# Sorting Algorithms

Matthew Boekamp, Richard Buckley, Sahil Chadha, Joseph Yanez
CSC 212 Data Structures and Abstractions
Professor Schrader
24 June 2023

**Table of Contents**

## 1. Introduction

In computer science, management and manipulation of large amounts of data are critical to the success of countless applications. Sorting algorithms lie at the center of modern computing, something which is evident when you look at web searches, database management, social media, and statistical analysis. All of these require efficient sorting and arranging of data to produce optimal results and are central to day to day life as we know it. In this report, we delve into the world of sorting algorithms, exploring their definition, implementation, and their time complexities.

Early in our brainstorming process, we settled on creating a sort visualizer to better show and explain the process of how each individual sorting algorithm works. When deciding this, we figured there would be no better way to illustrate how these algorithms work better than a visualizer. As these algorithms are central to day to day life, being able to fully grasp the concepts surrounding them is crucial. Within our visualizer, there are options to slow down the speed, move through the steps one at a time, and increase the number of elements being sorted. By visualizing the steps, and allowing for customization, we hope to make it easier to grasp the underlying concepts of the algorithms we explored.

## 2. Planning

Our planning process began with dividing up the workload by the amount of sorting algorithms. Each group member received a sorting algorithm being either *Quick Sort, Insertion Sort, Merge Sort* or *Radix Sort.* Each member would then break off and do their own research on it whether it was through the lecture slides we had through GitHub or online **(ex. geeksforgeeks).** The codes would then be put up through github for each of our IDE's. During the next meeting we had, we decided to use the graphics library SDL2 to create an early iteration of our visualizer and then go from there. After creating this, we decided to shift our visualizer to React to make it a web based program. This alleviated much of the concern over trouble with installation which we encountered preparing to make the visualizer. Through React we are able to look at the timing individually for each code through their own tabs. The timing is shown with bar graphs. The third step of our overall process was to work on the slides for our presentation. This presentation will include examples, time complexity and what each sort is doing in its logic. It will also include a timing code that was added to each algorithm, this tests each program and prints out a time to show the difference in the time. Our final meeting will serve as a series of test runs, recording several times to make the presentation in time and efficiently.

## 3. Theoretical Concepts

Time complexity is a crucial concept in computer science that measures the efficiency of an algorithm. It allows us to analyze and compare different algorithms of the same size, $n$, and

identify the more efficient algorithm to solve the issue at hand. Time Complexity is normally expressed in Big O notation. In this format, the time complexity is denoted as O(f($n$)) where f($n$) is the growth rate in terms of an input of $n$ size. What follows is a brief overview of the time complexities of the algorithms we chose. This chart is brief but this report delves further into the time complexities of Merge, Insertion, Quick, and Radix Sort.

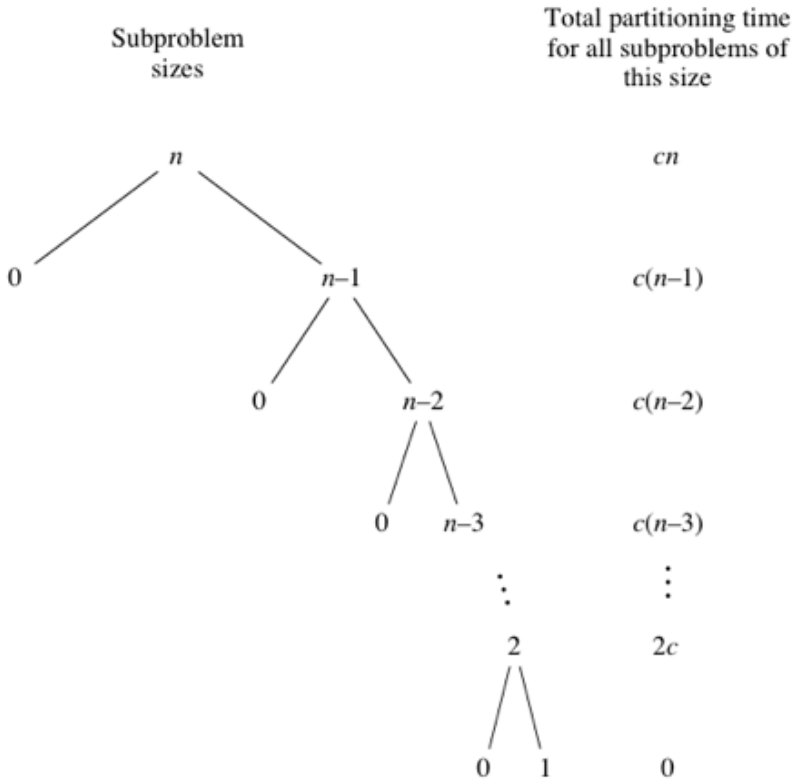
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
|---|---|---|---|
| Insertion Sort | O($n$) | O( $n^2$) | O( $n^2$) |
| Quick Sort | O($n$ log $n$) | O($n$ log $n$) | O( $n^2$) |
| Merge Sort | O($n$ log $n$) | O($n$ log $n$) | O($n$ log $n$) |
| Radix Sort | O($n$ * $d$) | O($n$ * $d$) | O($n$ * $d$) |

