Class: CS-325
Term: Fall 2017
Author: Jon-Eric Cook
Date: October 15, 2017
Homework: #3

1) (2 points) Rod Cutting: (from the text CLRS) 15.1-2

Show, by means of 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 pi / 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, have maximum density. It then continues by applying the greedy strategy to the remaining piece of length n - i.

Answer:
Below is a counterexample for the “greedy” strategy

Length i	1	2	3	4
Price p(i)	6	22	36	40
Density p(i)/i	6	11	12	10

Following the “greedy” strategy, if there was a rod of length 4, then, according to the above table, the first cut would be at 3, yielding a price of 36 and also the highest density of 12. From this, a length of 1 would be left over and combined with the length of 3, would result in an overall price of 42. The correct or optimal way is to cut the rod into two rods of length 2. By doing this, a total price of 44 would be achieved.

2) (3 points) Modified Rod Cutting: (from the text CLRS) 15.1-3

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.

Answer:
I will be modifying the pseudo located on page 366 of the text book

```
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 - i] - c)
                r[j] = q
        return r[n]
```

Things that were modified are as follows: making “c” an input for the ModifiedRodCutting function, setting q = p[j], making the second for loop go from 1 to j - 1, and setting q = max(q, p[i] + r[j - i] - c). By setting q = p[j], this takes care of when no cuts are made. When there are no cuts made, the revenue would simply be the value for the whole bar. Setting the second for loop to go from 1 to j - 1 is also done to handle when i = j, i.e no cuts were made. The main change was setting q = max(q, p[i] + r[j - i] - c). This was were the cost for a cut was deducted and then compared to see if the cut itself was worth it.

3) (6 points) 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.
b) Give the dynamic programming optimization formula OPT[j] for computing the product-sum of the first j elements.
c) What would be the asymptotic running time of a dynamic programming algorithm implemented using the formula in part b).

Answer:
a) 27 = 2 + 1 + 3*5 + 1 + 4*2

b) OPT[j] = max(OPT[j - 1] + vj, OPT[j - 2] + vj*vj-1). if j >= 2
= v1 if j = 1
= 0 if j = 0

c) The asymptotic running time of the dynamic programming algorithm is theta(n). This is so because to properly use the above equation, it needs to step through the entire array. To step through the entire array, a simple for loop is required and thus, leads to the asymptotic run time of theta(n).

4) (5 points) 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:

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

a) Describe and give pseudocode for a dynamic programming algorithm to find the minimum number of coins to make change for A.
b) What is the theoretical running time of your algorithm?

