Profiling Research Paper

Analysis of Sorting Algorithms Using a Code Execution Profiler

Course: CSCI 2210-201, Spring 2018

Author: Cullen Diebold

Profiling Research Paper

Table of Contents

Introduction 2
The Problem 2
Solution Plan 3
Tasks Completed 3
Results and Conclusions 3
Summary 5
Appendix 1 – Data and Graphs 6
References 12

Profiling Research Paper

Introduction

Most standard computer users would not care to learn about how their computer allocates specified resources. Computer allocation is a major issue within the computer science community. Computer scientists are always trying to better their code by making their code more efficient, faster, and require less resources on any given machine. With the ever-growing world of IoT (Internet of Things) devices, programmers need to learn how to write the most efficient, and smallest, code as possible. To find the most effective and efficient code, programmers need to use analysis tools on their code to find algorithms for their respected scenarios. This research study has compiled multiple algorithms and analyzed them individually for their best results. From the research collected, we can note that there is no single algorithm perfect for every scenario, but there are certain algorithms that are better to use in some situations.

The Problem

The main issue with the vast array of algorithms held within a programmer's tool-box, is that not a single algorithm will work for every use-case. This means that the programmer must decide which algorithm to use and where to use them to make their code as efficient as possible. Some collections of data may require a Tag sort to be more effective on a timely manner without moving data, while other sets of data may require a Quick sort to move the data in a fast and timely manner. The main issue as stated above is that it is difficult for the programmer to decide which sorting algorithm to use in which situation. Even though a Shell sort may be one of the fastest sorting methods, it would not be recommended to use a Shell sort on small amount of data.

One-way programmers can combat this problem is by analyzing their code to measure the Big O, Big Omega, and Big Theta values of each algorithm. Big O notation represents the upper bounds of a specified algorithm. This means that the algorithm in question will never exceed the arc of a Big O variable. The arcs measured start at a constant value (Big O (1)) and travel up to a factorial value (Big O(n!)). Programmers would want to use the lowest Big O complexity that they

could possibly grasp without exceeding the boundaries of the program, due to the code being in the most effective state as possible.

By using specific measurements, programmers can solve this everlasting problem within the programming community. Programmers can measure the effectiveness with certain profiling tools to select the best algorithm for the best situation.

Solution Plan

The solution for the problem stated above is to decide which sorting algorithm to use in certain scenarios. To conduct this research, each scenario tested every algorithm on a unique data set. The conducted research used three sets of data to measure the effectiveness of each of the individual data sets. The three data sets used in this research experiment were: a sorted numeric list, a semi-sorted numeric list, and a randomized numeric list ranging from 1 – 10, 1 – 100, and finally 1 – 1000. Once the data was run through each of the individual sorting algorithms, the results will be compared in order to form a solution to our stated problem. The research conducted is to find which algorithm worked best for certain scenarios.

Tasks Completed

There were many trivial and complex tasks that needed to be completed during this research operation. The first main task was to understand which sorting algorithms were best for which scenarios. To complete this task, many sub-tasks were involved in this process. The primary task to focus on was the researching of nine different sorting algorithms: Sink Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Quick Median of Three Sort, Shell Sort, Counting Sort, and Radix Sort. To test each one of these algorithms, I was tasked with creating individual data structures to run alongside other algorithms to be able to compare their run-time with their competing sorting methods. Another task that was dependent on running the algorithms was creating two out of three randomized data structures (Lists) to use as data sets.

To create these randomized sets, I used the Random class from the .NET standard library to randomize each of the lists. The final task was to create reliable and valid research to be used to compare against other sorting algorithms to further my knowledge in sorting algorithms.

