CS 325
Group Assignment 2
Ujjval Kumaria, Vijay Vardhan Tadimeti, Aashwin Vats

1. Description of how DP table is filled for coin change problem:

   First, we create an array (for this example, the array is called table) ranging from values 1 to (Value + 1). Initially, we set table[0] to zero and the remaining elements to infinite. This satisfies our base case, where the given value maybe 0 and hence the number of coins is also 0. Then, we start filling the dynamic table.
   The values for table[1] through table[9] are set considering the first coin[0] of value 1. This is the minimum number of coins required to get the amount by using the denomination '1'.
   Now, the value to be stored at index 10 of our table will be the minimum of the following:

   Either: table[9] + 1, when coins[0] = 1 → No. of coins used: 10
   Or: table[0] + 1, when coins[1] = 10 → No. of coins used: 1

   As the minimum number of coins that can be used to get the amount 10 is 1, we update our DP table with this value.

   Similarly, the loop goes on to calculate the minimum number of coins required to achieve the amount in the DP table. If there are multiple ways to get the amount, we compare the minimum number of coins that will be used to get that amount by every denomination given to us. The minimum value is then updated in the DP table. This goes on until we reach the ith iteration where i = value.

2. Pseudocode for greedy implementation:

```
FUNCTION change_greedy(coins, value):
    n <- len(coins)
    coin_counts <- [0] * len(coins)
    max_denomination <- coins[-1]
    curr_element <- n - 1
    while (value != 0):
        IF (max_denomination > value):
            max_denomination <- coins[curr_element - 1]
            curr_element -= 1
        ELSE:
```

```
            coin_counts[curr_element] += 1
            value <- value - max_denomination

    RETURN coin_count
```

Psuedocode for dp implementation:

```
FUNCTION change_dp(coins, value):
    table <- [0 for i in range(value + 1)]

    ans <- [-1 for i in range(value + 1)]

    table[0] <- 0
    for i in range(1, value + 1):
        table[i] <- sys.maxsize

    for i in range(1, value + 1):
        for j in range(len(coins)):
            IF (coins[j] <= i):
                sub_res <- table[i - coins[j]]
                IF (sub_res != sys.maxsize AND
                        sub_res + 1 < table[i]):
                    table[i] <- sub_res + 1
                    ans[i] <- coins[j]

    RETURN table[value], ans
```

3. Theoretical run-time analysis for greedy implementation:

   If we look at the highlighted elements of the above pseudocode. The while loop runs from 0 to V (where V is the value for which change is needed), let's say 'v' times. Hence, the run time for this algorithm is O(v) since there is only one loop. The same can be seen from the analysis on the above pseudocode as the number of coins is added once for every denomination.
   Time complexity: O(v)

   Theoretical run-time for DP implementation:

   Let's say that the length of coins array is n. Hence, the number of denominations present to us is 'n'.

From the above pseudocode, we can deduce that the running time is O(nv). We can represent table[i] as a fixed-size array that takes O(1) time to access any element of the table. It takes O(n) time to construct each table[i] for any i. There are O(v) values of i to consider.
Time complexity: O(nv)

4. Proof by induction that dp implementation is correct:

   Given a set *coins* = *{ coins[1], …, coins[m] }* $\subset N$ of coin denominations, for $i \in N_0$ let *T(i)* denote the minimum number of coins (with repetition) in *D* needed to obtain sum *i* (or *T(i)* = ∞ if it is impossible). Then clearly *T(i)* ≥ *0* for all *i* and *T(i)* = *0* ⇔ *i* = *0*.

   If *i* > *0* and a way to obtain *i* with *T(i)* coins uses at least one coin of denomination *coins[j]* (at least one such *j* exists), then by removing this coin we obtain a way to obtain *n* − *coins[j]* and hence conclude:

   T(i − *coins[j])* ≤ *T(i)* − *1* for at least one *1* ≤ *j* ≤ *m*

   On the other hand, if *1* ≤ *j* ≤ *m* and *i* ≥ *coins[j],* we obtain a way to obtain *i* with *T(i* − *coins[j])* + *1* coins by adding a coins[j] coin to an optimal way to get i − coins[j]. We conclude:

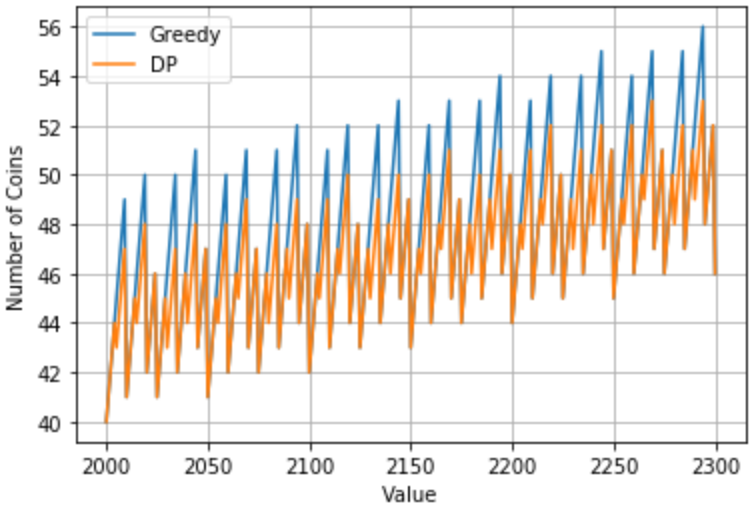
   T(i) ≤ *T(i* − *coins[j])* + *1* for all *1* ≤ *j* ≤ *m with i* ≥ *coins[j]*

