Empirical Comparison of Pessimal Sorting Algorithms: BozoSort vs. SlowSort

Awwal Ahmed, Natalie Lee, Elijah Olsson

Fall 2023

Abstract

This study sought to compare two lesser-known sorting algorithms: BozoSort and SlowSort. The primary interest was understanding their performance, given that neither is known for its speed. Tests were organized wherein both methods sorted different sizes of number lists and then recorded how long each method took. The results, which can be seen in Figure 1, showed clear differences. BozoSort had a hard time with number lists of 13 items, taking an average of two and a half hours. Sometimes, it even took close to four hours. On the other hand, SlowSort sorted the same size list in a very quick 0.8 milliseconds., making it about eleven million times faster than BozoSort. Testing BozoSort with larger lists than 13 items was not feasible due to its lengthy sort times, but SlowSort continued to be examined. When SlowSort dealt with lists close to 250 items, it began to slow down, taking almost twenty-five minutes for some of the bigger lists and over six hours for the largest ones.

From our tests, it is clear that SlowSort, even with its name, is much faster than BozoSort. This shows there can be big differences in how well different sorting methods work, especially as the size of the lists increases. While neither method might be the first choice for everyday tasks, this study offers an interesting look into the world of sorting methods.

Contents

1 Introduction

Sorting data is a problem fundamental to the field of computer science. Sorting algorithms play a pivotal role in determining how efficiently data can be organized and processed. There exist numerous sorting algorithms that have been streamlined for better performance. Most of us are familiar with algorithms like QuickSort or MergeSort. However, not all algorithms are created equal. On the flip side of these celebrated algorithms lie algorithms that are, to put it mildly, less than optimal named pessimal algorithms. This study delves into a run-time comparison of two such appallingly inefficient algorithms in the realm of sorting: BozoSort and SlowSort.

BozoSort and SlowSort break the mold of striving for optimization by implementing a level of unpredictability or inefficiency that goes against traditional sorting methods. The peculiar nature of these algorithms gives rise to several hindrances in their performance time and makes them interesting to compare. First, the non-deterministic behavior of BozoSort makes predicting its performance less straightforward than its conventional sorting algorithm counterparts. Secondly, the multiply and surrender method, a parody of the divide and conquer technique, may prove SlowSort slower than the randomized BozoSort. Ultimately, the aim to answer the question: **How does the average time complexity of BozoSort compare to SlowSort?** To determine this, a comparative analysis of the two algorithms will be preformed by running experiments in order to measure their run times. After implementing the two algorithms, testing data sets will be synthesized into varying sizes and run them on the algorithms.

There are inherent challenges in the study and comparison of sorting algorithms and even more when the randomization aspect of BozoSort is introduced. Capturing an accurate representation of the experimental run time of a sorting algorithm is largely dependent on the input data given. A variety of sizes of data inputs of representative "unsorted" cases will need to be managed. The data inputs will need to be reasonably limited to match the computational power that is available. Providing these algorithms with large input data sets may result in infinitely long run times on even the most powerful of computers, especially in the case of BozoSort. There will be a fine line between testing a large variety of inputs and getting enough usable data. The issues surrounding the collection of run time data using the metric of time may propel future work of additional theoretical analysis.

The motivation behind this research is multifaceted. Primarily, the aim is to provide results and comparisons for the experimental run times of the two algorithms. By putting these algorithms to the test in a controlled environment, the aim is to conclusively illustrate the magnitude of their inefficiencies. Additionally, understanding the intricacies and performance of these 'misfit' algorithms provides insight into what not to do in algorithm design. While these algorithms present as mere parodies, assessing their performance could shed light on the principles that differentiate effective algorithms from ineffective ones.

The comparison of sorting algorithms is not a novel undertaking. However,

to the best of our knowledge, no one has ever undertaken a comparative analysis of these particular algorithms against one another. This study thus seeks to fill this gap, offering a detailed insight grounded in empirical comparison evidence.

The significance of this exploration goes beyond the mere curiosity of how bad can an algorithm get. It serves as a teaching tool for budding computer scientists by providing an example of algorithmic techniques to avoid. As well as showcasing the run time consequences that result from poor algorithmic design.

