CPSC 320 Sample Soln: Memoization and Dynamic Programming, Part 2

March 1, 2017

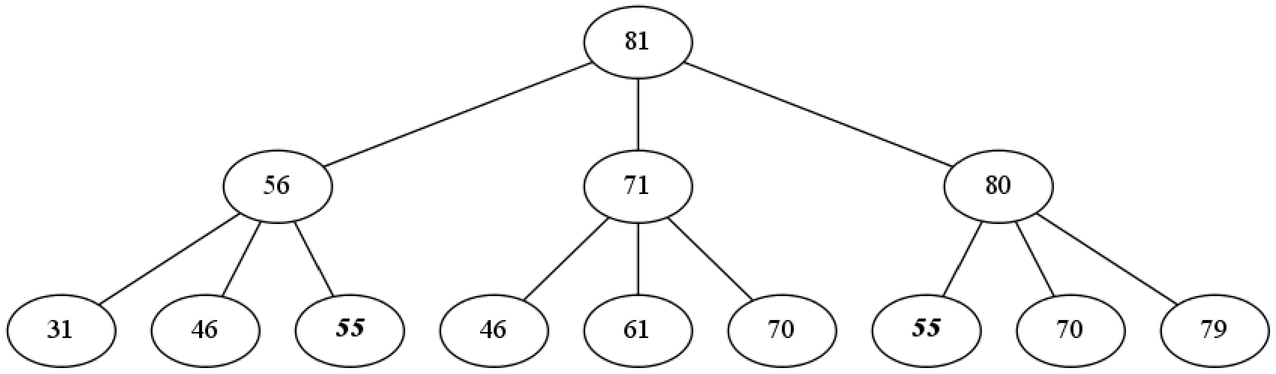
1 If I Had a Nickel for Every Time I Computed That

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** (for a given call to `CCC`).

   SOLUTION: Inline below:

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

  Initialize each element Soln[i] for 1 <= i <= n to: _-1_  // or any other "flag" value

  Return CCCHelper(n, Soln)


CCCHelper(n, Soln):

  If n < 0:

    Return infinity

  Else, If n = 0:

    Return _0_

  Else, n > 0:

    If (Soln[n] == -1):   // i.e., if we have not stored the answer for n
      // Compute and store the answer
      Soln[n] = min(CCCHelper(n-25, Soln) + 1,
                    CCCHelper(n-10, Soln) + 1,
                    CCCHelper(n-1, Soln) + 1)

