Erin Alltop
CS325 – Fall 2017
HW3

1. Show, by means of a counter example, that the "greedy" strategy does not always determine an optimal way to cut rods.

   A greedy strategy is one that always makes a choice that seems to be the best at that moment. In other words, it is an emphasis on locally optimal choices based on the current state. A counterexample of this might be:

length i	1	2	3	4
price p_i	1	10	12	15
density p_i/i	1	5	4	3.75

   If the rod is length i = 4, then if we are following the greedy strategy we will first cut a rod of length i = 3 for a cost of 12 dollars. Then we are left with a rod of length i = 1 which is 1 dollar. This leaves us with a total cost of 13 dollars. Looking at the problem though we can see that the most optimal solution is two rods of length i = 2 at 10 dollars each or a total cost of 20 dollars. Thus we can see that the greedy algorithm cannot always determine an optimal way to cut rods.

2. Consider a modification of the rod-cutting problem in which, in additional 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.

   Using Python:

```
def mod_cut(p, n, c):
        r = [0 for _ in xrange(n + 1)] //new array r
        for j in range (1, (n+1)]
                r[j] = p[j]
                for I in range(1, j):
                r[j] = max(r[j], p[i] + r[j – i] – c) //sum of the prices of the pieces minus the cost
of making the cuts
        return r[n]
```

3. Product-sum
   a. 27 = 2 + 1 + (3 x 5) + 1 + (4 x 2)
   b. OPT[j] = max{OPT[j – 1] + k_j, OPT[j – 2] + k_j * k_{j-1}} if j >= 2; k_j if j = 0; 0 if j = 0
   c.

   We can write some pseudocode of this algorithm to help us find the running time:

```
def prodSum(A[0…n], k, n)
    if(n = 0) return 0
    A OPT = new int [n + 1]
    OPT[0] = 0;
    OPT[1] = k[1]
```

```
for j in range (2, n)
    OPT[j] = max(OPT[j – 1] + k[j], OPT[j – 2] + k[j] * k[j – 1])
Return OPT[n]
```

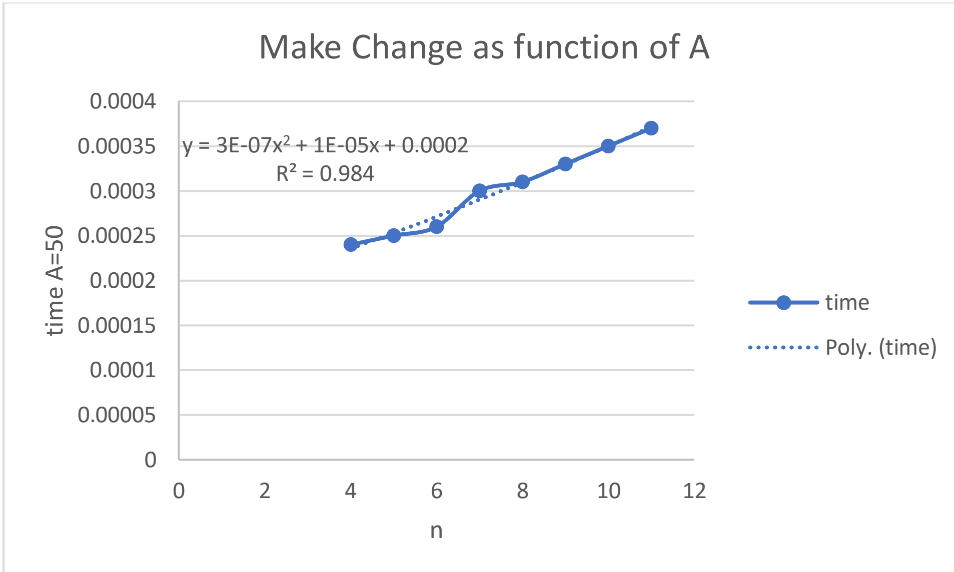
There is only one loop in this algorithm that has work of a constant time as it is simply assigning. Thus, the running time is O(n) as it iterates through the range 2 to n.

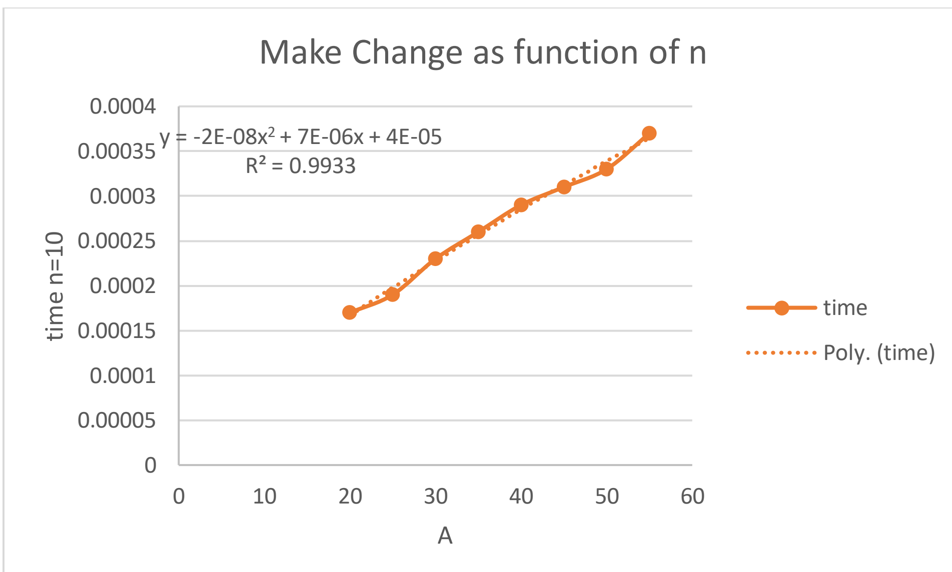
4.
   a. Pseudocode for a dynamic programming algorithm to find the minimum number of coins to make change for A.

