CPSC 320 Sample Soln: Memoization and Dynamic Programming, Part 1

October 21, 2017

You work for the First CitiWide Bank, a bank that makes change. That's just what you do.

1 Greedy Change

Assuming an unlimited supply of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent, once upon a time), give a greedy algorithm to make change for $n \geq 0$ cents using the smallest total number of coins. Prove your algorithm correct.

SOLUTION: 0 cents of change is a trivial example. How many coins does it take to give 0 cents in change? None. We might also think of 1, 5, 10, and 25 as trivial examples. Each takes the coin matching its denomination. Here are some more examples:

- 2: 2 coins (2 pennies)
- 4: 4 coins (4 pennies)
- 6: 2 coins (1 nickel, 1 penny)
- 16: 3 coins (1 dime, 1 nickel, and 1 penny)
- 33: 5 coins (1 quarter, 1 dime, and 3 pennies)
- 142: 9 coins (5 quarters, 1 dime, 1 nickel, 2 pennies)

Generally, it looks like we can just take the largest possible coin out first and then solve the remaining problem. Let's name our algorithm and describe it:

```
CoinsChange(n):
  if n = 0:
    return []  // empty list of 0 coins
  else if n >= 25:
    return [quarter] + CoinsChange(n-25)
  else if n >= 10:
    return [dime] + CoinsChange(n-10)
  else if n >= 5:
    return [nickel] + CoinsChange(n-5)
  else:
    return [penny] + CoinsChange(n-1)

// You could rewrite that recursive code as a loop.
// You might also note that we can use integer division
// and remainder to immediately figure out the number
```

This work is licensed under a Creative Commons Attribution 4.0 International License.
For license purposes, the author is the University of British Columbia.

```
// of quarters we can use and reduce the problem to
// one where we check on dimes next. We won't use that
// solution, since the other versions illustrate a
// handy point for the rest of our program!
```

Of the remainder of the problems, we'll give special emphasis to brute force. Here, (the recursive case of) a brute force algorithm looks something like this: Return the best of these possibilities. (1) Give a quarter and then $n-25$ cents of change. (2) Give a dime and then $n-10$ cents of change. (3) Give a nickel and then $n-5$ cents of change. (4) Give a penny and then $n-1$ cents of change.

That will sound eerily familiar when we put together our dynamic programming solution!

2 Brother, I Can't Spare a Nickel

A few years back, the Canadian government eliminated the penny. Imagine the Canadian government accidentally eliminated the nickel rather than the penny. (That is, assume you have an unlimited supply of quarters, dimes, and pennies, but no nickels.)

1. Adapt your greedy algorithm to this problem and then **challenge your approach** by designing and testing at least 2 examples that probe its weaknesses.

   SOLUTION: It's straightforward to simply eliminate the "nickel" case from the greedy algorithm above. It's not obvious that this breaks the algorithm, and yet it does!

   The first of our small cases above that fails is the 33 case. Our algorithm now says this is 9 coins (1 quarter and 8 pennies), but the optimal solution is only 6 coins (3 dimes and 3 pennies).

   Working from there, we can see that the smallest failing case is $n = 30$, for which the optimal solution is 3 dimes rather than 1 quarter and 5 pennies.

2. We can solve this problem with something like a divide-and-conquer algorithm. (In this case, using a brute-force, recursive approach.)

   (a) To make the change, you must start by handing the customer some coin. What are your options?
   SOLUTION: At this point (without nickels), our options are to hand out a quarter, a dime, or a penny.

   (b) Imagine that in order to make 81 cents of change using the fewest coins possible, you have to start by handing the customer a quarter. Clearly describe the problem you are left with (but **don't** solve it). It may help to give **names** to quantities and concepts in the problem if you haven't already!
   SOLUTION: If we **have** to start with a quarter, then I've already broken the problem down from the $n = 81$ case to the $n = 81 - 25 = 56$ case.
   Using some names. Let $N(n)$ be the number of coins needed to make n cents in change. Then, if we **must** use a quarter first, $N(81) = N(81 - 25) + 1 = N(56) + 1$.
   We note that even if a quarter isn't the right move, it still gives us an upper-bound on the number of coins (which is a lower-bound on the quality of the solution): $N(81) \leq N(81 - 25) + 1$.