In the subsequent sections, the two parody sorting algorithms: BozoSort and SlowSort, will be compared and analyzed. Section 2 will provide the background and literature review of the areas of sorting algorithms and pessimal algorithms. Section 3 will describe the experiment design and explain the testing environment will be used to ensure unbiased results. Section 4 will visualize the findings from the data collection tests in graphs and tables. Section 5 will describe the data and will include the resulting comparison analysis. Section 6 will conclude the work by summarizing the findings and including any proposed future research.

2 Background

2.1 Introduction to Sorting and Pessimal Algorithms

Sorting Algorithms

A sorting algorithm is an algorithm that arranges the elements of a list in a certain order, commonly in increasing or decreasing order [1]. These algorithms are used to make the processes of searching, inserting, and deleting in a list easier. To measure the performance of a sorting algorithm, a holistic method of analysis is used. The qualities of stability, adaptivity, and time and space complexity are all taken into consideration when analyzing a sorting algorithm. Stability refers to the sorting algorithms' ability to maintain the relative order of two identical list entries after sorting [1]. Adaptivity refers to the sorting algorithms' ability to sort lists that are mostly in order faster than lists that are extremely unordered [1]. Time and space complexity are measured by how much time or respectively how much memory an algorithm takes to complete depending on the size of the input data that is given [1].

Measuring the time complexity of a sorting algorithm is a popular, quantitative way to compare two algorithms. Conducting empirical comparison, or comparison based on system run time, must be done in a controlled manner as variability in machines can skew data collection [2].

Many well-performing sorting algorithms have already been discovered and some popular ones include Quick Sort, Selections Sort, and Insertion Sort. The field of research surrounding sorting algorithms is ever-growing and important as optimal sorting algorithms are needed in computing fields ranging from database management to artificial intelligence [3].

Pessimal Algorithms

The study of algorithms is most commonly done with the goal of finding the most optimal or the best algorithm for solving a problem. However, there exists a subsection of algorithms that aims to be the worst at solving a problem. Pessimal algorithms, a term coined in 1984 by Andrei Broder and Jorge Stolfi, are parody algorithms created for the amusement of observing their shortcomings [4]. Pessimal algorithms approach problems ranging from searching to sorting with no urgency to find a solution and even their best-case scenario time complexities try to be unfortunately large [5]. These algorithms are described as perversely awful and often include a randomized element [4].

The quintessential pessimal algorithm is BogoSort, also known as Stupid Sort [4]. The formula of BogoSort is constructed based on the idea of shuffle and pray. Similar to shaking the elements of a list in a bag and dumping them out until they miraculously appear in order, a permutation of the list is randomly generated until the list is sorted. Jokingly, if a program contains a bad algorithm, one can say they leveraged the BogoSort algorithm [4].

The time complexity of BogoSort has a lower bound on the average input case of $\Omega(n \cdot n!)$ [4]. Accomplishing a sorted list as the input list size grows becomes increasingly more costly and eventually improbable.

BozoSort and SlowSort are lesser-known pessimal algorithms and are described in more detail in the next subsection.

2.2 Background on BozoSort and SlowSort

BozoSort

A pseudocode representation of the BozoSort implementation reads as follows:

Algorithm 1 BozoSort

```
procedure BozoSort(array)
    while array is unsorted do
        index1 ← randomly selected index
        index2 ← randomly selected index
        Swap(array[index1], array[index2])
    end while
end procedure
```

The BozoSort algorithm checks to see if the elements of a list are sorted and if they are not it randomly selects two indexes of the list and swaps them [4]. The algorithm continues this process until the list is sorted [4]. As this sorting algorithm is closely related to the most popular pessimal algorithm, BogoSort, one formal analysis has been conducted on it which delved into several possible variations of the implementation and a theoretical time complexity analysis [4].

This work suggests that the theoretical time complexity of the BozoSort algorithm is $O(n!)$ in the average case, emphasizing its inefficiency in sorting

arrays of length n [4].

