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 data set 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 sub-data set one at a time.
- **Merge Sort** - Merge sort is a divide-and-conquer sorting algorithm that recursively divides the data set into smaller sub-data sets and then merges the sorted sub-data sets back together.
- **Quick Sort** - Quick sort is a divide-and-conquer sorting algorithm that recursively partitions the data set and then sorts the two resulting sub-data sets.
- **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 data set and moves it to the sorted portion of the data set.
- **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 data set and swapping them if they are in the wrong order. The algorithm starts at the beginning of the data set 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 data set is reached. At this point, the largest element in the data set will be in the last position. The algorithm then starts again at the beginning of the data set 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 data set 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.

Merge Sort Algorithm

Overview

The merge sort algorithm is a divide-and-conquer algorithm for sorting a data set. This algorithm sorts a data set by recursively breaking it down into smaller and smaller sub-data set until they are sorted. Once the sub-data sets have been sorted, they are sorted back together into one final sub-data set. The merge sort algorithm utilizes an auxiliary data-set (A temporary data-set) to store the sorted elements of the original data set. This auxiliary data-set is then copied back to the original data set. Similar to the quick sort algorithm, the merge sort algorithm requires the use of a separate algorithm in its implementation.

The algorithm that is used in the merge sort algorithm is the merge algorithm. This algorithm is responsible for taking two data sets and combing them into one data set. The merge algorithm can be seen below:

Merge Algorithm

Below is an example of the merge algorithm in the context of C++:

```
/*  merge - Merges two vectors into one final vector
*   Input:
*     left - Vector of integers that is passed by reference that is to be inserted into the
*     result vector
*     right - Vector of integers that is passed by reference that is to be inserted into the
*     result vector
*     result - Vector of integers that is passed by reference
*   Algorithm:
*     * Begin by calculating the size that is needed for the merged vector
*     * Set the positions of each vector that is going to be traversed through
*     * Create a temporary vector that will hold the merged values
*     * Iterate through the elements in the left and right vectors, copy the values into the
*       mergedNumbers vector
*     * Increment the merge position
*     * Merge the left and right vectors into the mergedNumbers vector
*     * Copy the elements from the mergedNumbers vector into the result vector
*   Output:
*     This function does not return a value, it merges two vectors into one, called result
*/
void Sorting::merge(vector<int>& left, vector<int>& right, vector<int>& result) {
  // Calculate the size of the needed vector for the merged vector
  int mergedSize = left.size() + right.size();
  // Set the positions of each vector
  int mergePos = 0;
  int leftPos = 0;
  int rightPos = 0;
  // Create a temporary vector that will hold the merged values
  vector<int> mergedNumbers(mergedSize);
  // Continue while loop as long as there are elements left in each vector to be merged
  while (leftPos < left.size() && rightPos < right.size()) {
    // Insert the element in the left vector into the merged vector if it is less than that
    // of the right vector
    if (left.at(leftPos) <= right.at(rightPos)) {
      mergedNumbers.at(mergePos) = left.at(leftPos);
      ++leftPos;
    }
    // Insert the element in the right vector into the merged vector if it is greater than
    // that of the left vector
    else {
      mergedNumbers.at(mergePos) = right.at(rightPos);
      ++rightPos;
    }
    // Increment the merge position
    ++mergePos;
  }
```

```
  // Merge the left vector into the mergedNumbers vector
  while (leftPos < left.size()) {
    mergedNumbers.at(mergePos) = left.at(leftPos);
    ++leftPos;
    ++mergePos;
  }
  // Merge the right vector into the mergedNumbers vector
  while (rightPos < right.size()) {
    mergedNumbers.at(mergePos) = right.at(rightPos);
    ++rightPos;
    ++mergePos;
  }
  // Copy the elements from the mergedNumbers vector into the result vector
  for (mergePos = 0; mergePos < mergedSize; ++mergePos) {
    result.push_back(mergedNumbers.at(mergePos));
  }
}
```

The main responsibility of the above algorithm is to sort the elements of the two data sets, insert them into the auxiliary data set, and then copy that auxiliary data set back to the original data set. The general procedure of this algorithm can be explained below:

- First calculate the needed size of the auxiliary data set and initialize the indexes of the left, right, and auxiliary data set
- The sub-data sets are then traversed through where the elements of the left and right sub-data sets are compared, after the comparison, the appropriate element is inserted into the auxiliary data set
- The above procedure is repeated until the index of either the left or right sub-data set reaches the back of the respective data set
- The remaining elements of the left and right sub-data sets are then added to auxiliary data set
- The auxiliary data set is then copied back to the resultant data set
- This process will repeat until the left and right sub-data sets are sorted

The merge algorithm is where the sorting of the merge sort algorithm occurs as well as where the merging of two sub-data sets occurs. The recursive nature of the merge algorithm will sort each halves of the original data set.

The next algorithm is the implementation of the merge sort algorithm. The merge algorithm is responsible for creating recursive calls to the input data set so that it can be sorted in a timely manner. This algorithm can be seen below:

Merge Sort Algorithm

Below is an example of the merge sort algorithm in the context of C++:

```
/*  mergesort - Uses the merge sort algorithm to sort elements in a vector
*   Input:
*     data - Vector of integers that is passed by reference that is going to be sorted
*   Algorithm:
*     * First check to see if the vector is empty or has one element
*     * Proceed to calculate the mid point of the vector
*     * Create a vector for the left and right half
*     * Recursively call the mergesort method on the left and right halves of the vector
*     * Merge the two vectors into one by copying the results to data
*   Output:
*     This function does not return a value, it sorts elements from a vector
*/
void Sorting::mergesort(vector<int>& data){
  // The vector is empty or only has one element
  if (data.size() <= 1) {
```

```
      return;
    }
    // The vector is greater than 1
    else {
      // Calculate the midpoint of the vector
      int midPoint = data.size() / 2;
      // Create a left half of the vector
      vector<int> leftHalf(data.begin(), data.begin() + midPoint);
      // Create a right half of the vector
      vector<int> rightHalf(data.begin() + midPoint, data.end());
      // Merge the left and right half of the vector, recursively
      mergesort(leftHalf);
      mergesort(rightHalf);
      // Create an empty vector
      vector<int> result;
      // Call the merge method and merge the two halves together
      merge(leftHalf, rightHalf, result);
      // Copy the results to data
      data = result;
    }
  }
```

This algorithm takes an input data set and recursively splits the input data set until it reaches a size of one. Once the sub-data set reaches a size of one, it then gets passed in as an input parameter into the merge algorithm so that it can eventually be copied back to the original data set that is passed into the algorithm. The general procedure for how this data set works is the following:

- The original data set that is passed into the algorithm is first checked if the data set is of size one, if it is, then the data set is returned, otherwise it continues on to the else block statement
- The midpoint of the data set is then calculated such that it can be split into two halves
- A left and right data set is then created to represent each have of the split data data set
- These sub-data sets are recursively fed into the function until the returned data set is of size one
- Once the returned sub-data set is of size one, it is fed into the merge algorithm so that the two sub-data sets can be combined into one data set
- The above process is repeated until the algorithm no longer merges two halves together, indicating that the data set has been sorted

This algorithm effectively splits an input data set until the returned data set is of size one. Once this happens, the merge algorithm then effectively merges two separate sub-data sets in a manner that they will be sorted. This process repeats over and over until all elements from the original data set have been sorted.

Similar to the quick sort algorithm, the merge sort algorithm is a highly efficient algorithm. Its recursive nature allows for less complex runtimes which makes it an optimal choice for sorting elements. The runtime complexities of this algorithm can be seen below:

Best Case $\Omega(n)$	**Average Case** $\Theta(n)$	**Worst Case** $\mathcal{O}(n)$
$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\mathcal{O}(n \log(n))$

Unlike the quick sort algorithm, the best case runtime complexity for the merge sort algorithm occurs when the list is already sorted. Because of nature of the merge sort algorithm, if the list is already sorted then it typically will only have to make n comparisons when sorting the list. On the contrary, the worst case time complexity for the merge sort algorithm occurs when the data set is sorted in reverse order. This is due to the splitting of the data sets and the recursive nature of the algorithm.

