Sorting Algorithms

Jake Murphy, Brendan Rice, Brian Larosiliere, Rushad Daruwalla

University of Rhode Island

CSC: 212

Professor Esteves

April 26, 2022

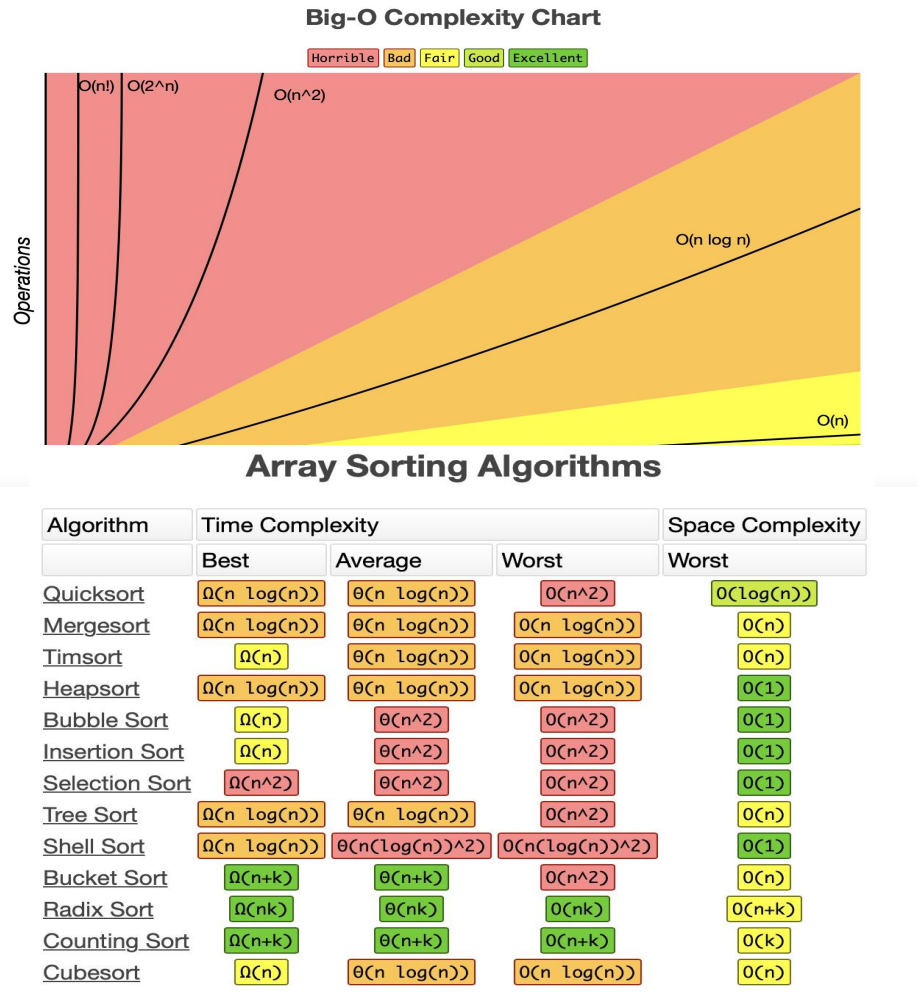
Abstract: Sorting algorithms are a fundamental part of computer science. However, there is not a plethora of efficient sorting algorithms even with them being so important. With arrays and vectors needing to be sorted often in many different programs, having efficient algorithms to do so is crucial. Especially when using arrays that contain millions of elements. This report goes in depth on four different sorting algorithms and how efficient they can be depending on the type and length of the array being sorted. When sorting arrays of smaller elements, all of the sorting algorithms perform efficiently. While this is the case with shorter arrays, longer arrays show the separation of greater algorithms. There are graphs to allow for visualization of these differences. All four of the sorting algorithms were tested, compared, and graphed to show how an efficient algorithm can allow for instantaneous sorting of arrays with thousands of elements.

