Theta(n\log(n))$ | $\mathcal{O}(n\log(n))$ |
| Quick Sort | $\Omega(n\log(n))$ | $\Theta(n\log(n))$ | $\mathcal{O}(n^2)$ |
| Radix Sort | $\Omega(n)$ | $\Theta(d(n+b))$ | $\mathcal{O}(d^2(n+b))$ |
| Selection Sort | $\Omega(n)$ | $\Theta(n^2)$ | $\mathcal{O}(n^2)$ |
| Shell Sort | $\Omega(n\log(n))$ | $\Theta(n^{3/2})$ | $\mathcal{O}(n^2)$ |

When using sorting algorithms, there are some inputs that will cause the algorithm to perform at its worst case run time. We call this situation the, "worst case data sequence". The worst case data sequences for these algorithms are:

| **Algorithm** | **Worst Data Sequence** |
|---|---|
| Bubble Sort | List of numbers in reverse order |
| Insertion Sort | List of numbers in reverse order |
| Merge Sort | Any list of number |
| Quick Sort | List of numbers in reverse order |
| Radix Sort | List of numbers with all different number of digits |
| Selection Sort | List of numbers in reverse order |
| Shell Sort | List of numbers in reverse order |

## Quick Summary of All Sorting Algorithms

In the context of this course, the sorting algorithms that are mentioned above all achieve the same outcome but they do so in a manner that are different than the others. For the sorting algorithms in this course, here is a brief summary of each algorithm and how it operates

- **Bubble Sort** - Bubble sort is a simple sorting algorithm that repeatedly compares adjacent elements in an array and swaps them if they are in the wrong order.
- **Insertion Sort** - Insertion sort is a simple sorting algorithm that inserts elements into a sorted subarray one at a time.
- **Merge Sort** - Merge sort is a divide-and-conquer sorting algorithm that recursively divides the array into smaller subarrays and then merges the sorted subarrays back together.
- **Quick Sort** - Quick sort is a divide-and-conquer sorting algorithm that recursively partitions the array and then sorts the two resulting subarrays.
- **Radix Sort** - Radix sort is a non-comparitive sorting algorithm that sorts elements by processing them digit by digit.
- **Selection Sort** - Selection sort is a simple sorting algorithm that repeatedly finds the smallest (or largest) element in the unsorted portion of the array and moves it to the sorted portion of the array.
- **Shell Sort** - Shell sort is a sorting algorithm that improves upon the performance of insertion sort by sorting elements that are far apart first, then gradually reducing the gap between elements to be compared.

In summary, the above sorting algorithms will all achieve the same end goal of sorting a list of elements. Choosing a sorting algorithm is largely predicated on what is needed most. If you are sorting a large set of data then you more than likely want to select a sorting algorithm that will have the best worst case runtime. If you are not stressed for computational speed then one can use a simpler sorting algorithm. The algorithm that one chooses to use is largely dependent upon the data set that is being run through the algorithm.

## Real World Applications of Sorting Algorithms

Sorting algorithms can be used for a wide variety of situations in the real world. Some examples of where these algorithms can be used are the following:

- Sorting contact lists
- Sorting files
- Sorting search results
- Sorting data in databases
- Sorting images
- Sorting medical records
- Sorting financial data
- Sorting traffic data

Sorting algorithms have a wide range of versatile uses that can be used in tandem with search algorithms as well. The above are only some utilization's of sorting algorithms.

## Sorting Versus Searching

In the context of sorting algorithms, one may ask what is the difference between a sorting algorithm and a search algorithm. In the simplest of terms, sorting algorithms are used to arrange a collection of data in a specific order, such as a ascending or descending. Search algorithms on the other hand are used to find a specific item in a collection of data. These algorithms have a range of purposes, complexities, and variations. We can summarize these topics with the following:

| **Feature** | **Sorting Algorithm** | **Searching Algorithm** |
|---|---|---|
| Examples | Bubble Sort, Merge Sort, Quick Sort | Binary Search, Hash Table Search, Linear Search |
| Purpose | Arrange a collection of data in a specific order | Find a specific item in a collection of data |
| Space Complexity | $\mathcal{O}(1)$ to $\mathcal{O}(n)$ | $\mathcal{O}(1)$ |
| Time Complexity | $\mathcal{O}(n\log(n))$ to $\mathcal{O}(n^2)$ | $\mathcal{O}(\log(n))$ to $\mathcal{O}(n)$ |

In summary, sorting algorithms sort data in data sets whereas searching algorithms find if data exists in data sets.