Answer:
a) The required dynamic programing algorithm to solve this problem needs to look through each element of the coin array and also each element that makes up the total of A and check if the current coin can go into the element of A. If the coin can go into the element in A, then it is subtracted and the remainder is assessed. The main formula of this algorithm is T[i] = min(T[i], 1+T[i-C[j]]) where T is an array from 0 to A and C is an array holding all the coins and their corresponding values. The main goal is to work through the A array, going from 0 to A and seeing if the currently selected coin from the coin array will lead to the minimal total number of coins needed to make up the value of the selected element in A. To get an array that contains how many individual coins were needed to make proper change, one has to go backwards and subtract the value of the used coin from the total and then continue to do this until there is nothing left. Below is some pseudo code.

```
MakeChange(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 length of V
                C[i] = 0

        for j = 0 to length of V
                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) The theoretical running time of this algorithm is sudo polynomial: O(A*V)

5) (10 points) Making Change Implementation

Submit a copy of all your files including the txt files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will only test execution with an input file named amount.txt.

You may use any language you choose to implement your DP change algorithm. The program should read input from a file named “amount.txt”. The file contains lists of denominations (V) followed on the next line by the amount A.

Example amount.txt:
1 2 5
10
1 3 7 12
29
1 2 4 8
15

In the above example the first line contains the denominations V=(1, 2, 5) and the next line contains the amount A = 10 for which we need change. There are three different denomination sets and amounts in the above example. A denomination set will be on a single line and will always start with the 1 “coin”.

The results should be written to a file named change.txt and should contain the denomination set, the amount A, the change result array and the minimum number of coins used.

Example change.txt:
1 2 5
10
0 0 2
2
1 3 7 12
29
0 1 2 1
4
1 2 4 8
15
1 1 1 1
4

In the above example, to make 29 cents change from the denomination set (1, 3, 7, 12) you need 0: 1 cent coin, 1: 3 cent coin, 2: 7 cent coins and 1: 12 cent coin for a total of 4 coins.

Answer:
See files submitted to TEACH

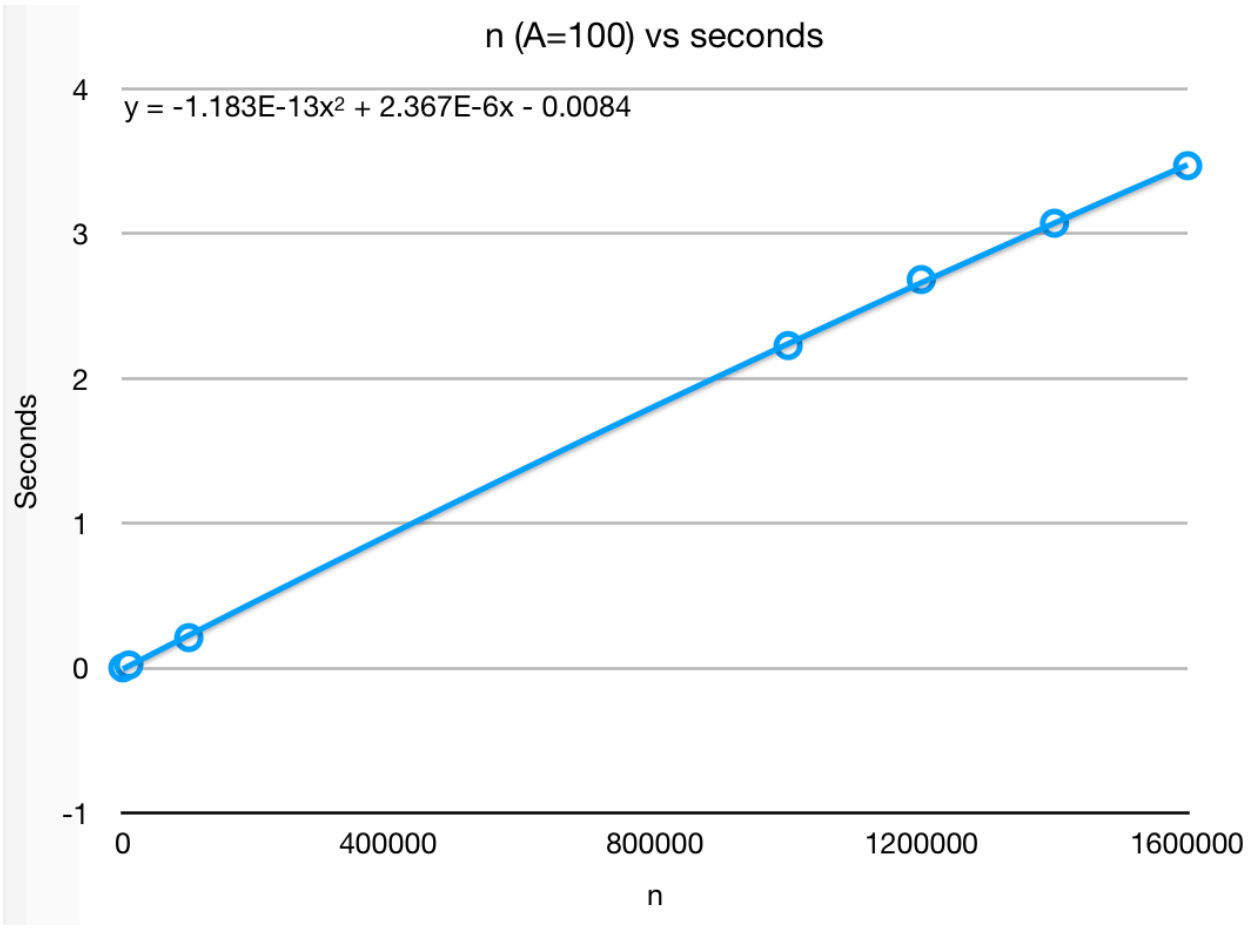
6) (4 points) Making Change Experimental Running time

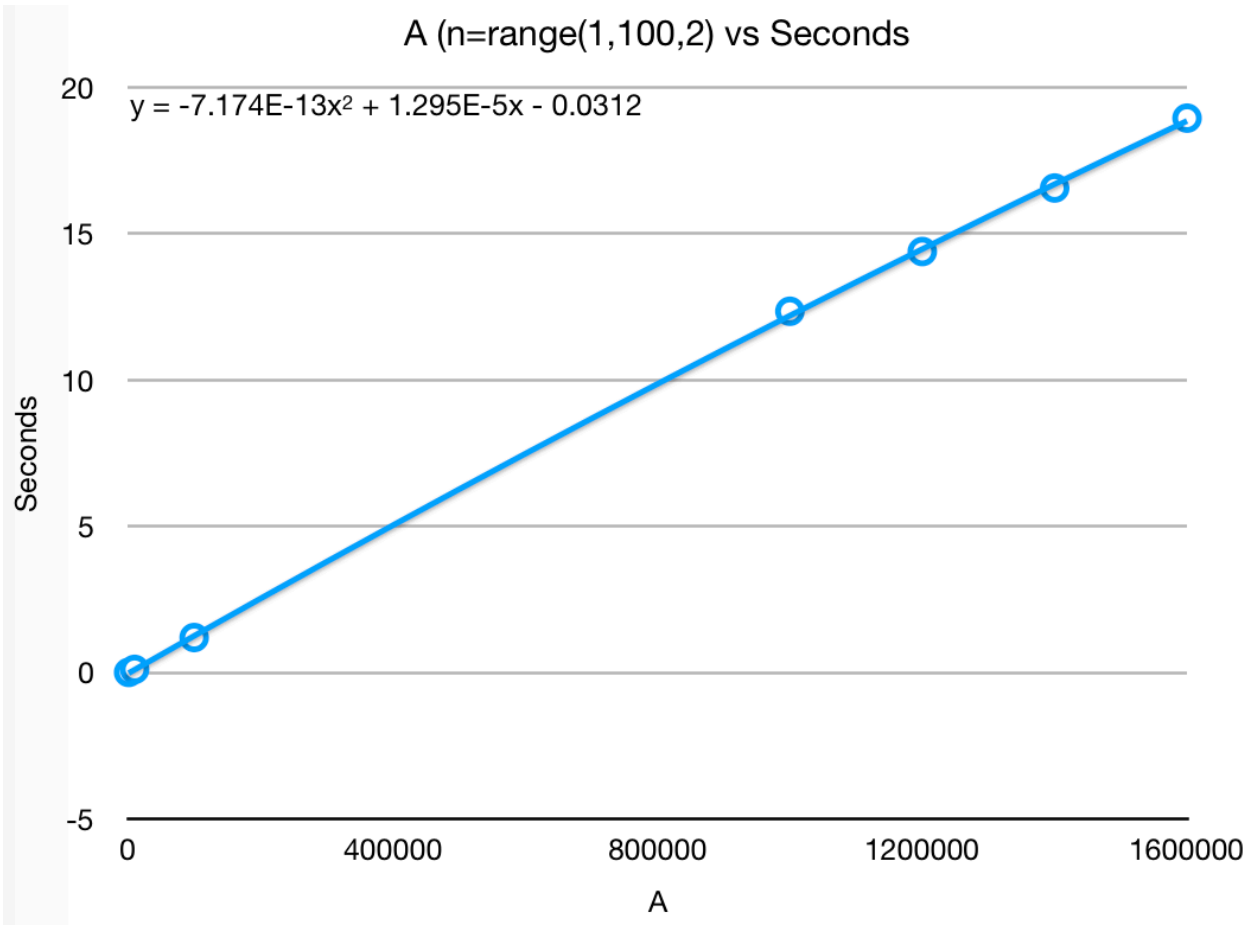
a) Collect experimental running time data for your algorithm in Problem 4. Explain in detail how you collected the running times.
b) On three separate graphs plot the running time as a function of A, running time as a function of n and running time as a function of nA. Fit trend lines to the data. How do these results compare to your theoretical running time? (Note: n is the number of denominations in the denomination set and A is the amount to make change)

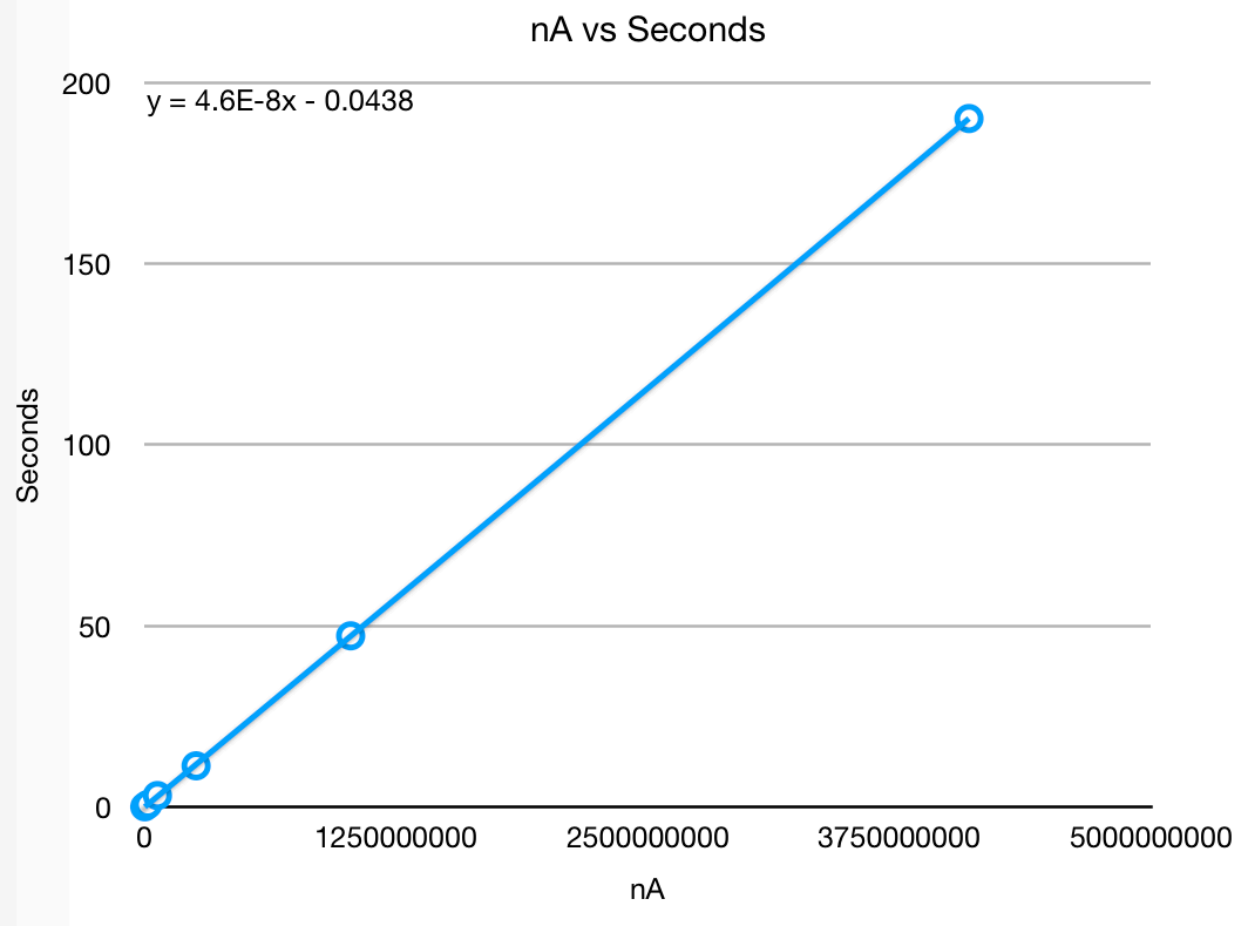
Answer:
a) The experimental data was collected by timing how long the make_change function took to execute for the following cases: a large array of values for V, increasing in size and a small number for A, a small array of values for V, increasing in size and a large number for A and a medium array of values for V, increasing in size and a medium number for A, increasing in size. See part b for data collected.

