Dynamic Programming

Many programs in computer science are written to optimize some value; for example, find the shortest path between two points, find the line that best fits a set of points, or find the smallest set of objects that satisfies some criteria. There are many strategies that computer scientists use to solve these problems. One of the goals of this book is to expose you to several different problem solving strategies. **Dynamic programming** is one strategy for these types of optimization problems.

The Greedy Method

A classic example of an optimization problem involves making change using the fewest coins. Suppose you are a programmer for a vending machine manufacturer. Your company wants to streamline effort by giving out the fewest possible coins in change for each transaction. Suppose a customer puts in a dollar bill and purchases an item for 37 cents. What is the smallest number of coins you can use to make change? The answer is six coins: two quarters, one dime, and three pennies. How did we arrive at the answer of six coins? We start with the largest coin in our arsenal (a quarter) and use as many of those as possible, then we go to the next lowest coin value and use as many of those as possible. This first approach is called a **greedy method** because we try to solve as big a piece of the problem as possible right away.

The greedy method works fine when we are using U.S. coins, but suppose that your company decides to deploy its vending machines in Lower Elbonia where, in addition to the usual 1, 5, 10, and 25 cent coins they also have a 21 cent coin. In this instance our greedy method fails to find the optimal solution for 63 cents in change. With the addition of the 21 cent coin the greedy method would still find the solution to be six coins. However, the optimal answer is three 21 cent pieces.

Optimization Through Recursion

Let’s look at a method where we could be sure that we would find the optimal answer to the problem. Since this section is about recursion, you may have guessed that we will use a recursive solution. Let’s start with identifying the base case. If we are trying to make change for the same amount as the value of one of our coins, the answer is easy, one coin.

If the amount does not match we have several options. What we want is the minimum of a penny plus the number of coins needed to make change for the original amount minus a penny, or a nickel plus the number of coins needed to make change for the original amount minus five cents, or a dime plus the number of coins needed to make change for the original amount minus ten cents, and so on.

The algorithm for doing what we have just described is shown below.

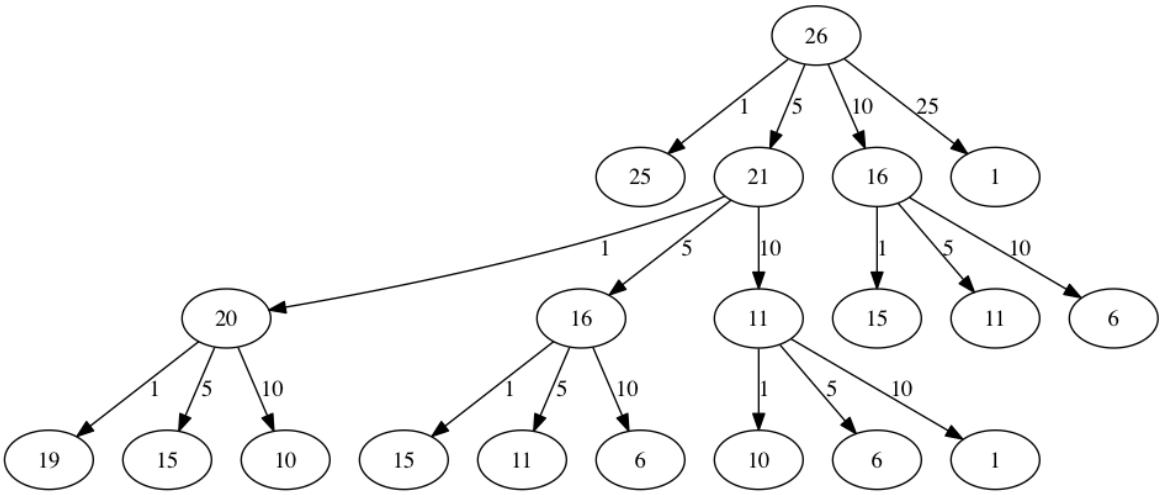
```
def recMC(coinValueList,change):
   minCoins = change
   if change in coinValueList:
     return 1
   else:
      for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recMC(coinValueList,change-i)
         if numCoins < minCoins:
            minCoins = numCoins
   return minCoins

print(recMC([1,5,10,25],63))
```

In line 3 we are checking our base case; that is, we are trying to make change in the exact amount of one of our coins. If we do not have a coin equal to the amount of change, we make recursive calls for each different coin value less than the amount of change we are trying to make. Line 6 shows how we filter the list of coins to those less than the current value of change using a list comprehension. The recursive call also reduces the total amount of change we need to make by the value of the coin selected. The recursive call is made in line 7. Notice that on that same line we add 1 to our number of coins to account for the fact that we are using a coin. Just adding 1 is the same as if we had made a recursive call asking where we satisfy the base case condition immediately.