## 4. Quick Sort

Overview

Quick Sort has a time complexity in the worst case O( $n^2$) and O($n$ log $n$) in its average case. It uses the idea of pivoting. The pivoting element in the array which in some cases is found by the rule of medians, the first, middle and last element are sorted and the middle element is chosen as the pivot. From there it will test the left side to make sure that all numbers are smaller than the pivot. If one is larger than the pivot it will save that position, it will do the same for the right side, but it will check for the smallest element and save that position. Once that is complete it will switch the two points. It will recursively run until both sides are less than or more than the pivot, if it is not sorted yet it will pick a new pivot through the right side.

Time Complexity Analysis




According to Khan Academy, who were in association in producing the following description with Dartmouth University's computer science department, Quicksort's worst case scenario time complexity is O($n^2$). The visual above is from their website and can be found here. According to their description, each recursive call on $n$ - 1 elements takes $c(n - 1)$ time. In this situation, $c$ is a constant. This relation continues on with $n$ - 2 elements taking $c(n - 2)$ time. This continues until 2 which takes $2c$ time and breaks down into 0 and 1. When adding up all the time taken, it results in a time complexity of O( $n^2$).

## Comparison to Other Sorting Algorithms

**Insertion Sort**

When compared to Insertion Sort, it can be seen that Quicksort is better suited for larger datasets while the simplicity and O($n$) best case time complexity of Insertion Sort makes it more appealing for smaller datasets.

**Merge Sort**

Both Merge Sort and Quicksort use divide-and-conquer methods. The main difference between the two is the time complexity. Both have an O($n$ log $n$) for best and average case scenarios. The difference occurs in the worst case scenario where Merge Sort still has O($n$ log $n$) while Quicksort has O( $n^2$). This makes Merge Sort the better algorithm in terms of time complexity.

**Radix Sort**

Radix Sort has a linear time complexity of $O(n * d)$ for sorting with $n$ being the number of elements and $d$ being the number of digits in the largest element. This time complexity is consistent across all scenarios. Radix Sort is specialized for numeric values while Quicksort can function with various different data types after a few alterations.

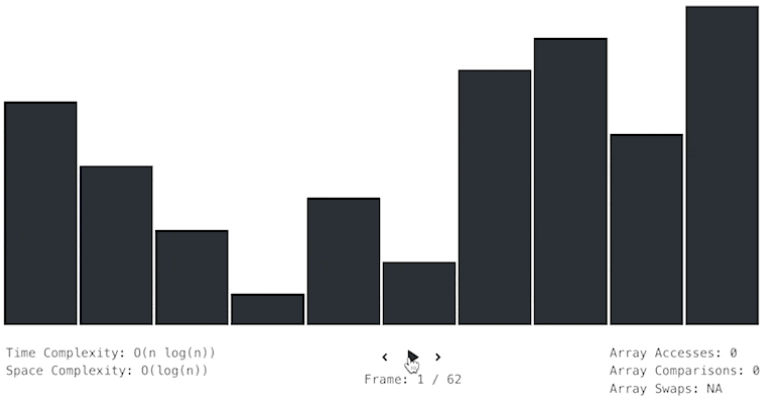
Implementation

```
int quickSort(std::vector <int> *quick, int left, int right) {
        int i = left;
        int j = right + 1;

        while (i < j) {

            // while quick[i] < pivot, increase i
            while (quick[++i] < quick[left])
                if (i == right) break;

            // while quick[i] > pivot, decrease j
            while (quick[left] < quick[--j])

                if (j == left) break;

            // if i and j cross exit theloop
            if(i >= j) break;

            // swap quick[i], quick[j]
            std::swap(quick[i], quick[j]);
        }

        // swap the pivot with quick[j]
        std::swap(quick[left], quick[j]);

        //return pivot's position
        return j;
}

// recursively calls quicksort
void r_quicksort(std::vector<int> *quick, int left, int right){
    if (right <= left) return;

    int pivot = quickSort(quick, left, right);

    r_quicksort(quick, left, pivot - 1);
    r_quicksort(quick, pivot + 1, right);
}
```

The first function in this Quicksort implementation is a helper function to the second. It takes in three parameters, a reference to an array, ‘left’ which is the starting index of the current subarray, and ‘right’ which is the ending index of the current subarray. Two variables are initialized with the left and right + 1 values . The first, i, is initialized with left and scans the array from the left towards the right. The second, j, is initialized with right + 1 and scans from right to left. The while loop increments i until an element greater than or equal to the pivot. The second does the same thing but decrements j until an element less than or equal to is found. Then, the if statement checks to see if i and j have crossed, meaning the scan is done. If this is not reached, the elements at index of i and index of j are swapped. The function then returns the index of the pivot.

