Consider the coin-change problem: Given a set of coin types and an amount of change to be returned, determine the fewest number of coins for this amount of change.

1) What "greedy" algorithm would you use to solve this problem with US coin types of \{1, 5, 10, 25, 50\} and a change amount of 29-cents?

2) Do you get the correct solution if you use this algorithm for coin types of \{1, 5, 10, 12, 25, 50\} and a change amount of 29-cents?

3) One way to solve this problem in general is to use a divide-and-conquer algorithm. Recall the idea of **Divide-and-Conquer** algorithms.
   Solve a problem by:
   - dividing it into smaller problem(s) of the same kind
   - solving the smaller problem(s) recursively
   - use the solution(s) to the smaller problem(s) to solve the original problem

   a) For the coin-change problem, what determines the size of the problem?

   b) How could we divide the coin-change problem for 29-cents into smaller problems?

   c) If we knew the solution to these smaller problems, how would be able to solve the original problem?
4) After we give back the first coin, which smaller amounts of change do we have?

![Diagram of possible first coins and smaller problems]

5) If we knew the fewest number of coins needed for each possible smaller problem, then how could determine the fewest number of coins needed for the original problem?

6) Complete a recursive relationship for the fewest number of coins.

\[
\text{FewestCoins}(\text{change}) = \begin{cases} 
\min( \text{FewestCoins}(\text{change} - \text{coin}) ) + 1 & \text{if change} \in \text{CoinSet} \\
\text{if change} \notin \text{CoinSet} 
\end{cases}
\]

7) Complete a couple levels of the recursion tree for 29-cents change using coin types \{1, 5, 10, 12, 25, 50\}.

![Diagram of recursion tree for 29 cents]

8) For coins of \{1, 5, 10, 12, 25, 50\}, typical timings using a recursive (backtracking) program:

<table>
<thead>
<tr>
<th>Change Amount</th>
<th>Run-Time (seconds)</th>
<th>Number of Call-Tree Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>0.92</td>
<td>236,583</td>
</tr>
<tr>
<td>300</td>
<td>33.23</td>
<td>8,617,265</td>
</tr>
<tr>
<td>320</td>
<td>64.12</td>
<td>16,676,454</td>
</tr>
<tr>
<td>340</td>
<td>116.8</td>
<td>30,370,729</td>
</tr>
</tbody>
</table>

Why the exponential growth in run-time?
9) As with Fibonacci, the coin-change problem can benefit from dynamic program since it was slow due to solving the same problems over-and-over again. Recall the general idea of dynamic programming:

- Solve smaller problems before larger ones
- Store their answers (so you never need to recompute them)
- Look-up answers to smaller problems when solving larger subproblems

a) How do we solve the coin-change problem using dynamic programming?
Dynamic Programming Coin-change Algorithm:

I. Fills an array `fewestCoins` from 0 to the amount of change. An element of `fewestCoins` stores the fewest number of coins necessary for the amount of change corresponding to its index value.

For 29-cents using the set of coin types \{1, 5, 10, 12, 25, 50\}, the dynamic programming algorithm would have previously calculated the `fewestCoins` for the change amounts of 0, 1, 2, ..., up to 28 cents.

II. If we record the best, first coin to return for each change amount (found in the “minimum” calculation) in an array `bestFirstCoin`, then we can easily recover the actual coin types to return.

```
fewestCoins[29] = minimum(fewestCoins[28], fewestCoins[24], fewestCoins[19], fewestCoins[17], fewestCoins[4]) + 1 = 2 + 1 = 3
```

a minimum for 29 given by 5-cent coin


b) Extend the lists through 32-cents.

```
fewestCoins: 4 17 19 24 28 29
bestFirstCoin: 0 12 24 29
```

```
fewestCoins: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
bestFirstCoin: 0 1 1 1 1 1 1 1 10 1 12 1 1 5 1 5 1 1 10 1 10 1 12 25 1 1 1 5
```

c) What coins are in the solution for 32-cents?