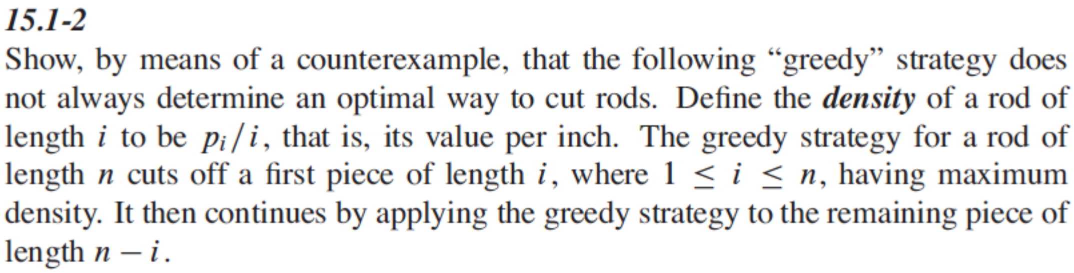
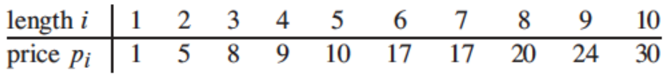
Christopher Ragasa
CS 325 – HW 3
4/22/18

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



15.1-2

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 $1 \le i \le n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.



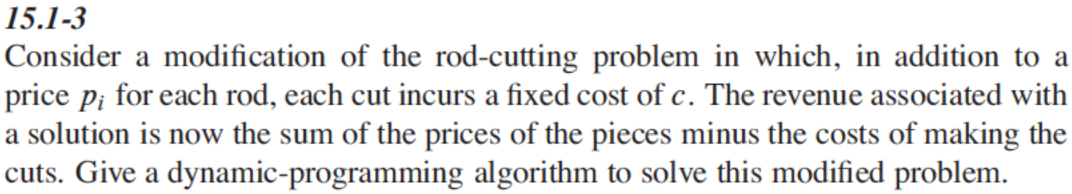
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Using the following price table given above, we can calculate the following densities for each given length/price:

Length, i	Price, pi	Density, pi/i
1	1	1.0
2	5	2.5
3	8	2.7
4	9	2.3

Let the given rod length be 4. Using the greedy strategy, we would first cut a rod of length i = 3, as it provides the maximum density of 2.7 for a price of $8. This leaves us with a rod of length 1 with a price of $1. The total price of this rod from derived from the greedy strategy is $9. However, the optimal way is to cut the rod into 2 pieces of length 2 at $5 per piece (total of $10). This shows that the following "greedy" strategy doesn't always determine an optimal way to cut rods.

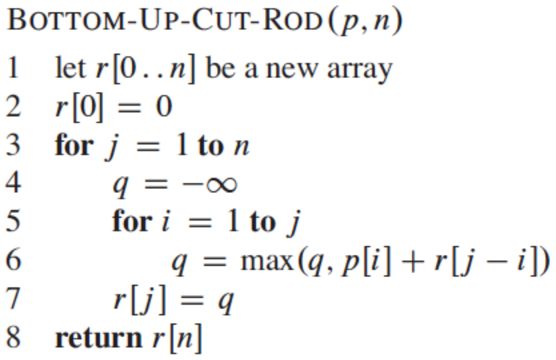
Problem 2: *(3 points)* **Modified Rod Cutting**: (from the text CLRS) 15.1-3



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.

Consider the bottom-up-cut-rod implementation given in the textbook (see below):



```
BOTTOM-UP-CUT-ROD$(p, n)$
1  let $r[0..n]$ be a new array
2  $r[0] = 0$
3  for $j = 1$ to $n$
4      $q = -\infty$
5      for $i = 1$ to $j$
6          $q = \max(q, p[i] + r[j - i])$
7      $r[j] = q$
8  return $r[n]$
```

