Adina Edwards
15th Oct. 2017
CS325

HW 3

1. Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the **density** of a rod of length i to be $\frac{p_i}{i}$, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

   Let us consider the price table given in $Figure\ 15.1$ from the textbook, but add a row for the density at each length.

Length, i	1	2	3	4	5	6	7	8	9	10
Price, p_i	1	5	8	9	10	17	17	20	24	30
Density, $\frac{p_i}{i}$	1	2.5	2.667	2.25	2	2.833	2.429	2.5	2.667	3

   If we start with a rod of 4 inches, that is, $i = 4$, and apply a greedy algorithm, it will first cut the rod at the highest density point of 3 inches. Since the remaining portion of the rod is $i = 1$ and $1 \leq i$, we are done cutting. This situation gives us $n = i_3 + i_1$ with a revenue of $r_n = p_3 + p_1 = 8 + 1 = 9$.
   In contrast, if the same rod is cut using a dynamic programming algorithm, it will consider all possible cuts and compare each potential solution to find the optimal one. In this case, the optimal one would be to cut the 4 inch rod at $i = 2$, which gives $n = i_2 + i_2$ with a revenue of $r_n = p_2 + p_2 = 5 + 5 = 10$.
   Since $9 < 10$, there is at least one situation where this greedy algorithm does not provide the most optimized solution.
2. Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

   Please note, pseudo code is python-esque.

```
extendedBottomUpCutRodWithCost(p, n, c)
        r = [0] * n
        s = [0] * n
        for j in range(1, n)
                q = float('-inf')
```

```
            for i in range(1, j)
                if q < p[i] - c + r[j-i]
                    q = p[i] - c + r[j-i]
                    s[j] = i
            r[j] = q
        return r, s
```

3. Given a list of integers $v_1, v_2, ..., v_n$, the product-sum is the largest sum that can be formed by multiplying adjacent elements in the list. Each element can be matched with at most one of its neighbors.
   For example, given the list $4, 3, 2, 8$ the product-sum is $28 = (4 * 3) + (2 * 8)$, and given the list $2, 2, 1, 3, 2, 1, 2, 2, 1, 2$ the product-sum is $19 = (2 * 2) + 1 + (3 * 2) + 1 + (2 * 2) + 1 + 2$.
   a. Compute the product-sum $2, 1, 3, 5, 1, 4, 2$.
      For any v_i where $v_i \geq 0$ and $1 \leq i \leq n$, $1 * v_i = v_i$ which will always be less than $1 + v_i$. This means, in order to find the largest possible product-sum, any v_i adjacent to a 1 should be added to that 1 instead of multiplied by that 1. Because 1 occurs at most every three values in this given input, this observation makes calculating the given input easier to do by hand, although this observation doesn't lend itself to thinking about the problem in terms of dynamic programming.
      $$2 + 1 + (3 * 5) + 1 + (4 * 2) = 27$$
   b. Give the dynamic-programming optimization formula $OPT[j]$ for computing the product-sum of the first j elements.
      The problem is trivial if $j < 2$. When $j \geq 2$, we have a choice of either multiplying or adding two adjacent values, v_j and v_{j-1}. If we multiply two adjacent values, v_j and v_{j-1}, then we must add this product to the subproblem that is finding the product-sum of the remaining values, $OPT[j-2]$, in order to follow the requirements of the original problem.
      $$OPT[j] = max(OPT[j-1] + v_j, OPT[j-2] + v_{j-1} * v_j)$$
   c. What would be the asymptotic running time of a dynamic-programming algorithm implemented using the formula in part b)?

```
productSum(V, n)
        OPT [] = [0] * n
        OPT[0] = 0
        OPT[1] = V[1]
        For j = 2 through j = n
                OPT[j] = max(OPT[j-1]+V[j], OPT[j-2] + V[j] * V[j-1])
        Return OPT[n]
```

The asymptotic running time is O(n) because it has to run through basically all values of n once in order to determine the optimal solution.

4. Given coins of denominations $1 = v_1 < v_2 < ... < v_n$, we wish to make change for an amount A using as few coins as possible. Assume that v_i's and A are integers. Since $v_1 = 1$ there will always be a solution.
    a. Describe and give pseudocode for a dynamic programming algorithm to find the minimum number of coins to make change for A.

    $T(A) = min(T[A],\ T[A - T[A-1]] + 1)$

    To be a dynamic program, we need to find the optimal situation for a given problem by considering the subproblems. We can make change for A by using A pennies, or by using $(A-1)$ pennies plus one $(+1)$ of a larger denomination coin. Whichever of those options in the minimum, will be the most optimal solution for the making change problem.

    Starting with $A = 1$ and assuming American currency, we can make change for A by providing a single penny. This is the base case. We record the fact that the minimum number of coins it takes to make change for $A = 1$ is a single penny, or $minCoin[1] = 1$. We also need another array to hold how many of each coin denomination is used to make change of a certain amount, A. We do this by creating an array $usedCoins$ and storing the index of the coin denomination needed to make that amount of change, so $usedCoins[1] = 1$ (where 1 means penny).

    At $A = 2$, we give 2 pennies, recording that $minCoin[2] = 2$, $usedCoins[2] = 1$ and so forth until $A = 5$ where we give the coin at the next highest index in the coin denomination list, a nickel, and record $minCoin[5] = 1$, $usedCoins[5] = 2$ where (2 means nickel). At $A = 6$, we give one nickel and one penny, so we record $minCoin[6] = 2$ and $usedCoins[6] = 1$. $A = 7$, one nickel and two pennies and so on until $A = 10$ where we give a dime.

    Each amount for which we make change can be dependent upon the previous amount, giving us $T[A - T[A-1]] + 1$; the time to find the change for amount A is $1+$ the time it takes to find the change for $A - 1$.

    For $a \leq A$
        $coinQty = a$
        $indexOfUsedCoin = 1$
        For $coinIndex$ in $coinList[]$
            If $coinQty$ is less than $minCoins[a - coinIndex] + 1$
                $coinQty$ is updated to $minCoins[a - coinIndex] + 1$
                $indexOfUsedCoin$ is updated to $coinIndex$
    $minCoins[amount]$ is updated to $coinQty$
    $usedCoins[amount]$ is updated to $indexofUsedCoin$
    Return $minCoins[A]$, $usedCoins[A]$

This will give us the minimum number of coins needed to make change, where the amount of change is given by the index, as well as the index of the coin needed to make said amount of change. In order to print out which coins are needed, given the quantity of each coin used, we need to run $usedCoins[A]$ through a function that writes out which coins are used.

While $coin > 0$
    $thisCoin$ is set to $usedCoins[coin]$
    Write out the value of $thisCoin$
    $coin$ is updated to $coin - thisCoin$

This goes through the $usedCoins$ array made in the previous function and prints out indexes for each coin used to make change for A.

b. What is the theoretical running time of your algorithm?
There are two factors upon which this algorithm is dependent: the total amount for which we're making change, A, and the number of coin denominations, n, from which to choose to make change. In order to make change for A, it will have to run through all values 1 through A and it will have to check all of these values against all possible coin denominations. This implies that it will take at most and at least An time, so the theoretical running time is $\Theta(nA)$.

5. Making Change Implementation
See submission on TEACH.
6. Making Change Experimental Running Time
    a. Collect experimental running time data for your algorithm in Problem 4. Explain in detail how you collected the running times.
    b. On three separate graphs, plot the running time as a function of A, running time as a function of n and running time as a function of nA. Fit trend lines to the data. How do these results compare to your theoretical running time?