Department of Electrical and Computer Engineering
North South University

CSE 373 Design and Analysis of Algorithms

Project Report

COIN CHANGE PROBLEM - Combination

JOY KUMAR GHOSH 2211424 6 42

KH BORHAN UDDIN SIYAM 2211918 0 42

YEASIN KHALILI 2132626 6 42

Mirza Mohammad Lutfe Elahi, Lecturer

Mahir Shahriar Tamim, Undergraduate Assistant

ECE Department

Spring, 2024

Submission Date: June 1, 2024

Chapter 1 Introduction

1.1 Problem Statement

Our given problem is related to the Coin Change Problem. After providing a coin set and a change amount, we need to determine how many combinations can be produced from this coin to make a successful change. Here, we need to count the possible combination, not the permutation. That means different orders will be counted as one combination.

This problem is significant for a cashier. They need to make changes with the minimum number of coins as fast as possible. But here, we are trying to find the number of combinations with different algorithms for a comprehensive analysis.

1.2 Objective

The primary goal of this project is to explore and analyze different algorithmic approaches for solving the coin change problem, explicitly focusing on finding the number of ways to make a given change amount using an infinite supply of given coin denominations and comparing and analyzing the performance of these approaches in terms of time complexity, space complexity, and practical execution time. And we need to find out which algorithm is better than others.

1.3 Scope

We have used a total of three algorithms for solving this problem.

1. Brute Force Algorithm
2. Dynamic Programming - Recursive Approach
3. Dynamic Programing - Iterative Approach

We considered some assumptions, constraints for this problem, and some limitations of the brute force algorithm.

In our problem, there will be two inputs,

1. Coin set with the total coin number N.
2. Change Amount, sum.

Assumption & Constraints:

1. Infinite Coin Supply
2. coin >= 1
3. sum >= 0
4. The coin set contains all unique coins; no duplicates are allowed.
5. Coin sets are sorted by increasing the value of the coin.

Limitations:
We use one sample data for both dynamic recursive and iterative. However, we can't use the same sample for the brute force algorithm because dynamic programming is faster than the brute force algorithm. The Brute Force Approach has exponential time complexity, making it impractical for larger inputs, and the recursion depth can lead to stack overflow for large sums and coin sets. So, we reduced the sample size only for the brute force algorithm for time limitations.

Chapter 2 Analysis of Algorithms

2.1 Algorithm 1: Brute Force Algorithm

2.1.1 Description:

In general, the brute force algorithm is used for the optimization problem. Brute Force Algorithm finds all possible combinations and then chooses the best one. In this problem, we will use a brute force algorithm to find all possible combinations to make the change and count the combinations number.

Let's,

coins = [1, 5, 8, 10] //set of coin

sum = 13

Then, all the possible combinations to make the change are:

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 13

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 5 = 13

1 + 1 + 1 + 1 + 1 + 8 = 13

1 + 1 + 1 + 5 + 5= 13

1 + 1 + 1 + 10 = 13

5 + 8 = 13

So, there are six possible combinations to make the change amount 13. The brute force algorithm will find all the combinations within these four coins and count the combinations that make it possible to make changes.

Then, what could be the approach for that?

If we choose a coin, reduce the amount from the sum, and call the same function with the remaining sum, we will end up with all possible permutations.

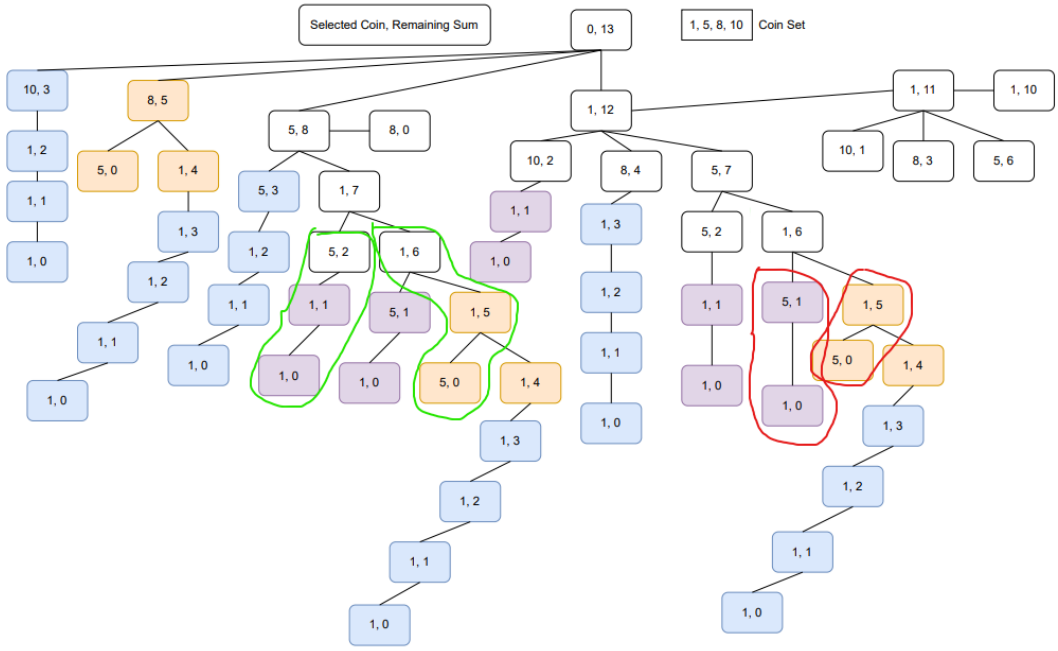

Figure 1- Difference between permutation and combinations.

In the diagram, we can see that for the sum amount 6, we can make it by choosing 5, 1 also by choosing 1, 5. The difference is the selection order, but both are the same combinations. So, how can we prevent that?

If we build our algorithm like that, after selecting one coin, it will not select the coin that is less than the previous coin chosen. That means it will not select the coin again after making all possible combinations by selecting one coin.

To do that, we need to control the coins' index number. Whatever we need to change or access in a recursive function, we must pass it through the recursive function parameter.

2.1.2 Pseudocode:

```
cashierBruteForceCombinations (coins, sum, startIndex)
        if sum became equal to 0
                return 1 //possible combination.
        if sum is negative or index >= length of coins
                return 0 //not a valid combination.
        with_startIndex = cashierBruteForceCombinations (coins, sum - coins[startIndex],
startIndex)
        without_startIndex = cashierBruteForceCombinations (coins, sum, startIndex+1)
        return with_startIndex + without_startIndex
```

2.1.3 Time Complexity Analysis:

In our algorithm, two recursive calls make decisions between two conditions. Select or exclude the coin representing a binary scenario like 1 or 0. The recursive tree's depth depends on the sum amount and the number of coins.

Because in each step, the problem is reduced by reducing the sum or the coin number. Also, in each step, the algorithm must make a maximum of N decisions. Depending on the decision, the problem is reduced to a smaller subproblem.

Potential Complexity Bounds:

- Each coin adds a binary decision point.

- The sum defines the depth of the recursion.

Therefore,

$$Time\ Complexity = 2^{(\text{sum} + \text{number of coin})}$$

Which represents an exponential graph.

2.1.4 Space Complexity Analysis:

In our computer processor, whenever we call a function, it will push the process info of the current state to a stack memory. Then, after completing the function, all the information from the stack memory will be restored. Here, whenever we call the brute force algorithm, it will store the current info in a stack and then run the function. Inside the function, when we call the recursive function, the current state will again be pushed to the stack.
In our algorithm, every recursive call will reduce the sum amount by the given coin value. So, if our coin set starts with 1, and if we provide the sum value as 5, then the (6th) line will reduce the sum by 1 and call recursively like this:
cashierBruteForceCombinations (coins, 4, 0)
cashierBruteForceCombinations (coins, 3, 0)
cashierBruteForceCombinations (coins, 2, 0)
cashierBruteForceCombinations (coins, 1, 0)
cashierBruteForceCombinations (coins, 0, 0) => Base case, will return

After the return, it will try to call another recursive call.
So, with the initial call, it will store the state info at most (Sum+1).
Therefore,

$$Space\ Complexity = O(sum + 1)$$

2.2 Algorithm 2: Dynamic Recursive Solution (Top – Down)

2.2.1 Description:

Dynamic Programming comes from control theory and is used for optimization problems. If, in a recursive problem, the algorithm needs to compute a repeating sub-problem too many times, then dynamic programming gives some significant enhancements. In dynamic programming, it calculates a subproblem, saves the result in an array, and uses it to solve bigger problems. There are two types of Dynamic Solutions. One is recursive, compute on demand, or use the top-to-bottom approach. Another one is iterative, compute all or bottom to top approach. Here, this algorithm is the recursive solution.