Wait! Is That Efficient?

The trouble with the algorithm is that it is extremely inefficient. In fact, it takes 67,716,925 recursive calls to find the optimal solution to the 4 coins, 63 cents problem! To understand the fatal flaw in our approach look at the figure below, which illustrates a small fraction of the 377 function calls needed to find the optimal set of coins to make change for 26 cents.

Each node in the graph corresponds to a call to `recMC`. The label on the node indicates the amount of change for which we are computing the number of coins. The label on the arrow indicates the coin that we just used. By following the graph we can see the combination of coins that got us to any point in the graph. The main problem is that we are re-doing too many calculations. For example, the graph shows that the algorithm would recalculate the optimal number of coins to make change for 15 cents at least three times. Each of these computations to find the optimal number of coins for 15 cents itself takes 52 function calls. Clearly we are wasting a lot of time and effort recalculating old results.



The key to cutting down on the amount of work we do is to remember some of the past results so we can avoid recomputing results we already know. A simple solution is to store the results for the minimum number of coins in a table when we find them. Then before we compute a new minimum, we first check the table to see if a result is already known. If there is already a result in the table, we use the value from the table rather than recomputing. The code below shows a modified algorithm to incorporate our table lookup scheme.

```
def recDC(coinValueList,change,knownResults):
   minCoins = change
   if change in coinValueList:
      knownResults[change] = 1
      return 1
   elif knownResults[change] > 0:
      return knownResults[change]
   else:
      for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recDC(coinValueList, change-i, knownResults)
         if numCoins < minCoins:
            minCoins = numCoins
            knownResults[change] = minCoins
   return minCoins

print(recDC([1,5,10,25],63,[0]*64))
```

Notice that in line 6 we have added a test to see if our table contains the minimum number of coins for a certain amount of change. If it does not, we compute the minimum recursively and store the computed minimum in the table. Using this modified algorithm reduces the number of recursive calls we need to make for the four coin, 63 cent problem to 221 calls!

Although the algorithm is correct, it looks and feels like a bit of a hack. Also, if we look at the `knownResults` lists we can see that there are some holes in the table. In fact the term for what we have done is not dynamic programming but rather we have improved the performance of our program by using a technique known as “memoization,” or more commonly called “caching.”

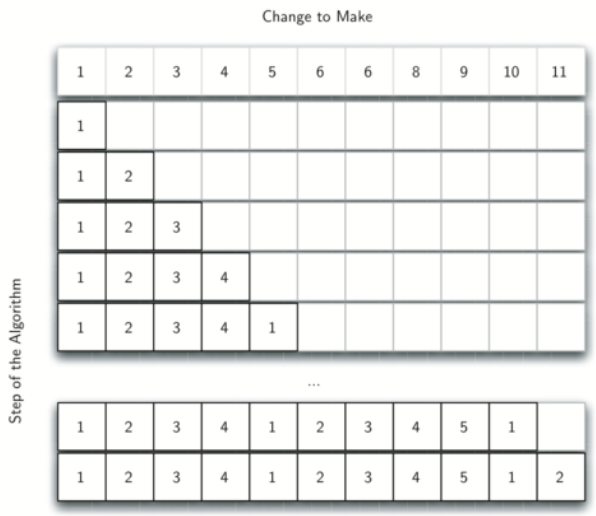
A Truly Dynamic Programming Solution

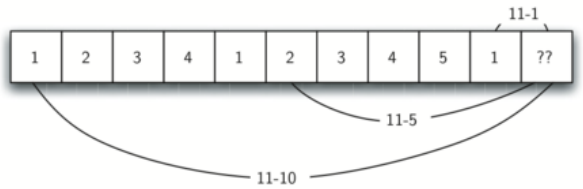
A truly dynamic programming algorithm will take a more systematic approach to the problem. Our dynamic programming solution is going to start with making change for one cent and systematically work its way up to the amount of change we require. This guarantees us that at each step of the algorithm we already know the minimum number of coins needed to make change for any smaller amount.

