CPSC 320: Memoization and Dynamic Programming I Solutions *

You want to make change in the world, but to get started, you're just ... making change. You have an unlimited supply of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent, once upon a time). You want to make change for $n \geq 0$ cents using the minimum number of coins.

1 Build intuition through examples.

1. Here is an optimal greedy algorithm to make change. Try it on at least one instance.

   function GREEDY-CHANGE(n)
   while $n > 0$ **do**
   if $n \geq 25$ **then** give a quarter and reduce n by 25
   else if $n \geq 10$ **then** give a dime and reduce n by 10
   else if $n \geq 5$ **then** give a nickel and reduce n by 5
   else give a penny and reduce n by 1

   SOLUTION: 0 cents of change is a trivial instance; no coins are given. Also, 1, 5, 10, and 25 are trivial instances; the algorithm gives the coin matching its denomination. Here are some more examples of what the algorithm gives:

   - 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 nickel, and 3 pennies
   - 142: 9 coins: 5 quarters, 1 dime, 1 nickel, 2 pennies

2. 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. Adapt algorithm GREEDY-CHANGE for the case where the nickel is eliminated, by changing the code above. Then see if you can find a counterexample to its correctness.

   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 Writing down a formal problem specification.

We will assume that a currency which includes the penny is fixed, with coins of value $1, v_1, \ldots v_k$ for some $k \geq 1$. We'll work with the currency $1, 10, 25$ in what follows, but want an algorithm that can easily be adapted to work more generally.

1. What is an instance of the making change problem?

   SOLUTION: Since the currency is fixed, an instance of the problem is a nonnegative integer n. Since the number of coins needed to make change is proportional to n, we'll use n as a measure of the problem size (even though the number of bits needed to represent n is $O(\log n)$).

2. This is an example of a *minimization problem*; what quantity are we trying to minimize?

   SOLUTION: We want to minimize the number of coins to make change for n. We'll denote this minimum by $C(n)$.

3 Evaluating the brute force solution.

As is often the case, this approach will lead us to even better approaches later on. It will be helpful to write our brute force algorithm recursively. We'll build up to that in several steps.

1. To make change, you must start by handing the customer some coin. What are your options?

   SOLUTION: Our options are to hand out a quarter, a dime, or a penny.

2. Imagine that in order to make $n = 81$ cents of change using the minimum number of coins, you can start by handing the customer a quarter. Clearly describe the subproblem you are left with (but **don't** solve it). You can use the notation above in the formal problem specification.

   SOLUTION: If we start with a quarter, then we are left with the subproblem of determining the minimum number of coins needed to make change for 56 cents. Then the total number of coins that we use is $1 + C(56)$.

3. Even if we're not sure that a quarter is an optimal move, we can still get an upper bound on the number of coins by considering the subproblem we are left with when we start with a quarter. What upper bound do we get on $C(81)$?

   SOLUTION: We get that $C(81) \leq 1 + C(56)$.

4. What other upper bounds on $C(81)$ do we get if we consider each of the other "first coin" options (besides a quarter), and the corresponding subproblem?

   SOLUTION: If we start with a dime, we are left with the subproblem of determining $C(81 - 10 = C(71)$. So, $C(81) \leq C(71) + 1$.

   With a penny: $C(81) \leq C(81 - 1) + 1 = C(80) + 1$.

5. There are three choices of coin to give first. Can you express $C(81)$ as the minimum of three options?

   SOLUTION:

   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 optimal solution:

   $$C(81) = \min\{C(81 - 25) + 1, C(81 - 10) + 1, C(81 - 1) + 1\}.$$

We can easily generalize that to a recursive formula for $C(n)$ for sufficiently large n:

$$C(n) = \min\{C(n-25)+1, C(n-10)+1, C(n-1)+1\}.$$

Notice that here we are using a recurrence to express the solution to a minimization problem. Recurrences are very useful for this purpose, as well as expressing runtimes of (recursive) algorithms.

6. Now, consider the more general problem of making change when there are $k+1$ different coins available, with one being a penny, and the remaining k coins having values $v_1, v_2, \ldots, v_k$, all of which are greater than 1. Let $C'(n)$ be the minimum number of coins needed in this case. For sufficiently large n, how can you express $C'(n)$ in terms of $C'()$ evaluated on amounts smaller than n?

