Making Change: Memoization and Dynamic Programming

In U.S. currency the problem of making change is easily solved using a "greedy" strategy which yields the smallest possible number of coins. For example, suppose we must make up 67 cents in change using quarters (25 cents), dimes (10 cents), nickels (5 cents) and pennies (1 cent). We first choose the largest coin less than or equal to the total (a quarter) and subtract as many multiples of it as possible while leaving a nonnegative total. In this case, we use 2 quarters, leaving a total of 17 cents. We then use the next coin that is less than or equal to the total (a dime) and subtract as many multiples of it as possible while leaving a nonnegative total. In this case, we use 1 dime, leaving a total of 7 cents. Repeating this operation, we use 1 nickel and then 2 pennies. This strategy uses 6 coins; it is "greedy" because it takes as big a "bite" out of the objective as possible at each step.

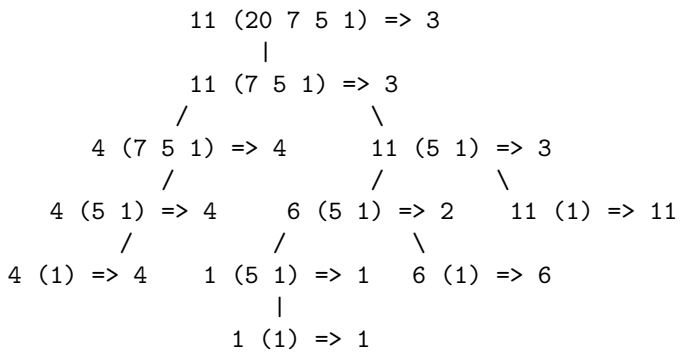
For a different currency a greedy strategy may not be optimal. In particular, if the currency has coins of values 20, 12, 6, 2, and 1 and the task is to make up a total of 37, then the greedy strategy will use a 20, a 12, two 2's and a 1, for a total of five coins. However, there is a better solution: three 12's and a 1, for a total of four coins.

We consider the problem of making change for an arbitrary coinage using as few coins as possible. The inputs are the total to be made and a list of the denominations of the coins. If the total cannot be made with the given coinage, the procedure should return `#f`.

A recursive solution can be based on dividing the problem into two cases: either the first coin is used at least once (which reduces the total) or not (which reduces the set of coin denominations.) A recursive procedure based on this idea is the following.

```
(define min-coins
  (lambda (total coins)
    (cond
      ((= total 0) 0)
      ((null? coins) #f)
      ((> (car coins) total) (min-coins total (cdr coins)))
      (else
        (special-min  (special-plus 1 (min-coins (- total (car coins)) coins))
                      (min-coins total (cdr coins)))))))

; if either x or y is #f, return the other; otherwise return (min x y)

(define special-min
  (lambda (x y)
    (cond
     ((not x) y)
     ((not y) x)
     (else (min x y)))))

; if either x or y is #f, return #f; otherwise return (+ x y)

(define special-plus
  (lambda (x y)
    (if (or (not x) (not y))
        #f
        (+ x y))))
```

As an example, consider the call tree for `(min-coins 11 '(20 7 5 1))`.



(Here we have abbreviated the computations of the form `n (1)`.) This tree suggests that there will in general be many repeated computations in this approach. If we memoize the procedure, how many different choices of arguments to `min-coins` can there be? The totals may vary from 0 to the original total to be made. The sets of coins may be any suffix of the original list of coins. Thus, an upper bound on the number of different choices of arguments is proportional to the original total times the number of coins. Hence memoization can be helpful here.

We may also consider a dynamic programming approach: can we fill in a table of the values of `min-coins` from the bottom up? If we consider the set of coin denominations to be fixed, then for each possible total from 0 to the original total, there is an optimal number of coins to make up that total. We can fill in the table from the bottom up; we consider the example of a total of 15 with the coins `(20 7 5 1)`.

total:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
min-coins:	0	1	2	3	4	1	2	1	2	3	2	3	2	3	2	3

The table is filled in as follows. The totals of 0 through 4 are made up with the corresponding number of coins of denomination 1. When we reach 5, we have an option of using either a coin of denomination 5 or a coin of denomination 1, so we choose the minimum of the solution for 0 and one the solution for 4 and add 1, getting 1 for the optimal solution for 5. In general, for a total of n, we enter one more than the minimum of the solutions for $n-1$, $n-5$, $n-7$ and $n-20$, (where we consider only those differences that are nonnegative.) Thus, for 14 we add one to the minimum of the solutions for 13 (which is 3), 9 (which is 3), and 7 (which is 1), which yields 2.

The time it takes to fill in the table is proportional to its length (which is one more than the original total) times the number of coins (since in general we have to subtract each coin from the index), which is the number of different arguments we considered in the memoization solution above.