    // By this point, we're guaranteed to have the answer stored.
    Return Soln[n]
```

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.

SOLUTION: Since $1 \leq x \leq n$, we'll hit the "else" case at the bottom of `CCCHelper`. Since this is not the first call for this value of x, we've already stored the result in `Soln`. Therefore, we do constant work checking that $x \not< 0$ and $x \neq 0$ to reach the "else" case and constant work checking that we've stored the solution, and constant work returning it.

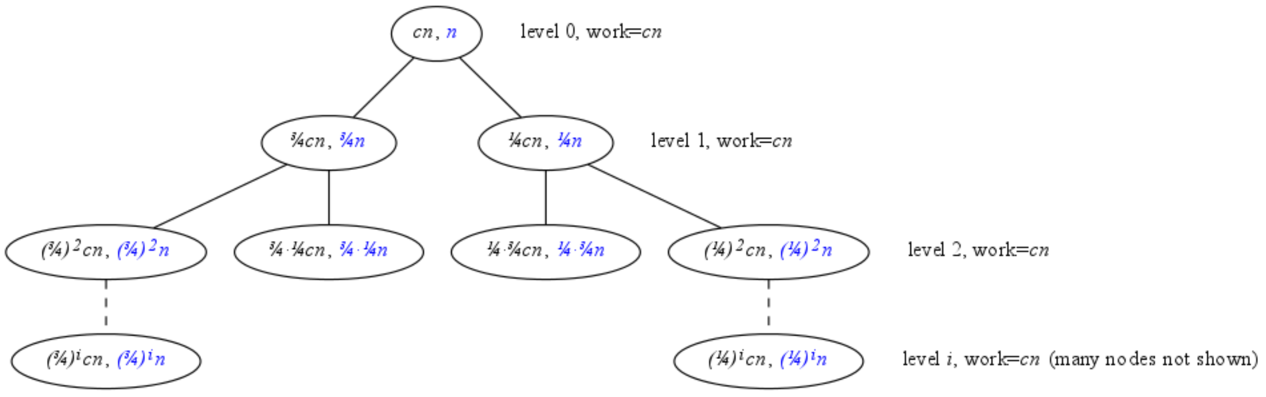
The runtime is in $\Theta(1)$

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.

   SOLUTION: In the first computation, we do the same constant work as before except we also take the minimum of the three recursive calls. Any of those recursive calls that are not **first** calls take constant time (per our answer to the previous part). Any of those recursive calls that **are** first calls aren't counted in this analysis.

   So, a first call **not counting its recursive calls' first computation** takes $\Theta(1)$ time.

   It may seem strange not to count other calls' first computations, but this is very similar to when we do the analysis of QuickSort and label a node with cn work (with work shown in black in this tree):


cn, n
level 0, work=cn
$\frac{3}{4}cn$, $\frac{3}{4}n$
$\frac{1}{4}cn$, $\frac{1}{4}n$
level 1, work=cn
$(\frac{3}{4})^2cn$, $(\frac{3}{4})^2n$
$\frac{3}{4}\cdot\frac{1}{4}cn$, $\frac{3}{4}\cdot\frac{1}{4}n$
$\frac{1}{4}\cdot\frac{3}{4}cn$, $\frac{1}{4}\cdot\frac{3}{4}n$
$(\frac{1}{4})^2cn$, $(\frac{1}{4})^2n$
level 2, work=cn
$(\frac{3}{4})^icn$, $(\frac{3}{4})^in$
$(\frac{1}{4})^icn$, $(\frac{1}{4})^in$
level i, work=cn (many nodes not shown)

   That cn work is the work done at that node, **not counting** the work done by its recursive calls. Why are we allowed to do that? Because we later sum up all the work at all the levels, which "counts back in" the work we ignored earlier.

   So, we just need to make sure we justify that we counted all the work before we're done.

4. Give a Θ-bound on the total cost of all these **first** computations. (That is, sum up the first computations.)

   SOLUTION: x can range from 1 to n, which is n values. There is at most one first call to each of these values of x. So, the total work is $\Theta(n)$.

5. Explain why this Θ-bound also bounds the total runtime of the algorithm. (That is, why do we not also need to include the cost of computations after the first one?)

   SOLUTION: We need to show that we've accounted for every node in the recursion tree.

   We've accounted for each "first call" to `CCCHelper` on any value $1 \le x \le n$ by adding them all up. (Some first calls to `CCCHelper` make recursive subcalls to `CCCHelper` that are also first calls. We didn't count those recursive subcalls when finding the runtime of the call itself, but by adding up all possible first calls, we **did** count these.)

   What about the second and later calls? No second and later call makes any recursive calls of its own; instead, it just returns the stored value. Therefore, every second and later recursive call is called by a first call. We counted that work in our analysis of the first call.

   Thus, we've counted every first call and every second and later call, which is **all** the calls.

   (Note: we never explicitly discussed base cases, but even the first call on the base case takes constant time.)

2 Growing from the Leaves

The technique from the previous part is called "memoization". Turning it into "dynamic programming" just 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.

   SOLUTION: Inline below.

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

Soln(0) = 0

Soln(i) = min(Soln(i-25)+1, Soln(i-10)+1, Soln(i-1)+1) 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]`?

   SOLUTION: Entries $i - 25$, $i - 10$, and $i - 1$.

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.

   SOLUTION: In this case, we need entry $i - 1$ before we can compute entry i; so, our only option is to compute the entries in order from smallest to largest: $1, 2, \ldots, n - 1, n$.

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

   SOLUTION: Inline below:

```
// 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 _infinity_

  Else If i = 0: Return _0_

  Else:          Return Soln[i]



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

  For i = _1 to n_:

    Soln[i] = the _minimum_ of:

      _Soln[i-25] + 1__,

      _Soln[i-10] + 1_, and

      _Soln[i-1] + 1_

  Return Soln[n]
```

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?

   SOLUTION: They both store an entry in Soln for each value from 1 to n. Assuming each entry takes one "unit" of memory, that's $O(n)$ memory.

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?

   SOLUTION: In the dynamic programming version, we refer back to only the last 25 entries at any given entry. So, we could keep a record of only those most recent 25 entries and update it (discarding the oldest entry) each time we compute a new entry. (A circular array—as in an array-backed queue implementation—might be a handy way to implement this.)

   In that case, we'd be using a constant number of "units" of memory: $O(1)$.

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.

SOLUTION: Our new CCC takes the array of k coins and a target value n. If $k = 0$, then the problem is trivial (we need n pennies). If $n = 0$, then we need no coins.

Here are a couple small examples:

- $[7, 8], 30$: 2 "eights" and 2 "sevens" make 30 cents in change. (This one would not work correctly with our greedy algorithm.)
- $[10, 100, 200], 333$: 1 "two hundred", 1 "one hundred", 1 "ten", and 3 pennies makes 333 cents in change. (This one **would** work correctly with our greedy algorithm, since we can see that any solution besides greedy can trade in multiple coins for a single coin used in greedy.)

We've already largely described the input above. We'll just produce the **number** of coins as output for now.

Here's an inefficient algorithm:

```
FC(c = [c1, c2, ..., ck], n):
  If n < 0:
    Return infinity
  Else If n = 0:
    Return 0
  Else:
    Let best = n // n pennies
    For i = 1 to k:
      Let with_ci = FC(c, n-c[i]) + 1
      If with_ci < best:
        best = with_ci
    Return best
```

This solution has exponential runtime once $k \geq 2$ (e.g., with $c = [7, 8]$).

Memoizing this just requires adding a table. Since the `c` parameter never changes, we don't need to worry about it in our table, and the table is still one-dimensional with n entries:

```
FC(c = [c1, c2, ..., ck], n):
  Create a new array Soln of length n  // using 1-based indexing
  Initialize each element Soln[i] for 1 <= i <= n to: -1
  Return FCHelper(Soln, c, n)



FCHelper(Soln, c = [c1, c2, ..., ck], n):
  If n < 0:
    Return infinity
  Else If n = 0:
    Return 0
  Else:
    If Soln[n] < 0:
      Let best = n // n pennies
      For i = 1 to k:
        Let with_ci = FCHelper(Soln, c, n-c[i]) + 1
        If with_ci < best:
          best = with_ci
      Soln[n] = best
    Return Soln[n]
```

This memoized solution takes $O(k)$ time to fill out each table entry the first time we call it on a particular value (not counting first subcalls). There are n such calls. So, it takes $O(kn)$ time total. It uses $O(n)$ "units" of table spaces.

We can convert this to dynamic programming in the same way we did for `CCC`. As an illustration, we won't use the `SolnCheck` helper here, but that would be a great approach as well.

```
FC(c = [c1, c2, ..., ck], n):
  Create a new array Soln of length n  // using 1-based indexing
  For i = 1 to n:
    Let best = i // n pennies
    For j = 1 to k:

      // Since we didn't use SolnCheck, we have to be
      // a bit careful about negative values here.
      Let n' = i - c[j]
      Let with_cj = infinity
      If n' >= 0:
        with_cj = Soln[n'] + 1

      If with_cj < best:
        best = with_cj
    Soln[i] = best
  Return Soln[n]
```

The dynamic programming solution has the same asymptotic characteristics as the memoized solution but will probably have lower constant factors on its runtime in practice. The DP version also facilitates truncating our `Soln` table (which need only be as long as the maximum element of c).