Results and Conclusions

The results found within this research study have been beneficial to my computing knowledge, and have peaked an interested within sorting algorithms. First, the most noticeable information in the below data is that there is no data for the Counting Sort method. During my process of writing the counting sort method, my computer ran out of virtual memory, causing an impossibility to further my research on Counting Sorts. This is due to the recursive nature found within some of the algorithms. That being stated, there are many more comparisons to reflect on within the data below. As noted by the below data and graphs, one can view that when using a sorting algorithm that compares low amounts of data (such as 10 integer values) that the following sorting methods are most effective: Insertion Sort, and Shell Sort. According to the data, each one of these sorts happens faster than 0.03 nanoseconds. On the other hand, the Merge sorting algorithm seemed to do extremely poorly with a small data set (averaging around 4.97 nanoseconds), but relatively well in larger data sets (such as 1000 integers). Another noting factor about these sets of data are that certain sorting methods exceed over 2000 method calls with a very low inclusive or exclusive time frame. As for data within the 100-integer values range, I noticed that Merge sorts, Radix Sorts, and occasionally Sink sorts all took around the same amount of inclusive/exclusive time to complete their tasks. The main winner according to the below data for the set of 100 integers is the Insertion Sort. The Insertion sort had the least number of method calls, while still sorting the data in a fraction of the time that it took other sorting algorithms to complete their sorts. Even though we only compared integer values from 10 to 1000, it is clear to state that Insertion sort was the most powerful sorting algorithm out of all the choices. This result shocked me, and lead me to believe that this was the most efficient sorting algorithm out of all the tested algorithms.

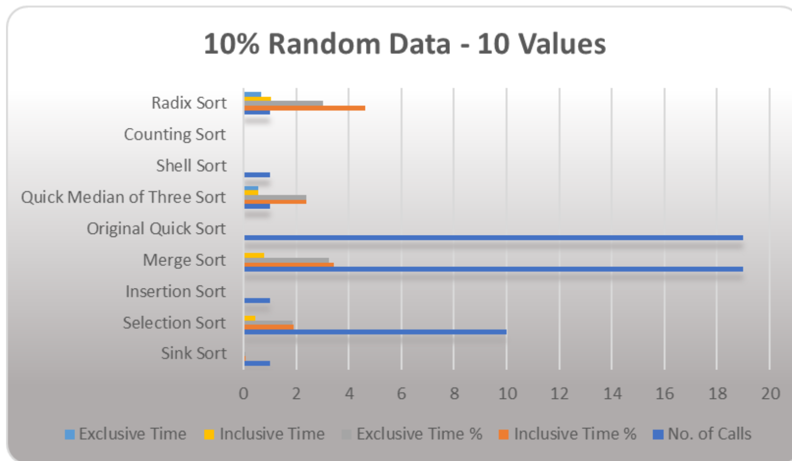
Another interesting result found within the collected data is that some sorts, such as the Sink sort, performed extremely well on small amounts of data, but when they were tested with large amounts of data became tremendously inefficient.