Introduction

Sorting algorithms are used for all sorts of arrays in computer science. They are very important because they are used so often. When sorting a shorter array, maybe containing 1-100 elements, the difference between a fast sorting algorithm and a slow algorithm is not very important. While using a larger array, containing more than 100,000 items, sorting algorithms become much more important and knowing when to use the right one is just as important. Every sorting algorithm has its downsides. Whether it suffers time sorting an array that is already sorted or from an array that is unsorted, they all have their issues. The best way to pick a sorting algorithm is not to look at what it does best, but to look at what it does worst. Finding an algorithm that is efficient with almost every type of array, and still effective with the worst case scenario is crucial. Luckily, people have already developed extremely efficient sorting algorithms for us to use. They are widely available for anyone. Some sorting algorithms include insertion sort, quicksort, merge sort, and radix sort. These are the sorting algorithms in which we will be focusing on. Not all of them are the most efficient, however, they are widely known and do not take too much time to learn and understand.

Our project will be focusing on implementing the insertion sort, quicksort, merge sort, and radix sort algorithms in numerical arrays. There are graphs that serve as visuals to help describe the speed of the algorithms mathematically. The insertion sort graphs were generated by using the algorithms that we implemented in our source code. However, the graphs are strictly located on the presentation and report. By using timestamps at each iteration of the algorithm, we inserted our findings into Microsoft Excel and created a simple visual. In our source code, we will test each algorithm on each type of array five times and average that time. It will be tested on arrays of length 100 and then arrays of length 50,000. The user will then be able to choose the

type of array they want to use. Array types are sorted, partially sorted, reversed, and random. The user will be able to input the length of the array and the type of sorting algorithm they want to test. The output will provide the user the sorted array and the time it took the algorithm to sort it. They will then be met with the opening prompt, again, asking which type of array they want to use or if they would like to exit. Selecting the type of array and sorting algorithm is done by pressing the 1, 2, 3, or 4 keys on the keyboard while they are able to enter any length they want.



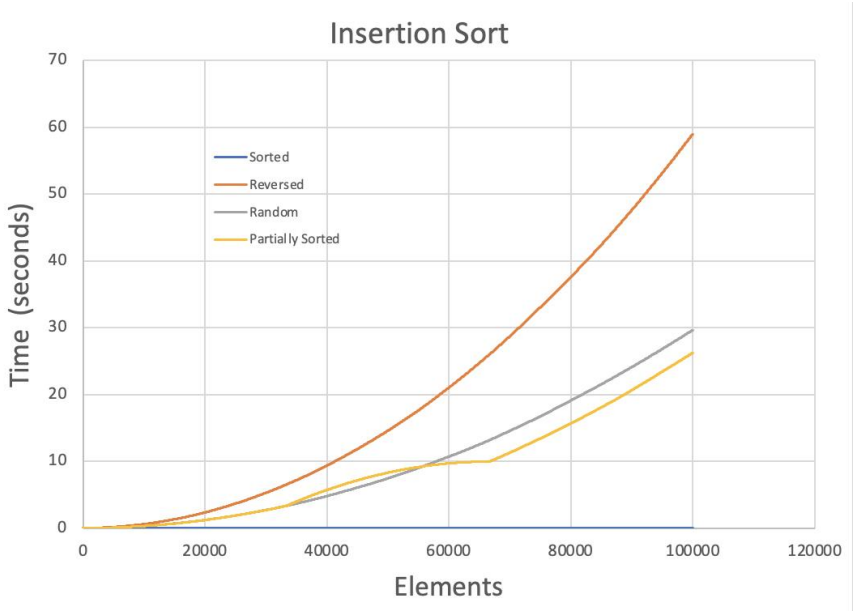
Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Graph of each time complexity function along with the time complexities of many different sorting algorithms including the four in this report.