SlowSort

A pseudocode representation of the SlowSort implementation reads as follows:

Algorithm 2 SlowSort

```
procedure SlowSort(array, start_idx, end_idx)
    if start_idx ≥ end_idx then
        return
    end if
    middle_idx ← ⌊(start_idx+end_idx)/2⌋
    SlowSort(array, start_idx, middle_idx)
    SlowSort(array, middle_idx + 1, end_idx)
    if array[end_idx] < array[middle_idx] then
        swap (array, end_idx, middle_idx)
    end if
    SlowSort(array, start_idx, end_idx − 1)
end procedure
```

The SlowSort algorithm uses the multiply and surrender paradigm, a directly comedic commentary on the divide and conquer strategy used by championed algorithms such as QuickSort [5]. SlowSort tackles sorting by recursively breaking down the problem until the sub-problem is menial (a sub-array of size one) and then sorting the sub-arrays through comparison and swapping [5].

The SlowSort algorithm was reviewed formally by the creators of Pessimal Algorithms where they dissected the implementation and provided theoretical run time analysis [5]. This work suggests that the theoretical lower asymptotic bound of the SlowSort algorithm is $\Omega\left(n^{\frac{n \log n}{2+\varepsilon}}\right)$ for any $\epsilon > 0$ [5].

2.3 Summary and Motivation

Sorting algorithms are fundamental to the field of computer science as they make data, one of today's world's most valuable assets, more usable [2]. Finding the perfect sorting algorithm is an ongoing endeavor. If anything, the range of sorting algorithms that have been analyzed has shown that some sorting algorithms champion specific use cases over others.

Pessimal algorithms act as a road map of what not to do when designing an algorithm. They provide comedic value in their terrible run times and limited value in real-life applications. The motivation to study pessimal algorithms is their educational value to beginner programmers. Beginners in the field of algorithm analysis can easily find the weak points of these algorithms and point out optimization steps that can be taken, making them great learning tools [4]. They also can humor anyone from beginners to experts in the field.

The algorithms covered in this paper are variants of pessimal algorithms. BozoSort is the slightly "optimized" sister of BogoSort, while still maintaining the pessimal algorithm quality of randomization. SlowSort showcases its sorting inefficiency through the excess recursive breakdown of a list. The current literature is missing an in-depth empirical analysis and comparison of BozoSort and SlowSort.

While the inherent uselessness of the two sorting algorithms in real-life applications is clear, the consistency and validity of the formal research process prevail. Fundamentally, our work highlights this phenomenon.

3 Methodology

3.1 Data Generation

To collect strong and compelling data for this research, arrays (or Python lists) will be the only applied data structure used during experimentation. To avoid added complexity, these arrays will be populated with values of integer type. As experimentation progresses, the challenge for these algorithms will increase.

The commonly used Python module, `random`, will be used to randomly generate integer arrays, with values ranging from 0 to 100. Previously generated data from different experiment runs will never be reused. After each successful execution of each algorithm, the program will use a different array, larger in size and populated with a different set of numbers than the last. Smaller data sets are used in particular due to the inefficient nature of these algorithms. There will be thirty-three sets of data, all shared by each algorithm, sizes being 1 - 15, 25, 30, 40, 50, 75, 100, 125, 150, 175, 200, 205, 210, 220, 225, 250, 300, 400, and 500.

3.2 Experiment Design

As previously mentioned, Python is the primary programming language used during the software development of this research. Python was chosen due to its versatility, extensive library support, and ease of readability and maintenance, enabling efficient code development. All code developed for research will be done in one of the many integrated development environments (Virtual Studio Code, Eclipse).

Within the source code that implements this experiment, Python scripts will be defined for both sorting algorithms. This allows a wrapper program, like a Bash script, to simply call one of the algorithms whenever needed. For every supplied dataset, each algorithm will sort the same dataset ten times, to collect meaningful data of each run.

Data collected during any single run will be put through a Python `pandas` data frame for organization. The most crucial data point that will be collected during any given run will be the elapsed time for each algorithm to complete with its supplied data array. The time delta is automatically written to a Comma

Separated Values (CSV) file. After which, another Python script is called to accurately plot that data into a graph to visualize these research findings using Python's `matplotlib` and `seaborn` modules.

3.3 Testing Environment

Building a solid, foul-proof development environment for this research is the key element in collecting truthful data. Testing anomalies found during experimentation may cause extended and unwanted development, which could also potentially lead data conclusions in the wrong direction.

