Project 4: The Change Making Problem
CS 3343: Analysis of Algorithms Summer 2024
Instructor: Dr. Mohammad Imran Chowdhury
Total: 100 Points
Due: 07/23/2024 11:59 PM

In this project, I invite you to solve the ``The Change Making Problem" using the various aspects of algorithmic thinking such as the Greedy approach, the Dynamic Programming approach, and Python programming discussed in class.

Problem Description:

The change-making problem involves **finding the minimum number of coins** from a set of denominations that add up to a given amount of money. For example, say you have coins available of denominations 1p, 2p, 5p, 10p, 20p, 50p, £1 and £2, as in the UK.

What is the minimum number of coins you need to make 24p? Using the greedy approach, the solution is 3: 20p + 2p + 2p

What about for £1.63?
Again, using the greedy approach, the solution is 5: £1 + 50p + 10p + 2p + 1p

Note: £1=100p, £2=200p, so on. Also, p stands for pence (p), £ stands for pound (£)

Greedy Approach

A greedy algorithm is one which makes locally optimal choices at any given point, and once a choice is made, does not revisit it. This can make the algorithm "short-sighted", and it may not find the optimal solution. However, there are advantages to the greedy approach such as often being quite fast, and relatively easy to implement.

Now to solve ``The Change Making Problem" using the **greedy approach you could follow** the pseudocode Algorithm 1 below to implement the idea:

Algorithm 1: *Change Making Problem* **(Greedy Approach)**

```
   input  : denominations, target_amount
   output: coin_count, values
1  coin_count ← 0                                  ▷ Initialize count;
2  values ← []                                     ▷ Store values here;
3  Sort the array of coins in decreasing order     ▷ Coin denominations are sorted in decreasing order;
4  forall coin in denominations do
5      while target_amount >= coin do
6                                  ▷ Use the current coin until its value is too large;
7          target_amount ← target_amount − coin    ▷ Decrease the remaining amount;
8          values.append(coin)                     ▷ Make a note of which coin was used;
9          coin_count ← coin_count + 1             ▷ Increase the coin count;
10 return coin_count, values;
```

Sample I/O:

Run 1:
Enter the coins : 200 100 50 20 10 5 2 1
Enter the target sum: 163
no. of coins: 5 , coins are: [100, 50, 10, 2, 1]

Impression: Very fast response time. This means the greedy approach is quite fast.

Run 2:
Enter the coins : 10 7 1
Enter the target sum: 15
no. of coins: 6 , coins are: [10, 1, 1, 1, 1, 1]

Impression: Not an optimal solution. See the Dynamic programming version's output in the next section. This means the greedy approach may not find the optimal solution. That is, the greedy algorithm (the bigger coins we use, the less count we need) doesn't work actually sometimes.

Run 3:
Enter the coins : 3 2
Enter the target sum: 4
no. of coins: 1 , coins are: [3] Greedy approach has failed to find the target sum 4 !!!

Impression: The greedy approach has **failed!** This means the greedy approach may not find the solution at all.

Note that you don't have to print the impression comments as part of your program output. This is just for better realization of the differences between the Greedy approach and the Dynamic programming approach.

Dynamic Programming Approach:

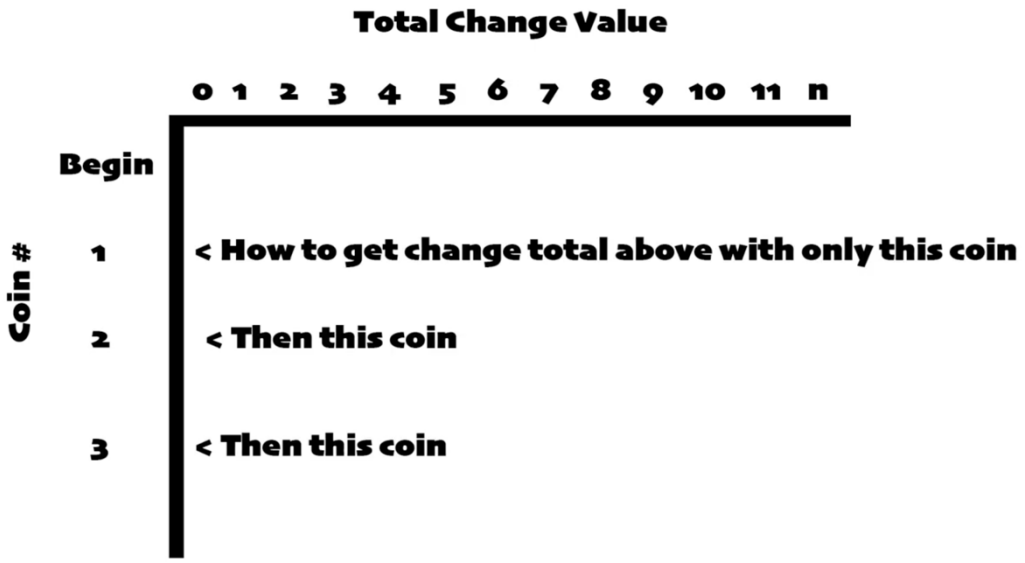
With UK coins, and many other sets, the greedy algorithm actually gives the optimal solution. This is not always the case, however, and with some combinations of available denominations, the greedy algorithm fails. For example, if you only had coins of denominations 2p and 3p, and wanted to make 4p, the **greedy algorithm** would select 3p first, then have no way to make the required value of 4p.

