Prime Coin Change Problem

Lucas Geurtjens | s5132841 | 18/04/2019

# Algorithm Design

## Overview

The prime coin change algorithm finds the total number of ways to pay a given amount from a list of coins and functions as seen in Figure 1. It achieves this differently depending on the input parameters as follows: 1 parameter tells it to find all combinations (not including duplicate combinations) to reach the amount, 2 parameters give it an exact count of coins it must use to reach the amount and 3 parameters give it a specific range of coins it must find the amount within. The list of coins used is created by appending all prime numbers until the amount, 1 and the amount itself (if not already a prime and in the list). From this, the algorithm calculates various solution states though recursive backtracking using DFS (Depth First Search) with pruning. Originally a dynamic implementation with Breadth First Search was used, however, due to its heavy use of list manipulation and poor speed efficiency, it was rejected. Finally, the algorithm is designed to be executed from the command line. It takes an input of the absolute location of a given input file and will output a output.txt file containing the solutions. Overall, the implementation was effective and had no obvious issues.



Figure : Overview Diagram of Algorithm

## Algorithm Description

Recursive Backtracking

The recursive backtracking algorithm is responsible for finding the solutions of a given amount. Firstly, because there are 3 parameters that change the criteria of the search(amount, exact/lower limit and upper limit), the algorithm checks how many parameters were used (given as input) and will then use a base case for that amount. The base case returns 1 for a valid solution and 0 for an invalid solution. If neither base case is met, the algorithm will them enter a loop which will recursively search over the coins. This acts as a pruned depth first search, as seen in figure 2.

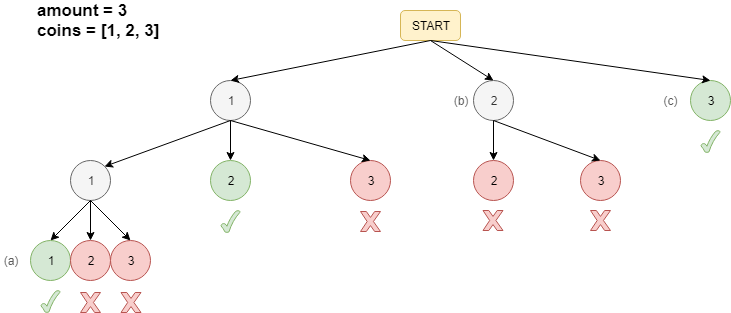


Figure : Example of Backtracking

The algorithm employs various pruning techniques to reduce its search space. Firstly, we employ the idea of a ‘current coin position’. This is initialised as the first coin in the list (0) and is incremented each time the loop is run, acting as the lower limit range of the loop. Since the algorithm is recursive, it can be observed in Figure 2 (a) that the first coin retains its current coin position of 0 until it reaches a base case. Similarly, as seen in Figure 2 (b), the algorithm skips the first coin. This pruning allows us to reduce the search space and remove duplicate permutated states. Alternatively, the second pruning method employed is an affect of reaching a base case. When one of the two base cases are reached, the algorithm will not search any further because it returns a base case value instead. These children node would not give a solution and hence can be discarded. Lastly, when parameter 2 and 3 are used, extra pruning is used to control the amount of coins used. Here, the algorithm checks if the number of coins used have exceeded either the exact or upper range coin limits. If so, it will return a base case 0 and hence stop searching the further child nodes.

Prime Checker

The prime checker algorithm allows us to check if a given number is prime which is useful for generating our list of prime number coins. Firstly, the algorithm checks very small and obvious base cases (1 is not a prime, while 2 and 3 are etc.) which will slightly speed the algorithm up for smaller numbers and won’t have to do any square rooting which is later used. From this, the algorithm will loop though and check each iterator against the number we are checking. This is done within the range of 2 and the square root of the number + 2. We start at 2 to avoid checking with 1 as we know it will give a remainder of 0 for all numbers (hence will inaccurately assign elements as not prime). The upper limit then begins at 2 to avoid the lower limit being smaller than the upper limit which occurs with squaring numbers like 2 and ensures it will execute at least once for numbers like 4. The square root is implemented in an effort to prune the algorithm by avoiding factors that will have already been checked (e.g. if we know 10, we can predict if 100 is prime too). If we are able to reach the end of the loop, then we must have a prime number.

Coin List Generator

The coin list generator creates a list of valid coins to use in the program (after running, this list is globally declared). Firstly, we append 1 to the list (which is required by the constraints). Next, we iterate until we reach the amount given, checking each time if the number is prime and adding it to the list if it is. After this, we check if the amount given is equal to the last element of the list (which can occur if the number is a prime), then append it if it is not (this is the required golden coin). From this we have a list of coins to use in the program.

