Prime Coin Change Problem

Lucas Geurtjens | s5132841 | 21/04/2019

Algorithm Design

Overview

The prime coin change algorithm was able to find the total number of ways to pay a given amount from a list of coins and methods as seen in Figure 1. This was achieved differently depending on the input parameters as follows: 1 parameter causes it to find all combinations (not including duplicate combinations) to reach the amount, 2 parameters cause it to find an exact count of coins it must use to reach the amount and 3 parameters give a specific range of coins it must find the amount within. The list of coins used was 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. Initially, 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 was 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 contained no obvious issues.

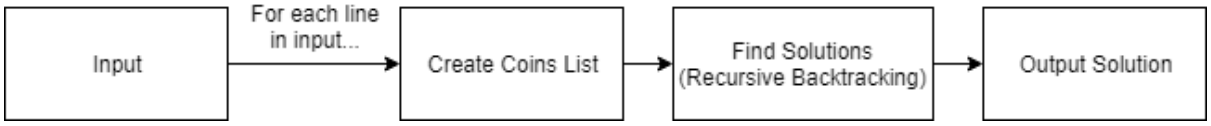

Figure 1: Overview Diagram of Algorithm

Algorithm Description

Recursive Backtracking

The recursive backtracking algorithm was responsible for finding the solutions for the given amount. This model was vaguely inspired by ONeilCode who exemplified the behaviour of recursive coin change problems. 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.

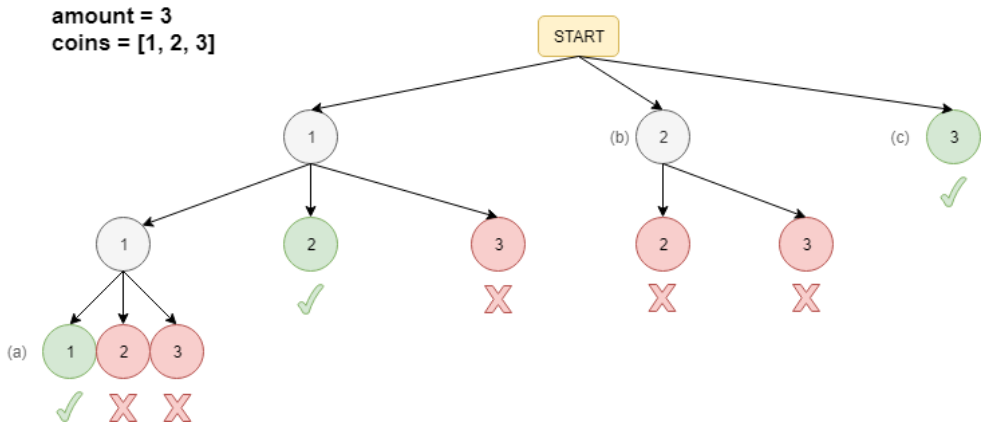

Figure 2: Example of Backtracking

The algorithm employed various pruning techniques to reduce its search space. Firstly, the idea of a 'current coin position' was used. The current coin position is initialised as the first coin in the list (0^{th} element) 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 (left most) coin retained its current coin position of 0 until it reaches a base case. Similarly, as seen in Figure 2 (b), the algorithm ignored the first coin and continued straight to the second coin. This pruning reduced the search space and removed duplicate permutated states. Alternatively, the second pruning method employed is an effect 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. Here, child nodes are not explored since a solution has already been found. 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, a base case 0 will be returned, hence stopping the searching of further child nodes.

Prime Checker

The prime checker algorithm enabled the ability to check if a given number is prime. This model was vaguely inspired by Saiful Islam who illustrated how to best check for prime numbers. Firstly, the algorithm checks very small and obvious base cases (1 is not a prime, while 2 and 3 are etc.) slightly speeding up the algorithm because a square root function (later used) can become avoided. From this, the algorithm loops though and checks each iterator against the number it is checking. This was done within the range of 2 and the square root of the number + 2. A start of 2 was used to avoid checking with 1 (as it will give a remainder of 0 for all numbers, hence will inaccurately assign elements as not prime). The upper limit then began at 2 to avoid the lower limit being smaller than the upper limit which occurs with squaring numbers such as 2 and ensures it will execute at least once for numbers such as 4. The square root was 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 the end of the loop is reached, a prime number is confirmed.

Coin List Generator

The coin list generator aimed to create a list of valid coins to use in the. Firstly, 1 is appended to the list (which is required by the constraints). Next, the algorithm iterates until the given amount is reached, checking each time if the number is prime and adding it to the list if it is. After this, a check is made to see if the amount given is equal to the last element of the list (this can occur if the number is a prime), then append it if it is not (this is the required golden coin). From this a list of coins can be globally declared and used throughout the program.

