Hunter Stout

Homework 4: Dynamic Program Report

11/16/2024

The Analysis of An Updated Sparse Price Rod Cutting Algorithm

Introduction

The traditional rod cutting algorithm is used on a continuous price list for all lengths from 1 to n. Although in most scenarios, lengths do not always have corresponding prices, and an input rod number can be larger than the highest value in a given length list. To help fix these problems, I updated the algorithm to use a sparse price list for calculating the values and added a method of handling an input rod length that is longer than the maximum length value, by splitting the rod into more manageable parts.

Algorithm Changes

Spare Price List Enhancements:

- Used a HashMap to easily map lengths to their price.
- Iterate through lengths that are only considered available, instead of automatically using every length. This dynamic programming approach reduces unnecessary load and allows all kinds of tables to be used.

Rod Length Longer Than Maximum Length:

- Rods that are longer than the maximum value in the Length list, are split into two parts.
  - The first part is the excess, which is basically the largest available value in the Length list, and it is repeated until there is a more manageable load.
  - The second part is the remaining half of the value, which is found by using the normal dynamic programming algorithm.
- These two parts are then combined to form a single price, regardless of the length overriding the length list.

Time Complexity Analysis

Spare Price Rod Cutting Algorithm:

- There are two loops in this function (Commented 1A in Functions.py). The outer loop iterates through the rod lengths provided, which are 1 to n. Next, the inner loop of this function only iterates through available lengths which we can mark as k.
- The overall time complexity of this function is $O(n \times k)$. Where n represents the rod length, and k represents the amount of available lengths.

- The space complexity of the dynamic programming functionality includes a Results array, which allocates a size of $n + 1$, requiring $O(n)$ space. It also includes a price HashMap, which stores price values using length keys. This requires $O(k)$ space.

Extended Algorithm For Rods Exceeding Maximum Length:

- When n is greater than the length max, this function is divided into two parts that were previously explained.
- The first being the excess part, which is a complexity of $O(1)$, using simple constant assignment.
- The second being the remaining part, which utilized the previous Spare Price Rod Cutting Algorithm, which is $O(n \times k)$.

The overall complexity of both Algorithms is $O(n \times k)$, the extended function adds only more constant time functionality.

Best, Worst, and Average Time Complexities

- The best case for this algorithm would consist of a condition where only one price is available. Meaning $k = 1$. The time complexity of this would be $O(n)$, because only one option would be calculated per rod length. The space would still be $O(n)$ as well.

Best Case Complexity: $O(n)$.

- The worst-case scenario would consist of a condition for the price list including every length. Meaning $k = n$. The time complexity here would be a hefty $O(n^2)$, because every length would be evaluated for every rod length. The space here is still $O(n)$.

Worst Case Complexity: $O(n^2)$.

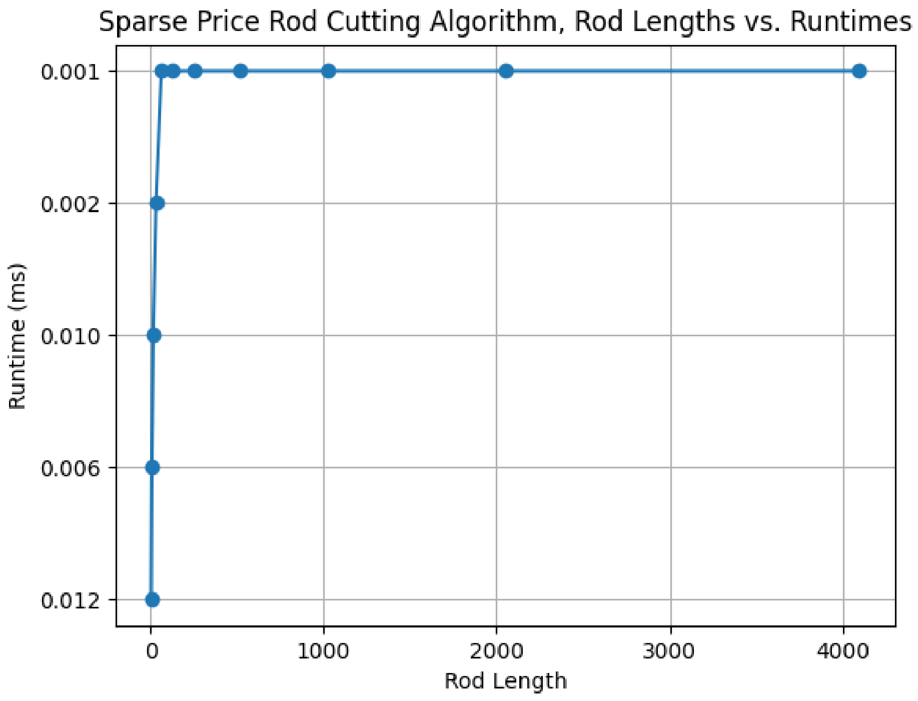
- The average case would consist of a condition for the price list containing an average number of lengths. The time complexity here would be $O(n \times k_{average})$. Meaning $k_{average}$ represents an average number of available length values. The space here would consist of $O(n + k)$.