There are two main changes we need to make to implementation above to account for the modification. In line 6 of the implementation above, q is assigned to the maximum of q or p[i] + r[j – i]. However, in the modified version, the revenue associated with a solution is now the sum of the prices of pieces minus the costs of making the cuts. Therefore, line 6 is modified below to assign the max of q or p[i] + r[j – i] – c. The second modification is in lines 4 and 5. In line 4 and 5 in the implementation above, q is set to negative infinity and the for loop runs from i = 1 to j. However, in the modified version, we need to account for the case in which no cuts are made. Therefore, the total revenue in this case is p[j] and the for loop instead runs from i to j – 1.

```
Modified-Cut-Rod(p, n, c)
    1       let r[0 .. n] be a new array
    2       r[0] = 0
    3       for j = 1 to n
    4           q = p[j]
    5           for i =  1 to j – 1
    6               q = max(q, p[i] + r[j – i] – c)
    7           r[j] = q
    8       return r[n]
```

Problem 3: *(6 points)* **Test Time**: OSU student, Benny, is taking his CS 325 algorithms exam which consists of n questions. He notices that the professor has assigned points { p1, p2, ..., pn } to each problem according to the professor's opinion of the difficulty of the problem. Benny wants to maximize the total number of points he earns on the exam, but he is worried about running out of time since there is only T minutes for the exam. He estimates that the amount of time it will take him to solve each of the n questions is { t1, t2, ..., tn }. You can assume that Benny gets full credit for every question he answers completely. Develop an algorithm to help Benny select which questions to answer to maximize his total points earned. **Note:** NO partial credit is assigned to problems that are only partially completed.

(a) Verbally describe a DP algorithm to solve this problem.

a. Given that Benny gets full credit for every question he answers completely and no partial credit is given to problems that are partially completed, this problem can be solved with the 0-1 knapsnack approach.

b. A 0-1 knapsnack problem entails a given set of items, each with a weight and benefit. In this case, the weight is the time it takes to solve each of the n questions {t1, t2, …, tn}, and the benefit is the/value is the assigned amount of points to each problem {p1, p2, …, pn}.

c. Verbal explanation of the algorithm:

   i. We first need to initialize a 2-d array, where the column size of the array is the total weight capacity (in the case of this problem, time) and the row size of the array is the number of items (in the case of this problem, n).

   ii. For each item, we grow/increase the total weight from 0 to max weight to test the best possible value we can get for that item.

   iii. We repeat this process for each item, however when there is leftover space in the knapsnack (for example, if you choose an item of weight 4 and you are comparing it to the scenario with a knapsack of weight capacity 7), we check for the best possible value we can achieve with the remaining amount of space – these values will be stored in our table.

   iv. This process is repeat for every possible scenario of items/weights until the best total outcome is found. In this case, we are trying to find how Benny can achieve the most amount of points given a certain amount of time and the time it takes to solve each problem.

(b) Give pseudo code with enough detail to obtain the running time, include the formula used to fill the table or array.

```
for w = 0 to W
    B[0,w]  = 0
for i = 0 to n
    B[i, 0] = 0
    for w = 0 to W
    if wi <= w
        if bi + B[i-1, w-wi] > B[i-1, w]
            B[i, w] = bi + B[i-1, w-wi]
        else
            B[i, w] = B[i-1,w]
    else B[i, w] = B[i-1, w]
```

(c) What is the running time of your algorithm? Explain.

Looking at the pseudocode above, we can see that we run the first for loop W times, which would giving a time complexity of O(W). We then run a second for loop n times, giving a time complexity of O(n). However, this second for loop has another nested loop that runs W times. Since the loop is nested, we must multiply the two runtimes of W and n. Therefore, the total time complexity of this algorithm is O(nW).

(d) Would Benny use this algorithm if the professor gave partial credit for partially completed questions on the exam? Discuss.

No, Benny would not use this algorithm if the professor gave partial credit for partially completed questions on the exam. The 0-1 knapsack algorithm is designed to give the optimal solution when you are constrained to either choosing or not choosing a whole item – in the case of this problem, Benny either finished the entire problem and received full credit or did not receive any credit at all; there are no partial credit opportunities or ratios allowed to top off the knapsack (hence the name, 0-1). Using the 0-1 knapsack approach would not give the optimal solution if partial credit was received.