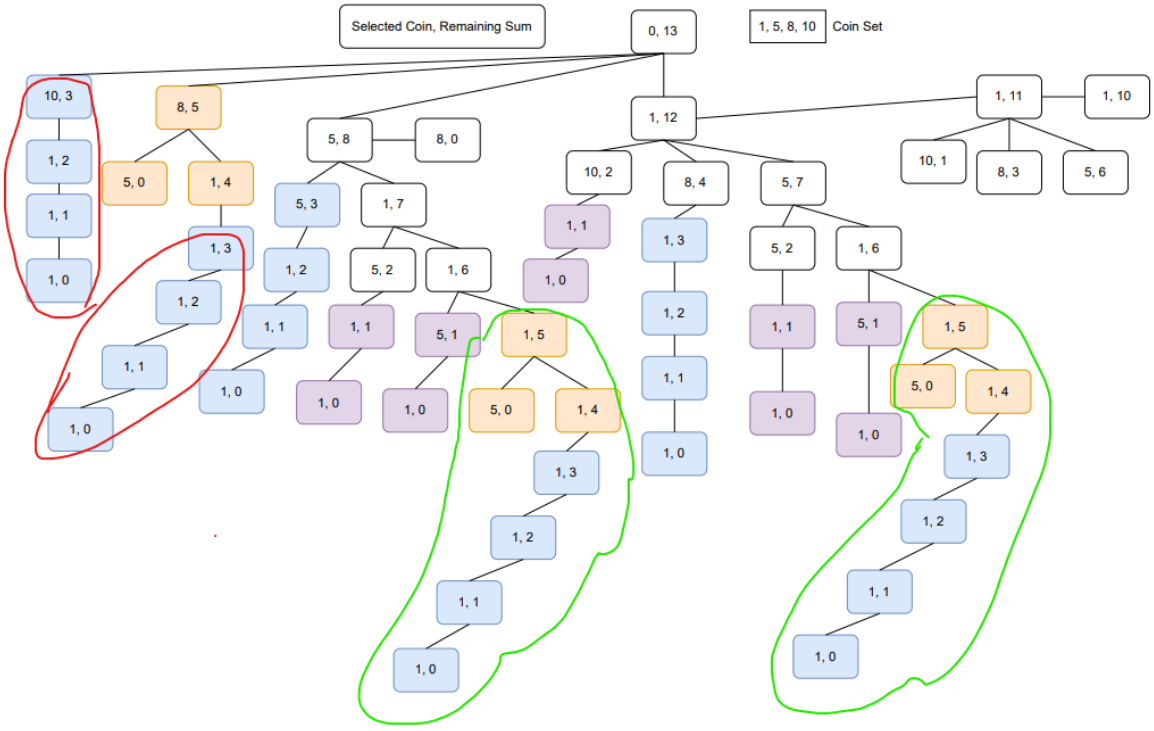

Figure 2 - Repeating Sub-Problems

Here, we can see that there are lots of subproblems that are repeatedly computing. We can store these in an array and use them on demand.

The algorithm will be the same as brute force. Additionally, we create an extra 2D array and store the result of each subproblem. We will add a base case to check whether it is computed.

2.3.2 Pseudocode:

```
cashierDynamicRecursive (coins, sum, startIndex)
        dp = [coin * sum] be a new global array
        if sum became equal to 0
                return 1 //possible combination.
        if sum is negative or index >= length of coins
                return 0 //not a valid combination.
        if dp [startIndex, sum] ≠ -∞
                return dp [startIndex, sum]
        with_startIndex = cashierBruteForceCombinations (coins, sum - coins[startIndex],
startIndex)

        without_startIndex = cashierBruteForceCombinations (coins, sum, startIndex+1)

        dp [startIndex, sum] = with_startIndex + without_startIndex
        return dp [startIndex, sum]
```

2.2.3 Time Complexity Analysis:

For the recursive call, its time complexity can be exponential. But, due to the use of a memorization system, we don't need to compute the repeated sub-problems. As a result, it will compute the output for each coin in depth to sum.
Therefore,

$$Time\ Complexity = O(number\ of\ coin * sum)$$

Which represents a linear graph.

2.2.4 Space Complexity Analysis:

It will already use the stack memory as we described earlier as O (sum + 1) for recursive calls.
However, in this algorithm, we used one extra 2D array, the size of which is (number of coins * sum).
Therefore,

$$Space\ Complexity = O\big((sum + 1) + (number\ of\ coins * sum)\big)$$
$$=> Space\ Complexity = O(number\ of\ coins * sum)$$

2.2 Algorithm 3: Dynamic Iterative Solutions (Bottom – Up)

2.3.1 Description:

Dynamic iterative solution: don't use any recursive call. It uses a loop to iterate and store all output from 0 to a given value. It is a bottom-to-top approach. It uses an array or table to store the output. Here, in our problem, we used a 1D array to store the solutions, which reduces the space complexity far better than recursive solutions.

2.3.2 Pseudocode:

```
cashierDynamicIterative (coins, sum)
        dp = [sum+1] be a new array

        dp [0] = 1 //initialize the base

        for each coin from coins

                for i = coin to sum+1

                        dp[i] = dp[i] + dp[i-coin]

        return dp[sum]
```

2.3.3 Time Complexity Analysis:

First "for loop" will run up to N (number of coins) times, and the inner loop will run up to the sum.
Therefore,

$$Time\ Complexity = O(number\ of\ coin * sum)$$

Which represents a linear graph.

2.3.4 Space Complexity Analysis:

There is no recursive call. As a result, no stack memory will be used. However, in this algorithm, we used one extra 1D array, the size of which is (sum + 1).
Therefore,

$$Space\ Complexity = O(sum + 1)$$

Chapter 3 Experimental Results

3.1 Algorithm 1: Brute Force Algorithm

3.1.1 Experimental Setup:

We used Python Language and JupyterLab as our IDE. It's very convenient, and we use our laptop hardware as the ipykernel.
Library:

1. random, used to generate a random dataset.
2. time, used to compute the execution time.
3. matplotlib, used for plotting the graph.
4. numpy, used to fit the graph in linear and polynomials.
5. pyperclip, used to copy the dataset on the clipboard.

We used different coins and sum amounts for the Brute Force Algorithm because it takes so much time to complete the execution. That is why we reduced the sample size for the brute force algorithm.

Coin Set used for the test:

Coin Set 1 (3 coins): [37, 51, 73]
Coin Set 2 (4 coins): [54, 73, 80, 93]
Coin Set 3 (5 coins): [6, 17, 25, 33, 64]
Coin Set 4 (6 coins): [6, 9, 11, 22, 29, 76]
Coin Set 5 (7 coins): [25, 26, 33, 47, 78, 90, 99]
Coin Set 6 (8 coins): [16, 28, 31, 64, 78, 83, 86, 96]
Coin Set 7 (9 coins): [4, 37, 44, 48, 53, 65, 67, 86, 94]
Coin Set 8 (10 coins): [12, 14, 21, 22, 29, 50, 57, 58, 74, 92]
Coin Set 9 (11 coins): [5, 10, 11, 16, 56, 66, 77, 82, 90, 94, 96]
Coin Set 10 (12 coins): [6, 10, 20, 23, 33, 38, 39, 41, 62, 87, 96, 98]

Sum used for the test:

random_sum = [150, 270, 351, 370, 385, 431, 510, 675, 831, 987]

fixed_sum = 987
coins = CoinSet [9]

In our problem, we have two variables.
So, first, we fixed a sum amount and varied the coin set to produce the graph of the Number of coins vs. Execution Time. As the complexity is exponential, we plot the graph using the line graph.

Here,
Sum = fixed
Coin Set = variable

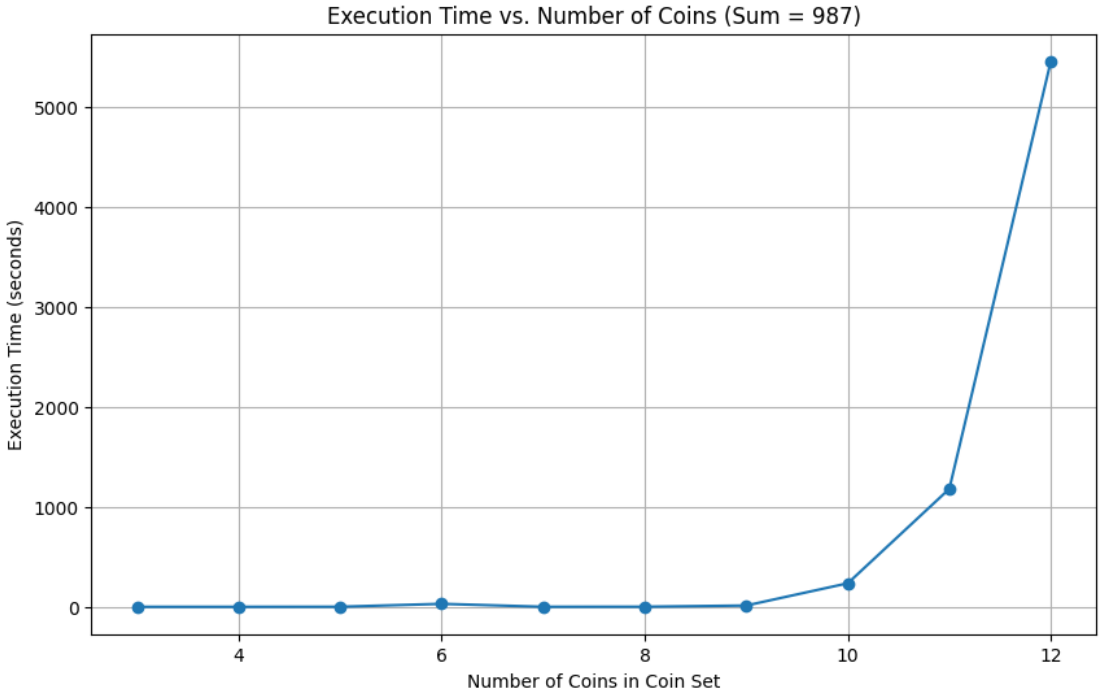

Figure 3 - Brute Force – Execution Time vs. Number of Coins (Fixed Sum)

Then, we use the same dataset to produce the complexity graph.