The second function from this Quicksort implementation is the main driver of the program. It takes in three parameters, a reference to an array, ‘left’ which is the starting index of the current subarray, and ‘right’ which is the ending index of the current subarray. The function sets left and right to values i and j. Then, there is the base if the right is less than or equal to the left. If this is true, the function returns and exits any other calls. Otherwise, the helped function is called to find the pivot. Then, the function is recursively called for the left and right subarrays.

Visualization




Advantages and Disadvantages

Advantages:

- **In-Place Sorting:** Quicksort does not require any additional memory beyond what is taken up for the input size. With small datasets, this is not relevant, but as the datasets become larger this could make a significant difference.

Disadvantages:

- **O( $n^2$) Worst Case Time Complexity:** When compared to the other divide-and-conquer algorithm, Merge Sort, Quicksort has a significantly worse worst case time complexity. This makes it less effective on larger datasets.

## 5. Insertion Sort

Overview

The second sorting algorithm we implemented was Insertion Sort. Insertion Sort is a simple and efficient comparison based sorting algorithm that builds a final sorted array one element at a time. As the array progresses through the sort, it is split into two halves, a sorted one and an unsorted one. Values from the unsorted part are picked and placed at the correct position. Insertion Sort’s significant worst case and average case runtime of O( $n^2$), contributes to it not being widely used. It is, however, effective with small quantities of data and data that is already

partially sorted due to its best case runtime of O($n$) and its easy implementation makes Insertion Sort a viable choice for the correct data set.

## Time Complexity Analysis

Insertion Sort has a time complexity of O( $n^2$) because of how it compares and shifts elements within the array. In the worst case scenario, every element being sorted has to be placed at the start of the array. This requires all elements to be shifted to the right. When looking at the comparisons, it can be seen that a similar relation exists. When considering both the comparisons and insertions, the worst case scenario in Insertion Sort can be looked at as follows: $(1 + 2 + 3 + \ldots + (n - 1)) + (1 + 2 + 3 + \ldots + (n - 1))$. This can be simplified to show the Insertion Sort time complexity is O( $n^2$).

## Comparison to Other Sorting Algorithms

**Quicksort**

Quicksort uses a divide and conquer approach and selects a pivot element from the array. From here it divides the elements into two subarrays and continues this process recursively to the new subarrays. Quicksort has a best and average time complexity of O($n$ log $n$) and has a worst case of O( $n^2$). Despite a poor worst case, this algorithm can outperform other sorting algorithms due to the good average case behavior. When compared to Insertion Sort, it can be seen that Quicksort is better suited for larger datasets while the simplicity of Insertion Sort makes it more appealing for smaller datasets.

**Merge Sort**

Similar to Quicksort, Merge Sort is also a divide and conquer algorithm. Merge Sort has a time complexity of O($n$ log $n$) in all cases. This makes it the more logical approach to larger datasets. But, Insertion Sort's best case of O($n$) once again makes it the better approach to smaller datasets or partially sorted ones.

**Radix Sort**

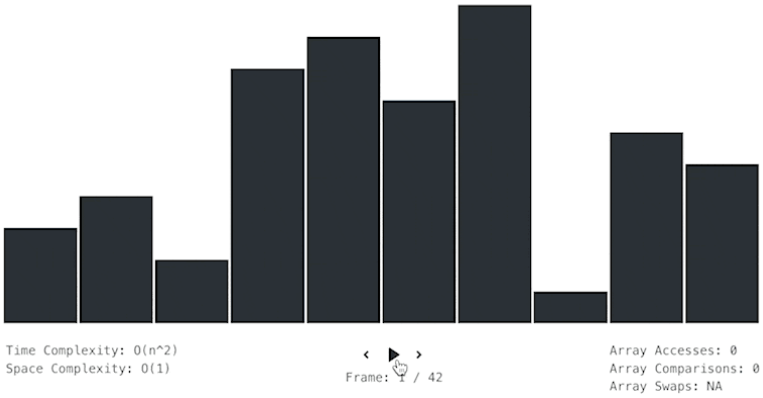
Radix Sort is a non-comparative sorting algorithm that has a linear time complexity of O($n * d$). In relation to Insertion Sort, Radix Sort can be faster with large integers however, it is fastest when the range of the values is not significant. Overall, just like the others, Insertion Sort's easy implementation makes it a better choice when the number of cases is small or partially sorted.

