BOGO-PI

worst project for the worst algorithm

Abstract

The point of this project is to examine the real-world inefficiency and horribleness of the bogosort algorithm. In this case, the randomized variant of bogosort will be used, as that was the easiest to implement while maintaining the inefficiency and horribleness. In this case, the algorithm will be implemented on a raspberry pi B+ and arrays of increasingly large size. A twitter bot will be used to show the results of each array being sorted including time elapsed and number of permutations.

Terminology

This section is used to outline some of the terms repeated throughout the document and program to increase the understanding of not only the code but my thinking behind it.

- Run : refers to an execution of the entire list of input sizes
- Efficiency : refers to how long an algorithm takes to complete given a certain input size

Introduction

Throughout my time studying computer science and programming, I, like most other students, have learned a lot about sorting algorithms, their implementation, and how efficient/inefficient they are. During my studies, I discovered the wikipedia page for bogosort, which detailed a bit about its implementation and inefficiency. However, I have not seen much about actually implementing it and running it for an extended period. Many of the implementations I have seen on the internet have been only running it overnight or for a couple of hours. This project aims to have a months, maybe even years, long implementation of the project. While I do not think that there is a whole lot to learn from this project, or that any huge leap in computer science will be made with this project, I do think that something could be learned from both the implementation and the results.

Algorithm

In this project, the specific version of the bogosort algorithm is randomized bogosort, this greatly affects the runtime of the algorithm and provides the algorithm with a new level of terribleness. Randomized bogosort is a very simple algorithm to implement and understand, as is its inefficiency. The algorithm functions by randomly shuffling the input array and then checking for sortedness. The output of this is then used again, as the input of the next try. This version of bogosort does not save any of the items that are correct, and it simply reshuffles the entire array. With this basis, the algorithm depends entirely on random chance and exhaustive permuting to find the sorted array.

From an analysis standpoint, there is a lot to unpack, yet it is very simple. Starting only looking at a single input array of size *n*, the likelihood of finding the sorted array is $\frac{1}{n!}$, given that there are *n!* possible permutations of the array. On average, the runtime of the algorithm should be $O(n \times n!)$, in which *n* is checking the array for sortedness, and *n!* is the number of permutations needed to be

calculated to find the sorted permutation. However, due to the fact that the algorithm does not save any correct items, the worst case runtime is exceedingly awful. The worst case complexity of randomized bogosort is $T(\infty)$ or the eventual heat death of the universe. This worst case complexity is objectively terrible and pretty much the worst it could be. To take the largest input, 52 and to show its average number of permutations needed and represent it decimally, it would look like:

80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,883,277,824,000,000,000,000

From a macro level, to get the average time complexity would be adding all of the average run times together. Mathematically, the equation would be:

$$O(\sum_{i=1}^{52}(n \times n!))$$

considering that the goal of the algorithm is to complete all input array sizes from 1 to 52 (the size of a full deck of cards). Along with this, the average number of permutations needed to find the sorted list of each input is:

$$\sum_{i=1}^{52} n!$$

this is a massive number, and unfortunately, this is only the average and not the worst case. The worst case is the eventual heat death of the universe, also known as long after anyone who is alive now has died. It should also be noted that some sources on the internet do state that the time complexity of bogosort is $O((n+1)! \times n)$ but I am not sure where the +1 comes from mathematically.

Implementation

This project is implemented in the Python programming language as it is easy to work with arrays and has shuffle as a built-in function. While other languages may provide faster runtimes and greater efficiency, I chose Python as it provides much of the functionality needed and is fast enough. Also this project is really about how awful the algorithm is and this will mostly be quantified with the number of permutations it takes compared to the average amount. While time will be kept, it will mostly be just to see and for the joke of seeing how long it took to do each. Additionally, a twitter bot has been implemented to tweet when it has completed an array. To complement this, a log file is created with each run to record the number of permutations and time elapsed for each input size along with the time the run was started.

Another facet of the implementation is the hardware used to actually run the code, which greatly affects the amount of time the program will take. In this instance, the program will be run on a Raspberry Pi Model B+, which has a 700 MHz processor and 512 MB RAM, while this pretty slow compared to the current average specifications, it was what I had on hand, and I think that the slowness of the hardware nicely complements the slowness of the algorithm. This hardware choice also plays into many of the considerations detailed later on in this writeup/paper.

