CS 325-400 Fall 17 Peter Moldenhauer
HW 3 moldenhp@oregonstate.edu
10/9/17

1. Rod Cutting: 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 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 <= i <= n, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length n – i.

   The following is a counterexample for the "greedy" strategy: Suppose the given rod is of length 4 and the corresponding prices per each length is 1 (length 1), 20 (length 2), 33 (length 3) and 36 (length 4). The resulting densities would be 1 (length 1), 10 (length 2), 11 (length 3) and 9 (length 4). Based on the definition of the "greedy" strategy, the first piece to cut is the piece with the largest density. Therefore, a rod of length 4 would first be cut at length 3 which has the highest density of 11. This gives a price of 33 for this first cut length. After this first cut we are just left with the remaining rod of length 1 since no more cuts are possible. This rod of length 1 gives a price of 1. The total price for the rod cut with the "greedy" strategy would then be 34 (33 + 1). However, this is less than if we were to use the dynamic programming strategy which would result in a total price of 40. Note: the dynamic programming strategy would cut the rod of length 4 into two pieces of length 2 which gives the total price of 40 (20 + 20).

Length	1	2	3	4
Price	1	20	33	36
Density	1	10	11	9

2. Modified Rod Cutting: 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.

   A dynamic-programming algorithm to solve the modified rod cutting problem could be based off of the bottom-up dynamic-programming approach that appears in the textbook (p. 366). The modified rod cutting algorithm would take in an additional input "c" which represents the cost for each cut. The total inputs would then be p (an array of prices), n (the length of the rod) and now also c (the cost per cut). The next change is that now its set that q = p[j]. This is set this way to handle the case in which no cuts are made. In this case, the price would be that of only the whole bar (with no deduction for cuts). Similarly, another change (setting the second loop to go from 1 to j – 1) handles the case when i = j. That is, when no cuts were made. The final (and biggest) change in this modified rod cutting

algorithm is the line where q = max(q, p[i] + r[j – i] – c). This is the logic that calculates and deducts the cost of making the cut. This modified rod cutting algorithm can be seen below:

```
ModifiedRodCutting(p, n, c)
        let r[0…n] be a new array
        r[0] = 0
        for j = 1 to n
                q = p[j]
                for i = 1 to j – 1
                        q = max(q, p[i] + r[j – 1] – c)
                r[j] = q
        return r[n]
```

3. Product Sum: Given a list of n integers, v1,…vn, 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 of 2, 1, 3, 5, 1, 4, 2

      2 + 1 + (3*5) + 1 + (4*2) = 27

   b) Give the dynamic programming optimization formula OPT[j] for computing the product sum of the fist j elements.

      The dynamic programming optimization formula is as follows:

$$OPT[j] = \begin{cases} \max\{OPT[j-1] + v_j,\ OPT[j-2] + v_j * v_{j-1}\} & \text{if } j >= 2 \\ v_1 & \text{if } j = 1 \\ 0 & \text{if } j = 0 \end{cases}$$

   c) What would be the asymptotic running time of a dynamic programming algorithm implemented using the formula in part b).

      The asymptotic running time of the dynamic programming algorithm that uses the above formula is Theta(n). This is because we would only need to loop through the array of the j elements to find the solution. Depending on the size of the array of j elements, we may have to “add” multiple for loops together through the array but it is not necessary for any nested loops (which would result in time greater than n). Therefore, simply looping through the array of n elements to get the solution would result in the running time of Theta(n).

4. Making Change: Given coins of denominations (value) 1 = v1 < v2 < … < vn, we wish to make change for an amount A using as few coins as possible. Assume that vi's and A are integers. Since v1 = 1 there will always be a solution.

Formally, an algorithm for this problem should take as input:

- An array V where V[i] is the value of the coin of the i^{th} denomination.
- A value A which is the amount of change we are asked to make

The algorithm should return an array C where C[i] is the number of coins of value V[i] to return as change and m the minimum number of coins it took. You must return exact change so

$$\sum_{i=1}^{n} V[i] \cdot C[i] = A$$

The objective is to minimize the number of coins returned or:

$$\text{m} = \min \sum_{i=1}^{n} C[i]$$

a) Describe and give pseudocode for a dynamic programming algorithm to find the minimum number of coins to make change for A.