   SOLUTION: Generalizing our previous recurrence, we get:

   $$C'(n) = \min\{C'(n-v_1)+1, C'(n-v_2)+1, \ldots, C'(n-v_k)+1, C'(n-1)+1\}.$$

7. Complete the following recursive brute force algorithm for making change:

   SOLUTION: Implemented inline below.

```
function BRUTE-FORCE-CHANGE(n)
    if n < 0 then
        return infinity
    else if n = 0 then
        return 0
    else if n > 0 then
        return the minimum of:
            BRUTE-FORCE-CHANGE(n − 25) + 1,
            BRUTE-FORCE-CHANGE(n − 10) + 1,
            BRUTE-FORCE-CHANGE(n − 1) + 1
```

8. Complete the following recurrence for the runtime of algorithm Brute-Force-Change:

   SOLUTION: For some constant $c > 0$,

   $$T(n) = \begin{cases} c & \text{for } n \le 0 \\ T(n-25) + T(n-10) + T(n-1) + c & \text{otherwise.} \end{cases}$$

   The constant c accounts for time needed to determine which of the three cases applies and, if in case 3, to to initiate the recursive calls and assemble the results of those calls.

9. Give a disappointing Ω-bound on the runtime of BRUTE-FORCE-CHANGE by following these steps:

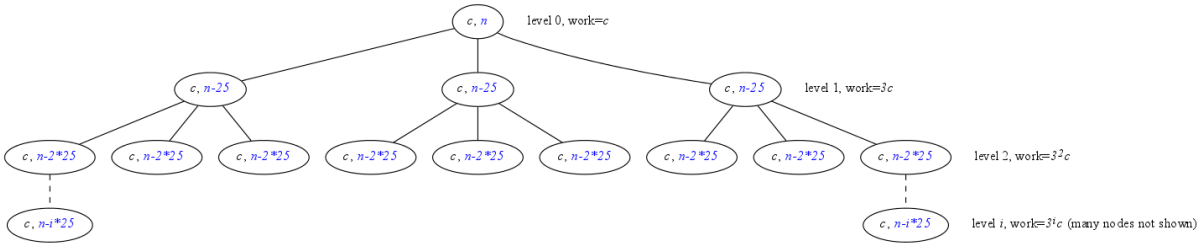
   (a) $T(n)$ is hard to deal with because the three recursive terms in part (8) above are different. To lower bound $T(n)$, we make them all equal to the smallest term. Complete the lower bound that we get for the recursive case when we do this:

   SOLUTION:

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

(b) Now, draw a recursion tree for the recurrence of part 9a and figure out its number of levels, work per level, and total work.

SOLUTION: Here's our tree:



Here, 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 one of our base cases: $n - i * 25 \approx 0$. Solving for i, we get $i \approx n/25$, which makes sense, as we're going down by quarters as long as possible.

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.)

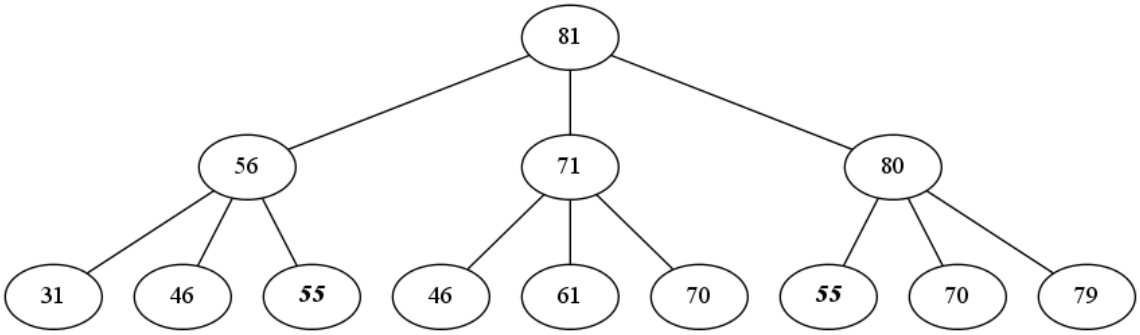
[Note: For recurrences $T(n)$ where there is just one term involving T on the right hand side, we could "unroll" the recurrence as follows:

$$\begin{aligned} T(n) &\geq 3T(n-25) + c \\ &\geq 3(3T(n-2*25)+c) + c = 3^2 T(n-2*25) + 3c + c \\ &\geq \ldots \\ &\geq 3^i T(n-i*25) + 3^{i-1}c + \ldots 3c + c \\ &= 3^i T(n-i*25) + c\sum_{j=0}^{i-1} 3^j. \end{aligned}$$

As in the recursion tree method, we see that we reach the base case when $i = n/25$, and so $T(n) \geq c\sum_{j=0}^{n/25} 3^j \geq 3^{n/25}$, as before.]