## Implementation

```
void insertionSort(std::vector <int> &array, int sizeN){
    // define our pointers
    int i, j, k = 0;

    // loop through array
    for (i = 1 ; i < sizeN ; i++){
        j = i;

        // store temp
        k = array[i];

        // move element until it is in order
        while (j > 0 and k < arr[j - 1]){
            array[j] = array[j - 1];
            j--;
        }

        // restore temp
        array[j] = k;
    }
}
```

This Insertion Sort function takes in two parameters, the reference to an array and the size of the sequence. Next, the variables i, j, and k are initialized with k being set to 0. The first for loop is responsible for selecting an element from the unsorted part of the array. This is why it begins at 1 and iterates until sizeN, the size of the array, is reached. Then the variable j is set equal to i. This will be used to find the correct position for the element at index i. Then, the value of the sequence at i is stored with the variable k. The while loop checks to see  if the value at index j - 1, which is one before, is less than the current value. If this is the case, then the element is moved to the right, creating room for the element at index i to be inserted. After finding the correct index, the element at index i is then inserted into the array at index j.

## Visualization




## Advantages and Disadvantages

Advantages:

- **Simplicity:** Compared to the other sorting algorithms explored in this report, Insertion Sort has the easiest implementation.
- **Good Time Complexity in the Best Case:** Insertion Sort's O($n$) time complexity at the best case is the quickest out of all the sorting algorithms in this report. However, it's high time complexity of O( $n^2$) in all other situations

Disadvantages:

- **Poor Time Complexity In Average and Worst Case:** It has the quickest best case time complexity but the slowest average and worst case complexity. The high complexity of O( $n^2$) in all situations besides the best case scenario limits the functionality of this algorithm. This makes it best suited for small datasets and partially sorted ones.

## 6. Merge Sort

### Overview

The third sorting algorithm we implemented was Merge Sort. Merge Sort is a divide-and-conquer algorithm created by John von Neumann in 1945. Since its inception, merge sort has been widely used due to simplicity, relatively easy implementation, and guaranteed time complexity of O($n$ log $n$). The process of Merge Sort can be broken down into the following parts:

1. **Divide**: The input list is divided into two equal halves (if the number of elements in the array is even) or two nearly equal halves. This is repeated until each element of the original list is by itself. This will put each element into its own sublist.
2. **Conquer**: In this part, the sublists are continually merged together to produce a new sorted sublist.
3. **Combine**: This is the final step of the algorithm where the larger sublists are combined. This continues until there is only a single sorted list remaining.

### Analysis of Time Complexity

As stated, Merge Sort has a time complexity of O($n$ log $n$) for the best case, average case, and the worst case scenarios when sorting $n$ objects. This is broken down by seeing that the first pass of Merge Sort merges subarrays of size 1. Then, the sort moves on to segments of size 2. Then to size 3 and so on until $i$, which merges subarrays of size $2^{i-1}$ . The total number of passes ends up being log $n$. Each merge takes O($n$) time which means the final time complexity is O($n$ log $n$).

### Comparison to Other Sorting Algorithms

### Quicksort

At its worst case, Merge Sort uses approximately 39% fewer comparisons than Quicksort does in its average case. Merge Sort's time complexity in its worst case (and all other cases) is the same as Quicksort's best and average case time complexity. Despite Merge Sort being the faster sort, Quicksort is an in-place sort, unlike Merge Sort.

### Insertion Sort

Merge Sort's O($n$ log $n$) time complexity trumps Insertion Sort's O( $n^2$) average and worst case time but falls short of its best case scenario. When looking at the efficiency of the two, Merge Sort is highly efficient with large datasets due to its divide-and-conquer approach. Insertion Sort, meanwhile, performs better on smaller datasets but the $n^2$ time complexity makes it less suitable for larger datasets. Insertion sort is both a stable and in-place algorithm, unlike Merge Sort which is only stable.

### Radix Sort

When looking at the time complexity of Radix Sort, we see that it is O($n$ * $d$). In this, $n$ refers to the number of elements in the list and $d$ refers to the number of digits in the largest number. Normally, Radix Sort is considered to be faster than Merge Sort's O($n$ log $n$) time complexity with smaller datasets where the number of digits is relatively small. Based on the timing program we developed, Radix Sort scored faster than Merge Sort consistently, not only with smaller data sets. Radix Sort ended up being the fastest sorting algorithm we implemented during this project.