A dynamic programming algorithm could be described as follows: To solve this problem, the algorithm must loop through each element of the array of coin values (V). The algorithm must also loop through each element that makes up the total of A to check if the current coin can go into A. If this coin can in fact go into A, then it is subtracted from A and the remainder as calculated in the same manner. In the pseudocode below, T represents the array (from 0 to A) to hold how many coins are needed per element, R represents the array to hold the last coin used on the corresponding T element, and C represents the array to hold all of the coins and their corresponding values. Most of the work in the algorithm is accomplished with the formula: T[i] = min(T[i], 1 + T[i-C[j]]). In short, the algorithm loops through the array of coins to check if the currently selected coin leads to the minimal total number of coins needed to make up the value in amount A. Finally, to get the number of individual coins needed to make the correct change, the algorithm subtracts the value of the used coin from the total and repeats until 0 is reached. The pseudocode for this algorithm can be seen below…

```
MakingChange(V, A)
    T[ ] = new array
    R[ ] = new array
    C[ ] = new array
    T[0] = 0
    R[0] = -1
```

```
for i = 1 to A
        T[i] = inf
        R[i] = -1

for i = 0 to V.length
        C[i] = 0

for j = 0 to V.length
        for i = 1 to A
                if i >= V[j]
                        if (T[i-V[j]] + 1 < T[i])
                                T[i] = 1 + T[i-V[j]]
                                R[i] = j
total = A
while total != 0
        C[R[total]] += 1
        total = total – V[R[total]]

return C
```

b) What is the theoretical running time of your algorithm?

The theoretical running time of this algorithm is pseudo polynomial: O(A*V) or similarly: O(total * number_of_coins). This said to be “pseudo polynomial” because the total is not in any way proportional to the number of coins.

5. Making Change Implementation

[Code submitted separately to 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.

I collected the running time data for my algorithm in the following manner. I wanted to time my algorithm based on three cases. For the first case I kept A the constant and relatively small while I increased the array of values for V larger and larger. The second case I kept the array of values for V relatively small while I increased A larger and larger. Then with the last case I kept V and A of “medium” size while incrementing both.

For case 1 I kept A at 100 and then I started with an array of size 1000 for V and increased the size by 5,000 each time until the final run of 30,000 for V.

For case 2 I kept V at an array size of 100 and started A at 1000 and increased by 5,000 each time until the final run of 30,000 for A
For case 3 I started V at an array size of 1000 and A at 1000. I then incremented both up by 5000 each run until the last run where V had an array size of 30,000 and A was 30000 as well.

Note: to prevent coin denominations of similar value, when I generate the array sizes for V, I skip all the odd or even numbers. If V is an array of increasing odd numbers for example, I would then assign an even number to A. All of my collected running time data can be seen in the charts below:

Case 1:

V (array size)	A	Time in seconds
1 – 1000	100	.005
1 – 5000	100	.036
1 – 10000	100	.039
1 – 15000	100	.061
1 – 20000	100	.082
1 – 25000	100	.085
1 – 30000	100	.099

Case 2:

V (array size)	A	Time in seconds
1 - 100	1000	.017
1 - 100	5000	.112
1 - 100	10000	.265
1 - 100	15000	.384
1 - 100	20000	.537
1 - 100	25000	.668
1 - 100	30000	.802

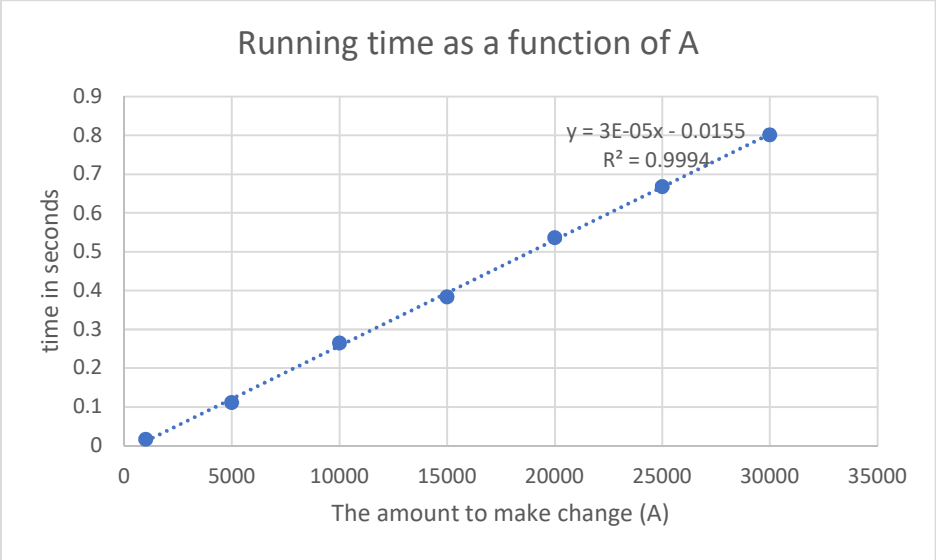
Case 3:

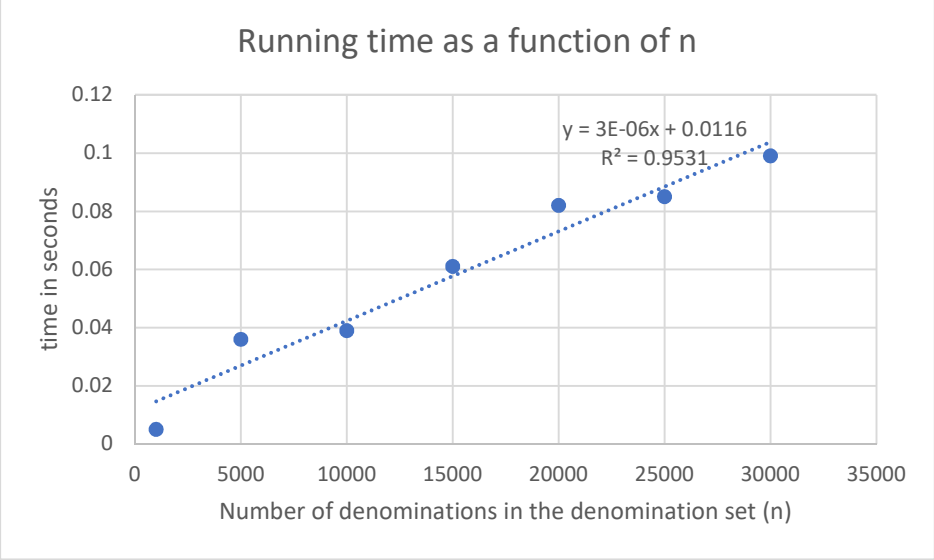
V (array size)	A	Time in seconds
1 – 1000	1000	.082
1 – 5000	5000	2.00
1 – 10000	10000	7.19
1 – 15000	15000	15.9
1 – 20000	20000	28.3
1 – 25000	25000	44.7
1 – 30000	30000	62.7

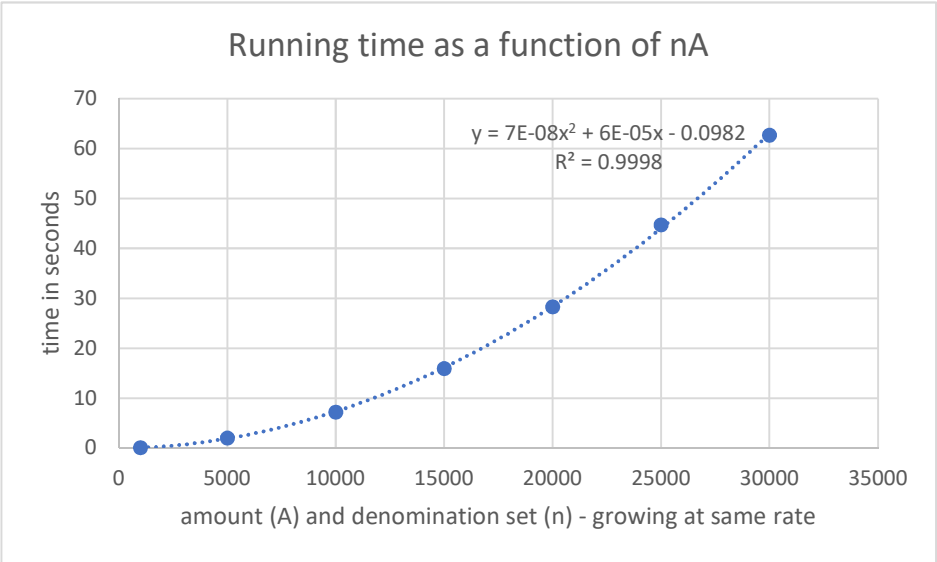
b) On three separate graphs plot the running time as a function of A, running time as a function of n and running time as a function of nA. Fit trend lines to the data. How do these results compare to your theoretical running time?

(Note: n is the number of denominations in the denomination set and A is the amount to make change).

The results that I collected are relatively similar to the theoretical running time of the algorithm. As mentioned earlier in a previous question, the theoretical running time of this making change algorithm is pseudo polynomial. My data was most similar to this running time when I plotted out the running time as a function of nA – that is when n and A were both large, close in value and increased at the same rate. However, when I plotted out the running time as a function of n and a function of A separately, both of these graphs were more closely related to linear running time. This makes sense because if A is small and n is large, then n would dominate and force the running time to be O(n). Similarly, I got the same result if n is small and A is large. You can see my graphs of my collected data below…






Running time as a function of nA
y = 7E-08x² + 6E-05x - 0.0982
R² = 0.9998
time in seconds
70
60
50
40
30
20
10
0
0
5000
10000
15000
20000
25000
30000
35000
amount (A) and denomination set (n) - growing at same rate