10. Why is the performance so bad? Does this algorithm waste time trying to solve the same subproblem more than once? For $n = 81$, draw the first three levels (the root at level 0 plus two more levels) of the recursion tree for BRUTE-FORCE-CHANGE to assess this. Label each node by the size of its subproblem. Does any subproblem appear more than once?

SOLUTION: Consider the first three levels of the recursion tree for BRUTE-FORCE-CHANGE(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).

In fact, if we draw out the whole tree, node 55 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 $n = 70$.)

4 Memoization: If I Had a Nickel for Every Time I Computed That

Here we will use a technique called **memoization** to improve the runtime of the recursive brute force algorithm for making change. Memoization avoids making a recursive call on any subproblem more than once, by using an array to store solutions to subproblems when they are first computed. Subsequent recursive calls are then avoided by instead looking up the solution in the array.
Memoization is useful when the total number of different subproblems is polynomial in the input size.

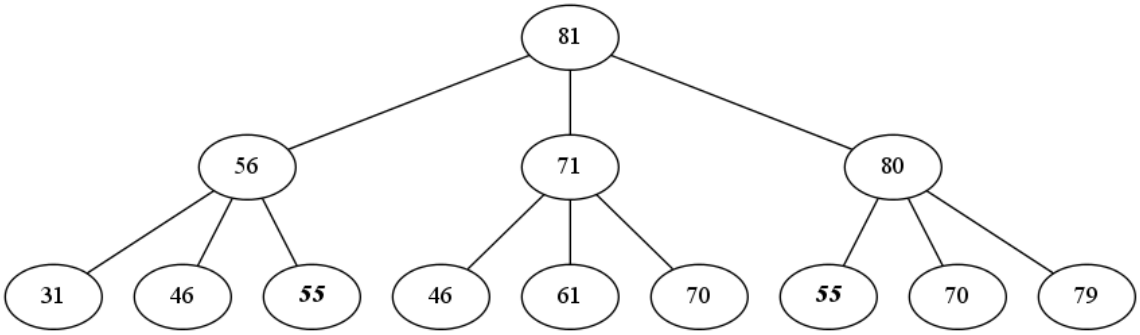
1. Rewrite Brute-Force-Change, this time storing—which we call "memoizing", as in "take a memo about that"—each solution as you compute it so that you **never compute any solution more than once**.

   SOLUTION:

   ```
   function Memo-Change(n)
       create a new array Soln[1..n]
       for i from 1 to n do Soln[i] ← −1          ▷ or any other "flag" value
       return Memo-Change-Helper(n)

   function Memo-Change-Helper(i)
       if i < 0 then
           return infinity
       else if i = 0 then
           return 0
       else if i > 0 then
           if Soln[i] == −1 then
               Soln[i] ← the minimum of:
                     Memo-Change-Helper(i − 25) + 1,
                     Memo-Change-Helper(i − 10) + 1,
                     Memo-Change-Helper(i − 1) + 1.
           return Soln[i]
   ```

2. We want to analyze the runtime of Memo-Change. In what follows, we'll refer back to this illustration of two levels of recursive calls for Memo-Change-Helper.



How much time is needed by a call to MEMO-CHANGE-HELPER, not counting the time for recursive calls? That is, how much time is needed at each node of a recursion tree such as the one above?

(Note: this is similar to 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). Here, however, we won't sum the work per level.)

SOLUTION: $\Theta(1)$ time is needed, to check which case of the `If` statement applies, the time to return the result, and, if in the third case when Soln$[n]$ is -1, the time to take the min of three values.

3. Which nodes at level two of the above recursion tree are leaves, that is, have no children (corresponding to further recursive calls) at level three? Assume that we draw recursion trees with the first recursive call on the left.

   SOLUTION: The leaves are the node labeled 46 that is in the middle subtree of the root, as well as nodes labeled 55 and 70 in the right subtree of the root. For example, since a node labeled 55 appears further to the left, Soln[55] is no longer equal to -1, and no recursive call is made.

4. Give an upper bound on the number of **internal** nodes of the recursion tree on input n.

   SOLUTION: There are at most n internal nodes, since each node is labeled by a distinct number between 1 and n.

5. Give a big-O upper bound on the number of **leaves** of the recursion tree on input n.

   SOLUTION: Each internal node can have at most three children that are leaves. So, there are $O(n)$ leaves.

6. Using the work done so far, give a big-O bound on the run-time of algorithm MEMO-CHANGE(n).

   SOLUTION: The recursion tree has $O(n)$ nodes, and the time per node is $O(1)$. So the total time is $O(n)$.