To combat these concerns, the sorting algorithms will be tested on a local Virtual Machine (VM) running on a Proxmox Hypervisor. The VM instance will be powered by 3 CPU cores from an AMD Ryzen 5 2600 at 3.4GHz clock speed. Matched with this power, will be eight allocated gigabytes of DDR4 RAM and ninety-six gigabytes of solid state storage. The same VM instance will be running a graphical desktop operating system, Debian 11. The Debian operating system meets the cost, system capabilities, and functionalities needed to perform the experiment. Included with Debian is the advanced package repository tool, `apt`, which is used to install prerequisite software packages for a functional testing environment. After the startup of Debian, the work environment will be populated with many packages to aid in the development and operation of the experiment.

Due to the nature of this experiment, collaboration is an essential role for proper development, and to successfully develop software with more than one person, version control is a requisite. Management of software developed for this project will be done through contributions to a GitHub repository [6]. The use of Git allows easy access and usability of the software developed for this research.

3.4 Experimental Variables

The experimental variables for this research are defined as follows:

Independent Variable

The independent variable for this research will be the algorithm tested, BozoSort or SlowSort.

Dependent Variables

The dependent variable for this research will be the elapsed time it takes for one instance of an algorithm to successfully complete.

Control Variable

The control variable for this research will be the size of the randomly generated array.

3.5 Experimental Procedure

Stage One (Boot-Up and Environment Setup)

1. Assuming the OS is already configured and log-in credentials are provided, turn on and boot-up Virtual Machine into Debian and SSH to log-in.
2. Next, log-in as root by typing `su -`. Enter password.
3. As root, type `apt-get upgrade`. Type "Y", then ¡ENTER¿.
4. Using the `git clone <repository-url` command, clone the GitHub repository that contains the experimental software for this research. [6]
5. Next, change to the `CS474/pessimal/code` working directory.

Stage Two (Experiment Setup and Software Execution)

1. With Stage One completed, verify all implementation files are present. There must be Bash script called `execute_algorithm_analysis.sh` to run the experiment.
2. Run the script by typing `nohup ./execute_algorithm_analysis.sh` into the command line. `nohup` is a command that will not send a HANGUP signal to the Bash script if the user were to log out of the shell.
3. After the script finishes, a line graph of the collected data will be present in the `CS474/pessimal/plots` directory.

With all of these steps completed in order, the experiment is considered to be finished.

4 Results

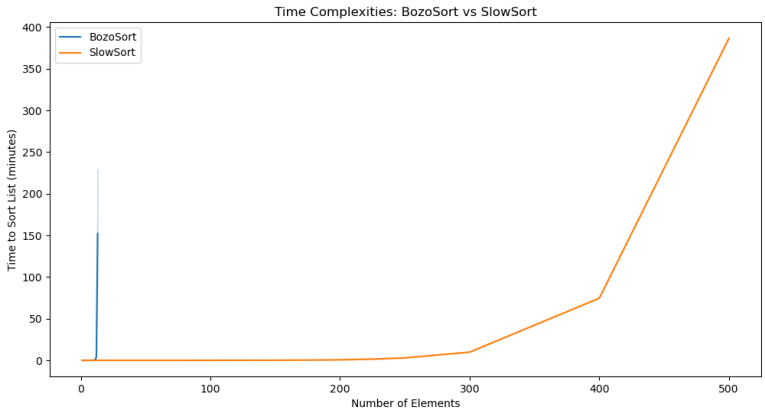

Figure 1: Array sizes of 1 - 15, 25, 30, 40, 50, 75, 100, 125, 150, 175, 200, 205, 210, 220, 225, 250, 300, 400, and 500 shared by both BozoSort and SlowSort.