   (c) Write down descriptions of the subproblems for each of your other "first coin" options (besides a quarter).
   SOLUTION: With a dime: $N(81) \leq N(81 - 10) + 1 = N(71) + 1$.
   With a penny: $N(81) \leq N(81 - 1) + 1 = N(80) + 1$.

   (d) Given an optimal solution to each subproblem, how will you tell which coin to choose first?
   SOLUTION: Not only is each of these an upper-bound on the number of coins we might use, they also represent all the possibilities. **ANY** way we give change must start with one of a quarter, a dime, or a penny. Therefore, whichever of these three is best **is** the best solution:

This work is licensed under a Creative Commons Attribution 4.0 International License.
For license purposes, the author is the University of British Columbia.

$N(81) = \min\{N(81-25)+1, N(81-10)+1, N(81-1)+1\}$
We can easily generalize that to a recursive formula for $N(n)$ for sufficiently large n: $N(n) = \min\{N(n-25)+1, N(n-10)+1, N(n-1)+1\}$.

3. It's hard to describe a recursive algorithm without naming it. We'll name the algorithm `CCC(n)` (for `CountCoinsChange(n)`). `CCC(n)` returns the **minimum number of coins** required to make n cents of change using only pennies, dimes, and quarters. Finish `CCC`'s implementation below:

   SOLUTION: Implemented inline below:

```
CCC(n):
  If n < 0:

    Return infinity

  Else, If n = 0:

    Return __0__

  Else, n > 0:

    Return the __minimum__ of these possibilities:

      __CCC(n - 25) + 1,__

      __CCC(n - 10) + 1, and__

      __CCC(n - 1) + 1.__
```

4. `CCC` does not actually return an optimal solution (the change to give), only the **number** of coins in an optimal solution. If we imagine allowing `CCC` to have two return values (e.g., returning a more complex object than an integer), it can also return the solution. Describe how.

   SOLUTION: In general, we can have our algorithm return a tuple of the quality of the solution (in this case the number of coins is the "badness" of a solution) and the solution itself. In this particular case, it would work well to return either a list of coins at each step, where the size of the list is the optimal number of coins and the list elements themselves are the change used to achieve that or to return a map like $\{\text{quarters} : 3, \text{dimes} : 1, \text{pennies} : 2\}$, which is a shorthand for $[\text{quarter}, \text{quarter}, \text{quarter}, \text{dime}, \text{penny}, \text{penny}]$ with the advantage that it takes asymptotically less space to store!

5. Finish this recurrence for the runtime of `CCC`:

   SOLUTION: Inline below...

```
T(n) = 1                                   for n _< 0_

T(n) = _T(n-25) + T(n-10) + T(n-1) + 1_ otherwise
```

   You might wonder whether the +1 on the end of the recursive case representing the constant amount of work done "at a node" in this recurrence is important. It will turn out to make no asymptotic difference for this particular recurrence, which is dominated by its leaves, but it's difficult to tell that until you've done the analysis. So, we recommend including it. (If you're not sure what it represents, look back up at the algorithm. **Does** it spend constant time preparing to make the recursive calls and assembling the results of those calls?)

This work is licensed under a Creative Commons Attribution 4.0 International License.
For license purposes, the author is the University of British Columbia.

6. Give a depressing Ω-bound on the runtime of `CCC` by following these steps:

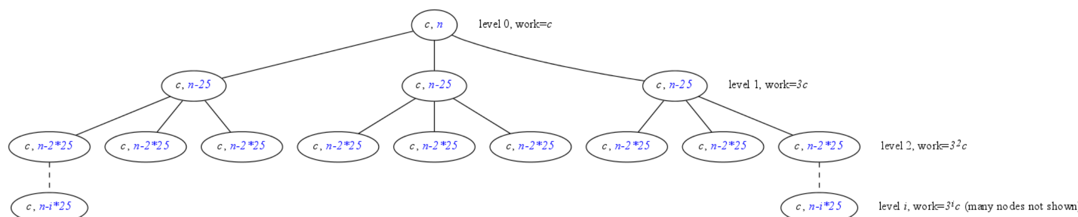
   (a) $T(n)$ is hard to deal with because it has very different-looking recursive terms. To lower-bound it, we can make them all look the same **as long as the resulting function gets smaller or stays the same**. Now, try to fill in the lower-bound on $T(n)$ below so the recursive terms all match:

   For the recursive case, $T(n) \geq$

   SOLUTION: As long as T is a non-decreasing function—which is often true for algorithms—we can say that $T(n) \geq T(n-1)$ for sufficiently large n. That means that $T(n-1) \geq T(n-10) \geq T(n-25)$, which lets us rewrite the recursive case $T(n) = T(n-25) + T(n-10) + T(n-1) + 1$ to $T(n) \geq 3T(n-25) + 1$.