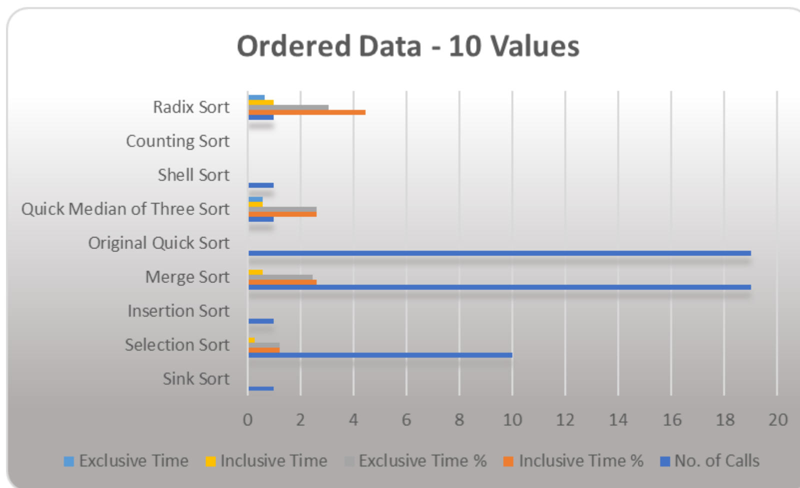
Summary

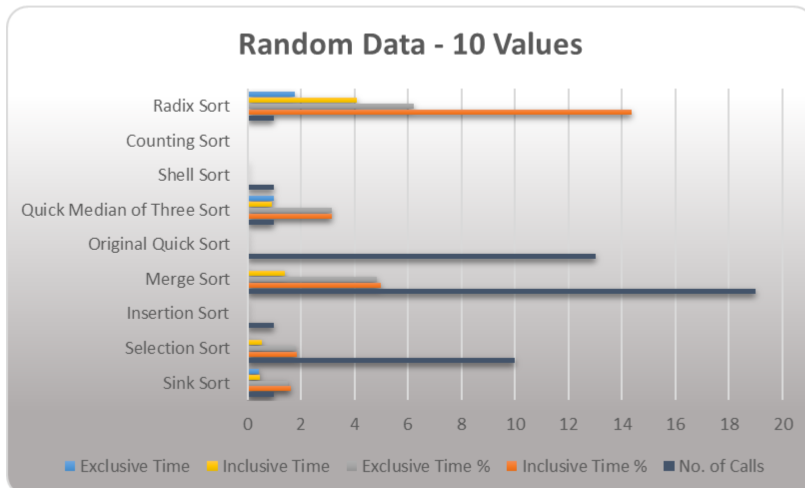
In conclusion, there is no best algorithm for every single scenario. There are many algorithms within the world of computing that are enormously efficient, and there are some algorithms that are dreadfully inefficient. It all depends on what type of data is being sorted, how it is being sorted, and the current state of the pre-sorted data. According to my studies, one can see that the Insertion sort was one of the most powerful sorting algorithms tested among the other eight algorithms. That is not to say that the other algorithms were less effective, but that the other algorithms may be better if tested on a different set of data. There were many algorithms that performed very sluggishly on large sets of data, such as, Quick Sort, Merge Sort, and Sink Sort. Once again it all depends upon the data being used, how the user wants to sort the data, and how the data is already sorted. After concluding my study, I can safely state that there is no golden-magical algorithm perfect for every sorting scenario.

