Shannon Jeffers
CS325
HW3

1. A counter example for this greedy strategy would be easy to provide. Let us assume the following chart corresponds to the prices and “densities” or value per inch of lengths of rods from 1 to 4.

Length i	1	2	3	4
Price p_i	1	18	30	32
P_i/i	1	9	10	8

   As you can see from this chart, the rod with the biggest density would be the rod with a length of 3, which means the greedy strategy would cut a length of 3 out of the length of 4 rod. This would only leave a length 1 section. When adding the values together this would give us a total value of $31. This value is less than both the option of not cutting the rod at all and the option of cutting the rod into two length 2 sections. The optimal solution would be cutting the rod into 2 length 2 sections giving us a total value of $36.

2. There isn’t much that needs to be changed in order for this algorithm to reflect a cost per cut. The first change would be to include the cost per cut in the parameters passed to the function. This way if the cost per cut changes it is not hard coded and can be used flexibly (even if the cost per cut is 0). The other change would be to include the cost for cut in the equation in the inner for loop and to account for a case in which we make no cutes. To account for making no cuts we will stop our for loop one number early and set q to p[j] before we start looping. Let’s call the cost per cut “cost” and using the psudocode from the book it would look something like this:

```
Mod_Cut_Rod(prices, rodlen, cost)
    results = []
    results[0] = 0
    for j in range (1, rodlen)
        q = prices[j]
        for I in range (I, j-1)
            q = max(q, prices[i] + results[j-i] –c)
        results[j] = q
    return results[rodlen]
```

3. Product Sum
   a. 2+1+(3*5)+1+(4*2) = 27
   b. If j >= 2 OPT[j] = max(OPT[j-1] + v_j, OPT[j-2] + v_j * v_{j-1})
      If j = 1 OPT[j] = v_1
      Otherwise OTP[j] = 0
   c. PsudoCode used to analize run time:

```
productSum(arr, n)
    if n = 0 return 0;
    let OTP be a new array
    OTP[0] = 0
    OTP[1] = arr[0] (if the array is 0 indexed)
    for j in range (2, n)
```

```
            OTP[j] = max(OTP[j-1] + arr[j-1], OTP[j-2] + arr[j-1]*arr[j-2]
        return OTP
```

The asymptotic runtime for this formula would be O(n) (maybe even theta(n) because it definitely will go through the entire array) because the formula is placed within a for loop that checks every value of n exactly once.

4. Coin Change

a. The coin change algorithm works by using a nested for loop to take each denomination of coin and see if it can be used to create each number from 1 through the total number of cents you wish to make change for. The solution for how many coins it takes to generate a certain amount cents is stored for every amount of cents between 1 and the total. These stored solutions are then used to deduct if larger amounts of cents can be made with fewer coins. After every iteration is complete the smallest number of coins it takes to make the total amount of change will be stored in the slot for the total change. At the end of the algorithm going back through the index array will allow us to count the number of each coin was used to get the final solution.

```
makeChange(V, A)
    let numCoins[A+1] to be a new array
    let index[A+1] be a new array
    numCoins[0] = 0
    index[0] = 0
    for i in range (1, A+1)
        numCoins[i] = infinity
        Index[i] = -1
    for j in range (0, v.length)
        for i in range (1, A+1)
            if i >= V[j]
                if numCoins[i - V[j]] + 1 < numCoins[i]
                    numCoins[i] = numCoins[i - V[j]] + 1
                    index[i] = j
    let C[V.length] be a new array
    while A != 0
        add 1 to C[index[A +1]]
        A = A - V[index[A]]
```

b. There are a couple loops in this algorithm that will help us determine the theoretical run time. The first important loop runs from 0 to the length of V, or the number of total denominations we are using to make change with. This gives this a theoretical run time of O(V) The loop nested inside of that loop runs from 1 to the size of A covering every cent value in between. This loop hols a run time of O(A) Because these two loops are nested within each other their runtimes are compounded giving us an overall theoretical run time of O(VA).

6. Experimental running times

a. The first set of experimental running times I collected was playing with the values for A. I left V set at 500 and started A at 5000. I took seven data points ranging from A= 5000 to A= 100000. The next set of experimental data I did the exact opposite of the first set. I left the value of A set to 500 while I slowly increased the value of V from 5000-100000. The final set of experimental data I did was to increase the values of both A and V at the same time at the same rate. I started both A and V at 5000 and slowly increased them across 7 data points until I reached a value of 100,000 for each.