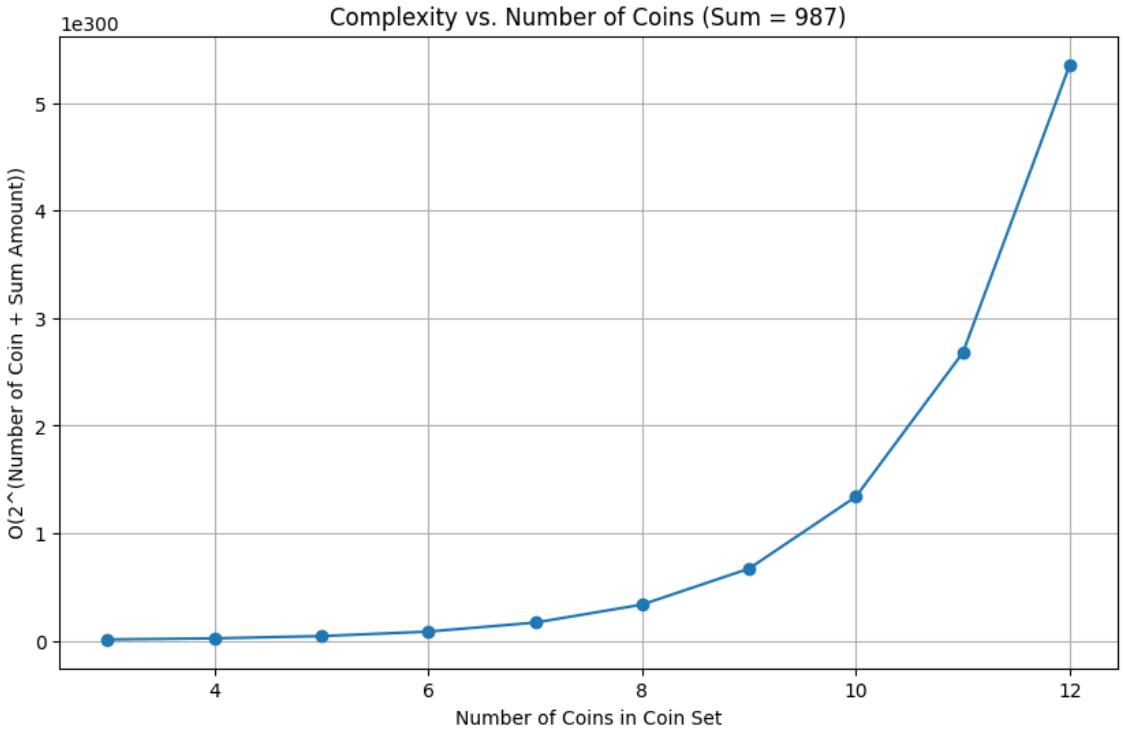

Figure 4 - Brute Force - Complexity vs. Number of Coins (Fixed Sum)

After that, we make a normalize data points and plotted two graphs in a single graph to show the similarity.

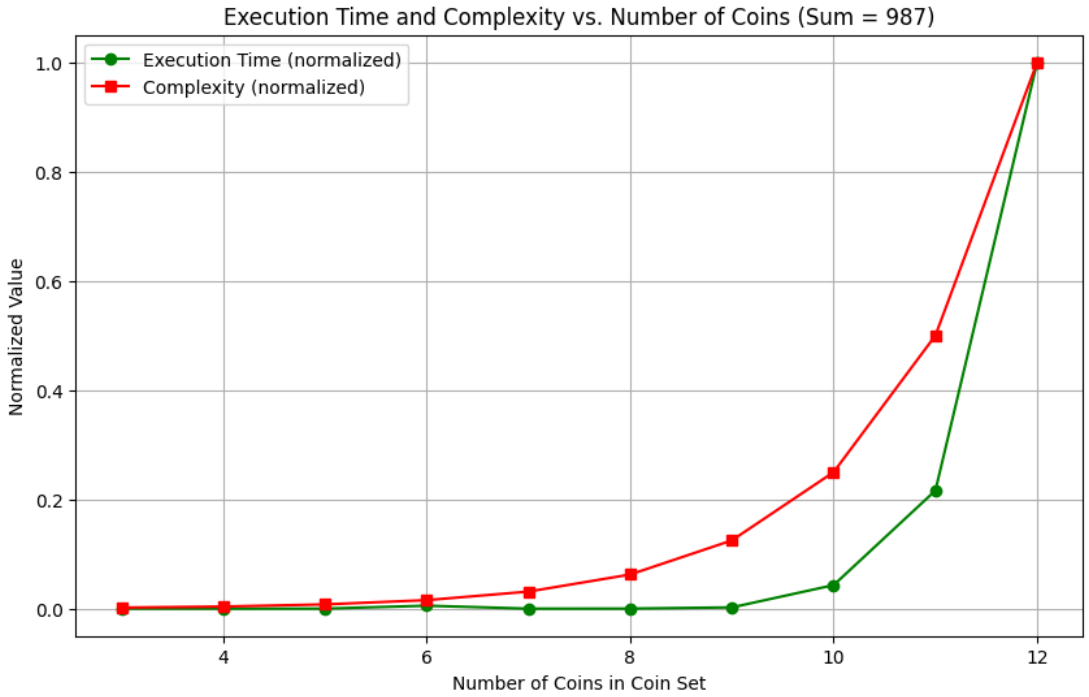

Figure 5 - Brute Force - Complexity vs. Execution Time (Fixed Sum)

Secondly, we fixed the coin set and varied the sum amount to produce the graph of Sum Value vs. Execution Time.
Here,
Sum = variable
Coin Set = fixed

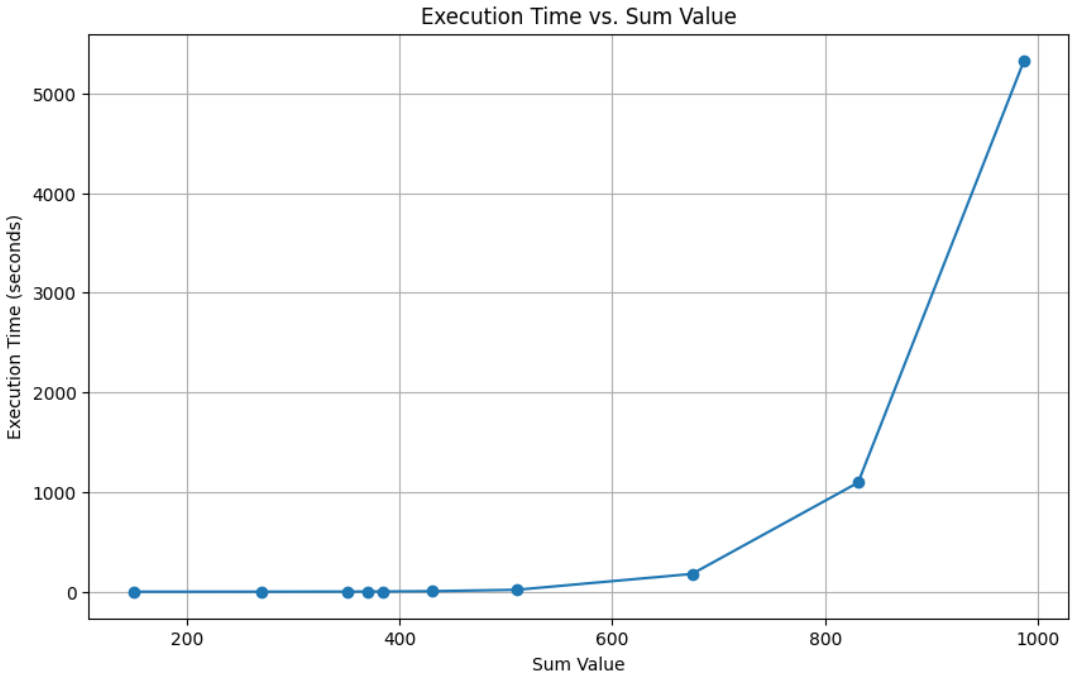

Figure 6 - Brute Force – Execution Time vs. Sum Value (Fixed Coin Set)