   (b) Now, draw a recurrence tree for $T(n)$ and figure out its number of levels, work per level, and total work.

   SOLUTION: Here's our tree:



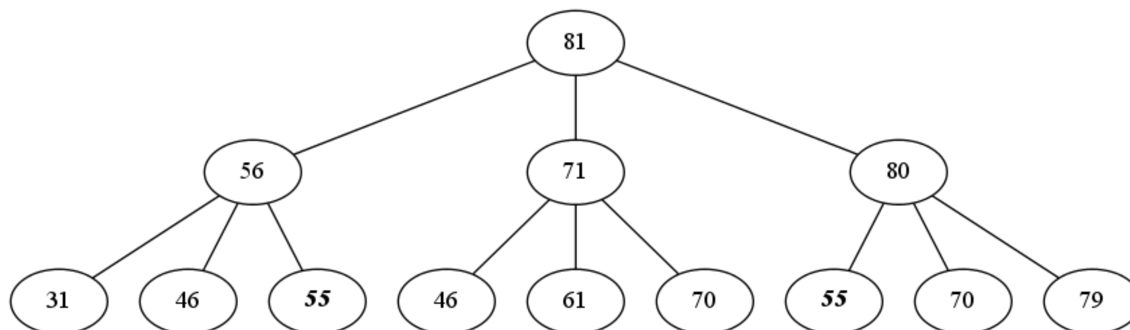
   The work in this tree forms a geometrically increasing progression. That means the work at the leaves will dominate. (The work at any level is almost three times as much as the work at **all** previous levels.)

   We reach the leaves when n reaches our base case: $n - i * 25 = 0$. Solving for i, we get $i = n/25$, which makes sense, as we're going down by quarters. It will take use $n/25$ quarters to give the change. (Note that we may overshoot our base case, which is fine; we can just count one level higher and still get a good lower-bound. Since that will be a difference of a factor of 3, and we're only concerned about asymptotics, we'll just assume here that we get to 0.)

   Thus, the work in the leaves is $3^{n/25}c = (3^{1/25})^n c \approx 1.045^n c$. While the base isn't much larger than 1, that's still exponential growth. For example, for $n = 500$, that's already $3486784401c$. For $n = 1000$, the coefficient has about 20 digits. Clearly, this scales poorly. (And, our original algorithm is exponential with a much larger base.)

7. Why is the performance so **bad**? (Hint: What subproblem do you get to if you try to give change with five dimes?)

   SOLUTION: Consider the first three levels of the recursion tree for `CCC(81)`:



   Notice the two nodes for $n = 55$ (in *italics*). The leftmost one appears as a child of the root's left child, but then the same value appears under the root's right child (and, although we didn't draw enough of the tree to see it, it appears additional times in all three subtrees of the root).

This work is licensed under a Creative Commons Attribution 4.0 International License.
For license purposes, the author is the University of British Columbia.

In fact, if we draw out the whole tree, that one node appears 48 times in the recursion tree. (How do we know? That's how many different ways you can make the 26 cents in change that get us from 81 cents to 55 cents: 2 ways with a quarter and a penny, 28 ways with two dimes and six pennies, 17 ways with one dime and sixteen pennies, and 1 way with twenty-six pennies. Each way of making the change is a path from the root to a node labeled 55.) So, however much that node costs, we pay its cost 48 times.

As we get deeper in the tree, the number of repeats of subtrees grows exponentially. We're spending essentially **all our time** recomputing the optimal solution to problems we've already solved!

(Even in these three levels, we can already see two other repeats, for $n = 46$ and $n = 70$.)

This work is licensed under a Creative Commons Attribution 4.0 International License.
For license purposes, the author is the University of British Columbia.