# Readme_Shoe

Version 1 8/22/24

A copy of this file should be included in the github repository with your project. Change the teamname above to your own

1. Team name: Shoe
2. Names of all team members: Alex Schumann
3. Link to github repository: https://github.com/CobblerOfShoes/NP-Coin-Problems
4. Which project options were attempted:
    a. Bin Packing/Knapsack Problem – Coin Jar Partition
5. Approximately total time spent on project:
    a. 6~7 hours
6. The language you used, and a list of libraries you invoked
    a. Language: Python
    b. Libraries: argparse, random, datetime, textwrap, os, csv, time, numpy, matplotlib.pyplot, scipy.optimize
7. How would a TA run your program (did you provide a script to run a test case?)
    a. The code is interactive with terminal commands.
    b. To generate test cases:
        i. python3 partitionTestCaseGeneratr_Shoe.py -f *outputDataFilename* -s *numberOfTestCasesPerCoinCount* -n *maxNumberOfCoins*
    c. To run the code on a test case:
        i. python3 partitionCoinJar_Shoe.py -f *inputDataFilename* -o *outputDataFilename* -i *optionalImageOutputFilename*
8. A brief description of the key data structures you used, and how the program functioned.
    a. The program reads in the test cases as a list, and pops off the first value as the test case index, and uses the rest as a list of available coins. It then takes these coins, and uses recursive calls to build a list for the left and right piles, effectively building a stack. It tries putting each coin on the left and right stack, and all following combinations, and then returns a dictionary with the test case index as the key and a tuple of the left pile, right pile, computation time, and number of coins as the value.
9. A discussion as to what test cases you added and why you decided to add them (what did they tell you about the correctness of your code). Where did the data come from? (course website, handcrafted, a data generator, other)
    a. I created a test case generator that loops through a range of a user given number of coins with a user given number of cases for each coin. This made it very easy to generate a wide variety of test cases of varying sizes. I made one test case by hand (“checkCoinJarTestCasesKnown_Shoe.txt”) that has 5 known failure cases and 5 known cases with a solution. I used this to get a rough idea that it was working properly.
10. An analysis of the results, such as if timings were called for, which plots showed what?

    a. The timings are what I expected on a large scale. I noticed that on a smaller scale, such as in the 3-5 coin range, the computations generally took 0 nanoseconds or 1 millisecond, which is a fairly large jump. Considering nothing can actually happen in zero time, I think this is a result of the time.time_ns() function not having enough precision to measure some of the operations happening in just a few clock cycles.

11. What was the approximate complexity of your program?
    a. The time complexity is approximately 2^n, where n is the number of coins. This is because for each coin, there are two choices: put it in the left or right pile. However, I used a simple exponential equation to graph the curve as I wasn't quite sure how to plot a 2^n regression.
12. A description of how you managed the code development and testing.
    a. I first saw that there were no test cases available, so I developed the random test case generator. I used this to generate all my test cases to test my code, and made it very interactive to be able to easily generate many different size tests. I then worked on the code to compute the partitioning of the jar into two piles, and once I got text output that was correct, I began working on graphing the data. After that, I created a simple python program that validates the output of the main program as a final test.
13. Did you do any extra programs, or attempted any extra test cases
    a. I submitted my test case generator to Prof. Kogge which was shared with the class (I since modified it a bit as I changed my mind on how I wanted to format the data). I was originally doing this as a subset sum problem with a target value to combine the coins to, but changed my mind as I found this problem more interesting. As an additional challenge, I tried a version of code that computes all subsets of the jar of coins, looks for a subset that sums to half the value of the jar of coins, and then returns that subset and its complement as the two piles. Conceptually, I found this approach very interesting, but could not quite get it to work as I was having issues generating subsets with duplicate values.