## Bubble Sort Algorithm

### Overview

The bubble sort is a simple sorting algorithm that works by repeatedly comparing adjacent elements in an array and swapping them if they are in the wrong order. The algorithm starts at the beginning of the array and compares the first two elements. If the first element is greater than the second element, they are swapped. The algorithm then moves on to the next two elements and repeats the process. This continues until the end of the array is reached. At this point, the largest element in the array will be in the last position. The algorithm then starts again at the beginning of the array and repeats the process. This time, the second largest element will be moved to the second-to-last position, and so on. The algorithm continues until all of the elements in the array are sorted in ascending order.

**Bubble Sort Algorithm**

Below is the bubble sort algorithm in the context of C++:

```
/*  bubblesort - Algorithm that sorts elements utilizing the bubble sort algorithm
*   Input:
*     data - Vector of integers that is passed by reference that is to be sorted
*   Algorithm:
*     * Begin by iterating through the vector "data"
*     * Compare adjacent elements of the vector at the current index
*     * Check to see if the next element in the vector is greater than that of the current element
*       * If it is, swap them, otherwise, move on to the next element
*   Output:
*     This function does not return a value, it sorts the elements in "data"
*     using a bubble sort algorithm
*/
void Sorting::bubblesort(vector<int>& data){
// Iterate through the vector "data"
for (int i = 0; i < data.size() - 1; i++) {
    // Compare adjacent elements in the vector
    for (int j = 0; j < data.size() - i - 1; j++) {
    // Check to see if next element in vector is greater than that of the current element
    if (data.at(j) > data.at(j+1)) {
        // Create a temporary integer value for that of the current element
        int temp_val = data.at(j);
        // Assign the current element with that of the next element
        data.at(j) = data.at(j+1);
        // Assign the next element with that of the current element
        data.at(j+1) = temp_val;
    }
    else {}
    }
}
}
```

The bubble sort algorithm is a relatively inefficient sorting algorithm when discussing runtime complexity. The runtime complexity of the bubble sort algorithm is as follows:

| **Best Case** $\Omega(n)$ | **Average Case** $\Theta(n)$ | **Worst Case** $\mathcal{O}(n)$ |
|---|---|---|
| $\Omega(n)$ | $\Theta(n^2)$ | $\mathcal{O}(n^2)$ |

At its best, the bubble sort algorithm will run at a linear runtime complexity. In the average and worst case, the bubble sort algorithm will run at a quadratic runtime complexity. These runtime complexities are not considered great, but the given the simplicity of the algorithm it is rather efficient.

### Optimization & Utilization

Although the bubble sort is not efficient as it is written above, there are ways to optimize the bubble sort algorithm. The two main ways that a bubble sort can be optimized are as follows:

- **Early Termination -** Early termination means that a loop does not execute due to a specific condition. One way early termination can occur is by keeping track if a swap has happened in a recent pass through of the data set. If no swap has occurred, then this means that the list is already sorted and the loop does not need to execute. This is possibly the simplest way one can optimize the bubble sort algorithm.

- **Variable Loop Length -** Variable loop length means that a loop will only run up to the point of the last known element that was swapped. This is done by keeping track of the elements that have been swapped and instructing one of the loops to only run up to the given index of that element that has been swapped. This is a cheeky way of optimizing the bubble sort algorithm than can cut down on the total time that the algorithm runs.

Even with these optimizations, the best runtime complexity of the bubble sort algorithm is still $\Omega(n)$, a linear runtime complexity.