On the contrary, the space complexity of the merge sort algorithm tends to be of the order n, $\mathcal{O}(n)$. This is because each time the recursive function is called it has to occupy memory. If the merge sort algorithm uses the 'divide and conquer in place' strategy, where the sorted halves of the data set are stored in the original data set, the space complexity will decrease.

Optimization & Utilization

As mentioned previously, the merge sort algorithm is a divide-and-conquer algorithm that is highly efficient in the context of runtime complexity. Because this algorithm breaks the problem of sorting into smaller and smaller parts, the spacetime complexity is linear. Due to the spacetime complexity of the merge sort algorithm, there is a situation where the algorithm can cause a stack overflow due to the recursive nature of the algorithm. On the contrary, small data sets can be less efficient because of the comparisons and splitting of the data set that is needed for the algorithm to operate. For small data sets this can be over kill because of the nature of how the algorithm operates.

The merge sort algorithm does not rely on making comparisons like a bubble sort algorithm. Instead, it breaks the data set into smaller and smaller data sets and only makes comparisons when the sub-data sets are being merged into a final data set. The merge sort algorithm can be used with linked lists. The fact that the merge sort algorithm does not have to traverse the list in order to sort it, it makes sorting it relatively easy. The main difference between sorting a linked list and sorting a data set like an array or vector is that when the individual linked lists have to be merged, the pointers of the lists have to point to the correct node in the list that is being merged. Comparing the previous runtime complexities for sorting linked lists, here is how they compare to one another:

Algorithm	**Worst Case Time Complexity** $\mathcal{O}(n)$	**Worst Case Space Complexity** $\mathcal{O}(n)$	**Stability**
Bubble Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Stable
Merge Sort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n)$	Stable
Quick Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$	Not Stable

Stability in this context refers to the ability for an algorithm to preserve the relative order of equal elements in the sorted data set. This means if two elements in the data set are equal, an unstable algorithm may swap the relative order of sorted elements in the data set. If we look at these three algorithms, the overall comparison of all the data sets are:

Algorithm	**Best Use**	**Complexity**	**Space Complexity** $\mathcal{O}(n)$	**Stability**	**Time Complexity** $\mathcal{O}(n)$
Bubble Sort	Small Arrays	Simple	$\mathcal{O}(1)$	Stable	$\mathcal{O}(n^2)$
Merge Sort	Large Arrays	Complex	$\mathcal{O}(n)$	Stable	$\mathcal{O}(n \log(n))$
Quick Sort	Medium-Sized Arrays	Simple	$\mathcal{O}(n)$	Not Stable	$\mathcal{O}(n^2)$

Overall, the merge sort algorithm is a highly efficient algorithm. It is versatile in a number of ways and does not really have a lot of instances where it is inefficient. There are occasional situations where the algorithm may cause a stack overflow on large amounts of data but that problem can be mitigated with modifications to limit the number of times function has to make a recursive call. It works great with almost all data types and can be used with dynamic memory data structures such as linked lists.