1. Insertion Sort

Insertion sort is one of the least efficient sorting algorithms in coding. The algorithm works by looping through the array with nested nested loops. At each iteration, the algorithm checks if the current number in the array is less than the number at one element prior in the array. If the current number is less than the number one element prior, they are swapped. The swapping repeats until the number one element prior is greater than or equal to the current number or if it reaches the first element in the array. It continues looping through the rest of the array and doing this until it reaches the end of the array. Insertion sort performs poorly with every type of array except for one, a sorted array. It is not something that is very useful because the way to use it efficiently would be to check if the array is already sorted before calling insertion sort. The problem with this is if the array is already sorted, then there is no point in calling insertion sort because it would be O(2n), rather than O(n) for only checking if it is sorted. If the array is not sorted then it would be significantly more efficient to call something else.

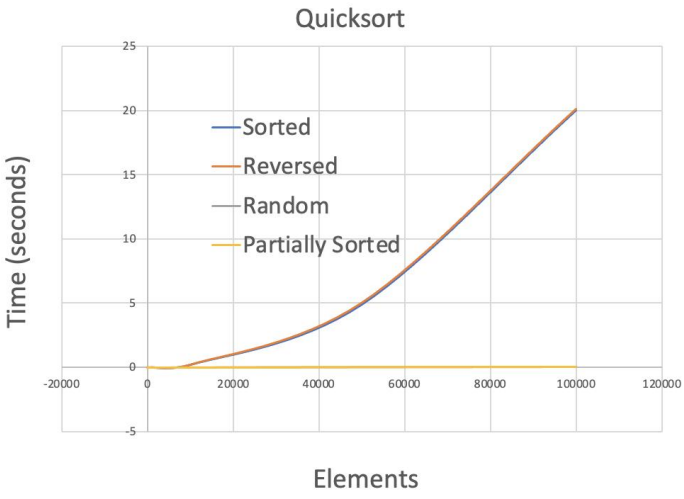
Insertion sort was tested on five attempts at sorting the different types of arrays (sorted, partially sorted, reversed, and randomized) and averaged over those five attempts. For arrays of length 100, it averaged 0.0000 seconds for a sorted array, 0.0001 seconds for a partially sorted array, 0.0001 seconds for a reversed



array, and 0.0001 seconds for a randomized array. For arrays of length 50,000, insertion sort averaged 0.0006 seconds on a sorted array, 6.4964 seconds on a partially sorted array, 14.6213 seconds on a reversed array, and 7.3317 seconds on a randomized array. Sorted arrays are the best cases for insertion sort while reversed arrays are the worst case for the algorithm. When using a small amount of data or sorted arrays, insertion is usable and efficient. However, on any other type of array, it is not.

2. Quicksort

In comparison to insertion sort, selection sort, and other inefficient sorting algorithms, quicksort is a great algorithm to use. However in the grand scheme of things, and looking at other more efficient algorithms such as radix sort which will always have a linear time complexity, quicksort can be viewed as a moderately efficient algorithm when sorting numbers. When sorting categorical data, it is considered one of the fastest sorting algorithms (Wang, 2016). Quicksort is at peak efficiency when the array has been scrambled into a random order. Quicksort is a recursive search algorithm, meaning that it calls itself until reaching a base case which ensures that the array is sorted properly (hi <= lo.) The algorithm splits the array into two separate arrays, and recursively solves itself using a partition function. This function



takes the low and high value of the subarray, and returns a pivot to be used to continue sorting.

