CPSC 320 Notes: Memoization and Dynamic Programming, Part 2

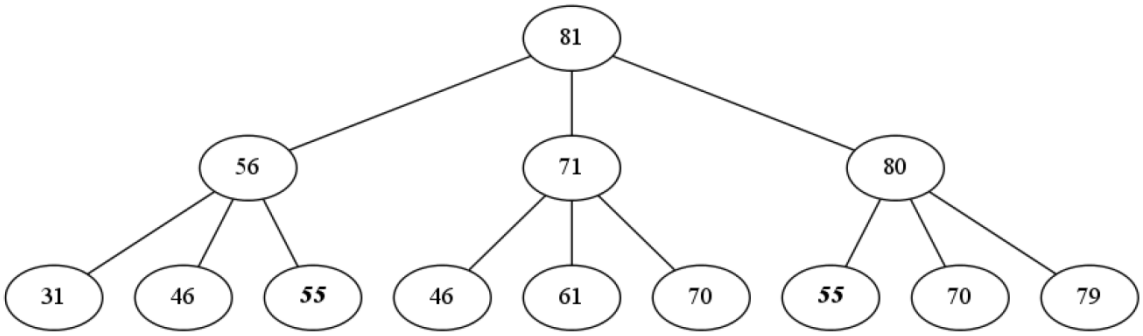
July 21, 2019

1 If I Had a Nickel for Every Time I Computed That

1. Rewrite `CCC`, this time storing—which we call "memoizing", as in "take a memo about that"—each solution as you compute it so that you **never compute any solution more than once**.

```
CCC(n):
  Create a new array Soln of length n  // using 1-based indexing

  Initialize each element Soln[i] for 1 <= i <= n to: ________________________

  Return CCCHelper(n, Soln)


CCCHelper(n, Soln):

  If n < 0:

    Return infinity

  Else, If n = 0:

    Return _______________

  Else, n > 0: // FILL IN THE REMAINING CASE
```

2. Consider this portion of the recursion tree for `CCCHelper` called on 81, where two calls to `CCCHelper` with the argument 55 are italicized:



Since we draw recursion trees with the first recursive call on the left, the left subtree finishes before the middle, which finishes before the right. Therefore, the left-hand 55 node is the first call to `CCCHelper` with the value 55. The right-hand 55 node is one (of many!) calls to `CCCHelper` with the value of 55 that happen after that first call.

Give a Θ-bound on the runtime of calls to `CCCHelper` like the right-hand one that are on a value x (where $1 \leq x \leq n$) and are **not** the first call to `CCCHelper` on that value.

3. Not counting the cost of any **other** call's **first** computation, give a good Θ-bound on the runtime of calls like the left-hand one that are the **first** computation of `CCCHelper` on a value x.

(Note: this is just like the analysis we did of QuickSort's recursion tree where we labelled the cost of a node (call) without counting the cost of subtrees (recursive calls), except we do count the inexpensive recursive calls that are not first computations.)

4. Give a Θ-bound on the total cost of all these first computations. (That is, sum up the first computations to get the total work in the tree.)

2 Growing from the Leaves

The technique from the previous part is called "memoization". Turning it into "dynamic programming" requires changing the order in which we consider the subproblems.

1. Finish this formula for `Soln(i)` in terms of smaller entries in `Soln`. (This is **also** a recurrence, just like the ones we use to measure performance!) Make it **as similar as you can** to your recursive code above.

```
Soln(i) = infinity                                  for i < 0

Soln(0) = 0

Soln(i) = ___________________________________________

          ___________________________________________

          ___________________________________________ otherwise
```

2. If we were to store this in the `Soln` array, which entries of the array need to be filled in before we're ready to compute the value for `Soln[i]`?

3. Give a simple order in which we could compute the entries of `Soln` so that all previous entries needed are **already** computed by the time we want to compute a new entry's value.

4. Take advantage of this ordering to rewrite `CCC` without using recursion:

```
// Note: It's handy to pretend Soln has 0 and negative entries.
//       We use SolnCheck to do that.
SolnCheck(Soln, i):

  If i < 0:      Return _______________

  Else If i = 0: Return _______________

  Else:          Return Soln[i]



CCC(n):
  Create a new array Soln of length n // using 1-based indexing

  For i = _____________________________________:

    Soln[i] = the ____________ of:

      ______________________________________,

      ______________________________________, and

      ______________________________________.

  Return Soln[n]  // assumes n > 0
```

5. Both the dynamic programming and memoized versions of `CCC` run in the same asymptotic time. Asymptotically in terms of n, how much **memory** do these versions of `CCC` use?

6. Imagine that you only wanted the **number** of coins returned from `CCC`. In the dynamic programming version how much of the `Soln` array do you **really** need at one time? If you take advantage of this, how much memory does it use, asymptotically?

3 Foreign Change

Design a new version of `CCC` so that it handles foreign currencies where you receive the target amount n and an array of coin values $[c_1, c_2, \ldots, c_k]$. Assume that the penny is always available. (So, for pennies, dimes, and quarters, the array would look like $[10, 25]$.)

Analyse the runtime of your algorithm in terms of n and k.

TAKE IT STEP BY STEP! That means to write trivial and small examples, describe the input and output, design an inefficient recursive version, memoize it, and transform that into a dynamic programming solution.

4 Challenge

1. How would you alter your algorithm for the "foreign change" problem if pennies were **not** guaranteed to be available? What unusual cases could arise in solutions?

2. Modify the dynamic programming solution to return both the number of coins used **and** the solution while using only constant memory. *Hint:* it helps when storing partial solutions that you don't care what order you give the coins out in.

3. Count the **number of different ways** to make `n` cents in change using quarters, dimes, nickels, and pennies (again, using memoization and/or dynamic programming).

   (a) First, assume that order matters (i.e., giving a penny and then a nickel is different from giving a nickel and then a penny).

   (b) Then, assume that order does not matter.

4. Solve the "minimum number of coins" change problem if you do **not** have an infinite supply and instead are given the available number of each coin as a parameter `[num_quarters, num_dimes, num_nickels]`. (Assume an infinite number of pennies.)

5. Prove that you can take at least one greedy step if the foreign change algorithm takes only two distinct coin values `[c1, c2]`, and `n` is at least as large as the least common multiple of `c1` and `c2`.

6. Extend this "least common multiple" observation to more coins.