# CPSC 320 Notes: Memoization and Dynamic Programming

March 2, 2015

You work for the First CitiWide Bank, a bank that makes change. That's just what you do.

## 1 Greedy Change (on your own, outside class)

Assuming an unlimited supply of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent, once upon a time), give a greedy algorithm to make change for $n \geq 0$ cents using the smallest total number of coins. Prove your algorithm correct.

## 2 Brother, I Can't Spare a Nickel

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.)

1. Prove that the natural adaptation of the greedy algorithm is not optimal (i.e., does not use the minimum number of coins).

2. We can solve this problem with something like a divide-and-conquer algorithm.

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

   (b) Imagine that to make 81 cents of change using the fewest coins possible, you start by handing the customer a quarter. Clearly describe the problem you are left with.

   (c) Write down descriptions of the subproblems for each of your other "first coin" options.

   (d) Given an optimal solution to each subproblem, how will you tell which coin to choose first?

3. It's hard to describe a recursive algorithm without naming it. We'll name the algorithm `CCC(n)` (for `CountCoinsChange(n)`). `CCC(n)` returns the **minimum number of coins** required to make $c$ cents of change using only pennies, dimes, and quarters. Finish `CCC`'s implementation below:

```
CCC(n):
  If n < 0:

    Return infinity

  Else, If n = 0:

    Return _______________

  Else, n > 0:

    Return the _______________ of these possibilities:

      ----------------------------------------

      ----------------------------------------

      ----------------------------------------
```

4. `CCC` does not actually return an optimal solution (the change to give), only the **number** of coins in an optimal solution. If we imagine allowing `CCC` to have two return values (e.g., returning a more complex object than an integer), it can also return the solution. Describe how.

5. Finish this recurrence for the runtime of `CCC`:

```
T(n) = 1                          for n ________

T(n) = ____________________________ otherwise
```

6. Give a depressing $\Omega$-bound on the runtime of `CCC`. *Hint:* try replacing $T(n)$ with a simpler one that lower-bounds $T(n)$ but makes the recursion tree much easier to draw.

7. Why is the performance so **bad**?

## 3 If I Had a Nickel for Every Time I Computed That

1. Rewrite `CCC`, this time storing (memoizing) each solution as you compute it so that you **never compute any solution more than once** (for a given call to `CCC`).

```
CCC(n):
  Create a new array Soln of length n

  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:
```

2. Not counting the cost of the **first** computation of `CCCHelper(x,...)` for all `x < 81`, give a good $\Theta$-bound on the runtime of `CCCHelper(81,...)`.

3. Not counting the cost of any **other** call's **first** computation, give a good $\Theta$-bound on the runtime of the first comutation of `CCCHelper(x,...)` for all $1 \leq x \leq c$, where $c$ is the amount requested on the first call to `CCCHelper`.

4. Give a $\Theta$-bound on the total cost of all these **first** computations.

5. Explain why this $\Theta$-bound also bounds the total runtime of the algorithm.

# 4 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!)

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

Soln(i) = ___________________________________________ otherwise
```

2. Which entries of the `Soln` 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

  For i = _____________________________________:

    Soln[i] = the ___________ of:

      ____________________________________,

      ____________________________________, and

      ____________________________________.

  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?

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?

## 5 Foreign Change

Modify the memoized and dynamic programming versions of `CCC` so that it handles foreign currencies where you receive the target amount `n` and an array of coin values `[c1, c2, ..., ck]`. (So, for dimes and quarters, the array would look like `[10, 25]`.)

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

# 6 Challenge

1. 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.

2. 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.

3. 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.)

4. Prove that you can take at least one greedy step if the algorithm takes two coins `[c1, c2]`, and `n` is at least as large as the least common multiple of `c1` and `c2`.

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

6. Find the SNL video on the First Citiwide Bank of Change, hampered by your presence in Canada.