Let’s look at how we would fill in a table of minimum coins to use in making change for 11 cents. We start with one cent. The only solution possible is one coin (a penny). The next row shows the minimum for one cent and two cents. Again, the only solution is two pennies. The fifth row is where things get interesting. Now we have two options to consider, five pennies or one nickel. How do we decide which is best? We consult the table and see that the number of coins needed to make change for four cents is four, plus one more penny to make five, equals five coins. Or we can look at zero cents plus one more nickel to make five cents equals 1 coin. Since the minimum of one and five is one we store 1 in the table. Fast forward again to the end of the table and consider 11 cents. The graph below shows the three options that we have to consider:

1. A penny plus the minimum number of coins to make change for $11-1=10$ cents (1)
2. A nickel plus the minimum number of coins to make change for $11-5=6$ cents (2)
3. A dime plus the minimum number of coins to make change for $11-10=1$ cent (1)

Either option 1 or 3 will give us a total of two coins which is the minimum number of coins for 11 cents.





The code below is a dynamic programming algorithm to solve our change-making problem. `dpMakeChange` takes three parameters: a list of valid coin values, the amount of change we want to make, and a list of the minimum number of coins needed to make each value. When the function is done `minCoins` will contain the solution for all values from 0 to the value of `change`.

Listing 8

```
def dpMakeChange(coinValueList,change,minCoins):
   for cents in range(change+1):
      coinCount = cents
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
      minCoins[cents] = coinCount
   return minCoins[change]
```

Note that `dpMakeChange` is not a recursive function, even though we started with a recursive solution to this problem. It is important to realize that just because you can write a recursive solution to a problem does not mean it is the best or most efficient solution. The bulk of the work in this function is done by the loop that starts on line 4. In this loop we consider using all possible coins to make change for the amount specified by `cents`. Like we did for the 11 cent example above, we remember the minimum value and store it in our `minCoins` list.

Although our making change algorithm does a good job of figuring out the minimum number of coins, it does not help us make change since we do not keep track of the coins we use. We can easily extend `dpMakeChange` to keep track of the coins used by simply remembering the last coin we add for each entry in the `minCoins` table. If we know the last coin added, we can simply subtract the value of the coin to find a previous entry in the table that tells us the last coin we added to make that amount. We can keep tracing back through the table until we get to the beginning.

The last chunk of code shows the `dpMakeChange` algorithm modified to keep track of the coins used, along with a function `printCoins` that walks backward through the table to print out the value of each coin used. This shows the algorithm in action solving the problem for our friends in Lower Elbonia. The first two lines of `main` set the amount to be converted and create the list of

coins used. The next two lines create the lists we need to store the results. `coinsUsed` is a list of the coins used to make change, and `coinCount` is the minimum number of coins used to make change for the amount corresponding to the position in the list.

Notice that the coins we print out come directly from the `coinsUsed` array. For the first call we start at array position 63 and print 21. Then we take 63−21=42 and look at the 42nd element of the list. Once again we find a 21 stored there. Finally, element 21 of the array also contains 21, giving us the three 21 cent pieces.

```
def dpMakeChange(coinValueList,change,minCoins,coinsUsed):
   for cents in range(change+1):
      coinCount = cents
      newCoin = 1
      for j in [c for c in coinValueList if c <= cents]:
         if minCoins[cents-j] + 1 < coinCount:
            coinCount = minCoins[cents-j]+1
            newCoin = j
      minCoins[cents] = coinCount
      coinsUsed[cents] = newCoin
   return minCoins[change]

def printCoins(coinsUsed,change):
   coin = change
   while coin > 0:
      thisCoin = coinsUsed[coin]
      print(thisCoin)
      coin = coin - thisCoin

def main():
   amnt = 63
   clist = [1,5,10,21,25]
   coinsUsed = [0]*(amnt+1)
   coinCount = [0]*(amnt+1)
```

Source: [Problem Solving and Algorithms in Python (http://interactivepython.org/runestone/static/pythonds/index.html#)](http://interactivepython.org/runestone/static/pythonds/index.html#) from Bradley Miller on [www.interactivepython.org (http://interactivepython.org/runestone/static/pythonds/index.html#)](http://interactivepython.org/runestone/static/pythonds/index.html#).