Appendix 1 – Data and Graphs

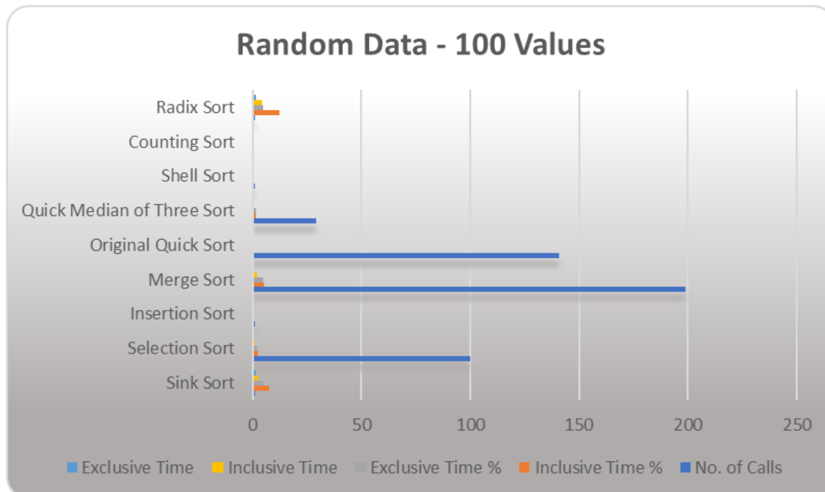
Data Collected from 10 integers

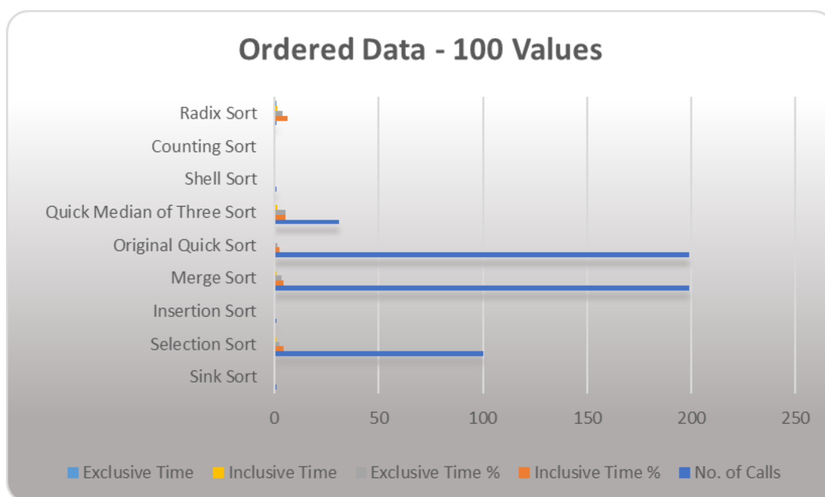


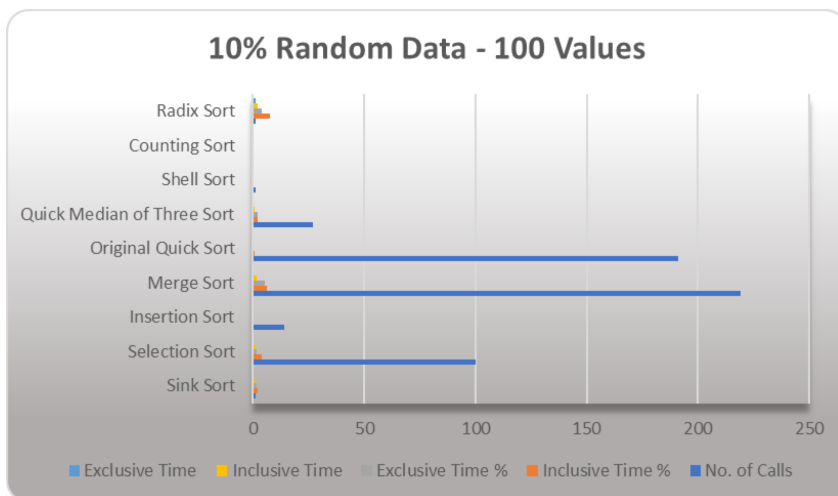




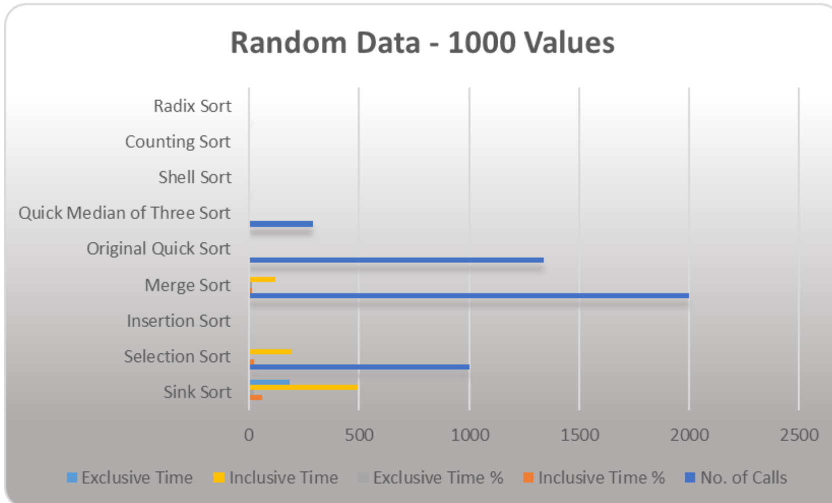
Data Collected from 100 integers

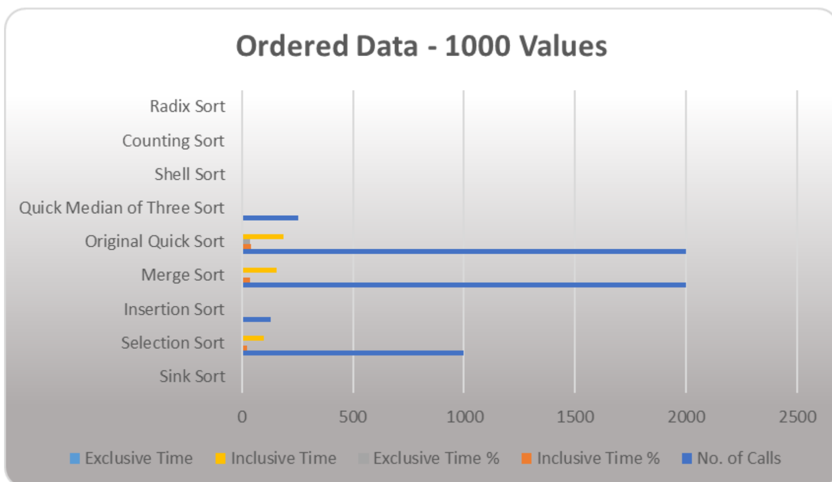


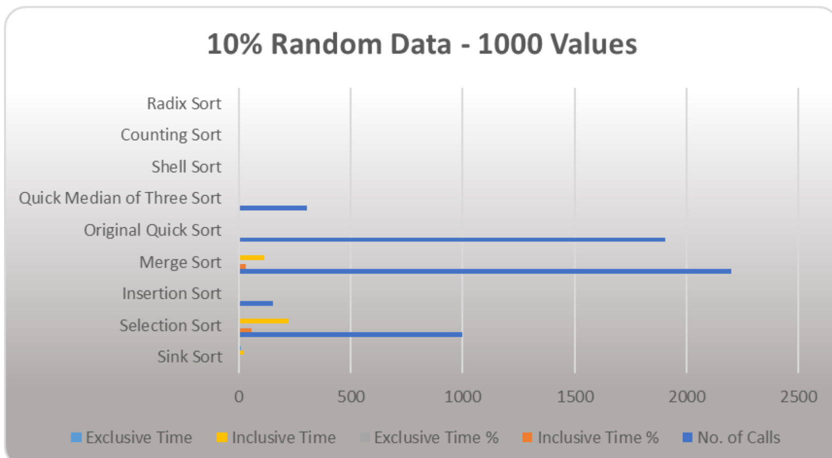




Data Collected from 1000 integers







Profiling Research Paper

Function Name	No. of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Avg Elapsed Inclusive Time	Avg Elapsed Exclusive Time	N Value	Order
Sink Sort	1	1.61	1.53	0.46	0.43	10	Random
Selection Sort	10	1.85	1.82	0.53	0.05	10	Random
Insertion Sort	1	0.03	0.02	0.01	0	10	Random
Merge Sort	19	4.97	4.83	1.41	0.07	10	Random
Original Quick Sort	13	0.02	0.01	0	0	10	Random
Quick Median of Three Sort	1	3.17	3.15	0.9	0.99	10	Random
Shell Sort	1	0.01	0	0	0	10	Random
Counting Sort						10	Random
Radix Sort	1	14.35	6.2	4.08	1.76	10	Random
Sink Sort	1	7.41	4.83	2.65	1.73	100	Random
Selection Sort	100	2.67	1.98	0.96	0.01	100	Random
Insertion Sort	1	0.02	0.01	0.01	0.01	100	Random
Merge Sort	199	5.35	4.61	1.91	0.01	100	Random
Original Quick Sort	141	0.23	0.2	0.13	0	100	Random