Problem 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 ith 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:

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

a) Describe and give pseudocode for a dynamic programming algorithm to find the minimum number of coins to make change for A.
    a. The minimum change problem uses a dynamic programming approach to derive a solution. We can use a bottom up approach. The algorithm iterates through each coin denomination and uses a minCoins table to determine if a more optimized solution can be used for each total value (from 0 to A).
    b. See the pseudocode below:

```
for each value from 0 to A
    assign current coin value to coinCount
    assign 1 to newCoin
        for each value denomination and if that value <= cents
            if the stored minCoin val + 1 is less than the current coinCount
                set coinCoint equal to the current amount – current coin value + 1
                set newCoin = current coin value
        assign coinCount in minCoins table
        assign newCoin in coinsUsed table
    return minCoins[A]
```

b) What is the theoretical running time of your algorithm?
    a. Looking at the pseudo-code, we see that we run the first for loop A amount of times, with A being the total value of money we are trying to create change from – we can say that this runs in O(N) time. There is a second for loop nested inside the first one that runs an m amount of times, with m being the number of

different denominations – we can say that this runs in O(m) time. Since the loops are nested, we need to multiply their individual time complexities to find the total time complexity of the algorithm. Therefore, the theoretical running time of the algorithm is pseudo-polynomial, O(N*m).

Problem 5: *(10 points)* **Making Change Implementation**

See python file provided.

Problem 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.

In order to calculate experimental running times for minChange as a function of A, I ran the algorithm with the same list of denominations while making changes to the change (A). In order to calculate experimental running times for minChange as a function of n, I ran the algorithm with the same amount of change while making changes to the number of denominations. In order to calculate experimental running times for minChange as a function of nA, I ran the algorithm while making adjustments to both the list (n) and change (A). Please see the tables below for details:

minChange runtimes as a function of A

list	change	time
1,2,3,4,5	100	0.000152
1,2,3,4,5	500	0.000498
1,2,3,4,5	1000	0.001006
1,2,3,4,5	1500	0.002274
1,2,3,4,5	2000	0.002056
1,2,3,4,5	2500	0.002587
1,2,3,4,5	3000	0.000301
1,2,3,4,5	3500	0.003831

minChange runtimes as a function of n

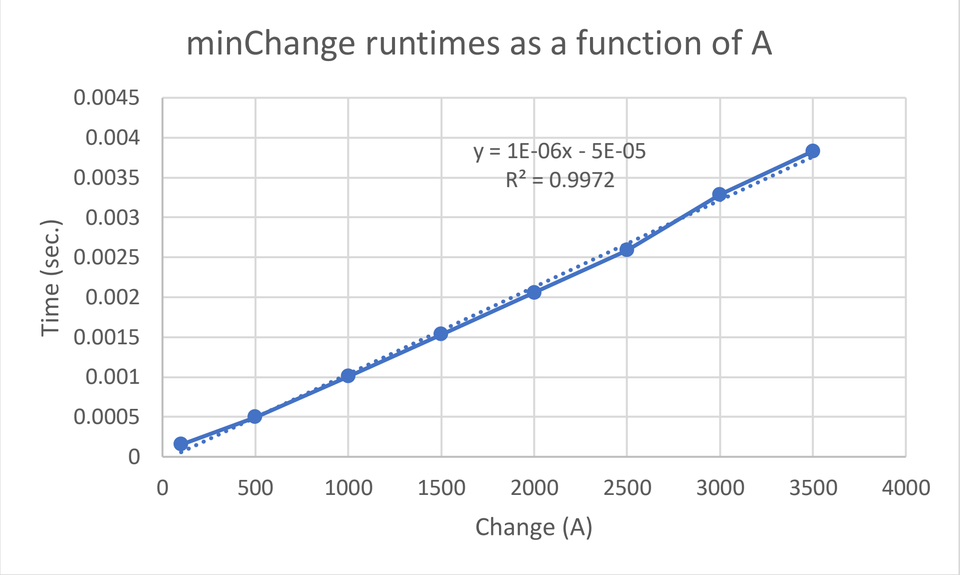
list	change	time
range(1, 11)	5000	0.008994
range(1, 21)	5000	0.01438
range(1, 31)	5000	0.020395
range(1, 41)	5000	0.023866
range(1, 51)	5000	0.033239
range(1, 61)	5000	0.034299
range(1, 71)	5000	0.038241
range(1, 81)	5000	0.046569