The bubble sort algorithm can be used for a variety of different data structures. In the context of this assignment, we used the bubble sort algorithm to sort a vector. It can just as easily be used on an array. The main data structures that a bubble sort algorithm can be used with are the following:

- **Arrays & Vectors -** Arrays and vectors are the most common data structures that can be utilized with a bubble sort algorithm. Because of their easy nature in terms of data manipulation, arrays and vectors are prime candidates for utilizing a bubble sort algorithm.

- **Linked Lists -** Linked lists can be sorted with the use of the bubble sort algorithm as well. In order to use the bubble sort algorithm, one has to perform other operations in the context linked list manipulation for the algorithm to work efficiently.

- **Strings -** Strings can be sorted with a bubble sort algorithm due to the fact that a string can be represented as an array of characters.

Other data structures that can be used in a bubble sort are queues and stacks. The runtime complexity of these other data structures is still pretty poor, but nonetheless, the use of it is still possible.

## Quick Sort Algorithm

### Overview

The quick sort algorithm is a divide-and-conquer algorithm for sorting a data set. This algorithm works by repeatedly partitioning the data set around a pivot element, and then recursively sorting the two sub-arrays created by the partition. The pivot element that is chosen for this algorithm is typically the middle element of the array, but it is not necessarily required for the algorithm to work. When the partitioning of elements occurs, elements that are smaller than the pivot are moved to the left of the pivot. On the contrary, the elements that are larger than the pivot move to the right of the pivot. Once this partition is completed, the two sub-arrays are recursively sorted. The quick sort algorithm is often regarded as a very efficient sorting algorithm and works well on large data sets.

The quick sort algorithm requires a separate algorithm than the main algorithm for it to operate correctly. The secondary algorithm that is required is the ‘quicksort_partition’ algorithm. The primary role of this algorithm is to iterate through the vector and partition it so that elements can be sorted in a manner that is appropriate. This algorithm can be seen below.

**Quick Sort Partition Algorithm**

Below is the quick sort partition algorithm in the context of C++:

```cpp
/*  quicksort_partition - Determines the pivot point inside a given vector and returns the
*   updated highest index value
*   Input:
*     data - Vector of integers that is passed by reference where the pivot index is being
*     found
*     low_idx - Integer value that represents the lower index of the vector for where we
*     will search through
*     high_idx - Integer value that represents the higher index of the vector for where we
*     will search through
*   Algorithm:
*     * Find the middle index of "data" and assign it to "mid_idx"
*     * Find the pivot value by looking at the value in the "data" vector at the middle index
*       "mid_idx"
*     * Traverse through the vector by incrementing "low_idx" until a value is greater than or
*       equal to that of the pivot
*     * Traverse through the vector by decrementing "high_idx" until a value is less than or
*       equal to that of the pivot
*     * If the lower index value "low_idx" is greater than or equal to that of the higher index
*       "high_idx", then we exit the while loop
*     * If the lower index is less than the higher index, then we do the following:
*       * Create a temporary integer "temp_val" for the value in "data" at the "low_idx" element
*         location
*       * Swap the value that is found on the left side of the pivot with the value that is on
*         the right side of the pivot
*       * Assign the value "temp_val" to that of the value at "high_idx" in the "data" array
*       * Increment the lower index by one and decrement the higher index by one
*   Output:
*     high_idx - Integer value that represents the updated higher index value of our vector that
*     we are searching through
*/
int Sorting::quicksort_partition(vector<int>& data, int low_idx, int high_idx){
    // Define variables to track elements in the "data" vector
    int mid_idx = low_idx + (high_idx - low_idx) / 2;
    int pivot = data.at(mid_idx);
    bool finished = false;
    // Begin traversing through the vector
    while (!finished) {
    // Increment lower index until value greater than or equal to that of the pivot is found
    while (data.at(low_idx) < pivot) {
        low_idx += 1;
    }
    // Dectrement hihger index until value less than or equal to that of the pivot is found
    while (pivot < data.at(high_idx)) {
        high_idx -= 1;
    }
    // If no elements remain, then exit the loop
    if (low_idx >= high_idx) {
        finished = true;
    }
    // Begin the process of swapping values
    else {
        // Keep track of the current value at the lower index
        int temp_val = data.at(low_idx);
        // Swap the values of lower and higher indexes
        data.at(low_idx) = data.at(high_idx);
        data.at(high_idx) = temp_val;
        // Increment and decrement index values
        low_idx += 1;
        high_idx -= 1;
        }
    }
    return high_idx;
}
```

