Ruolin Li
31764160

CPSC 320: Memoization and Dynamic Programming II *

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

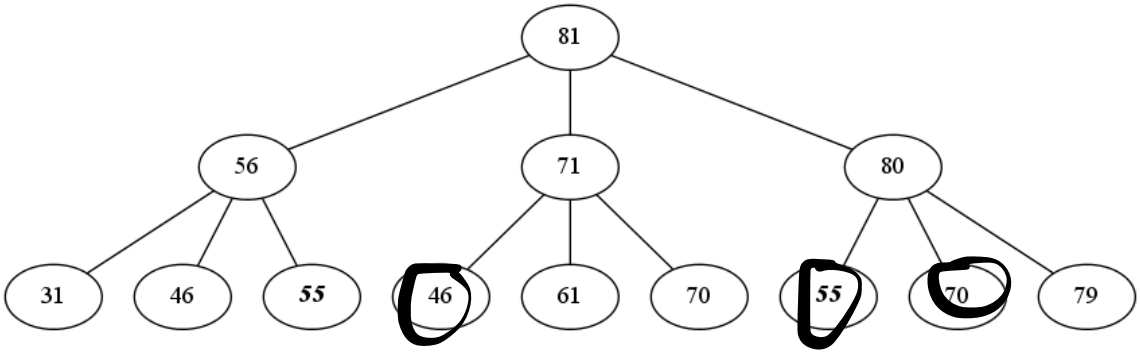
Here you'll use a technique called **memoization** to improve the runtime of the recursive brute force algorithm for making change. Memoization avoids making a recursive call on any subproblem more than once, by using an array to store solutions to subproblems when they are first computed. Subsequent 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] ← -1      //We use -1 to indicate if this slot has been edited 'flag' number
    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] = 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.)

Time per node is constant, we use c to represent it

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.

See above, circled in black

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

The number of internal nodes is ≤ n, because for each number i, 1 ≤ i ≤ n

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

There are ≤ n internal nodes, each has at most 3 children that are leaves
So the total number of leaves is ≤ 3n

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

work for internal nodes is O(n)
work for leaves is O(n)
So total is O(n)

2 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:

$$\begin{array}{ll} \text{Soln[i]} = \text{infinity}, & \text{for } i < 0 \\ \text{Soln[0]} = 0, & \\ \text{Soln}[i] = 1 + \min\{\text{Soln}[i-25], \text{Soln}[i-10], \text{Soln}[i-1]\} & \text{otherwise.} \end{array}$$

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

   We need Soln[i-25], Soln[i-10], Soln[i-1]

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.

   We could use an array in ascending order to store the entries.

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** **0**
   else return Soln[i]

   function DP-CHANGE(n)
   if $n \leq 0$ **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**
   Soln'[i-25]+1
   Soln'[i-10]+1
   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.

```
function Calculate_Change(n)
    set #of Q(quarter) = 0
     set #of D(dime) = 0
    set #of P(penny) = 0
    while n > 0
      If Soln'[n-25] ≤ Soln'[n-1] AND Soln'[n-25] ≤ Soln'[n-10]
          #of Q ++
          n = n - 25
      else if Soln'[n - 10] ≤ Soln'[n - 1] AND Soln'[n - 10] ≤ Soln'[n - 25]
          #of D ++
          n = n - 10
      else
          #of P ++
          n  = n - 1

    return #of Q, #of D, #of P
```

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?

O(n) since we need to store the number from 1 to n

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?

O(1)

To compute Soln[n] in the change function, we just need array entries back to Soln[n - 25], and if we don't store array entries that are no longer needed, we can get O(1)

3 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.

4 More 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 $[c_1, c_2]$, and n is at least as large as the least common multiple of c_1 and c_2.

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