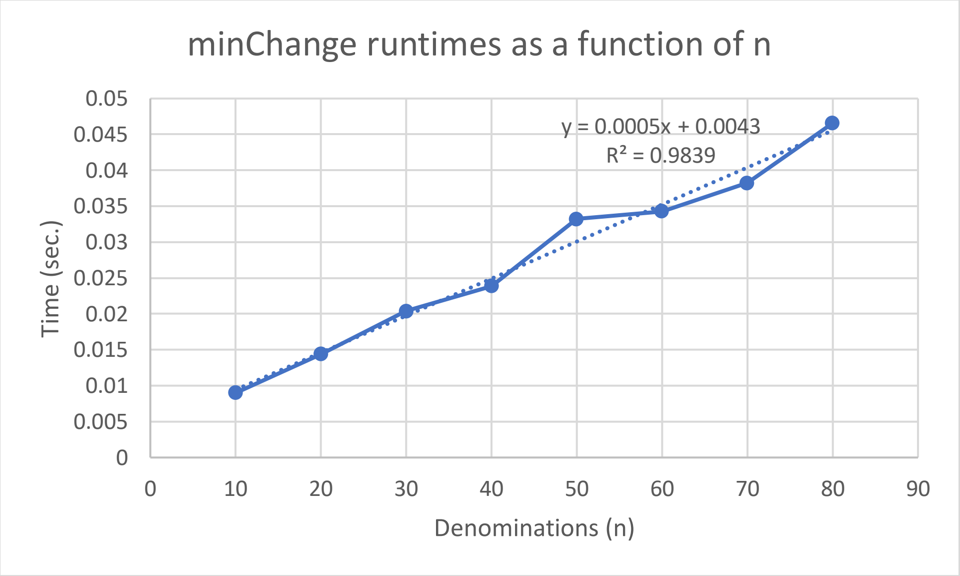
minChange runtimes as a function of nA

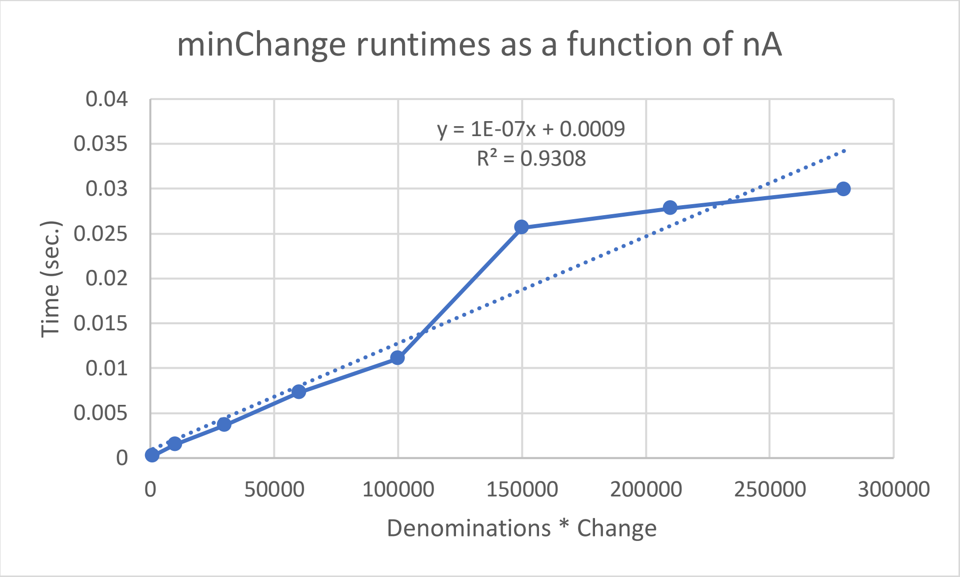
list	change	time
range(1, 11)	100	0.000231
range(1, 21)	500	0.001489
range(1, 31)	1000	0.003658
range(1, 41)	1500	0.007285
range(1, 51)	2000	0.011105
range(1, 61)	2500	0.025668
range(1, 71)	3000	0.027815
range(1, 81)	3500	0.029927

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)*

   a. I previously determined that the theoretical running time was pseudo-polynomial. However, looking at the graphs below, it appears that they all have a linear relationship. For example, the first graph has an R^2 value of 0.9972 to a linear fit, the second graph has an R^2 value of 0.9839 to a linear fit, and the third graph has an R^2 value of 0.9309 to a linear fit. Please see the three graphs below:






minChange runtimes as a function of nA
y = 1E-07x + 0.0009
R² = 0.9308
Time (sec.)
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0
50000
100000
150000
200000
250000
300000
Denominations * Change