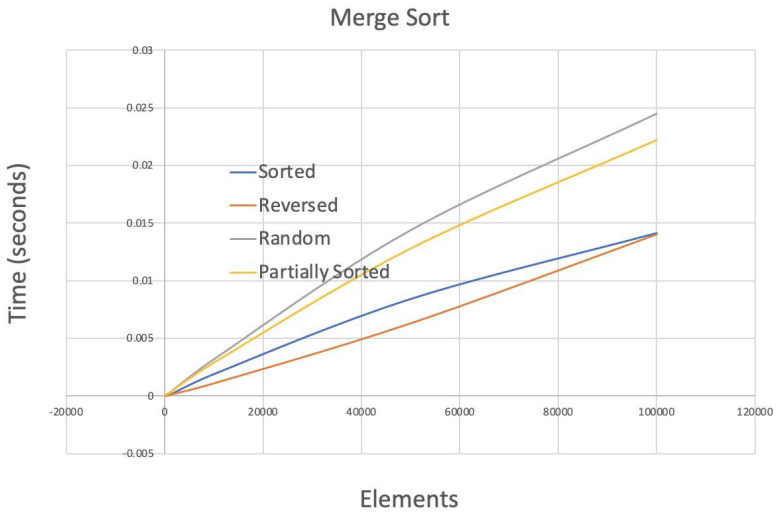
After being tested on the four different types of arrays of length 100 and 50,000 five times each and averaged over those five times, quicksort performed moderately. On arrays of length 100 it averaged 0.0001 seconds on sorted arrays, 0.0000 seconds on partially sorted arrays, 0.0001 seconds on reversed arrays, and 0.0000 seconds on randomized arrays. Looking at these four times on arrays of length 100, it was safe to assume that quicksort would probably suffer more time on sorted and reversed arrays of length 50,000 compared to partially sorted and randomized arrays. That is precisely what happened. Quicksort, tested on arrays with 50,000 elements, performed on average 4.8594 seconds on sorted arrays, 0.0177 seconds on partially sorted arrays, 4.9173 seconds on reversed arrays, and lastly 0.0101 seconds on randomized arrays. Quicksort is extremely powerful on partially sorted and randomized arrays, but suffers greatly on sorted and reversed arrays. This makes quicksort virtually unusable unless the array being sorted is shuffled.

3. Merge Sort

Merge sort is considered to be one of the more efficient sorting algorithms. This algorithm is running at best, worst, and average cases in O(n log n) time (“Quicksort vs Merge Sort”, 2021). This means that in almost all cases merge sort should be one of the preferred algorithms. With bigger arrays this will constantly run efficiently, although not the most efficient out of all common sorting algorithms. The speed of this algorithm is much more constant. As shown in the graph the time complexity between sorted, reversed, random, and partially sorted

arrays is basically the same sort of look. The space complexity is a little taxing as it requires two arrays to be passed in. One with the randomly sorted list, and another which acts as a placeholder in order to compare the children of the tree.

Essentially this is a sorting algorithm that takes the original array and is constantly breaking it into sub arrays of split lengths from the array before using recursion. The dividing of the arrays will stop recurring once the base case has been achieved where there is only one node (one number). Once the entire tree is created all the way down to individual items we start sorting. We compare the items side by side (in most cases, besides the bottom level this will be the array of numbers). We look at the children being compared to one another and go in chronological order because that portion of the array will already be sorted. So it compares the left most number in each array and then puts the lower value in first. It continues to exit each call slowly putting the split array back together until it is in order.

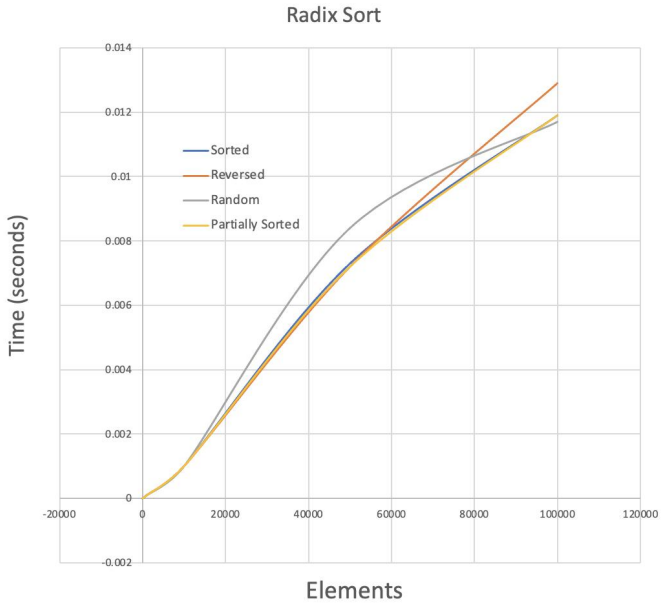


Merge sort was extremely efficient. On arrays of length 100, it performed at 0.0000 seconds on sorted and reversed arrays. On partially sorted and randomized arrays of the same length, it took 0.0001 seconds to sort. On 50,000 element arrays, merge sort took 0.0084 seconds on a sorted array, 0.0128 seconds on a partially sorted array, 0.0063 seconds on a reversed array,

and 0.0144 seconds on a randomized array. As seen from these results, merge sort is an efficient and consistent sorting algorithm.