Average Case Complexity: $O(n \times k_{average})$.

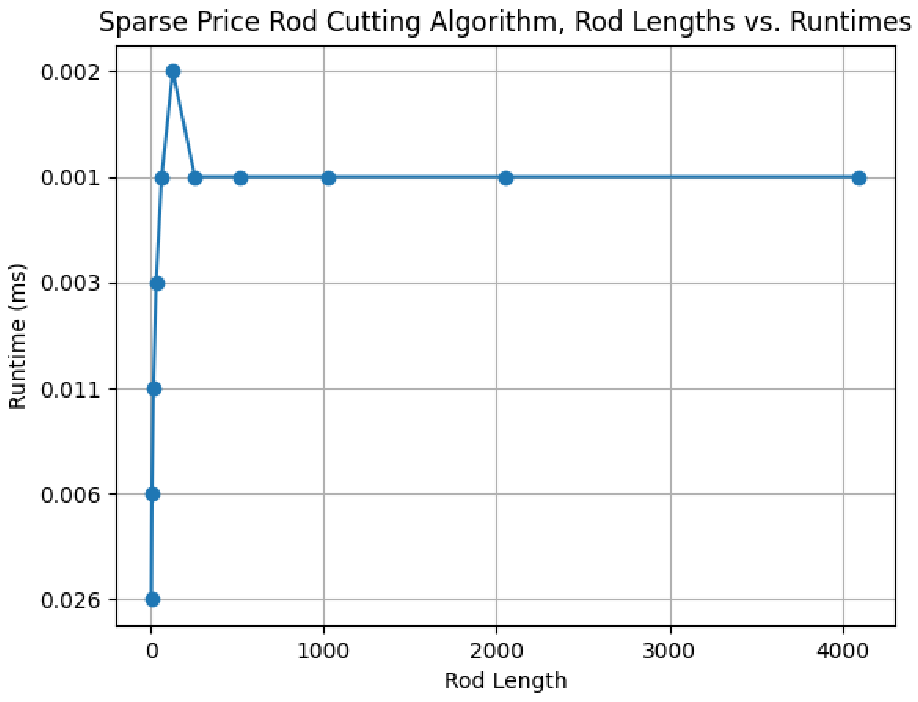
Plotting Runtimes

The Spare Price Rod Cutting Algorithm Extended was performed on several different rod lengths, consisting of [4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]. The runtimes of these different lengths were recorded, and plotted together on graphs, through several trials. I noticed that the values could vary depending on discrepancies out of my control, so I performed three total trials.