   Together this gives us:

   T(i) = min{ T(i − *coins[j])* + *1* ∣ *1* ≤ *j* ≤ *m, i* ≥ *coins[j] }* for all *i* > *0* ...(1)

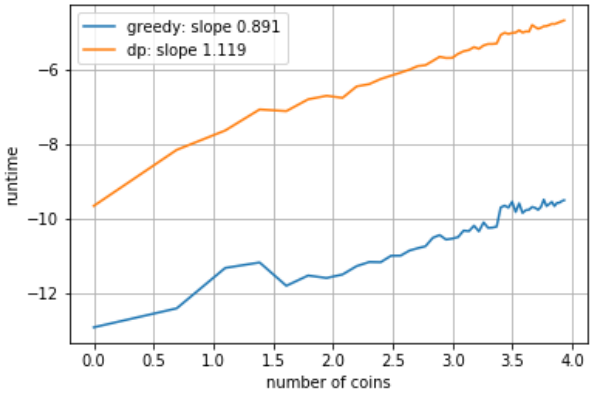
   (with the min ∅ = ∞ understood). And (1) together with T(0) = 0 is precisely what the DP algorithm uses to compute T(i) recursively.

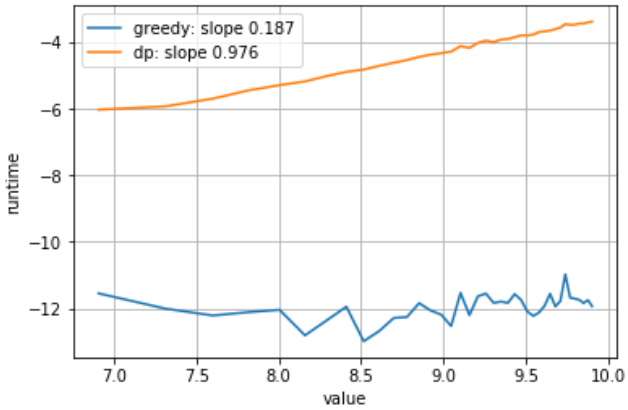
5. Plot



The dynamic programming implementation seems to edge the greedy implementation in a majority of the cases by providing a solution with a lesser number of coins.

6. Plot





The above graphs show our run-time with value and number of coins analysis for each algorithm to compare both approaches. The DP approach gives out a slope of 0.976(as compared to 0.187 in case of greedy) that correlates to an increase in value by 1. This is what our theoretical analysis also indicated above. At the same time, we can see that both approaches have a strong correlation with run time as the number of coins increases with an increasing runtime.
We can say from our observation of the outputs of code implementation that the greedy approach doesn't necessarily produce an optimal solution. However, from the above graph, it can be seen that in terms of running time, the greedy algorithm is going to be faster than the DP algorithm. Dynamic programming will always produce the optimal solution regardless of values.

7. To solve the problem by calculating the number of possible ways to calculate a specific amount(value), we would solve it using recursion where we take into consideration to include or not include a coin in the solution.
   We start by creating a dynamic programming table that's indexed from 0 to (Value+1). Note that the value here, is the amount specified by the user to calculate the combinations of coins. Now, we will try to fill the dp table with the number of combinations that can get you the specified amount. Here is how we'll do it,
   If Amount> Coin then,
   Table[Amount] += Table[Amount-Coin]

   So for example, if we have the following coin array,
   [1, 2, 5]
   Amount = 5
   So, there are a number of ways to get this amount from the above example,
   [5], [1, 1, 1, 1, 1, 1], [1, 2, 2], [1, 1, 1, 2]
   So, the output is **4**(number of combinations to get the amount 5)**.**
   We know for every coin whether to include or exclude it in the answer, we check if the value of the coin is <= to the total. If that is the case, we can find the number of ways by including and excluding that coin. When we include the coin: We reduce the total by the value of the coin and use the subproblem answer (total-v[i]) (from dp). When we exclude the coin: we would consider the answer for the same total without considering that coin. That means, If the value of the coin is greater than the total then we cannot consider that coin, hence the answer will be without considering that coin.
   At total 0, we would return the empty set for making the change, (This is 1 way of making the change). If we do not have any coins, then it means that there are no ways (0 ways) to change the total.