Algorithm Pseudocode

```
# Recursive Backtracking to Find Solutions
# NOTE: we assume 'coins' is a global variable with our list of coins
# Input:
#    - An amount to reach
#    - The current coin position (starts at 0)
#    - No. of coins used (starts at 0)
#    - parameter upper and lower limits (0 if they don't exist)
#    - parameter type used (1, 2 or 3)
# Output: A total number of solutions

function coin-search(amount,
                     current coin pos,
                     coins used,
                     lower limit,
                     upper limit,
                     param type)

if parameter type == 1 # find amount with any amount of coins

    if amount == 0 # amount found
        return 1
    if amount < 0 # too big, not an amount
        return 0

else if parameter type == 2 # find amount for an exact amount

    if amount == 0 and amount == coins used # amount found and right coins used
        return 1
    if amount < 0 or coins used > lower limit # amount too big or too many coins
        return 0

else if parameter type == 3 # find amount within a range.

    if amount == 0 and lower limit <= coins used <= upper limit # amount found and within right range
        return 1
    if amount < 0 or coins used > upper limit # amount too big and too many coins.
        return 0

solution count <- 0

# recursively look though coin combos
for coin in range(current coin pos, size of coins list)
    solution count += coin-search(amount - coins[coin], # minus the coin from the amount
                                  current coin pos, # update current coin position
                                  coins used ++, # another coin was used
                                  lower limit,
                                  upper limit,
                                  param type)

    current coin pos ++ # used next coin (avoid duplicate combos)

return solution count
```

Figure 3: Pseudocode for Recursive Solution Finder

```
# Checking If Number is Prime
# Input: A number we want to check
# Output: True or False (was it prime?)

function is-prime(number)

if number == 1
    return True # not prime

if number == 2 or 3 or 4 or ... # etc small quick prime numbers.
    return True # prime

for i in range(2, sqrt(number) + 2) # +2 to avoid our upper limit being less than 2.

    if number mod i == 0:
        return False # not Prime

return True # prime
```

Figure 4: Pseudocode for Prime Checker

```
# Create List of Coins
# Input: A given amount to reach
# Output: A list of coins for that amount

function create-coins(amount)

coins <- blank list we will fill with coins

append 1 to coins # add 1 to the list

# look at numbers from 1 to the amount and add the ones that are prime
for i in range(1, amount)
    if is-prime(i) # use a function to see if the number is prime
        append i to coins

# sometimes our amount might be a prime and hence would duplicate
if amount not in coins
    append amount to coins

return coins
```

Figure 5: Pseudocode for Coin List Generator

Results and Algorithm Analysis

Speed Test Results

#	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 6: Speed Test Results

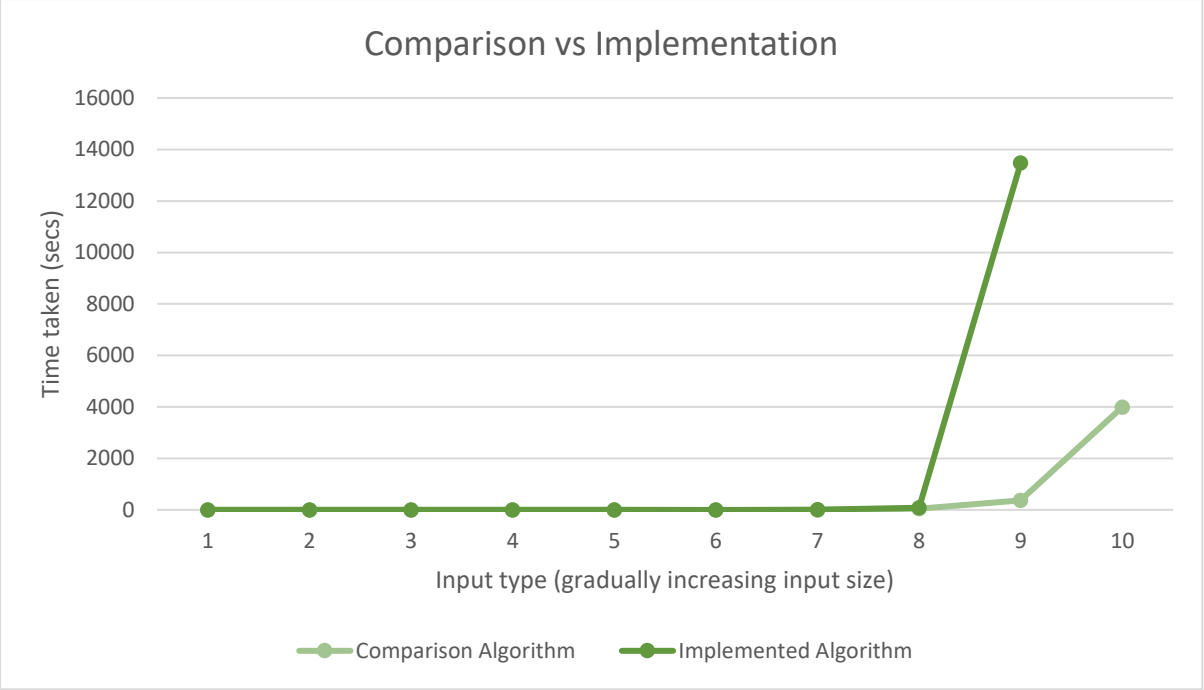

Figure 7: Comparison vs Implementation Speed Test

The program was run using a previously solved test input file (see appendix Figure 8) and enabled the ability to collect results (see Figure 6) and make comparisons between solutions and speed tests. Firstly, it appeared that all collected solutions accurately matched the known solutions. This suggests the algorithm has the ability to accurately predict the number of solutions for any given amount. Alternatively, the time taken to find all solution states was partially different (as seen in Figure 7). The algorithm was observed to quickly solve inputs wherein the first parameter was < 100, most results solved under a second. Nevertheless, for bigger inputs such as 300, the algorithm took a significant amount of time (approximately 4 hours). As such, the final input "300 10 15" was in-complete due to its time exceeding 4 hours. Similarly, the comparison speed has a significantly longer speed (compared to its other speeds) for this specific input. These results suggest an exponential time complexity is present within both models, with the implemented model containing slightly weaker pruning methods.