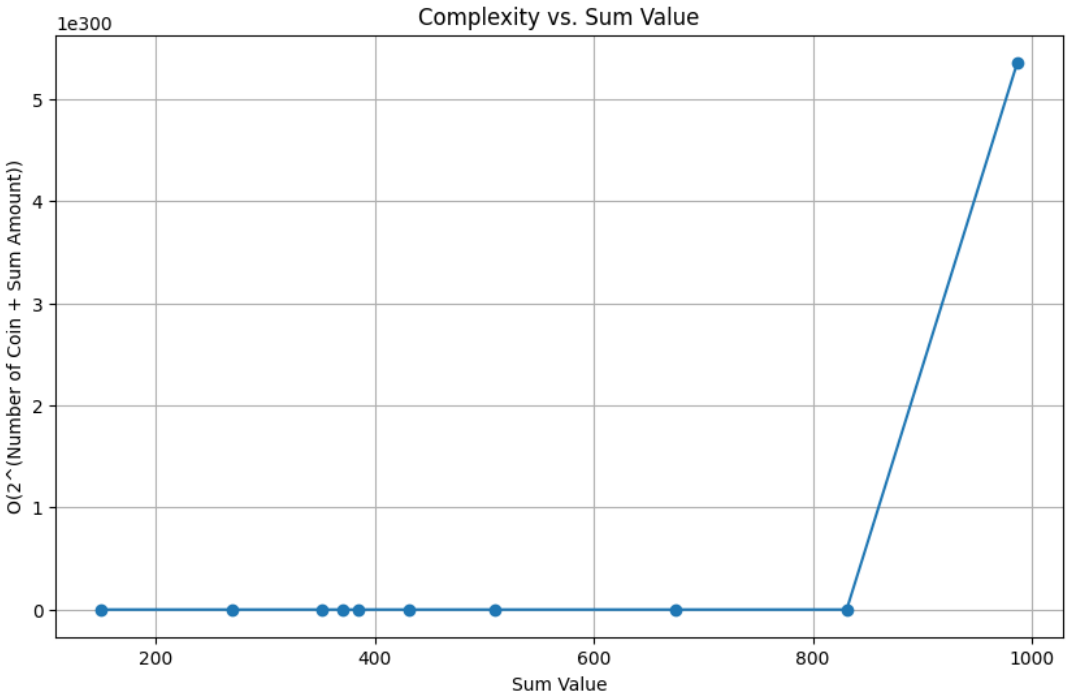

Figure 7 - Brute Force - Complexity vs. Sum Value (Fixed Coin Set)

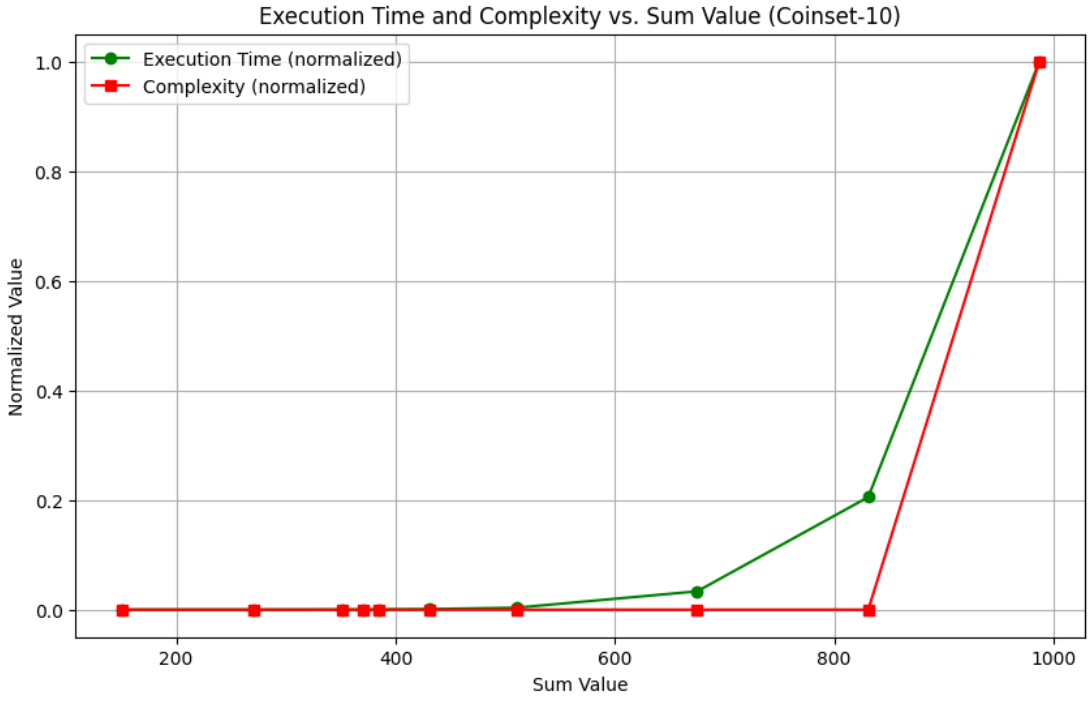

Figure 8 - Complexity vs. Execution Time (Fixed Coin Set)

In our problem, we are using a brute force algorithm. Whatever the input is, it doesn't matter; it will check all possible combinations to produce the final output. Therefore, our algorithm does not have a best-case or worst-case. All cases will have the same complexity.

3.2 Algorithm 2: Dynamic Programing – Recursive Approach

3.2.1 Experimental Setup:

All the libraries used are the same as the brute force algorithm. However, as dynamic programming is much faster than the brute force algorithm, we generate a large sample set to test the time complexity.

Coin Set used for the test:

Coin Set 1 (15 coins): [37, 47, 218, 231, 349, 367, 429, 452, 457, 536, 539, 708, 828, 871, 1000]

Coin Set 2 (19 coins): [68, 114, 123, 171, 199, 255, 295, 537, 553, 627, 631, 659, 661, 687, 866, 913, 929, 945, 966]

Coin Set 3 (21 coins): [11, 82, 119, 296, 332, 339, 348, 388, 404, 421, 436, 447, 472, 486, 543, 561, 570, 818, 845, 891, 991]

Coin Set 4 (30 coins): [25, 56, 84, 223, 341, 375, 401, 410, 452, 464, 514, 517, 520, 535, 559, 596, 720, 760, 785, 814, 829, 831, 857, 866, 893, 895, 912, 933, 953, 993]

Coin Set 5 (35 coins): [2, 10, 38, 59, 61, 63, 121, 160, 162, 166, 202, 205, 237, 263, 290, 313, 401, 402, 414, 459, 512, 576, 584, 614, 642, 648, 650, 671, 686, 689, 779, 798, 811, 870, 990]

Coin Set 6 (58 coins): [46, 71, 102, 103, 109, 140, 143, 172, 174, 178, 203, 218, 264, 266, 286, 287, 293, 343, 378, 384, 404, 455, 464, 467, 482, 493, 497, 498, 507, 525, 539, 550, 586, 602, 603, 606, 664, 679, 683, 694, 714, 751, 754, 810, 813, 814, 815, 865, 895, 899, 912, 919, 936, 953, 967, 970, 991, 995]

Coin Set 7 (72 coins): [3, 16, 21, 39, 47, 50, 86, 89, 104, 110, 129, 135, 166, 184, 217, 227, 253, 256, 264, 296, 313, 324, 351, 354, 360, 361, 382, 395, 468, 479, 481, 483, 488, 496, 498, 509, 527, 537, 562, 571, 582, 583, 599, 610, 641, 642, 644, 649, 669, 671, 682, 725, 734, 745, 782, 816, 834, 847, 862, 892, 896, 898, 904, 914, 917, 935, 937, 941, 949, 987, 989, 998]

Coin Set 8 (82 coins): [8, 12, 32, 40, 42, 46, 53, 65, 80, 83, 116, 118, 119, 150, 151, 158, 174, 175, 196, 198, 222, 225, 235, 238, 239, 256, 287, 296, 299, 300, 302, 308, 333, 347, 374, 381, 397, 403, 415, 433, 445, 482, 492, 495, 511, 512, 553, 564, 579, 583, 597, 600, 603, 659, 683, 686, 691, 697, 707, 711, 725, 730, 736, 750, 776, 778, 789, 830, 835, 836, 841, 853, 867, 868, 883, 893, 935, 971, 978, 990, 996, 998]

Coin Set 9 (93 coins): [1, 13, 15, 20, 21, 35, 55, 56, 74, 96, 112, 162, 170, 181, 186, 198, 200, 201, 211, 213, 216, 218, 233, 240, 249, 268, 276, 279, 295, 298, 299, 312, 322, 335, 337, 344, 347, 356, 371, 399, 407, 408, 412, 422, 432, 434, 448, 460, 468, 484, 489, 494, 524, 532, 537, 553, 556, 559, 562, 566, 583, 593, 599, 613, 624, 634, 639, 667, 680, 682,

688, 691, 701, 705, 721, 732, 748, 767, 808, 811, 812, 821, 850, 862, 868, 872, 889, 894, 901, 921, 934, 966, 1000]

Coin Set 10 (97 coins): [16, 19, 47, 61, 76, 78, 108, 121, 129, 143, 146, 155, 162, 164, 170, 180, 181, 182, 186, 227, 230, 236, 241, 243, 244, 258, 276, 282, 283, 284, 287, 293, 307, 334, 355, 378, 384, 386, 389, 395, 416, 439, 450, 454, 461, 473, 477, 480, 489, 498, 505, 518, 527, 550, 555, 563, 567, 577, 585, 599, 610, 616, 625, 626, 646, 663, 666, 692, 693, 708, 715, 724, 727, 729, 743, 744, 748, 750, 768, 790, 806, 823, 833, 840, 850, 854, 855, 874, 887, 892, 913, 924, 953, 968, 981, 994, 995]

Sum used for the test:

random_sum = [148379, 203158, 363103, 457966, 472585, 663706, 741838, 910374, 950201, 978368]

fixed_sum = 978368
coins = CoinSet [9]

We also modified the recursive max depth of the ipykernel as given below:

$$recursive\ maximum\ depth = fixed\_sum * 10$$

We experiment with the same way used for the brute force algorithm. We changed the dataset. As the complexity is Linear, we plot the graph using the Linear Fit graph.

