CPSC 320: Memoization and Dynamic Programming I *

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

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.

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?

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

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?

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.

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 $C(81)$?

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?

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

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?

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

function BRUTE-FORCE-CHANGE(n)
  if $n < 0$ **then**
    return _____
  if $n = 0$ **then**
    return _____
  if $n > 0$ **then**
    return the minimum of:

    _____,

    _____, and

    _____

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

$$T(n) = \begin{cases} \text{\_\_\_\_\_} & \text{for } n \leq 0 \\ \text{\_\_\_\_\_} & \text{otherwise.} \end{cases}$$

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:

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

   (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.

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?

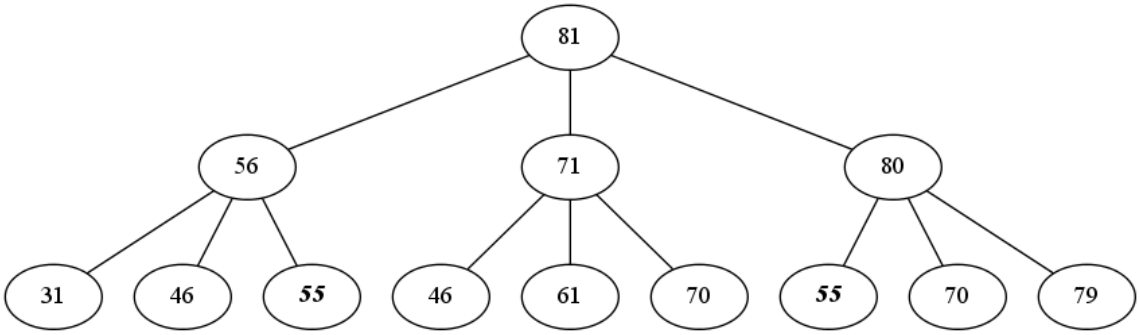
4 Memoization: If I Had a Nickel for Every Time I Computed That

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**.

```
function Memo-Change(n)
    create a new array Soln of length n // using 1-based indexing

    for i from 1 to n do: Soln[i] ← _____
    return Memo-Change-Helper(n)

function Memo-Change-Helper(i)
    if i < 0 then
        return infinity

    else if i = 0 then

        return _____

    else if i > 0 then
        if Soln[i] == −1 then
```

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.)

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.

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

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

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

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.

3. Take advantage of this ordering to rewrite BRUTE-FORCE-CHANGE without using recursion:

```
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 _____
    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 _____ do

            Soln[i] ← the _____

                    _____

                    _____

                    _____
    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.

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?

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?

6 Challenge: Foreign Change

Design a new version of DP-CHANGE that handles foreign currencies more generally. An instance of the problem is a target amount n and an array of coin values $c[1, ..k]$. Assume that the penny is always available and is not included in the array. 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 finally transform that into a dynamic programming solution.