Values for A	Time

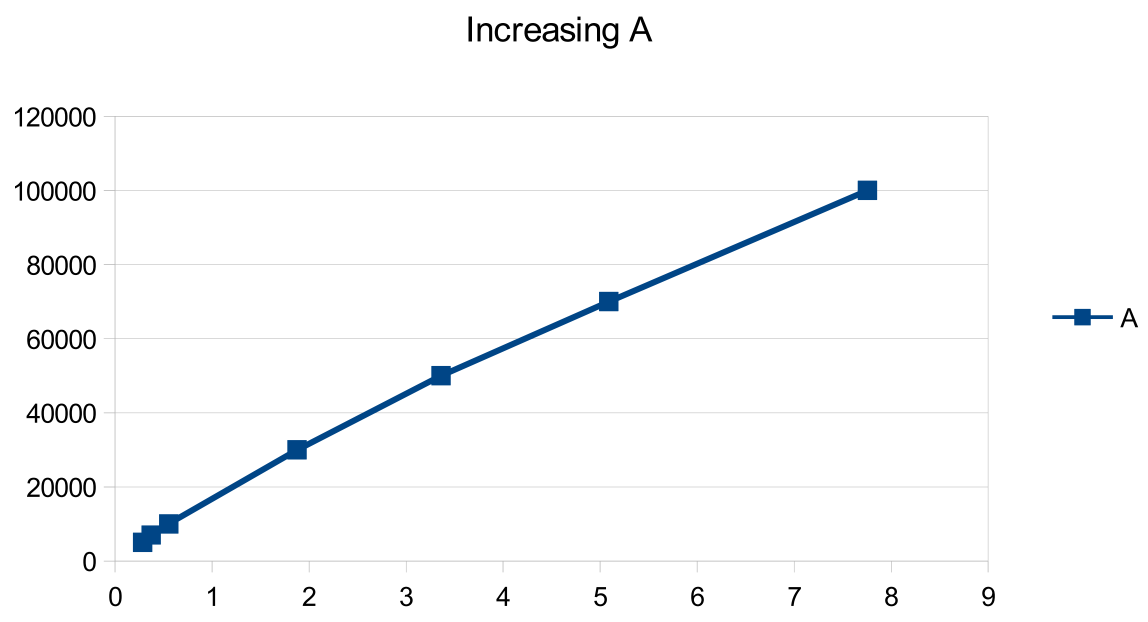
5000	0.283999919891
7000	0.371999979019
10000	0.552999973297
30000	1.87700009346
50000	3.35899996758
70000	5.08999991417
100000	7.75600004196

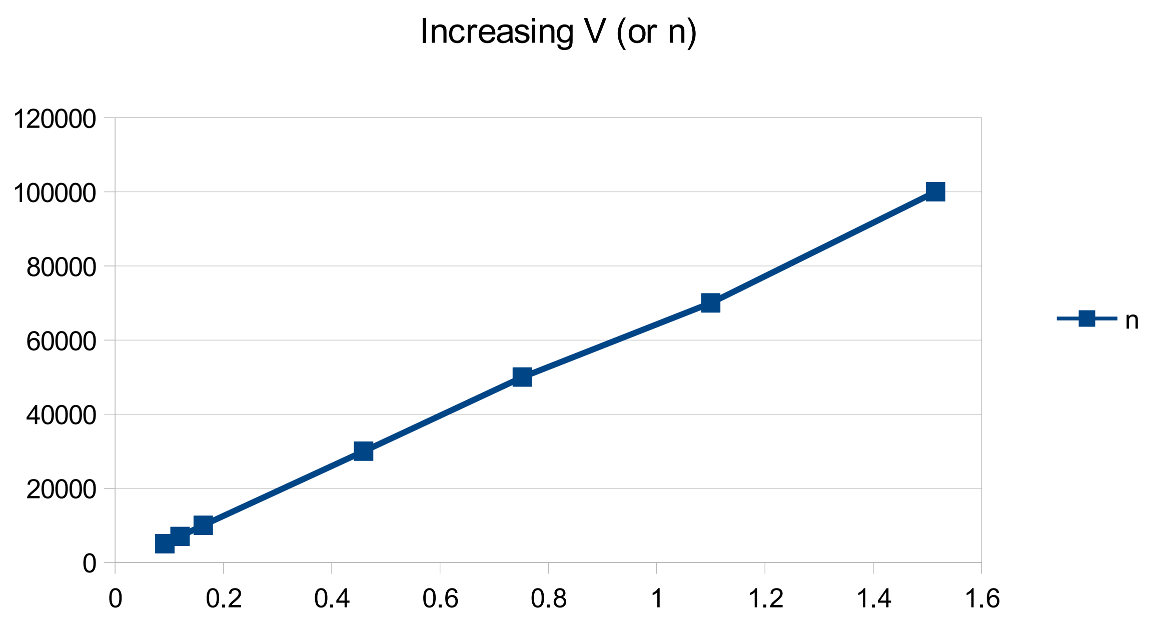
Values for V	time to run
5000	0.092000007629
7000	0.119999885559
10000	0.163000106812
30000	0.458999872208
50000	0.751999855042
70000	1.09999990463
100000	1.5150001049