Here,
Sum = fixed
Coin Set = variable

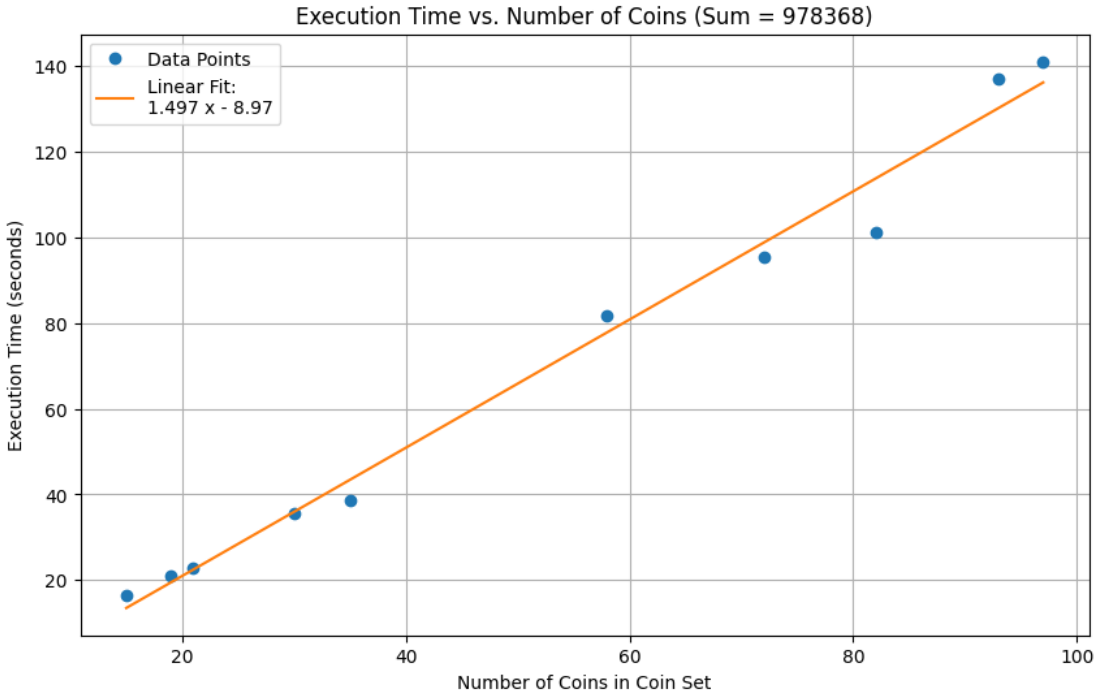

Figure 9 - Dynamic Recursive - Execution Time vs. Number of Coins (Fixed Sum)

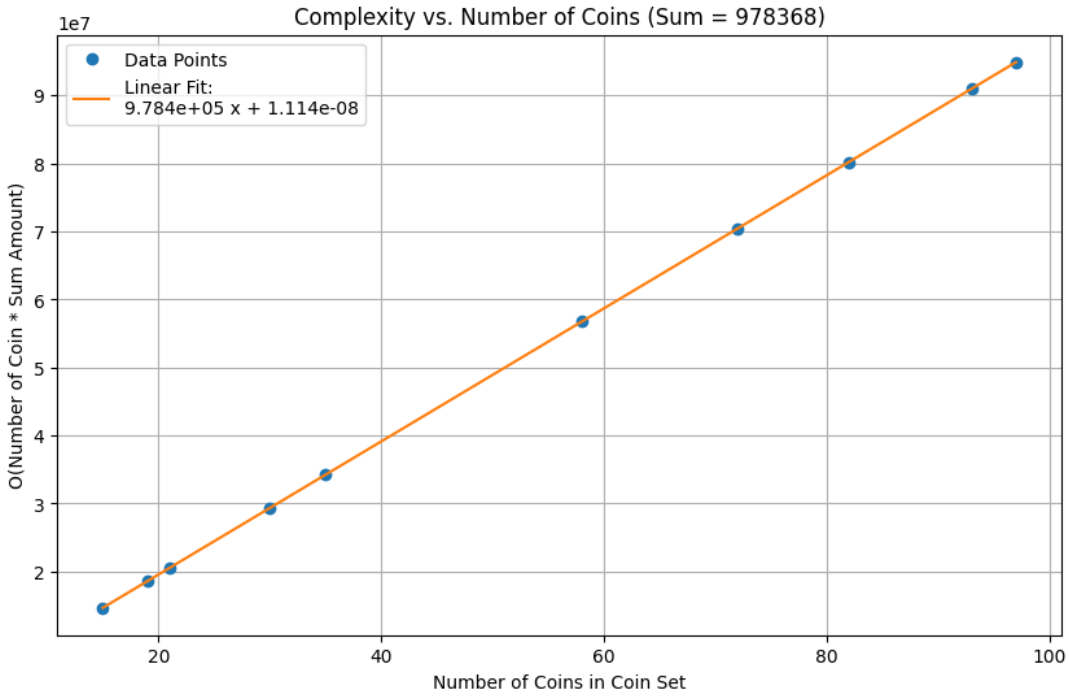

Figure 10 - Dynamic Recursive - Complexity vs. Number of Coins (Fixed Sum)

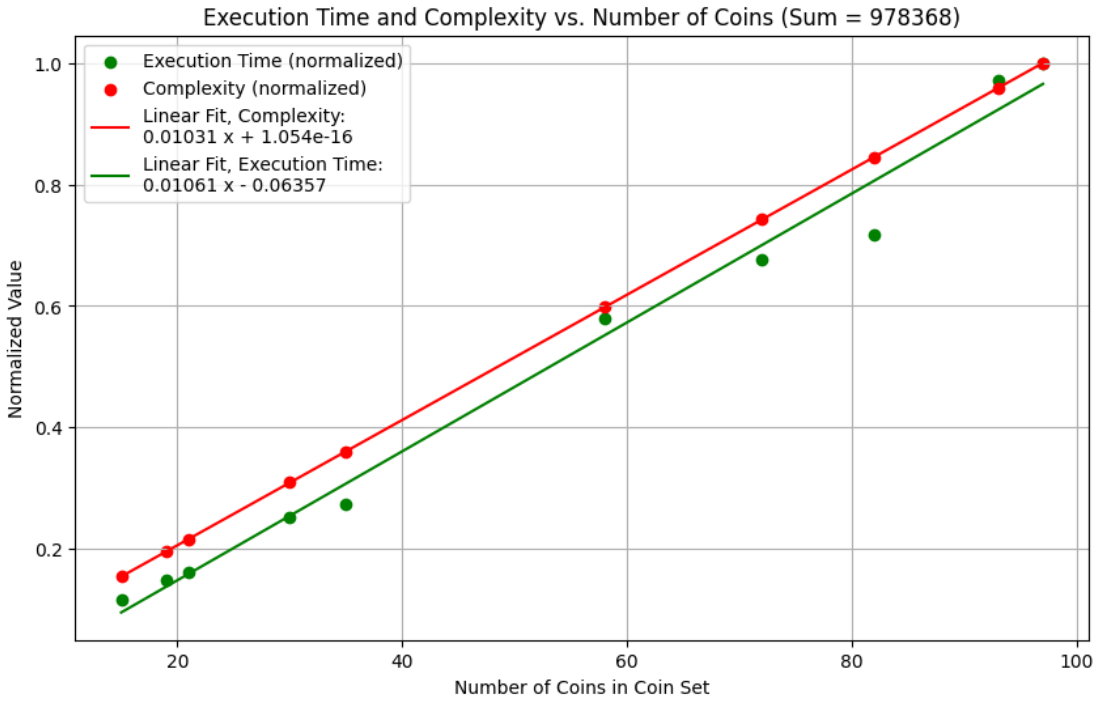

Figure 11 - Dynamic Recursive - Execution Time vs. Complexity (Fixed Sum)

Here,
Sum = variable
Coin Set = fixed

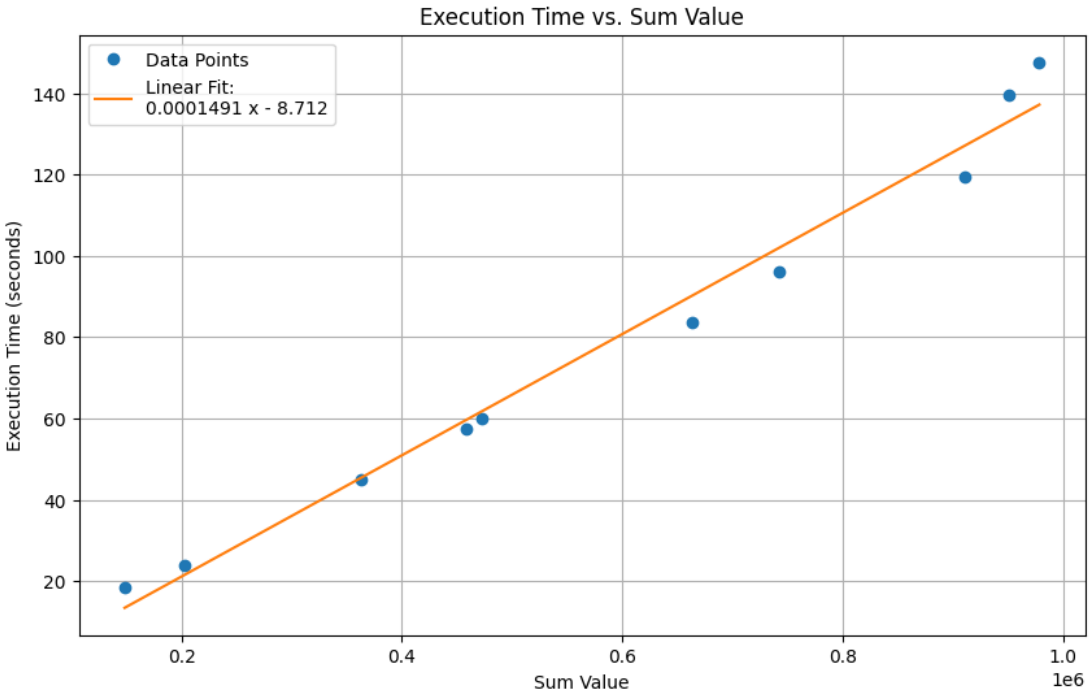

Figure 12 - Dynamic Recursive - Execution Time vs. Sum Value (Fixed Coin Set)

