Empirical Comparison of Pessimal Sorting Algorithms: BozoSort vs. SleepSort

Awwal Ahmed, Natalie Lee, Elijah Olsson

Fall 2023

Contents

1 Introduction **2**

2 Background **3**

2.1 Introduction to Sorting and Pessimal Algorithms 3

2.2 Background on BozoSort and SleepSort 4

2.3 Summary and Motivation 5

3 Methodology **6**

3.1 Data Generation 6

3.2 Experiment Design 6

3.3 Testing Environment 7

3.4 Experimental Variables 7

3.5 Experimental Procedure 8

Bibliography **9**

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 SleepSort.

BozoSort and SleepSort 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 purposeful halting of execution for SleepSort makes it unclear if this tactic will make it slower than the randomized BozoSort. Ultimately, we aim to answer the question: **How does the average time complexity of BozoSort compare to SleepSort for small and large input data sets?** To determine this will will perform a comparative analysis of the two algorithms by running experiments in order to measure their run times. After implementing the two algorithms, we will synthesize testing data sets of varying sizes and run them on the algorithms.

There are inherent challenges in the study and comparison of sorting algorithms and even more when we introduce the randomization aspect of BozoSort. Capturing an accurate representation of the experimental run time of a sorting algorithm is largely dependent on the input data given. We will need to manage a variety of sizes of data inputs of representative "unsorted" cases. We will also need to reasonably limit our data inputs to match the computational power that we have access to. Providing these algorithms with too large of 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, we will provide results and comparisons for the experimental run times of the two algorithms. By putting these algorithms to the test in a controlled environment, we aim 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. Our 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, we will analyze and compare two parody sorting algorithms: BozoSort and SleepSort. Section 2 will provide the background and literature review of the areas of sorting algorithms and pessimal algorithms. Section 3 will describe our methodology procedure and explain the testing environment we will use to ensure unbiased results. Section 4 will visualize our findings from our data collection tests in graphs and tables. Section 5 will describe our data and will include our resulting comparison analysis. Section 6 will conclude our work by summarizing our 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 Bubble 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 SleepSort are lesser-known pessimal algorithms and are described in more detail in the next subsection.

2.2 Background on BozoSort and SleepSort

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 [4].

SleepSort

A pseudocode representation of the SleepSort implementation reads as follows:

Algorithm 2 SleepSort

```
procedure SleepSort(array)
    for element in array do
        Sleep for element seconds
        Print element
    end for
end procedure
```

The SleepSort algorithm sorts lists by creating threads for each element of the input array [6]. Each thread sleeps for the amount of time proportional to its element's value and then prints [6]. This results in a list where elements of a lower value are printed first and, in turn, sorted in the list [6]. This algorithm lacks any formal literature on its analysis and is more popularly examined on community forums such as Stack Overflow or Reddit.

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. SleepSort sorts a list by iterating over its elements and printing values after making them "sleep" or waiting for the number of seconds of the element. The current literature is missing an in-depth analysis and comparison of BozoSort and SleepSort.

While the inherent uselessness of the two sorting algorithms is clear, the consistency and validity of the formal research process to extend to even the most menial of tasks prevails. Fundamentally, our work highlights this phenomenon.

3 Methodology

3.1 Data Generation

To collect strong and compelling data for this research, arrays (or lists) will be the only applied data structure used during experimentation. To avoid complexity, these arrays will be populated with values of integer type. As experimentation progresses, the challenge for these algorithms will become exponentially larger. The size of the array will double after each successful execution of each algorithm.

The commonly used Python module, *random*, will be used to randomly generate integer number arrays, with values ranging from 1 to 999. Previously generated data will never be reused. After each successful execution of each algorithm, the program will create new arrays, larger in size and populated with a different set of numbers. This is the general logical structure for each sequence of iterations:

Sequence 0: Run the program x times for sample size n.

Sequence 1: Run the program x times for sample size $2n$.

Sequence 2: Run the program x times for sample size $2n$.

Sequence 3: Run the program x times for sample size $4n$.

...

Sequence z: Run the program x times for sample size $(2^z)n$.

3.2 Experiment Design

As previously mentioned, Python is the primary programming language used during 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 Python file that contains the logical implementation, a function (or method) will be defined for both sorting algorithms. This allows a higher-level program to simply call the algorithm whenever needed. The only parameter needed for each function will be the generated input data array.

All experimental tests will be run through a wrapper program that uses the Python module, *pandas*, for data collection and organization. The most crucial data point that will be collected during any given test will be the elapsed time for each algorithm to complete with its supplied data array. Additionally, the amount of memory used for a successful execution of the called algorithm will be collected. These data points will then be accurately plotted on graphs to visualize research findings using Python's *matplotlib* module.

3.3 Testing Environment

Building a solid, foul-proof infrastructure 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 2 CPU cores from an AMD Ryzen 5 2600 at 3.4GHz clock speed. Matched with this power, will be 4 allocated gigabytes of DDR4 RAM and 32 gigabytes of solid state storage. The same VM instance will be running a non-GUI, console-only operating system, Debian 12.1. The Debian operating system meets the cost, system capabilities, and functionalities needed to perform the experiments. 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 testing environment will be populated with OpenSSH, Vim, and the latest version of Python3.

Due to the nature of these experiments, collaboration is an essential role for proper development, and to successfully develop software with more than one person, version control is needed. Management of software developed for this project will be done through contributions to a GitHub repository. 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 SleepSort.

Dependent Variables

The dependent variable for this research will be the elapsed time it takes for one instance of an algorithm to successfully complete. Additionally, the amount of memory used will be recorded as a dependent variable.

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 12.1 and SSH to log-in.

2. Using the *git* command, clone the GitHub repository that contains the experimental software for this research.

3. With the clonetree copied onto the VM, run the *setup.ksh* script to download the necessary environment packages.

Stage Two (Experiment Setup and Software Testing)

1. With Stage One completed, verify all implementation files are present in the clonetree. There will be a file for each algorithm and a wrapper program that will invoke the algorithmic functions.

2. Run the wrapper program in "TESTING" mode, ensuring that the program successfully completes and data is recorded, plotted, and displayed to the user.

3. Once the wrapper program exits with no errors in "TESTING" mode, check to see if other processes are running. If nothing else is running, move to Stage Three.

Stage Three (Experiment)

1. With all other stages completed, run the wrapper program in "PRODUCTION" mode.

2. Observe any and all console output, looking for any run-time errors.

3. After the wrapper program finishes in "PRODUCTION" mode, secure copy all graphical charts and data files to local machine for analysis.

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

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] "Sleep sort – the king of laziness / sorting while sleeping," GeeksforGeeks, 06 2016. [Online]. Available: https://www.geeksforgeeks.org/sleep-sort-king-laziness-sorting-sleeping/