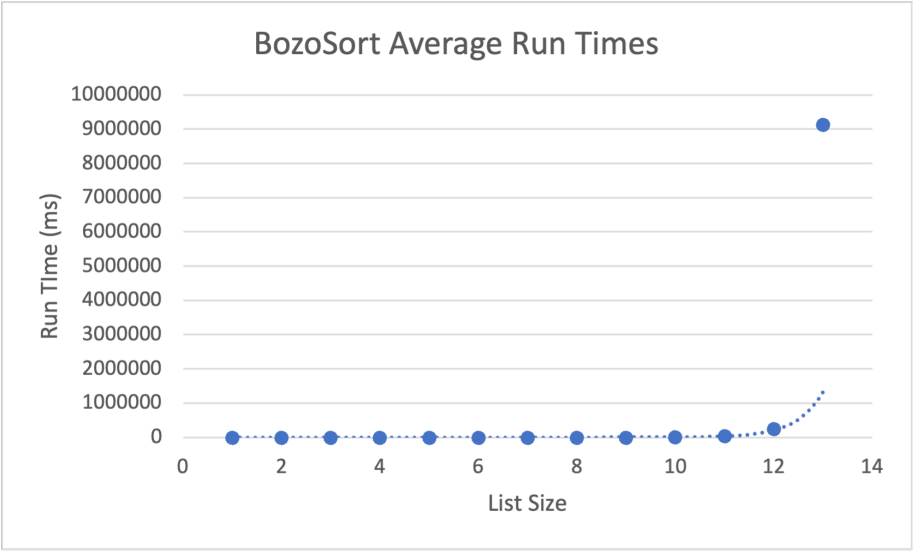

Figure 2: A plot of the averages of the run times for BozoSort.

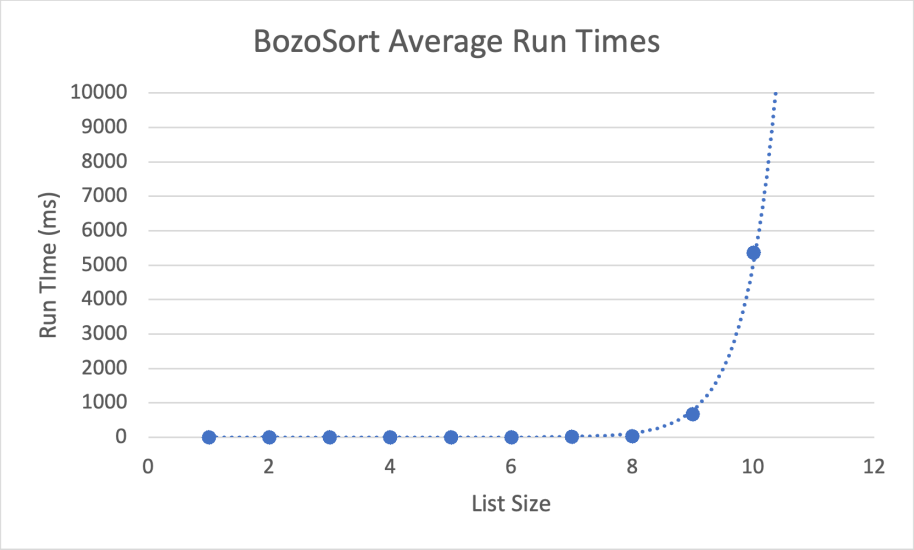

Figure 3: A scaled version of Figure 2.

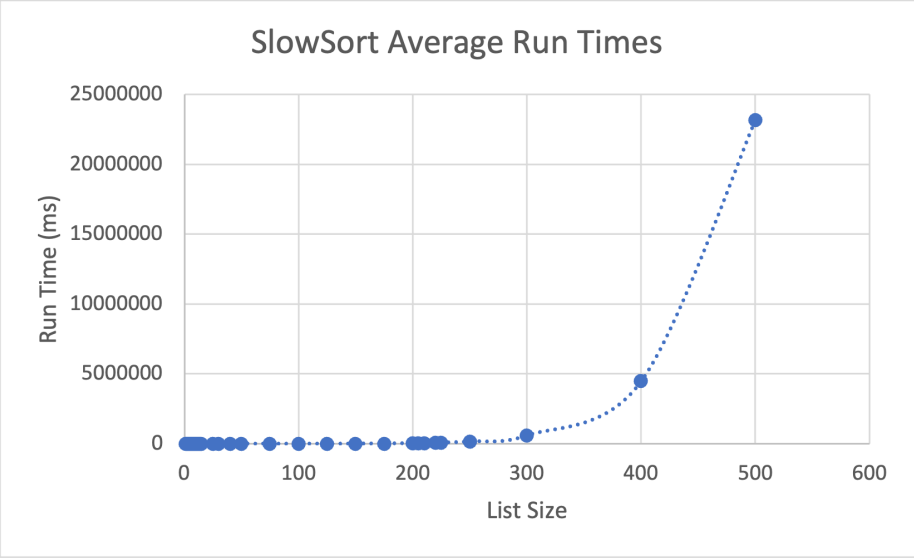

Figure 4: A plot of the averages of the run times for SlowSort.