Quick Median of Three Sort	29	1.7	1.42	0.61	0.02	100	Random
Shell Sort	1	0.2	0.12	0.07	0.04	100	Random
Counting Sort						100	Random
Radix Sort	1	12.28	4.94	4.39	1.77	100	Random
Sink Sort	1	59.45	21.9	494.16	182.03	1000	Random
Selection Sort	1000	23.15	2.61	192.45	0.02	1000	Random
Insertion Sort	1	0	0	0.03	0.02	1000	Random
Merge Sort	1999	14.15	13.95	117.63	0.06	1000	Random
Original Quick Sort	1337	0.17	0.14	1.44	0	1000	Random
Quick Median of Three Sort	289	0.22	0.15	1.84	0	1000	Random
Shell Sort	1	0.05	0.03	0.43	0.27	1000	Random
Counting Sort						1000	Random
Radix Sort	1	0.52	0.29	4.36	2.44	1000	Random

The above data represents every sort method, when sorting an array of randomized integers in the values of 10 integers, 100, and 1000 integers. We can see that according to this graph, when dealing with smaller amounts of data that a merge sort seems to be less useful than other sorts. While on the other hand, when dealing with larger amounts of data, a merge sort seems like a more viable option. We can also note that the insertion sort method may be the best sorting algorithm to use when dealing with randomized integers.