## Implementation

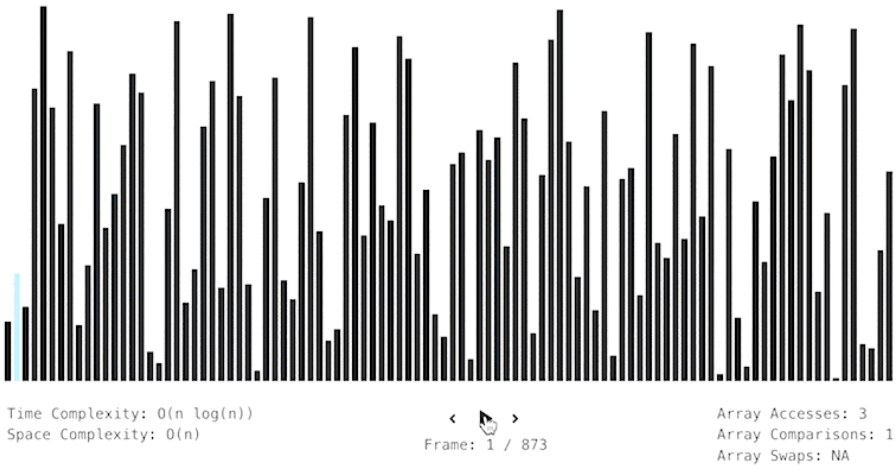
```cpp
void Sorts::merge(std::vector<int>& arr, int left, int mid, int right) {
    int i = left;       // Index for left subarray
    int j = mid + 1;    // Index for right subarray
    int k = 0;          // Index for temporary array

    std::vector<int> temp(right - left + 1); // Temporary array

    // Merge the two subarrays into the temporary array
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    // Copy the remaining elements of the left subarray, if any
    while (i <= mid)
        temp[k++] = arr[i++];

    // Copy the remaining elements of the right subarray, if any
    while (j <= right)
        temp[k++] = arr[j++];

    // Copy the merged elements back to the original array
    for (i = left, k = 0; i <= right; i++, k++)
        arr[i] = temp[k];
}

 void mergeSort(std::vector<int>& arr, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2; // Calculate the mid-point

            // Recursively divide the array into two halves
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

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

The first function that is used is a merge function which merges two sorted subarrays into a temporary array. This function takes four parameters. Firstly, ‘arr’ is a reference to the original vector containing the array to be sorted. Then, ‘left’ is the left bound index of the first subarray. Following this, ‘mid’ is taken in. This is the index representing the right boundary of the first subarray and the left boundary of the second subarray. Finally, ‘right’ is the index representing the right boundary of the second subarray. From here, the function creates a vector called temp that is initialized to the size of left - right + 1 to be able to hold all of the merged values. The first while loop merges the two subarrays into the temporary vector. It compares and inserts the smallest element into the vector while advancing the indices accordingly. The second and third while loops check for any additional elements left in the array. This is used to correctly implement subarrays which have an uneven number of elements. The final for loop copies the sorted elements back to the original array, completing the merge step.

The second function, titled mergeSort, implements the recursive divide-and-conquer approach. This function takes in three parameters. Firstly, ‘arr’ is a reference to the original vector containing the array to be sorted. Then the next two are ‘left’ and ‘right’. Left’ is the left bound index of the subarray. While right is the index representing the right boundary of the subarray. The first if statement checks if left is less than right which checks if the current segment of the array has more than one element. From here, the midpoint is calculated and then

the merge sort function is recursively called for the two halves. After this, the merge function is called to merge the sorted left and right subarrays.

Visualization




Advantages and Disadvantages

Advantages:

- **Stable performance**: Merge Sort has a time complexity of O(*n* log *n*) for the average and worst-case scenarios. This makes it a good choice when dealing with larger datasets.

Disadvantages:

- **Space complexity**: Merge Sort requires additional memory for the temporary arrays used during merging. This is not a concern with smaller datasets but can be one when dealing with very large ones.