One thing that makes ``The Change Making Problem" interesting is that it can be solved in other ways which avoid some of the pitfalls of the greedy approach. For example, it can be solved using **dynamic programming**.

Using dynamic programming you would be able to find the solution for the above-mentioned example on which the greedy approach has failed. That is if you only had coins of denominations 2p and 3p, and wanted to make 4p, the **dynamic algorithm** would select 2p the first time, then select 2p again for the second time to make the required value of 4p.

That is unlike the greedy approach, once a choice is made, the dynamic programming approach revisits it to get the target solution.

Hence a simple analogy to solve ``The Change Making Problem" using the dynamic programming (tabulation) approach is depicted in the following figure (that **you could follow)**:



That is to solve ``The Change Making Problem" using the **dynamic programming (tabulation)** approach **you could follow** the pseudocode Algorithm 2 below to implement the idea:

Algorithm 2: *Change Making Problem* **(Dynamic Programming)**

```
input  : coins, amount
output: table[amount]
table ← [];
forall k in range(amount+1) do
    table.append([]);
table[0].append([])                    ▷ table[0] = [[]], if you prefer;
forall coin in coins do
    forall x in range(coin,amount+1) do
        forall solution in table[x-coin] do
            table[x].append(solution + [coin]);
return len(min(table[amount], key = len)), min(table[amount], key = len));
```

Note: Your program should match the exact sample input output format like mine.

Sample I/O:

Run 1:
Enter the coins : 200 100 50 20 10 5 2 1
Enter the target sum: 163
no. of coins: 5 , coins are: [100, 50, 10, 2, 1]

Impression: Slow response time. This means the Dynamic programming approach is quite slower whereas the greedy approach is generally faster.

Run 2:
Enter the coins : 10 7 1
Enter the target sum: 15
no. of coins: 3 , coins are: [7, 7, 1]

Impression: Optimal solution. This means the Dynamic Programming approach **guarantees** the optimal solution when compared to the Greedy approach. You can match the output with the greedy solution mentioned above i.e., no. of coins: 6 , coins are: [10, 1, 1, 1, 1, 1].

Run 3:
Enter the coins : 3 2
Enter the target sum: 4
no. of coins: 2 , coins are: [2, 2]

Impression: The Dynamic programming approach has **found the solution!** This means the Dynamic Programming approach guarantees the solution whereas this is not the case for the Greedy approach always. You can match the output with the greedy solution mentioned above i.e., no. of coins: 1 , coins are: [3] Greedy approach has failed to find the target sum 4 !!!

Note that you don't have to print the impression comments as part of your program output. This is just for better realization of the differences between the Greedy approach and the Dynamic programming approach.

The submission grading rubric is as follows (points out of 100 total):

Project element	**Points**
Code readability such as the usage of comments in code	10
Proper coding implementation • Greedy Approach • Dynamic Programming Approach	(30+40) = 70 30 40
Screenshots of the program output	10
Program's sample Input/Output format matching as per project writeup	10

Submission Instructions: Create a compressed file (.zip or .tar.gz files are accepted) with all your source files such as .py files. Within this compressed .zip folder, you should provide some screenshots of your running program's output as proof. Generally speaking, to complete ``The Change Making Problem", you need a maximum of two .py files, one for the Greedy approach and the other one for the Dynamic programming approach. But it's better to submit everything as a compressed file. Submit the compressed file to Canvas.

Late submission policy: As described in the syllabus, any late submission will the penalized with 10% off after each 24 hours late. For example, an assignment worth 100 points turned in 2 days late will receive a 20-point penalty. Assignments turned in 5 or more days after the due date will receive a grade of 0.