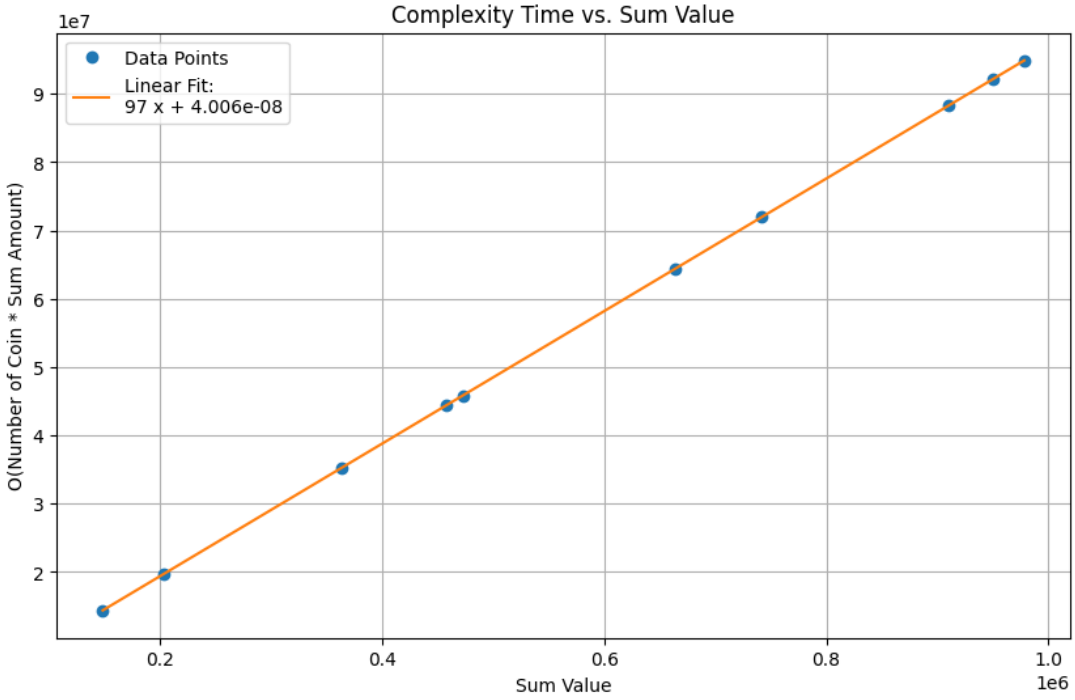

Figure 13 - Dynamic Recursive - Complexity vs. Sum Value (Fixed Coin Set)

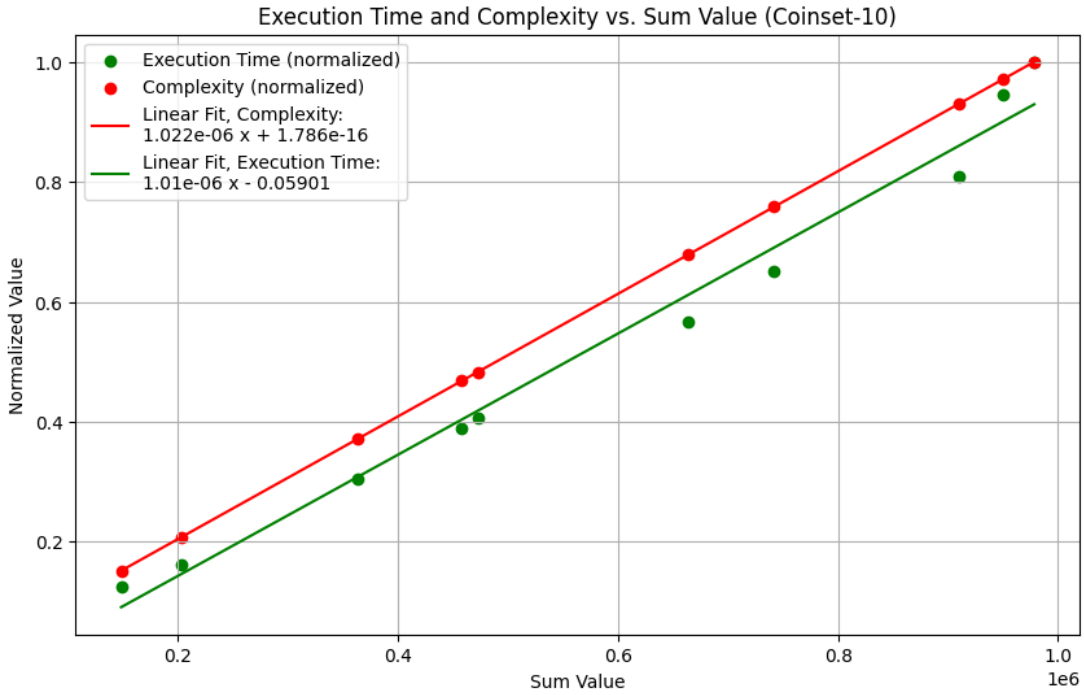

Figure 14 - Dynamic Recursive - Execution Time vs. Complexity (Fixed Coin Set)

There is no best case or worst case in our algorithm. All cases have the same complexity.

3.3 Algorithm 3: Dynamic Programming – Iterative Approach

3.3.1 Experimental Setup:

We used the same dataset as the dynamic recursive solutions. Here, we don't change anything; we just run the algorithm and capture the execution time for the plot. As the complexity is Linear, we plot the graph using the Linear Fit graph.

Here,
Sum = fixed
Coin Set = variable

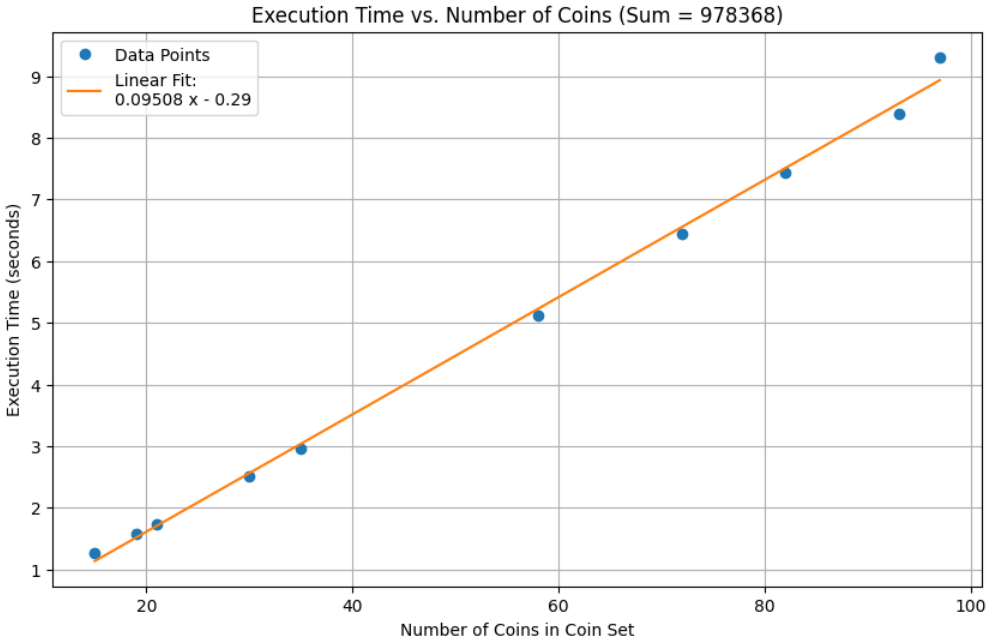

Figure 15 - Dynamic Iterative - Execution Time vs. Number of Coins (Fixed Sum)