5 Discussion

After our experimental trials, the question of which sorting algorithm is quicker was quickly solved. Reflecting their theoretical run times, these algorithms became very slow, very fast. As presented in Figure 1, it was found that BozoSort had experienced lots of trouble with arrays of size 13. On average, an array of this size took BozoSort roughly two and a half hours to sort. With this same array, it took SlowSort an average of 0.8 milliseconds to sort, roughly eleven million times faster than BozoSort. In Figure 1, transparency on the blue line indicates outliers, meaning that a list of size 13 did not always take two and a half hours to sort, but instead could take a lot longer. One execution of BozoSort with an array of size 13 took nearly four hours to complete. Unfortunately, due to the computational resource constraints of this research, the analysis of BozoSort did not make it past the array of size 13. Despite our predictions that the heuristic elements of BozoSort may lead to faster empirical run times, the theoretical factorial time complexity of the algorithm endured.

On the other end of Figure 1, we have SlowSort, which clearly is the faster pessimal sorting algorithm. When SlowSort reaches array size 250, there is a significant spike in sorting time. Increasing the array size from 250 to around 300 shows a big jump in time from under a minute to almost twenty-five minutes to sort. From there, the time it takes sort exponentially grows, reaching execution times of six and half hours.

After doing our experimentation, we are confident that no further analysis is needed to show that SlowSort is the better candidate for sorting integer arrays.

6 Conclusion

Our research embarked on a quest to compare two lesser-known sorting algorithms: BozoSort and SlowSort. We wanted to determine their empirical run times, even though they are not renowned for speed. To understand these algorithms in action, we conducted a series of experimental trials that involved sorting arrays of varying sizes and analyzing the time each algorithm took. Our results, depicted in Figure 1, painted a clear picture of the performance of each algorithm. BozoSort struggled immensely with arrays of size 13, taking on average a staggering two and a half hours to complete its sorting task.

In contrast, SlowSort, when handling the same array size, completed the task in a swift 0.8 milliseconds, proving to be approximately eleven million times faster than BozoSort. While BozoSort's performance stagnated due to our computational resource constraints, preventing us from testing it with arrays larger than 13, SlowSort continued to be the subject of our exploration. SlowSort, while being the superior of the two, did exhibit a sharp increase in sorting time as the array size neared 250. Beyond this size, its performance deteriorated rapidly, taking almost twenty-five minutes to sort arrays just over 250, and eventually demanding over six and a half hours for larger ones.

Given our findings, it's evident that SlowSort outperforms BozoSort signifi-

cantly. The data suggests a profound difference in their efficiencies, with SlowSort clearly standing out as the more viable pessimal sorting algorithm when dealing with integer arrays. Although both algorithms have their limitations, and neither is ideal for practical real-world applications, our experiments have provided valuable insights into the landscape of pessimal sorting algorithms, emphasizing the vast differences that can exist even among these infrequently employed methods. Looking ahead, it's intriguing to explore the reasons behind such distinct differences in sorting times. Future research could involve comparing these methods with other less common sorting algorithms, and investigating if there are specific situations where they might perform more effectively than anticipated.

References

[1] Y. Chauhan and A. Duggal, "Different sorting algorithms comparison based upon the time complexity," *International Journal of Research and Analytical Reviews(IJRAR)*, vol. 07, pp. 114–121, 2020.

[2] N. Akhter, M. Idrees, and F. ur Rehman, "Sorting algorithms – a comparative study," *International Journal of Computer Science and Information Security*, vol. 14, pp. 930–936, 12 2016.

[3] A. Zutshi and D. Goswami, "Systematic review and exploration of new avenues for sorting algorithm," *International Journal of Information Management Data Insights*, vol. 1, no. 2, p. 100042, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667096821000355

[4] H. Gruber, M. Holzer, and O. Ruepp, "Sorting the slow way: An analysis of perversely awful randomized sorting algorithms," in *Fun with Algorithms*, P. Crescenzi, G. Prencipe, and G. Pucci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 183–197.

[5] A. Broder and J. Stolfi, "Pessimal algorithms and simplexity analysis," *ACM SIGACT News*, vol. 16, 08 2000.

[6] "Github repository for pessimal algorithms," https://github.com/natalieswork/CS474/tree/main/pessimal.