Time Complexity

In order to find the time complexity of the algorithm, a recurrence relation was calculated. This was achieved by looking at the simplest model base case (the 1 parameter model). This was because the remaining models' base cases also take the same time complexity in their worst case and allowed the calculation to become simpler. A master theorem was not used to help ensure the calculation was correct as this recurrence relation did not employ a divide and conquer approach hence does not meet its criteria (see appendix Figure 9 for reference). The following was used in the calculation:

- C : A constant variable which represents how many times the recursive function will be called from the loop. Since we don't know how many times it is run, a constant is used to denote this.
- T(n – 1) : The recursive function which represents the reduction of 1 unit in the input size each call. The constant 1 is used since decreasing by 1 will be the worst case.
- 1: A constant used to represent the minor calls that are made each iteration. These include declaring variables, addition and if statement checks.

- T(0) = 1: The base case state representing when the amount has been reached, hence, the amount minus the decremented factor will be equal to 0 and return 1 for a solution state.

$$T(n) = C \cdot T(n-1) + 1$$

$$= C[C \cdot T(n-2) + 1] + 1$$

$$= C^2 T(n-2) + C^1 + C^0$$

$$= C^2[C \cdot T(n-3) + 1] + C^1 + C^0$$

$$= C^3 T(n-3) + C^2 + C^1 + C^0$$

....

$$= C^k T(n-k) + C^{k-1} + C^{k-2} + \ldots + C^1 + C^0$$

Here, $n - k = 0$, hence $n = k$

Now, using $T(0) = 1$

$$= C^k \cdot T(0) + C^{k-1} + C^{k-2} + \ldots + C^1 + C^0$$

$$= C^k + C^{k-1} + C^{k-2} + \ldots + C^1 + C^0$$

$$= C^{k+1} - 1$$

$$= C \cdot C^k - 1$$

$$\leq C \cdot C^k$$

$$\therefore O(C^n)$$

From this, it can be noted that the recurrence relation follows an exponential time complexity of $O(C^n)$. This confirms what was observed in the results.

Correctness

In order to check the correctness of the algorithm complexity, proof by induction was used.

1.) Proposition

$$T(n) = C^n \text{ where } n \geq 0$$

2.) Base Case

Since $T(0) = 1$,

$$T(0) = C^0 = 1 \text{ where } n = 0$$

$\therefore$ True for n = 0

3.) Assumption

Assume True for n = k

$T(k) = C^k$

4.) Inductive Step

$T(k+1) = C^{k+1}$

5.) Conclusion

Thus, we can conclude that $T(n) = C^n$

In-place and Stable

The solution finder and prime checker algorithms used were in-place. This could be observed as no significant extra memory was utilized outside of small variables. This was achieved by avoiding the use of list manipulation (a characteristic significantly used by the dynamic and poorly optimized prototype algorithm). Nevertheless, it may be argued that the coin generation algorithm used was not in-place, as it used additional memory to create its list of coins.

Alternatively, all algorithms used were stable. This was true as none of the algorithms swapped list elements, hence, could never swap equal valued elements and become unstable.

P or NP Complexity

After observation of the algorithm, it was concluded that the problem followed an NP (Non-Polynomial) complexity. NP is a problem infinitely difficult to solve but easy to check if a given solution is correct. This was indicated by the requirement to calculate a list of infinitely large prime numbers and the find combinations of summed coins to equal an amount. There existed no easy, polynomial method of calculating primes or coin combinations, but it was easy to confirm said solutions were true, thus, the problem was NP.

References

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

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

[3] *HegartyMaths - Understanding how to perform proof by induction – 21/04/2019 -* *https://www.youtube.com/watch?v=yBbKFSLHlFw*

Appendix

Results for Python Implementation

Input	Output	CPU Time (Secs)	Input	Output	CPU Time (Secs)
5	6	0.000523	20 10 15	57	0.007611
6 2 5	7	0.001361	100 5 10	14839	0.912590
6 1 6	9	0.001741	100 8 25	278083	39.795810
8 3	2	0.000537	300 12	4307252	372.670860
8 2 5	10	0.001271	300 10 15	32100326	3992.311160

Figure 8: Input.txt Comparison Results

Master Theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), T(1) = 1$$

where $a \geq 1$, $b > 1$ and $f(n) = \Theta(n^k \log^p n)$.

Figure 9: Master Theorem (not implemented).