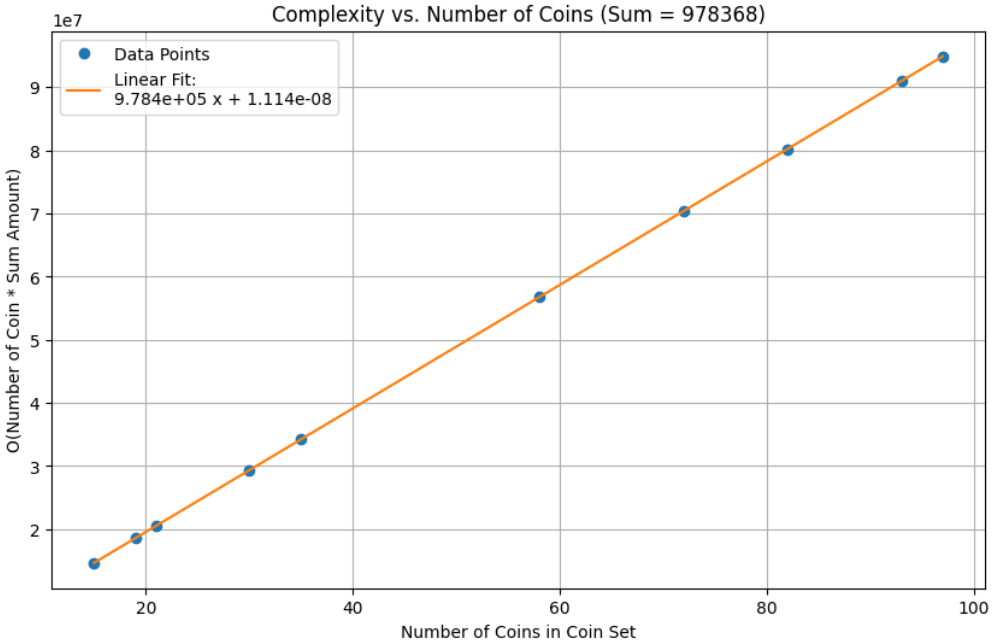

Figure 16 - Dynamic Iterative - Complexity vs. Number of Coins (Fixed Sum)

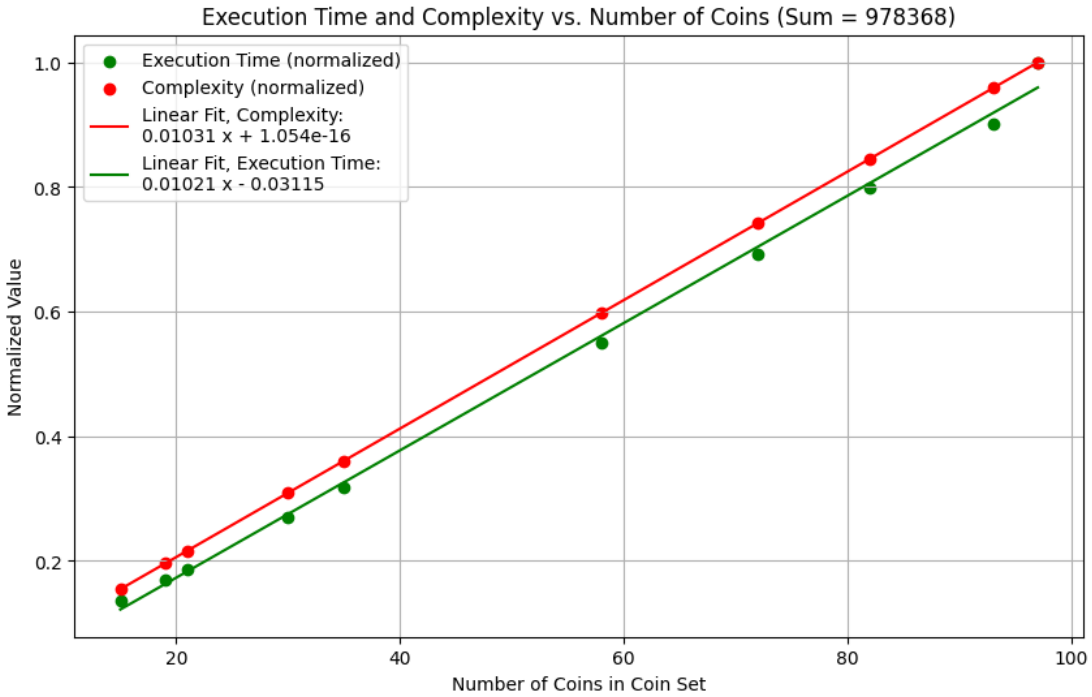

Figure 17 - Dynamic Iterative - Execution Time vs. Complexity (Fixed Sum)

Here,
Sum = variable
Coin Set = fixed

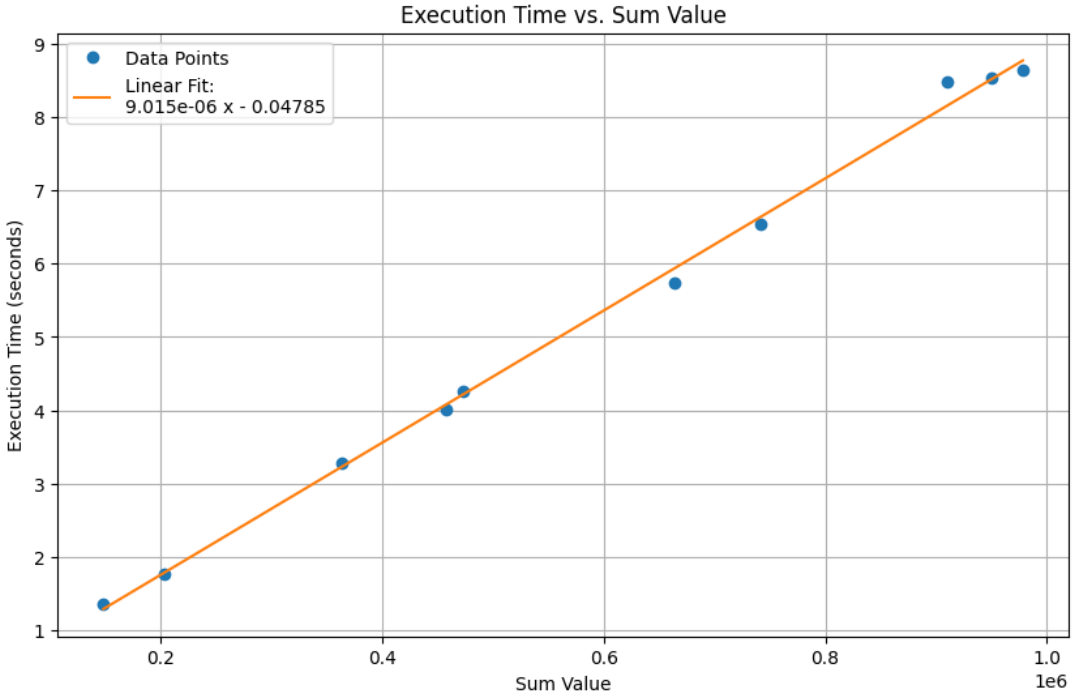

Figure 18 - Dynamic Iterative - Execution Time vs. Sum Value (Fixed Coin Set)

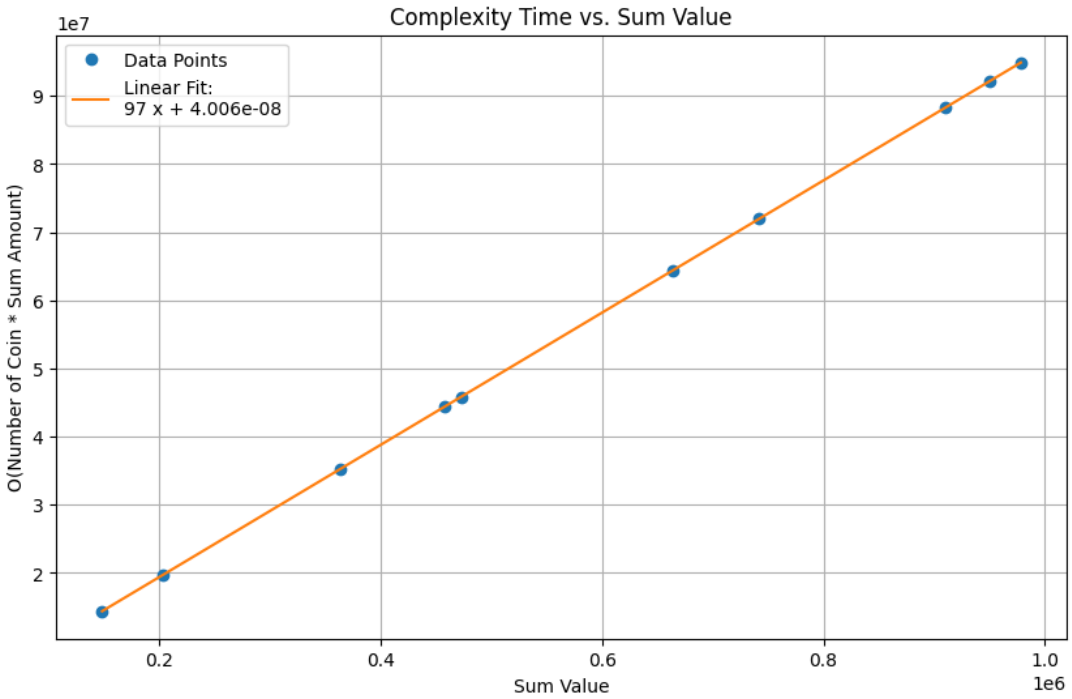

Figure 19 - Dynamic Iterative - Complexity vs. Sum Value (Fixed Coin Set)