- **Slower for smaller lists**: Merge Sort is less efficient for smaller lists due to the recursive calls and merging. Other algorithms like Insertion Sort function better for these cases.

## 7. Radix Sort

Overview

The final sorting algorithm we implemented was Radix Sort. Radix Sort dates back all the way to 1887 with tabulating machines, however the first memory efficient computing method for this sort was developed in 1954 at MIT. Radix Sort is a non-comparative algorithm that functions by distributing elements into containers based on individual digits at different positions within the numbers. It is typically most effective when sorting numbers with a fixed number of digits. The algorithm works by performing a series of passes with each pass sorting the element

based on a digit position. The process is repeated until all the numbers have been appropriately placed.

Though very powerful, this non-comparative sorting algorithm does not see anywhere near as much use as other sorts, such as Quicksort, or Merge Sort because it is not as versatile. Radix sort falls short when dealing with doubles, and long numbers. If radix sort was implemented in a way that supported doubles, the sorting algorithm could potentially take much, much, longer, as numbers such as π, or $1.00000000000000000032$ are obtainable. This is a major disadvantage for a non-comparison based sorting algorithm, as this would increase the number of digits, or times that the algorithm would need to run before determining that the array was sorted.

## Analysis of Time Complexity

This algorithm has a time complexity of O($n * d$) where $n$ is the number of elements in the array and $d$ is the number of digits in the largest value in the array. The code runs an outer loop $d$ times and within this loop, values are placed into containers within another for loop. This loop runs $n$ times. This contributes to the overall time complexity of O($n * d$).

## Comparison to Other Sorting Algorithms

### Quicksort

Radix Sort has a linear time complexity of O($n * d$) for sorting with $n$ being the number of elements and $d$ being the number of digits in the largest element. Quicksort has a time complexity of O($n \log n$) for the best and average case while having O($n^2$) with the worst case scenario. Radix Sort is specialized for numeric values while Quicksort can function with various different data types after minor alterations.

### Merge Sort

Normally, Radix Sort is considered to be faster than Merge Sort's O($n \log n$) time complexity with smaller datasets where the number of digits is relatively small. Based on the timing program we developed, Radix Sort scored faster than Merge Sort consistently. Radix Sort ended up being the fastest sorting algorithm we implemented during this project.

### Insertion Sort

Insertion Sort has a time complexity of O($n^2$) for the worst and average case while having O($n$) for the best case. Although insertion sort hsa a best case of $O(n)$, this only occurs when the array is already sorted, or partially sorted. Overall, with numerical values, the more efficient choice is

Radix Sort. But, with smaller datasets, the simplicity of the Insertion Sort implementation makes it the more logical choice.

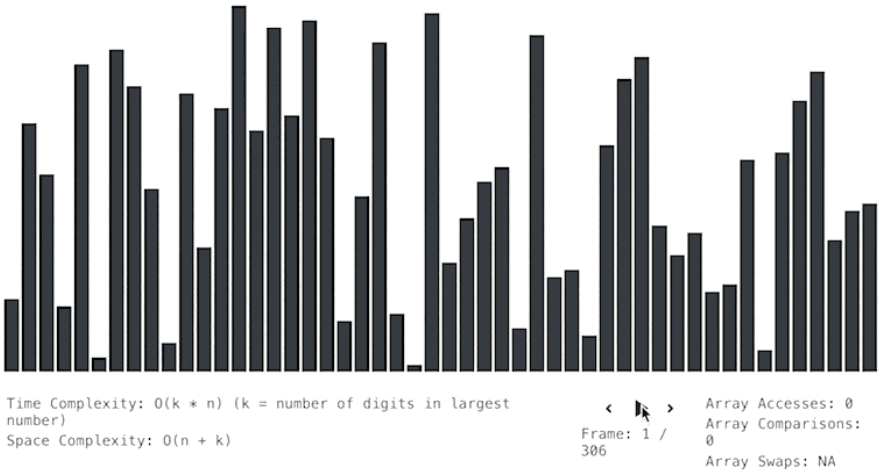
## Implementation