b) The results seen below are relatively close to that of the theoretical running time. Seeing how the theoretical running time is sudo polynomial, the polynomial trend line was selected. Although, to the eye it appears to be more linear than anything. One can imagine that if A was small and n large, then n would dominate and lean the running time to be O(n) and vice versa. Now, if both A and n were large and close in value, the running time may lean towards O(k^2), were k is approximately equal to n and A. I believe my last graph shows little signs of being O(k^2) as not enough time was allowed to pass and the value of nA was not large enough.

n (A = 100)	sec	A (n = range(1,100,2))	sec	nA	sec
1000	0.00244	1000	0.00767	1000000	0.04386
10000	0.02179	10000	0.11005	4000000	0.15844
100000	0.21061	100000	1.19897	16000000	0.66213
1000000	2.22701	1000000	12.34656	64000000	3.15528
1200000	2.68295	1200000	14.38872	256000000	11.36361
1400000	3.07194	1400000	16.55835	1024000000	47.27168
1600000	3.46913	1600000	18.94592	4096000000	190.13332


n (A=100) vs seconds
y = -1.183E-13x² + 2.367E-6x - 0.0084
Seconds
4
3
2
1
0
-1
0
400000
800000
1200000
1600000
n


A (n=range(1,100,2) vs Seconds
y = -7.174E-13x² + 1.295E-5x - 0.0312
20
15
10
5
0
-5
Seconds
0
400000
800000
1200000
1600000
A


nA vs Seconds
y = 4.6E-8x - 0.0438
200
150
100
50
0
Seconds
0
1250000000
2500000000
3750000000
5000000000
nA

Reference: The link below was used to help understand and complete part of this assignment.
https://www.youtube.com/watch?v=NJuKJ8sasGk