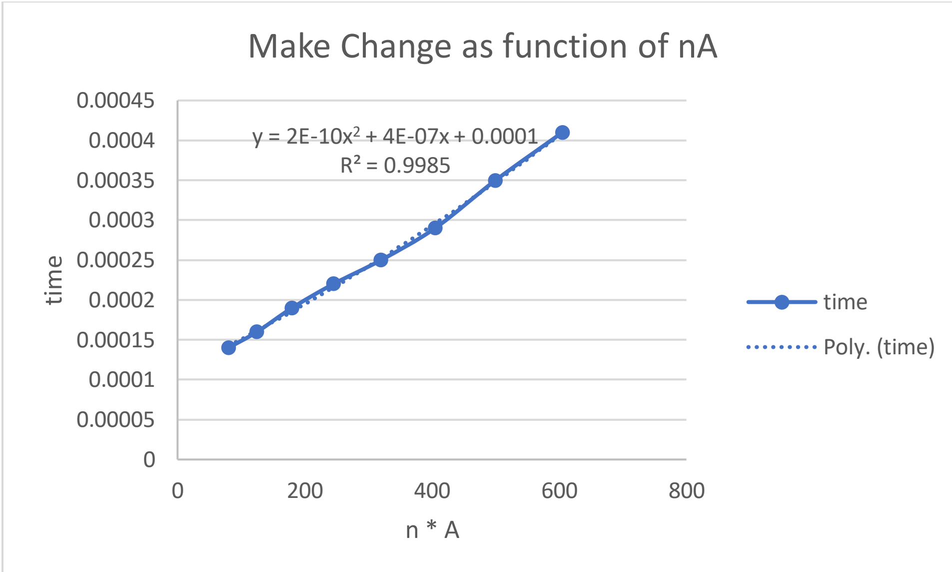
```
minCoins(A[0…n], m, v) //m = length of array, v = total value of coins
    coins = [v + 1] //stores min coins needed for i value
    coins[0] = 0 //base case
    for x in xrange(1, v)
            coins.append(x) //setting coins array values to infinite
            x = inifinity
            for y in xrange(0, m) //find least amount of coins needed for value
                    if(arr[y] <= x)
                            sub_res equals coins[x – arr[y]]
                            if(sub_res != infinity and sub_res+1 < coins[x])
                                    coins[x] = sub_res+1
    return coins[v] // return last value – min coins
```

   b. The theoretical running time of the algorithm is O(nk). Based on the pseudocode, we have a nested for loop that goes from 1 to v (total amount) in the first loop and 0 to m (length of coin array) in the second, which has a run time of O(nk). The work needed to assign or reassign variables is O(1). This gives us O(nk) + O(1) which ends up being O(nk).

5. Submitted to TEACH
6. Experimental running time data for algorithm in problem 4.
   a. For the running time of the Make Change algorithm as a function of n, I kept n as a constant number, specifically 10, and my A changed. For example, I had an array of [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] and changed A (the total amount) for each test. Similarly, for the running time data as a function of A, I kept A the same throughout (it stayed constant at 50. And for the nA running time I had A and n be the same values as they were for their respective running time experiments. For the graph I multiplied n and A per the recommendation.
   b. All three trendlines for these graphs shows a high R-factor for polynomial trendlines. This is similar to the theoretical running time which was also a polynomial.


Make Change as function of A
y = 3E-07x² + 1E-05x + 0.0002
R² = 0.984
time A=50
0.0004
0.00035
0.0003
0.00025
0.0002
0.00015
0.0001
0.00005
0
0
2
4
6
8
10
12
n
time
Poly. (time)


Make Change as function of n
y = -2E-08x² + 7E-06x + 4E-05
R² = 0.9933
time n=10
0.0004
0.00035
0.0003
0.00025
0.0002
0.00015
0.0001
0.00005
0
0
10
20
30
40
50
60
A
time
Poly. (time)


Make Change as function of nA
y = 2E-10x2 + 4E-07x + 0.0001
R² = 0.9985
time
0.00045
0.0004
0.00035
0.0003
0.00025
0.0002
0.00015
0.0001
0.00005
0
0
200
400
600
800
n * A
time
Poly. (time)