Sink Sort	1	0.07	0.07	0.02	0.02	10	In Order
Selection Sort	10	1.23	1.22	0.27	0.03	10	In Order
Insertion Sort	1	0	0	0	0	10	In Order
Merge Sort	19	2.6	2.45	0.57	0.03	10	In Order
Original Quick Sort	19	0.02	0.02	0	0	10	In Order
Quick Median of Three Sort	1	2.62	2.61	0.58	0.58	10	In Order
Shell Sort	1	0	0	0	0	10	In Order
Counting Sort						10	In Order
Radix Sort	1	4.48	3.06	0.99	0.67	10	In Order
Sink Sort	1	0.15	0.13	0.04	0.03	100	In Order
Selection Sort	100	4.4	2.56	1.09	0.01	100	In Order
Insertion Sort	1	0.07	0.03	0.02	0.01	100	In Order
Merge Sort	199	4.25	3.25	1.05	0	100	In Order
Original Quick Sort	199	2.26	1.18	0.56	0	100	In Order
Quick Median of Three Sort	31	5.49	5.43	1.36	0.04	100	In Order
Shell Sort	1	0.09	0.04	0.02	0.01	100	In Order
Counting Sort						100	In Order
Radix Sort	1	6.01	4.03	1.48	0.99	100	In Order
Sink Sort	1	0.05	0.04	0.23	0.19	1000	In Order
Selection Sort	1000	21.19	4.28	97.63	0.02	1000	In Order
Insertion Sort	128	0.01	0.01	0	0	1000	In Order
Merge Sort	2,001	34.02	0.29	156.72	0	1000	In Order
Original Quick Sort	2001	40.43	36.95	186.25	0.09	1000	In Order
Quick Median of Three Sort	255	0.29	0.24	1.32	0	1000	In Order
Shell Sort	1	0.04	0.02	0.2	0.08	1000	In Order
Counting Sort						1000	In Order
Radix Sort	1	0.45	0.26	2.08	1.21	1000	In Order

The above data represents every sort method, when sorting an array of ordered integers in the values of 10 integers, 100, and 1000 integers. We can once again note that the insertion sort method is one of the fastest methods to use when dealing with ordered integers. Something interesting noted by the above data is that when a quick sort algorithm is used on ordered data, it seems to be useless. The magnitude of measurement for a quick sort upon ordered data is much larger than most other sorting algorithms.

Sink Sort	1	0.06	0.06	0.01	0.01	10	10% Random
Selection Sort	10	1.9	1.88	0.43	0.04	10	10% Random
Insertion Sort	1	0	0	0	0	10	10% Random
Merge Sort	19	3.43	3.24	0.77	0.04	10	10% Random
Original Quick Sort	19	0.04	0.02	0.01	0	10	10% Random
Quick Median of Three Sort	1	2.4	2.39	0.54	0.54	10	10% Random
Shell Sort	1	0.01	0	0	0	10	10% Random
Counting Sort						10	10% Random
Radix Sort	1	4.62	3.03	1.04	0.68	10	10% Random
Sink Sort	1	2.06	1.79	0.54	0.47	100	10% Random
Selection Sort	100	3.81	1.64	1	0	100	10% Random
Insertion Sort	14	0.07	0.04	0	0	100	10% Random
Merge Sort	219	6.24	5.39	1.64	0.01	100	10% Random
Original Quick Sort	191	0.61	0.46	0.16	0	100	10% Random
Quick Median of Three Sort	27	2.07	1.9	0.54	0.02	100	10% Random
Shell Sort	1	0.36	0.18	0.1	0.05	100	10% Random
Counting Sort						100	10% Random
Radix Sort	1	7.52	3.91	1.97	1.03	100	10% Random
Sink Sort	1	5.54	2.39	21.3	9.2	1000	10% Random
Selection Sort	1000	57.38	4.51	220.7	0.02	1000	10% Random
Insertion Sort	152	0.01	0.01	0	0	1000	10% Random
Merge Sort	2199	29.88	0.96	114.93	0	1000	10% Random
Original Quick Sort	1905	0.69	0.57	2.67	0	1000	10% Random
Quick Median of Three Sort	303	0.28	0.24	1.08	0	1000	10% Random
Shell Sort	1	0.08	0.04	0.32	0.16	1000	10% Random
Counting Sort						1000	10% Random
Radix Sort	1	0.97	0.53	3.72	2.02	1000	10% Random

The above data represents every sort method, when sorting an array of semi-ordered integers (90% sorted) in the values of 10 integers, 100, and 1000 integers. According to the above graph, we can see that when sorting semi-ordered lists of 10, a merge sort seems to be the least effective, while the sink sort and insertion sort are the most effective. Finally, the insertion sort trumps the data set with large numbers, due to its effective sorting patterns.

References

Bailes, Don. “Lecture 11 – Algorithm Analysis” East Tennessee State University. March 2018. PowerPoint presentation.

Bailes, Don. “Lecture 12 – Sorting Algorithms” East Tennessee State University. March 2018. PowerPoint presentation.