Considerations

Throughout this project, there have been many changes to the implementation of the algorithm and the program itself. Changes to the logging system and how to deal with the fact that the runtime is virtually infinite have been the largest changes with the implementation at the time of writing. Many

of the considerations were resolved as a compromise between ease of implementation and staying true to the algorithm and the original idea of this project.

Originally, comprehensive logs, containing each permutation and whether or not it is sorted for each input value. However, it is not really feasible as the size of the files would be excessively large, with during a test run, the log file for input of size 13 quickly reached 200 gigabytes. As the project will be implemented on a raspberry pi, which only has 32 gigabyte sd card for storage, there simply is not enough room even if the files are compressed upon completion. For now, a shortened log file is created upon beginning a run, and is appended to whenever an input is completed. The log file is named for the time when it is created and also contains the time when the run has been started, in seconds, which can then be turned in to human-understandable time, using an equation, that changes seconds into days, hours, minutes, and seconds.

At one point, the idea of implementing both the algorithm and the entire process recursively seemed plausible and even easier than the iterative approach. Unfortunately, this is probably not possible and more complicated than the iterative approach. The Python language has a coded in limit for recursion depth, which will most certainly be exceeded as the program reaches larger input sizes. While the limit can be increased, there does not seem to be a manner in which to remove the limit completely, which would be necessary. Due to this, the program and algorithm will be implemented iteratively as there is no limit that I have found thus far in my testing.

As this project could be running for the rest of my life and will be running constantly, loss of power is an eventuality. Given this, a function has been added to restart the program with the next largest input size, it will not take into account any of the work done on this input size. This might also be used when updating the code or other similar tasks, as the aggregate time will be calculated with the time elapsed of each size. Along with this, a function could be implemented to calculate the aggregate time by using the log file created for each run. Initially, using some sort of small UPS (uninterruptible power supply) build for the raspberry pi was considered but this seems a bit overkill and the restart function remedies similar issues and also deals with updating and other similar issues.

To make checking the progress of bogo-pi easier, a twitter bot has been created which will create a tweet whenever an input has been sorted. This means that I do not have to constantly check the progress by being physically at the raspberry pi or SSH into the machine to read the log files. Remotely accessing the machine via SSH will still be possible to best update the software and to work with the data created by this program. Specific changes and how they will be done, will be determined after some extended testing and this document will be updated to reflect these changes. Another addition to the twitter bot is the implement the option of a daily logging tweet, to make sure the program is still running and functioning correctly.

On Randomness

It should be mentioned that there is no such thing as a totally random number, and that random shuffling implemented in the program is a built-in function of python and uses an algorithm to determine its output. Due to this, bogo-pi does not fully implement randomized-bogo yet it does the best possible with the language as it stands. While there may be better ways to implement, again this is acceptable for this application, but it does stand to be mentioned in a writeup.

On the Code

The actual is fairly simple, as is the algorithm, which makes it pretty easy to understand once the algorithm is understood. Much of the code written is actually the logging, resume, and twitter bot functions. None of the code within these functions is very complex or relies on comprehensive knowledge of programming or computer science or mathematics. Yet, I would like to explain it anyways to get used to justifying my choices and also I feel that it could be useful to look at later on.

The worker function simply calls the randomized_bogo function, which is the implementation of the algorithm. All it does is watch return value, a boolean, and stops the loop when it returns true -- that the input is now sorted.

The logger function was probably the most complex, as it does some file operations and calls another python script, the twitter bot. Again, these are not really complex and rely on pretty basic python stuff except for maybe the twitter bot.

The twitter bot script is not very complex but it does have some confusing elements, including a configuration file and some api calls. However, this is not really necessary to understand the project or the underlying algorithm. The twitter bot has been updated for simplicity and to best reflect the needs of the project. In many ways, the earliest implementation of the bot provided the functionality required by the project, but it did it in an inefficient way and was not really completing its function with any options for easy improvement or extension.