4. Radix Sort

Radix sort is a digit by digit sorting algorithm that avoids comparisons and uses counting sort. Since it is a non-comparison algorithm, it can not be used to sort anything other than numbers. This sorting algorithm checks the digits of a number from left to right and sorts between iterations. Given an array, radix sort will focus on a single digit from right to left, counting and incrementing the occurrences of each digit. Using arrays holding the occurrences of each digit, the algorithm creates a prefix sum array that tells us how the original array should be sorted. After sorting the array based on the last digits of the numbers, the algorithm will repeat itself until it reaches the end (beginning) of the array's lengthiest number. At this point the array will be sorted. Radix sort has a time complexity of O(nk), with the k being the length of the longest digit. Thus, it is a linear sorting algorithm.



Like the other algorithms, radix sort was run using 4 different kinds of arrays. Those arrays are sorted, partially sorted, reverse sorted, and random. On all types of arrays of size 100, radix sort performed under 0.0001 of a second. It is extremely fast on a short array, similar to the other algorithms. For sorted, partially sorted, reverse sorted, and random arrays of length

50,000, radix sort performed in 0.1318, 0.1324, 0.1326, and 0.1322 seconds respectively. This makes radix sort one of the fastest sorting algorithms, beating most sorting algorithms.

Results

All of the four tested sorting algorithms have their benefits and disadvantages. Based on our testing of the algorithms the fastest and most consistent sorting algorithm is radix sort. As seen in the graph, radix sort is a linear algorithm no matter how many elements the array being sorted contains. It can only be used on numerical data, due to its non-comparison sorting. Merge sort is slightly faster than quicksort, but only when sorting large arrays ("Quicksort vs Merge Sort", 2021). Quicksort's best case is when the elements are shuffled prior to using it. Lastly, insertion sort is the slowest sorting algorithm out of these four. It performs linearly on an already sorted array, but it is not realistic to use because of its inefficiency on any other type of array.

Contributions

Jake and Brendan came up with the layout of the code using a class for the sorting functions. Jake implemented the layout and insertion sort algorithm. Brendan implemented the quicksort algorithm. Brian implemented the radix sort algorithm. He searched and found multiple sorting algorithms and found radix to be the most interesting one. Rushad implemented the merge sort algorithm and time complexity and also helped with the graphs for visualization. Jake wrote the majority of this report and wrote the insertion sort section. He also created a separate program printing timestamps to an output file to time insertion sort and radix sort algorithms to graph them and provide visualization. For merge sort and quick, he benchmarked them using the source code and graphed them for a similar visual. Brendan contributed to the quicksort section. Brian contributed to the radix sort section. Rushad created and presented the slides on merge sort.

Brendan created the majority of the presentation and wrote about the quicksort algorithm. Jake, Brian, and Rushad wrote about insertion sort, radix sort, and merge sort respectively.

References

"Know Thy Complexities!" *Big*, https://www.bigocheatsheet.com/.

"Merge Sort vs. Quicksort: Algorithm Performance Analysis." *Interview Kickstart*, https://www.interviewkickstart.com/learn/merge-sort-vs-quicksort-performance-analysis.

"Quicksort vs Merge Sort." *GeeksforGeeks*, 29 Apr. 2021, https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/.

Radix sort. (2013, September 2). GeeksforGeeks. https://www.geeksforgeeks.org/radix-sort/

Wang, Esha. "Quicksort - the Best Sorting Algorithm?" *Medium*, Human in a Machine World, 17 Apr. 2016, https://medium.com/human-in-a-machine-world/quicksort-the-best-sorting-algorithm-6ab461b5a9d0.