```
void radixSort(std::vector<int> &array, int numDigits) {

    for (int i = 0 ; i < numDigits ; i++) {

        // create empty containers that we will organize into
        std::vector< std::vector<int> > containers(10);

        // for num in arr
        for (int v : array)
            containers[(v / (int)std::pow(10, i)) % 10].push_back(v);

        // clear out arr
        array.clear();

        // Rinse & Repeat
        for (const std::vector<int> &c : containers)
            array.insert(arr.end(), c.begin(), c.end());

    }

}
```

The ‘radixSort’ function takes in two parameters, a reference to a vector of integers, and the number of digits of the largest number in that vector. The variable numDigits is calculated in a helper function which determines the largest number and then calculates the number of digits within that number. From here, the value is passed to this function. The function begins with a for loop which runs from 0 to the parameter numDigits. With each run through the for loop, a vector of 10 containers is created. This is used to organize the elements based on their digits (0 to 9). The first pass of the array the elements are distributed based on their least significant digit. Then the second pass arranges them based on their second least digit and so on. After each element has been placed into its appropriate container, the original array is cleared. This allows for the new sorted elements to be placed into the array. This step occurs in the final for loop. This cycle continues from 0 to numDigits, ensuring that every element is correctly sorted.

The variable numDigits is calculated in a helper function to allow for easily sorting arrays with an absolute number of digits in each number. This can be helpful if you want to sort an array, where we know that each element in the array is exactly 10 digits long. It is also worth noting that finding the maximum value and getting the number of digits of it adds an $O(n)$ time complexity, making it much more advantageous to know the exact amount of digits in the largest number of the array.

## Visualization




Advantages and Disadvantages

Advantages:

- **No Comparisons and Linear Time Complexity:** The linear time complexity of Radix Sort can combine with ideal values in the array to produce a much faster sorting time than other comparison based sorts.

Disadvantages:

- **Lack of Flexibility:** Other sorting algorithms mentioned within this report can be slightly modified to accommodate other forms of data rather than just numerical values. Radix Sort, on the other hand, uses digits within the numerical values given to determine placement. Radix Sort could be modified to function in other ways but it is not as simple as the other algorithms.

## 8. Visualizer

For this assignment, we ended up making two different visualizers, one online with Javascript, and another on a local machine with SDL. We did this so we could both easily show a visual representation of our algorithm (online), but also have a very performant local visualizer that shows a similar visualization of arrays being sorted.

### 8a. SDL Visualizer

To make the SDL visualizer, we used the SDL2 library which allows for graphics. The visualizer adopts a concept of using each element in an array as a bar where

each bar's height corresponds to the value being represented. The visualizer works by depicting the state of the array at swaps within the algorithm. This is easily visible when looking at Merge Sort, Insertion Sort, and Quicksort. It is, however, more difficult to see on the Radix Sort. This is because Radix Sort is a non-comparison algorithm.

Beyond the sorting algorithms, which were taken from sorts.cpp on GitHub, the code has an additional function to visualize the bars called renderBars. This function takes in the current state of the array, as well as several global variables to create a visualization of the current state. Calls of this function are placed throughout the four sorts to accurately depict data rearranging during the sort. The global variables were used to easily change the size and speed of the visuals.

## 8b. Web Visualizer

To make the visualizer, we used the React javascript library, which allowed us to deeply simplify the components needed to create the visualization. One component was `Bar` and the other was `Container`. We created a bar for each value in the array, and updated each bar's value on each frame. After applying a transition and some styling, we arrived at our end result.

To actually sort these arrays however was a problem in itself. We did not implement a sort as we render approach, as this could go badly… javascript is not as powerful as c++, and this could bottleneck with too much going on at the same time in lower end computers / phones. To solve this problem, we generated the array in steps, saving each step whenever we moved indexes, with colors representing the state of the sorting algorithm.

To accomplish this, we created a class `SortingAlgorithm` which was implemented by each sort… `class InsertionSort extends SortingAlgorithm …`, and then we would just override the sort() function. And that was it. We just added a sleep function to wait a certain amount of time before rendering the next step. To finish the program, we matched the theme of the visualization to the theme of the presentation, added some different ways to sort the initial array, and that was it.

## 9. Timing Program

For the timing program we decided to use python to run a cpp file, which would output the times for each of the algorithms to sort a given array. The reason for using python was so we could analyze more in depth the results of each array. Furthermore, python allowed for rapid development to occur. Though, it is not without its faults, it would most certainly be more efficient for the c++ program to output the results to the appropriate csv file, however, when we interface this with python, we are able to send the results to separate `.csv` files for different cases (e.g. different sort, different test case, different column structure, etc).

We also used code from stack overflow here (which we cited), in efforts to make sure our results were very accurate. After the code was generated by python, we sent it to a google sheet and performed an analysis. To analyze the results, we first took the mean and standard deviation of each algorithm. We then calculate z-scores (which is how far an algorithm deviates from the mean), which is the right choice to benchmark, as the hardware inside each machine is different. This allows us to obtain repeatable results on different computers.