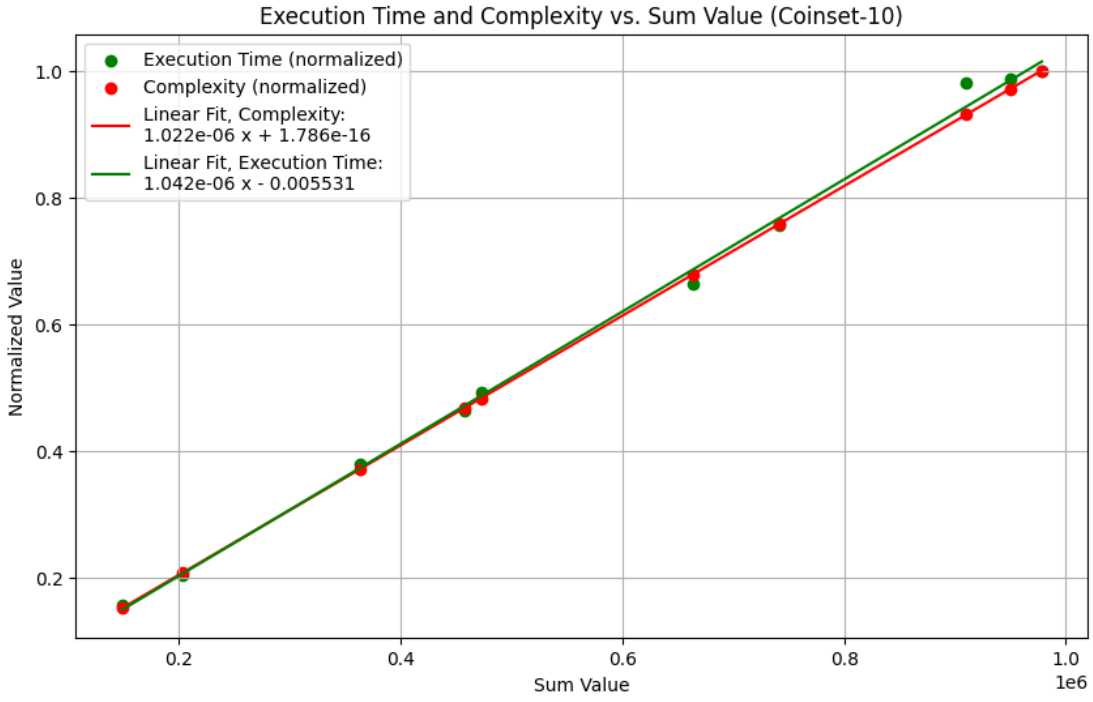

Figure 20 - Execution Time vs. Complexity (Fixed Coin Set)

As this algorithm computes for all 0 to sum, our algorithm has no best or worst case. All cases have the same complexity.

Chapter 4 Conclusions

4.1 Summary

In the brute force algorithm, we used a small sample size for the experiment and found that it takes at most 5 thousand seconds, which is too large than other algorithms. We use a sample size of at least ten times larger for the dynamic solution, but it still takes 147 seconds. Therefore, dynamic programming solutions are much better than the brute-force algorithm.

We used two dynamic algorithm types: top-to-bottom (recursive) and bottom-to-top (iterative). We provided the same sample data for the experiment. Here is the summary of these two algorithms represents in the graph:

Here,
Sum = fixed
Coin Set = variable

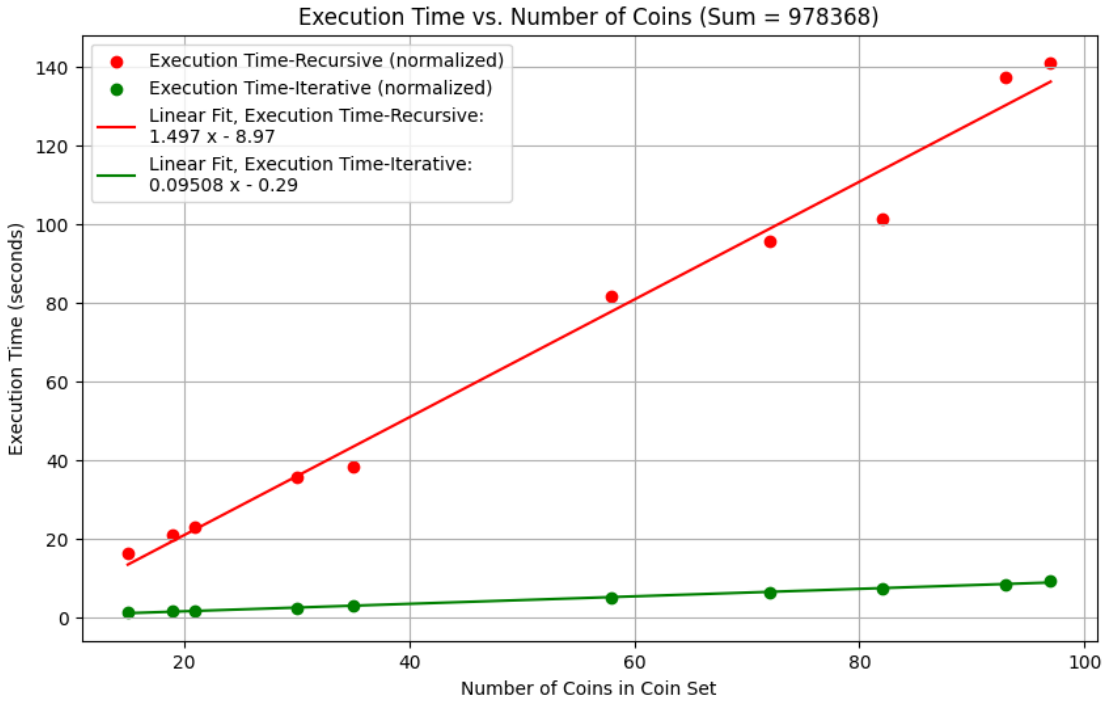

Figure 21 - Dynamic Recursive vs. Iterative Execution Time (Fixed Sum)

Here,
Sum = variable
Coin Set = fixed

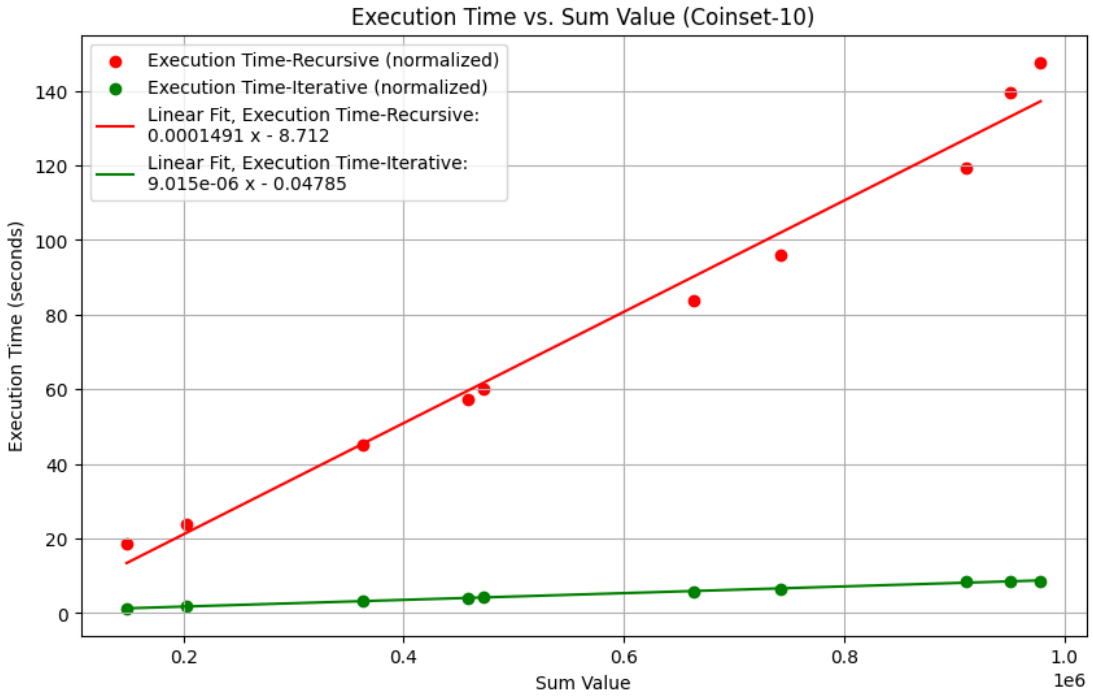

Figure 22 - Dynamic Recursive vs. Iterative Execution Time (Fixed Coin Set)

Here, we can see that the dynamic programming iterative approach is significantly better than the recursive approach.

4.2 Insights

So far, we have seen that the dynamic programming iterative approach gives the best output when comparing execution time and space complexity.

For space complexity, brute force and dynamic recursive are the same, and the dynamic iterative is less than others.

For the time complexity, dynamic recursive and iterative have the same big O notation, but iterative still takes less time than the recursive approach. This overhead time happened due to the push pop time in the stack for the recursive call. In the iterative approach, it doesn't call any recursive call nor use the stack. It's just iterating through the loop. Therefore, the dynamic programming iterative approach is most suitable for this problem.

We learned from this analysis that if a problem has to compute some repeating sub-problem, then dynamic programming is the best choice. The iterative approach is also the best choice.

References

1. https://www.geeksforgeeks.org/
2. ChatGPT
   For algorithms and plotting parameters only.