**Experimental Analysis on Dynamic programming approach solving Change-Making Problem**

**Introduction:**

In continuation to the previous analysis of Naïve recursive and Greedy approaches of solving change-making problem, this report explores the dynamic programming approach for solving the same problem.

**Importance:**

So, as per the earlier analysis, we figured that naïve recursive and Greedy approaches have their own constraints in finding the solution. So, it is still important to find the effective approach to find the solution. We will try to use dynamic programming approach if it overcomes the constraints of the other two approaches in this experiment.

**Theoretical Analysis:**

Let’s start with the theoretical analysis in which we will go through the pseudocodes of Dynamic programming algorithm that tries to solve coin-change problem.

*Change-Making Problem:* The problem involves making change for n cents using the fewest number of coins. Assume that each coin’s value is an integer. Assume that the coin system consists of k denominations of coin, and they are d1, d2, …, dk.

Note: *Let us assume that given denomination set can solve* ***any*** *given n amount. We assume that no country will design a weird coin system such as the example below in the real world.*

Example: let us say n = 9, k = 2, d = [2,4]. We cannot solve n using d. we assume such weird coin system input will not be given to the algorithms. (We can still handle this situation by adding some lines of code and giving output as “we can’t get solution for given coin system”.)

*Pseudocode of Dynamic Programming approach:*

Change\_Dp(n, d, k)

1. p = [0]
2. s = [0]
3. for i in range(1,n+1)
4. count = +ve infinity #Considering positive infinity value as max default count
5. for j in range(0,k)
6. if d[j] <= i
7. if count > p[i-d[j]]+1
8. count = p[i-d[j]]+1
9. coin\_index = j
10. p.append(count)
11. s.append(coin\_index)
12. out = []
13. while n>0 #listing the minimum number of denominations used for given amount
14. out.append(d[s[n]])
15. n=n-d[s[n]]
16. return out

where n is amount for which change is required, d is denomination list and k denote number of denominations. The program returns the coins used list that vending machine gives to the users.

p = [0] --- list to store the minimum number of denominations needed to make at p[i] for i cents.

s = [0] --- list to store index of least possible denomination to provide change for i cents.

*Based on Line 3 and line 5, we can see that algorithm takes Θ(n\*k) and from Line 13 to 15, algorithm takes* Θ(n) at most.

Hence, we can say that Time complexity of algorithm is *Θ(n\*k)*

**Practical Analysis:**

The table below represents the output of Dynamic programming approach for a weird Coin system that has denominations [25,23,10,5,1]:

|  |  |  |
| --- | --- | --- |
| **Amount(n)** | **Min num of coins used** | **Time in nano seconds** |
| 11 | [10,1] | 406400 |
| 23 | [23] | 407300 |
| 31 | [25, 5, 1] | 399900 |
| 51 | [25, 25, 1] | 358400 |
| 73 | [25, 25, 23] | 417600 |
| 83 | [25, 25,23 10] | 398400 |
| 91 | [25,23,23,10,10] | 432700 |
| 99 | [25,25,25,23,1] | 574300 |

*The output for the input amount (n) = 69 is [23,23,23] in time 408700 nano seconds.*

*The graph represents the plots of input amount(n) vs time in nano seconds using dynamic programming algorithm.*

**Correctness:**

The important phase is to talk about the correctness of actual results given by Dynamic programming algorithm.

**For US coin system:**

|  |  |  |
| --- | --- | --- |
| **Amount(n)** | **Expected** | Actual output |
| 11 | [10, 1] | [10, 1] |
| 23 | [10,10,1,1,1] | [10,10,1,1,1] |
| 31 | [25, 5, 1] | [25, 5, 1] |
| 51 | [25, 25, 1] | [25, 25, 1] |
| 73 | [25, 25, 10, 10, 1, 1, 1] | [25, 25, 10, 10, 1, 1, 1] |
| 83 | [25, 25, 25, 5, 1, 1, 1] | [25, 25, 25, 5, 1, 1, 1] |
| 91 | [25,25,25,10,5,1] | [25,25,25,10,5,1] |
| 99 | [25,25,25,10,10,1,1,1,1] | [25,25,25,10,10,1,1,1,1] |

**For Weird Coin System:**

|  |  |  |
| --- | --- | --- |
| **Amount(n)** | **Expected** | Actual output |
| 69 | [23, 23, 23] | [23, 23, 23] |

Hence, we can say that Dynamic programming approach gives the correct output for all kinds of valid coin systems.

Naïve Recursive always gives the correct output and Greedy approach does not always give correct!

**Comparison:**

From the above results we can observe that Dynamic Programming algorithm and Naïve recursive algorithm give the correct output. But Naïve recursive does take exponential time and more space (as recursion uses stack memory) which is not a good thing for any organization to use. Dynamic Programming algorithm takes ***Θ(n\*k)*** which if better than naïve recursive.

Greedy algorithm works for some type of coins and will not for some coin systems but takes linear time(***Θ(k)*** super-fast). We can say that the Greedy approach is also not a good way to use any coin system blindly.

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Correctness** |
| **Dynamic Programming** | ***Θ(n\*k)*** | **Always** |
| **Naïve Recursive** | ***exponential*** | **Always** |
| **Greedy Algorithm** | ***Θ(k)*** | **Sometimes** |

**Performances:**

As per my understanding the theoretical and practical analysis (by observing graphs) are consistent. I did not find any surprises but was excited in learning these things.

Dynamic programming approach performs well in all kinds of valid coin systems!

We can say naïve recursive performance is not at all good in terms of time complexity.

The Greedy algorithm is absolutely working fine for the US Coin System but sadly not working for some weird coin systems, but its performance is appreciable as it takes linear time.

**Conclusion:**

To conclude with the analysis of results, Dynamic programming approach can be used to find the solution for applying it to vending machines with which we can blindly use without worrying of any kind of denominations. Whereas Greedy algorithm can be used in coin systems like US to make the job done quicker.