Though this code seems very long, a lot of it is comments. Furthermore, this program is not entirely complete ~ although it is straightforward, if more time was allocated, we would add a function to directly output a results table with the given z-scores.

## 10. Contributions

| **Group Member** | **Contribution** | **Start Date** | **End Date** |
|---|---|---|---|
| Richard Buckley | Radix Sort | 7/6/2023 | 7/8/2023 |
| Matthew Boekamp | Merge Sort | 7/6/2023 | 7/8/23 |
| Matthew Boekamp | SDL Visualizer | 7/10/2023 | 7/24/2023 |
| Richard Buckley | Web Visualizer | 7/18/2023 | 7/23/2023 |
| Richard Buckley | Timing Program | 7/22/2023 | 7/23/2023 |
| Joseph Yanez | Quick Sort | 7/6/2023 | 7/6/2023 |

| | | | |
|---|---|---|---|
| Sahil Chadha | Insertion Sort | 7/6/2023 | 7/6/2023 |
| All | Report | 7/19/2023 | 7/23/2023 |
| All | Presentation | 7/19/2023 | 7/23/2023 |

A quick note: please don't dock points from us for making an additional visualizer with react, though that seems like a lot more work (it was), I mainly did it for my portfolio, as I want to get an internship next year (fingers crossed), and it was a cool project. Also I had a couple flights and plenty of time. - Rich

## 11. Conclusion

In conclusion, this report explored four fundamental sorting algorithms: Merge Sort, Insertion Sort, Quicksort, and Radix Sort, along with a visualizer to help grasp the functionality of the algorithms. Through testing, implementation, and research we have gained an improved understanding of each algorithm's implementation, advantages and disadvantages. Merge Sort, along with its divide-and-conquer approach proved to be an effective option for all scenarios. Having an O($n$ log $n$) time complexity for all scenarios makes it a logical choice for any situation. Insertion Sort's easy implementation as well as great best case scenario time complexity of O($n$) makes it the best choice when dealing with small datasets or partially sorted ones. Quicksort, similar to Merge Sort, exhibited great performance on best and average case scenarios due to its O($n$ log $n$) time complexity. It is, however not a great option for larger datasets due to the O($n^2$) worst case scenario. Radix Sort surprised us all by time and time again proving to be the quickest algorithm in our testing process. (We did find an implementation of a hybrid Radix sort in the Boost cpp library, so this helps affirm our results) The visualizer we developed provided us with a deeper understanding of the step-by-step execution of each algorithm. Overall, our implementation of the sorting algorithms, visualizer, and timing program and the knowledge that we gained in the process will allow us to make informed decisions on which algorithm is most suitable for future problems we encounter.

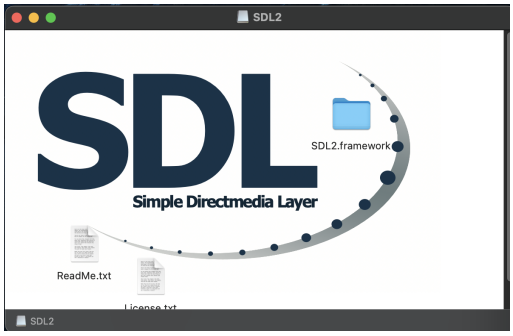
## 12. Appendix

Installation of SDL2 for SDL Visualization

Steps on macOS:

1. Go to the SDL website: https://www.libsdl.org/
2. Click SDL Releases under Downloads on the left side of the website, this will direct you to SDL's latest release on GitHub
3. Click the download that is correct for your computer
    a. To skip this step click the following for Macbooks: SDL2-2.28.1.dmg

   b. Clicking here will download the correct file
4. Following the download, open the file. It should look like this:




5. From here, open a new finder window by right clicking the finder icon and selecting New Finder Window near the top
6. On the top of your screen, select Go
7. Then, select Go to Folder
8. This will bring up a text box saying Go to Folder, click into this and paste the following: **/Library/Frameworks**
9. Hit enter, then you will see the Frameworks folder on your Macbook
10. Go back to the open SDL2 file and drag the framework into the Frameworks folder
11. For these next few steps, I am assuming that you have the compiler g++ installed as well as an IDE
12. Open a new file and copy in the code located here
13. You will see lots of errors, this is normal and to be expected from doing it this way because the install of SDL2 is not completely linked to your IDE.
14. Now, you are ready to compile
15. When compiling, use this format, replacing the file name with the file of your choosing **g++ <file_name>.cpp -o program -L /Library/Frameworks/SDL2.framework/Headers -F /Library/Frameworks/ -framework SDL2**
16. With our project, use the following line:
    **g++ visual.cpp -o program -L /Library/Frameworks/SDL2.framework/Headers -F /Library/Frameworks/ -framework SDL2**
17. Once compiled, enter the following line into your terminal with the correct input file name and the correct mode:
    **./program <file_name>.txt <mode>**
18. There are four text files in the GitHub to be tested
19. You can use the following modes:
    a. 0: Insertion Sort
    b. 1: Merge Sort
    c. 2: Quicksort

    d. 3: Radix Sort
20. Now, after running, a separate window will open, showing a sort occuring

## Installation of Timing Program

1. Install a python interpreter, and the g++ compiler
    a. If you do not have the python pandas package installed, install it with `pip install pandas`
2. Download the repo as a zip file from the Github Repository
3. Make any necessary changes within the file… e.g. change number of times to run each test, change test conditions…
4. Open the terminal
    a. `cd {CSC212REPO}/timing`
    b. `python main.py`
    c. Wait for the program to finish, and then you will find `.csv` files inside of a `./tests` directory, which reports the time that it took to sort each array.

## Link to Sort Visualizer Website

https://rbuckley-sorting.netlify.app/
https://github.com/rhbuckley/react-sort

1. Why a website? Well, for convenience. With C++ and STL, multiple dependencies need to be properly installed, and then configured, whereas with a website, you click the link, and that's it.
2. Though this was not written in C++, we still applied aspects of object oriented programming using Typescript classes. We ended up with creating a SortingAlgorithm class which was implemented by each algorithm. This allowed us to easily track things such as array accesses, swaps & record each step with different colors that we wanted to show the user.