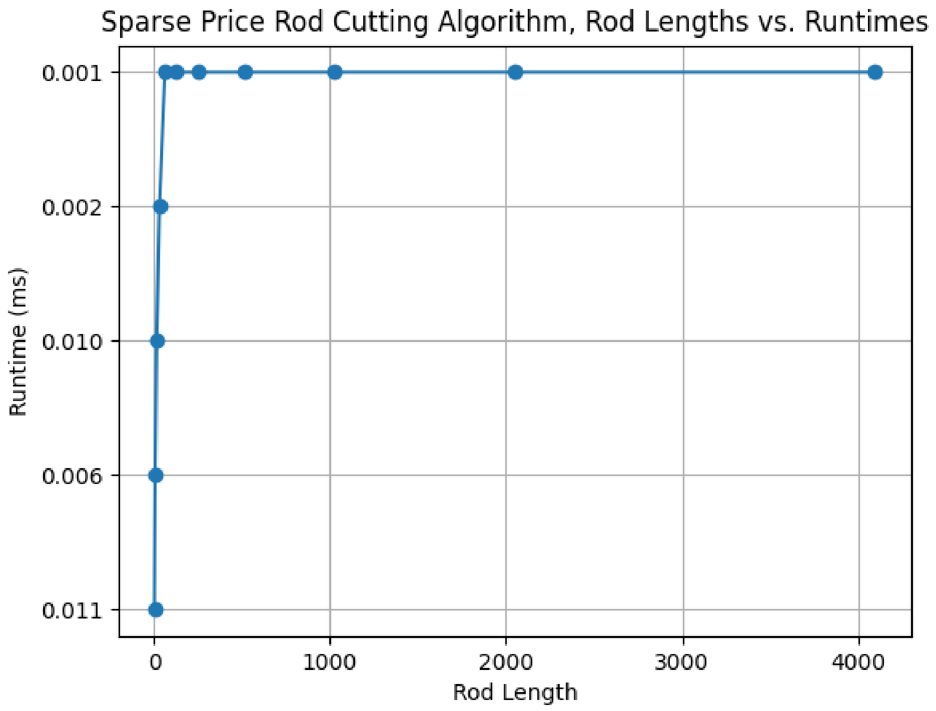
Trial 1:



Trial 2:



Trial 3:



These trials performed as expected, you can see in real time the algorithm steadily start and soft cap at a respectable runtime, within the time complexity's predictions. Considering small discrepancies outside of my power for this assignment. Overall, my work was successful in demonstrating the power of scalability for real-world scenarios where price lists are sparse, and rods could potentially exceed limits that were predefined.

The Runtime Analysis and Comparison of Different Coin Change Algorithms

Introduction

The purpose of this experiment is to test the performance of two different Coin Change Algorithms. The coin change algorithm finds the different combinations of a fixed coin list to achieve a specific amount. The original coin change dynamic programming solution counts hypothetical unique solutions, and the reconstructed version finds all the unique solutions.

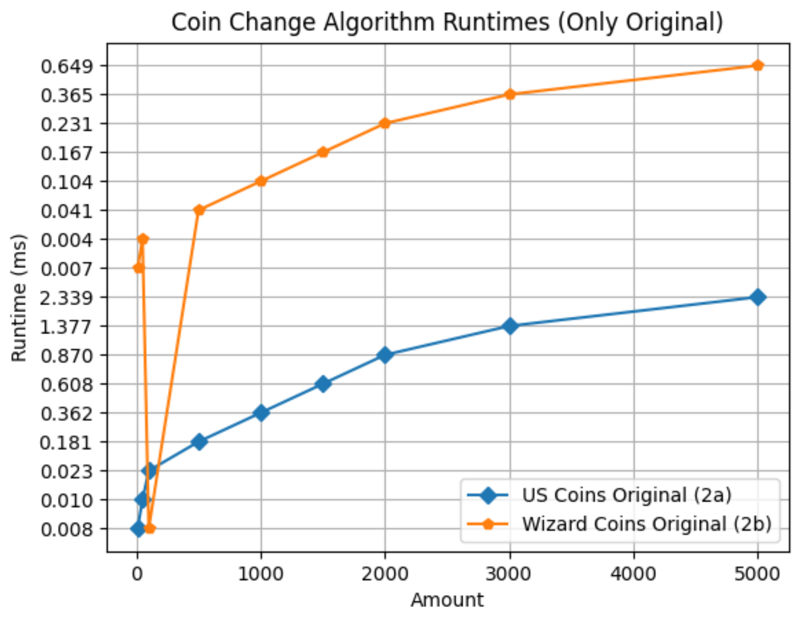
This experiment is going to compare the original coin change algorithm, against the reconstructed algorithm in terms of Datasets vs. runtime. The two datasets that this code will be using are:

- US Coins: [1, 5, 10, 25, 50, 100, 200, 500, 1000, 2000]