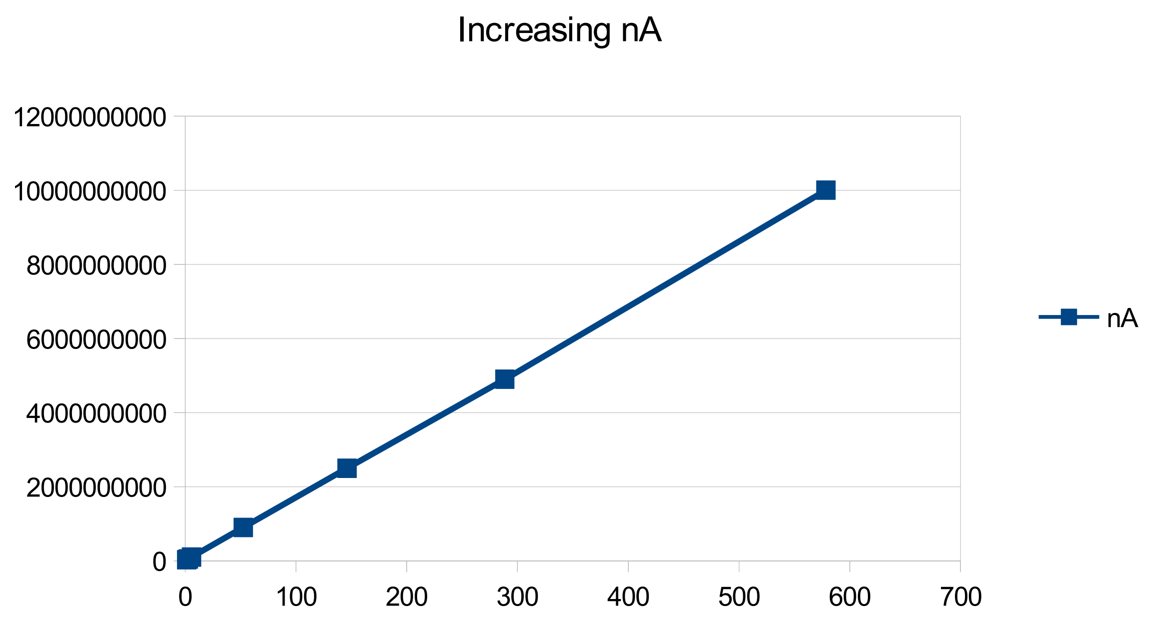
VxA	time to run
V = 5000 A = 5000	1.47900009155
V = 7000 A = 7000	2.88199996948
V = 10000 A = 10000	5.8630001545
V = 30000 A = 30000	52.5339999199
V = 50000 A = 50000	146.2490000072

V = 70000 A = 70000	288.639000177
V = 100000 A = 100000	578.4930000031

b.







At first I felt like all three graphs looked rather linear and I was a little bit confused by this. I was expecting the graph representing the running time of of A and the one representing the running time of n where I kept one rather low and increased the other value over a series of points to be linear because then the run time would basically be O(A) or O(n) for large enough values for A and n. The nA graph is the one that really through me off however. I expected this one to look much more polynomial because I kept the values for n and A the same and increased them at the same rate. I expected it to look much more like an n^2 type curve, but I realized the reason that It doesn't look much more n^2 is because I plotted n*A on one axis rather than just plotting x or A. Had I just plotted x or A it would look much more polynomial instead the run times are just reflecting "n". After discussing it in the homework group I decided to re-plug in the data into a better program at work(excel rather than open office) and do a fit for a polynomial curve on my data. I was very surprised to realize all three curves are very very close to an r^2 value of 1 when using the polynomial curve, which actually makes sense because the overall run time of this algorithm should be O(nA) or psudopolynomial, so in a way I actually got data that represents the theoretical run time. My values for A fit a polynomial code at an r^2 value of .999, n fit at .993 and nA actually fit exactly at 1. I meant to exchange the graphs on here for the ones at work because they looked better, but I forgot to send them home to myself, but these still do a decent job at showing the data.