## Algorithm Pseudocode

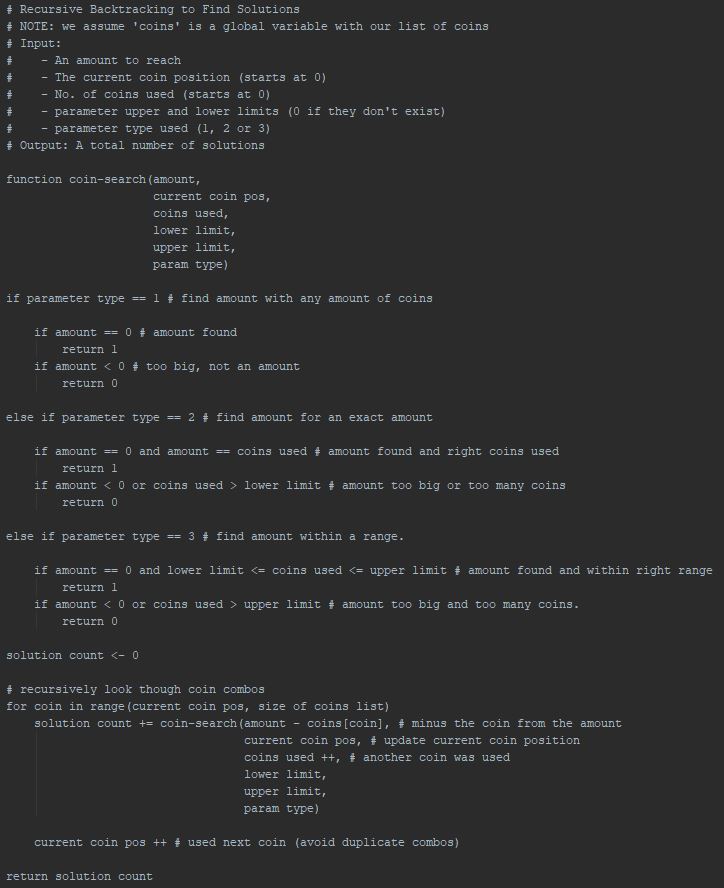


Figure : Pseudocode for Recursive Solution Finder

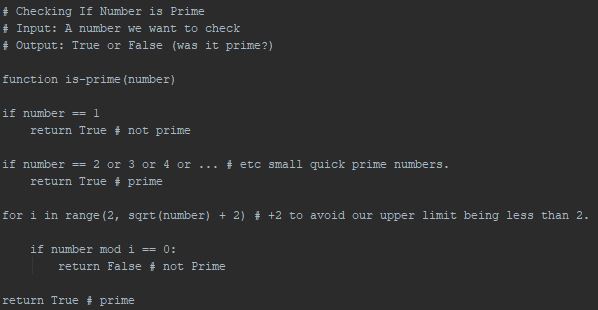


Figure : Pseudocode for Prime Checker

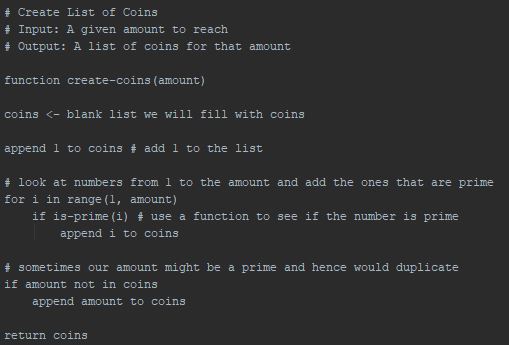


Figure : Pseudocode for Coin List Generator

# Results and Algorithm Analysis

## Speed Test

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Speed (secs)** |
| 1 | 5 | 6 | 0.0000 |
| 2 | 6 2 5 | 7 | 0.0000 |
| 3 | 6 1 6 | 9 | 0.0000 |
| 4 | 8 3 | 2 | 0.0005 |
| 5 | 8 2 5 | 10 | 0.0005 |
| 6 | 20 10 15 | 57 | 0.0090 |
| 7 | 100 5 10 | 14839 | 7.6174 |
| 8 | 100 8 25 | 278083 | 76.0504 |
| 9 | 300 12 | 4307252 | 13470.9674 |
| 10 | 300 10 15 | N/A | 14400+ |

Figure : Speed Test Results

Figure : Illustration of Algorithm Speed Test

Discuss the sample input.txt results

It looks like it’s exponential

Why didn’t we get higher times?

Was it finding the correct solutions? Compared to known solutions provided.

Look at results for sample input.txt

Explain why we didn’t get to the higher times.

Why is this faster than an unpruned model?

Perhaps compare it to your dynamic implementation.

## Time Complexity

Calculate the recursive relation

Big O?  
Big Alpha?  
Big Omega?

Big Theta?

Correctness?

Completeness?

Stable?

In-Place?

How do you know?

What’s the verdict? Does it use a lot of storage? Does it use a lot of CPU?

# References

Understanding how to check for prime numbers - 09/04/2019 - *Tuesday drop-in session with Saiful.*

Understanding how a recursive coin change problem works - 14/04/2019 - [*https://www.youtube.com/watch?v=k4y5Pr0YVhg*](https://www.youtube.com/watch?v=k4y5Pr0YVhg)