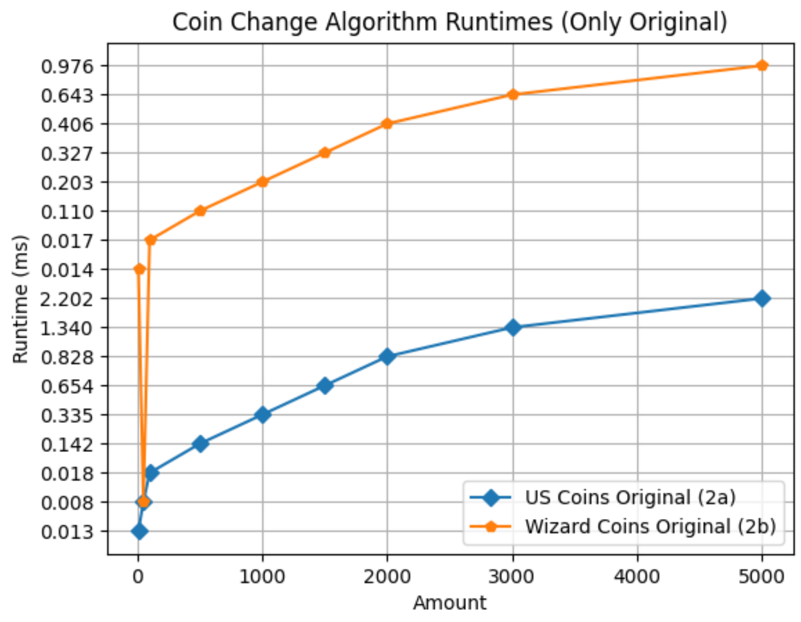
- Wizarding World Coins: [1, 29, 493]
- Original Amounts (Price Totals): [10, 50, 100, 500, 1000, 1500, 2000, 3000, 5000]
- Reconstructed Amounts (Price Totals): [10, 25, 50, 100].

Plot Results

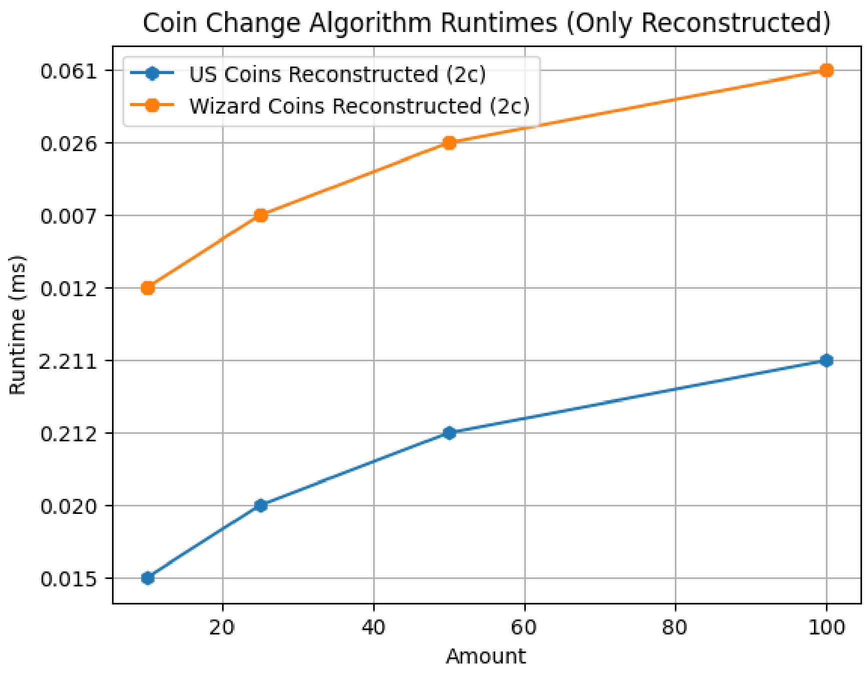
- 2A & 2B Only:
    - Trial 1:



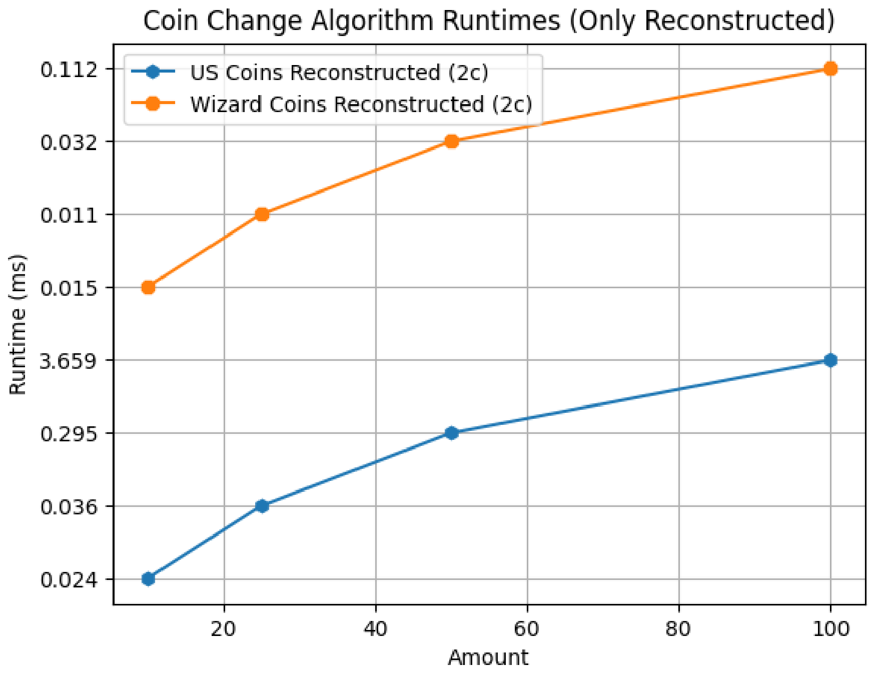
    - Trial 2:



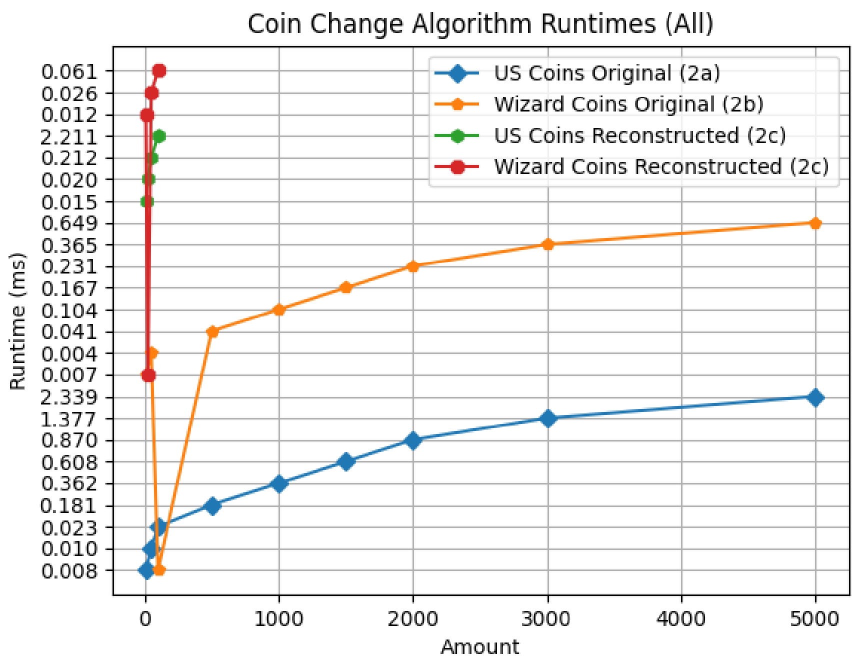
- 2C Only:
    - Trial 1:



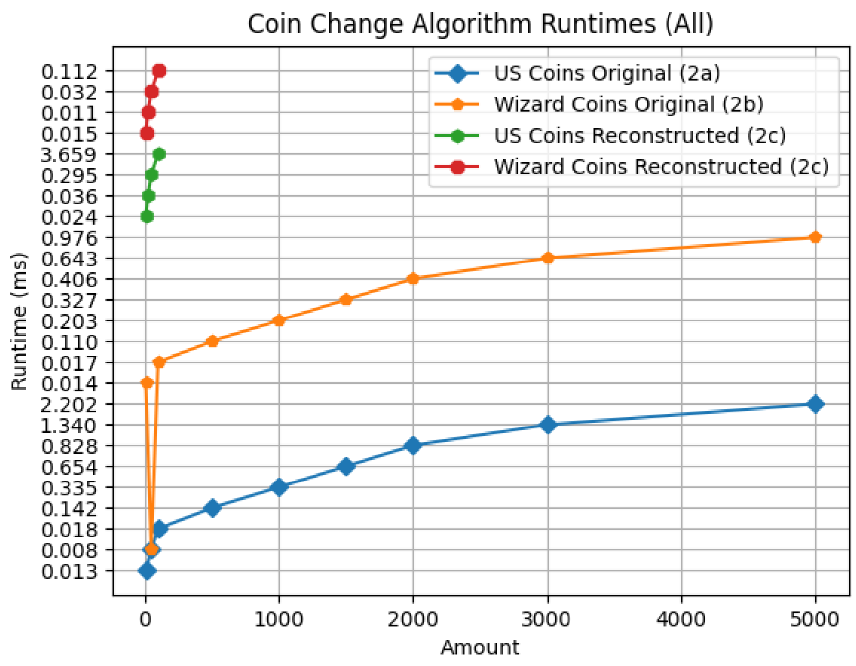
    - Trial 2:



- 2A, 2B, & 2C (All):
    - Trial 1:



    - Trial 2:



Overall, the runtimes are performing as expected. The results help convey that a larger list of coins contributes to the higher number of unique combinations. When there are more unique combinations that need to be processed, this is going to cause the runtime to increase as the code will need longer to process more calculations. This is the same in both algorithms, as it seems the US Coins takes a longer amount of runtime compared to the Wizard Coins when using each respective algorithm. It's difficult to compare the Original vs. Reconstructed algorithms in this given scenario, as the List of Amounts (Price totals we are finding unique combinations for), decreases to a smaller size per Homework instructions. I'm sure that if we used the same List of Amounts, it would take significantly longer for the reconstructed algorithm, as there are quite a lot more computations happening. This will be further broken down, as I begin to discuss the different time complexities for each algorithm.

Time Complexity Breakdown

Original Coin Change Algorithm:

Code	Explanation	Complexity
MaxAmount = amountToCheck + 1	Define the max size.	$O(1)$
Results = [0] * MaxAmount	Allocate Result array of the max size.	$O(n)$
Results[0] = 1	Define the base case.	$O(1)$
for i in coinList:	This is the outer loop m. m is the size of coinList.	$O(m)$
for j in range(i, MaxAmount)	This is the inner loop, which runs until the max size n.	$O(n)$
Results[j] += Results [j-i]	This updates the value inside of the results array per iteration.	$O(1)$

- The total complexity for this algorithm is $T(n, m) = O(m \times n)$.
- The space complexity is $O(n)$
- $n = MaxAmount.$
- $m = coinList\ Length.$

Reconstructed Coin Change Algorithm:

Code	Explanation	Complexity
MaxAmount = amountToCheck + 1	Define the max size.	$O(1)$
Results = [[] for _ in range(MaxAmount)	Allocate Result array of the max size.	$O(n)$
Results[0] = [[]]	Define the base case.	$O(1)$
for i in coinList:	This is the outer loop m. m is the size of coinList.	$O(m)$
for j in range(i, MaxAmount)	This is the inner loop, which runs until the max size n.	$O(n)$
for x in Results[j-i]	This is the inner-inner loop, it runs through the current stored combinations for the amount x.	$O(k)$ $(number\ of\ combinations)$
Results[j]. append(newCombination)	Adds a new combination to the Results array.	$O(1)$

- The total complexity for this algorithm is $T(n, m, k) = O(m \times n \times k)$.
- The space complexity is $O(n \times k)$.
- $n = MaxAmount$.
- $m = coinList\ Length$.
- $k = Average\ number\ of\ Combinations$

Best, Worst, and Average Time Complexities

- The best-case scenario for the two algorithms would consist of a condition for very few large denominations. For example, [25, 40, 80]. Leaving out values like [1, 2, 5]. The complexity of this best-case scenario for the original & reconstructed algorithm would be $O(m \times n)$. The reason it's the same complexity for the reconstructed function is because $k = 1$ in this scenario.

- The worst-case scenario for the two algorithms would have a condition opposite of the one above, consisting of many small denominations like [1, 2, 5]. This would make quite a lot more unique combinations for higher total prices. The time complexity of the original would still be $O(m \times n)$, although the reconstructed function's complexity would be $O(m \times n \times k)$. In this situation, the k value would grow exponentially with the value of n.
- The average case scenario's condition would contain denominations that are moderately configured. A combination of both good and bad scenarios. An example of this would be [1, 5, 10, 25, 50, 100]. This list has small values that will make the functions produce more unique combinations, although it has higher values to break down those large total prices easier. The time complexity of the original function would still be $O(m \times n)$, although the complexity of the reconstructed function would be $O(m \times n \times k_{average})$.