The general method for how this algorithm works is the following:

- The function calculates the middle index of the 'data' vector
- This function then sets the pivot element at the element of the middle index
- We then iterate through the the data set until we find a value on the left side of the pivot that is either greater than or equal to that of the pivot value
- After this, we iterate through the data set until we find a value on the right side of the pivot that is less than or equal to that of the pivot value
- We then check to see if the two sub-vectors (The values to left and right of the pivot) are sorted with the 'if (low_idx $\geq$ high_idx)' statement
  - If this statement evaluates to true, we exit the while loop
  - If this statement is false, we swap the values of the lower index with that of the value at the higher index, and proceed to continue partitioning the data set
- The above process will repeat until the we the previous if statement evaluates to true
- 'high_idx' is in turn the pivot point indicating that the values that are to left of it are less than it and the values to right of it are greater than or equal to it

The above algorithm correctly places the elements that are less than that of the pivot to the left of it and the values that are greater than or equal to that of the pivot to the right of it. This process repeats itself until the condition previously mentioned is achieved. Once this condition is met, then we recursively sort the elements on the left and right of the pivot.

The next algorithm, which is the implementation of the quick sort algorithm, recursively sorts the sub arrays that are created with the partitioning algorithm. This is done by making recursive calls to itself while utilizing the 'quicksort_partition' algorithm to partition the left and right sub arrays of the data set. This algorithm can be seen below.

## Quick Sort Algorithm

Below is the quick sort algorithm in the context of C++:

```
    /*  quicksort - Algorithm that sorts elements in a vector for a given low and high index
    *   Input:
    *     data - Vector of integers passed by reference that are to be sorted
    *     low_idx - Integer value that represents the lower index of the vector that is to be
```

```
*     sorted
*     high_idx - Integer value that represents the higher index of the vector that is to be
*     sorted
*   Algorithm:
*     * If the lower index is greater than or equal to that of the higher index, then stop
*       execution
*     * Otherwise, find the pivot point with "quicksort_partition" and assign this as the
*       upper index of the lower partition
*     * Recursively call the algorithm to sort the lower end of the partitioned vector
*     * Recursively call the algorithm to sort the higher end of the partitioned vector
*   Output:
*     This function does not return a value, it modifies the "data" vector
*/
void Sorting::quicksort(vector<int>& data, int low_idx, int high_idx){
    // Check to see if the lower index is greater than or equal to that of higher index
    if (low_idx >= high_idx) {
        return;
    }
    else {
        // Determine the pivot point of the current vector
        int low_end_idx = quicksort_partition(data, low_idx, high_idx);
        // Recursively sort the lower half of the vector
        quicksort(data, low_idx, low_end_idx);
        // Recursively sort the higher half of the vector
        quicksort(data, low_end_idx + 1, high_idx);
    }
}
```

---

The general method for how this algorithm works is the following:

- First check to see if the lower index is greater than or equal to that of the higher index indicating that the data set has been sorted
- If the data set is not sorted, then we partition the data set such that it will have a lower half and upper half that can be sorted after
- The function then proceeds to finding the partition point (The point where elements to the left of it are less than it and the elements to the right are greater than or equal to it) with the 'quicksort_partition' algorithm
- After the partition point has been found, the algorithm proceeds to recursively sort the left and right halves of the data set until all elements have been sorted

The goal of the quick sort algorithm is to partition a data set to the point such that the elements to the left of the pivot are less than or equal to the pivot and the elements to the right of the pivot point are greater than or equal to that of the pivot. Once this is achieved for a specific data point, we recurisvely sort each partition until the all the partions are of size one. when this is achieved, we have effectively sorted our data set.