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-data sets created by the partition. The pivot element that is chosen for this algorithm is typically the middle element of the data set, 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-data sets 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++:

```
/*  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 the previous if statement evaluates to true
- 'high_idx' is returned from the partition algorithm to represent the index where elements to the left of it are less than or equal to it as well as the elements to the 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-data sets 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 data sets 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 their is only one element in the data set that is being sorted, otherwise continue to the next logic block
- If the data set is greater in size than one element, 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 that are set with the help of the partition point until the partition reaches size one

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. This process will repeat until the data set that is being partitioned is of size one. After the data set has been partitioned, it is sorted recursively with a call to 'quicksort'.

The quick sort algorithm is a very efficient algorithm for data sets of all sizes. Its recursive nature allows for different sizes of data sets to be sorted efficiently in a timely matter implicating that it is a optimal choice for a sorting algorithm. The runtime complexity of the quick sort algorithm is as follows:

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

The quick sort algorithm typically will have a runtime complexity of $\Theta(n \log(n))$ with the worst runtime complexity of $\mathcal{O}(n^2)$. The worst case runtime complexity occurs when the data set that is being fed into the algorithm is sorted in reverse order.

Optimization & Utilization

When using a quick sort algorithm there are a myriad of choices that can affect the runtime of the algorithm. One choice that can affect the algorithm directly is how the pivot point is calculated. For instance, if we choose a pivot point such that it is always the smallest or largest element in the data set, the algorithm is only able to sort one element at a time. This can

lead to a very poor runtime complexity and in turn it decreases the overall efficiency of the algorithm. If the pivot point is chosen carefully, it can lead to the average case runtume complexity. The careful selection of the pivot point can result in partitioning sub data sets of relatively the same size on each recursive call. The choice of the pivot point is contingent upon the data set that is being fed into the algorithm. The impact of the choice of pivot point can be seen in the following table:

Choice of Pivot	**Average Case Runtime** $\Theta(n)$	**Worst Case Runtime** $\mathcal{O}(n)$
First Element	$\Theta(n \log(n))$	$\mathcal{O}(n^2)$
Median, Middle, or Last Element	$\Theta(n \log(n))$	$\mathcal{O}(n \log(n))$
Random Element	$\Theta(n \log(n))$	$\mathcal{O}(n \log(n))$
Smallest or Largest Element	$\Theta(n \log(n))$	$\mathcal{O}(n^2)$

These are just some examples of choosing elements for the pivot point and how it affects the runtime of the algorithm. In general, the average runtime of the algorithm is still very efficient but there are specific selections of the pivot point where the algorithm can perform poorly.

Occasionally elements in the data set will have duplicates. When this happens, the element that is a duplicate will be placed in a data set that is the same as that of the pivot. Generally duplicates do not affect the runtime efficiency too greatly, however, if the data set possesses a lot of duplicates it can lead to the worst case runtime complexity.

The quick sort algorithm can be used recursively as well as iteratively. If one chooses to use the quick sort algorithm iteratively it is usually done with the use of stacks. The procedure for using the quick sort iteratively is:

- Initialize the stack with the data set to be sorted
- While the stack is not empty:
  - Remove the top element from the stack
  - Partition the data set at the pivot point
  - Push the left data set onto the stack
  - Push the right data set onto the stack

Once the data set is sorted, the stack will end up empty. This method of utilizing an iterative method over a recursive method can lead to some advantages and disadvantages. Here is how the iterative method is both advantageous and disadvantageous:

- Advantages:
  - The iterative method does not use recursion, which can be beneficial for some programming languages, such as Python, which have a limited stack size.
  - The iterative method is easier to understand and debug than the recursive method.
- Disadvantages:
  - The iterative method is slightly slower than the recursive method, since it has to do an extra loop to push and remove the sub data sets onto the stack.
  - The iterative method is not as flexible as the recursive method, since it cannot be easily used to sort a data set in place.

Occasionally, the input data set that is fed into the quick sort algorithm is nearly sorted or sorted. When this happens, neither the recursive or iterative method is efficient in sorting the data set. This is because of the nature of the partitioning algorithm that is present in the quick sort algorithm. A way to limit the runtime complexity of the quick sort algorithm with a sorted or near sorted data set is to use random sampling in the recursive call of the quick sort algorithm. Random sampling refers to randomly selecting the pivot value that is being used. When this technique is used it can limit the negative effects of dealing with a sorted or nearly sorted data set.

There are several ways to make mistakes with using the quick sort algorithm. Some examples of errors in implementation of the quick sort algorithm are the following:

- **Choosing The Wrong Pivot Point** - As previously mentioned, if the pivot point is not carefully chosen, the algorithm can perform poorly and lead to its worst case runtime complexity.
- **Not Handling Duplicates** - If duplicates are not handled correctly in the quick sort algorithm then it can lead to inaccurate results or inefficient runtimes.
- **Not Using Recursion** - When recursion is not used, this can lead to runtime complexity that is far worse. Using recursion allows memory allocation that is efficient and thus speeds up the runtime of the algorithm.

Overall, the quick sort algorithm can be optimized in a variety of ways. When dealing with different input data sets, one can optimize the algorithm with handling cases that are appropriate for specific input data sets. This changes for different inputs but the general procedure is the same; find a comparison method, pivot selection, or some other form of algorithmic enhancer that will accommodate for the given input that is being fed into the algorithm.