5 Dynamic programming: Growing from the leaves

The recursive technique from the previous part is called *memoization*. Turning it into *dynamic programming* requires avoiding recursion by changing the order in which we consider the subproblems. Here again is the recurrence for the smallest number of coins needed to make n cents in change, renamed to Soln:

$$\text{Soln}[i] = \begin{cases} \infty & \text{if } i < 0 \\ 0 & \text{if } i = 0 \\ 1 + \min\{\text{Soln}[i-25], \text{Soln}[i-10], \text{Soln}[i-1]\} & \text{otherwise.} \end{cases}$$

1. Which entries of the Soln array need to be filled in before we're ready to compute the value for Soln[i]?

   SOLUTION: We need Soln[$i-25$], Soln[$i-10$], and Soln[$i-1$] (assuming that $i > 25$).

2. 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: We can calculate the entries in increasing order.

3. Take advantage of this ordering to rewrite Brute-Force-Change without using recursion:

   SOLUTION:

```
function Soln'(i)
    ▷ Note: It would be handy if Soln had 0 and negative entries.
    ▷ We use this function Soln' to simulate this.
    if i < 0 then return infinity
    else if i = 0 then return 0
    else return Soln[i]

function DP-Change(n)
    if n ≤ 0 then return Soln'(n)
    else
        ▷ Assumes n > 0; otherwise, just run Soln'
        create a new array Soln[1..n]
        for i from 1 to n do
            Soln[i] ← the minimum of
                    Soln'(i − 25) + 1,
                    Soln'(i − 10) + 1, and
                    Soln'(i − 1) + 1
    return Soln[n]
```

4. Assume that you have already run algorithm Memo-Change(n) or DP-Change(n) to compute the array Soln[$1..n$], and also have access to the Soln' function above. Write an algorithm that uses the values in the Soln array to return the number of coins of each type that are needed to make change with the minimum number of coins.

   SOLUTION:

```
function Calculate-Change(n)
    NumQuarters ← 0; NumDimes ← 0; NumPennies ← 0
    while n > 0 do
        if Soln'(n − 25) ≤ Soln'(n − 10) and Soln'(n − 25) ≤ Soln'(n − 1) then
            NumQuarters ← NumQuarters + 1
            n ← n − 25
        else if Soln'(n − 10) ≤ Soln'(n − 1) then
            NumDimes ← NumDimes + 1
            n ← n − 10
        else
            NumPennies ← NumPennies + 1
            n ← n − 1
        return "Use" NumQuarter "quarters," NumDimes "dimes and" NumPennies "pennies."
```

5. Both MEMO-CHANGE and DP-CHANGE run in the same asymptotic time. Asymptotically in terms of n, how much **memory** do these algorithms 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 want the number of coins returned from BRUTE-FORCE-CHANGE, and don't need to actually calculate change. For the DP-CHANGE algorithm, how much of the Soln array do you **really** need at one time? If you take advantage of this, how much memory does the algorithm use, asymptotically?

   SOLUTION:

   In DP-CHANGE, we refer back to only the last 25 entries when computing Soln[i]. So, we could keep a record of only those most recent 25 entries and update this record (discarding the oldest entry) each time we compute a new entry. A queue data structure provides a handy way to implement this. With this implementation, we use a constant number of "units" of memory: $O(1)$.

SOLUTION: Our new BRUTE-FORCE-CHANGE 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 of 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.)

Here's an inefficient algorithm to produce the **number** of coins as output.

```
Brute-?????-CCC(c[1..k], 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 = Foreign-CCC(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 $c = [7, 8]$).

We can memoize this using a Soln array as before:

```
Memo-Foreign-CCC(c[1..k], 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(n)

FCHelper(n)
  If n < 0
```

```
    Return infinity
  Else If n = 0
    Return 0
  Else
    If Soln[n] = -1
      Let best = n // n pennies
      For i = 1 to k
        Let with_ci = FCHelper(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 compute an entry of `Soln`

The first time that an entry of `Soln` needs to be computed, i.e., when the entry is -1 when FCHelper is called, it takes time $\Theta(k)$ to compute the entry (because of the For loop in the FCHelper code). There are n entries in the `Soln` array. So, it takes $O(kn)$ time total. The algorithm uses $\Theta(n)$ memory to store the `Soln` array.

We next convert this memoized algorithm to a dynamic programming algorithm in a similar way that we did for `CCC`. We won't use the `Soln`' helper here, but that would be a great approach as well.

```
DP-Foreign-CCC(c[1..k], n)
  Create a new array Soln of length n  // using 1-based indexing
  For i = 1 to n
    Let best = i // i pennies
    For j = 1 to k

      // Since we didn't use Soln', 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 running time as the memoized solution but will probably have lower constant factors on its runtime in practice. The DP version also facilitates truncating our `Soln` array (which need only be as long as maximum value stored in array c).