CS 325 Homework 3, Lyell Read

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

> The counterexample that I'll provide is a rod of length 3. The values of each rod length from 1 to 3 are: {{1, 2}, {2, 2}, {3, 5.9}}. The algorithm will choose to cut a length 1 segment, as the density is 2, as opposed to densities 1 and <2 for lengths 2, 3 respectively.

Question 2: 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.

```
o = [INT_MIN for x in n]
n = length of rod
p = price per length
c = cost of cut

optimal(n, p, o, c):
     if n == 1:
          o[1] = p[1]
          return o[1]

     if o[n] == (INT_MIN):

          O = p[n]
          for k in range (1, n):
               R = (p[k] + optimal(n-k, p, o, c) - c)
               if R > O:
                    O = R
          o[n] = O
          return O

     else:
          return o[n]
```

Question 3

A. For each below, (a) will be the naive recursive, and (b) will be the dp recursive.

a. This algorithm starts with the last element in the list and will check to see if it is done with the list, and if it can fit the item in the bag. If it cannot, it will move on to the next item down the list (n-1). If it can fit the item, it will call both a recursive call to itself with and without having selected that item, and it will return the max wealth gained between those two.

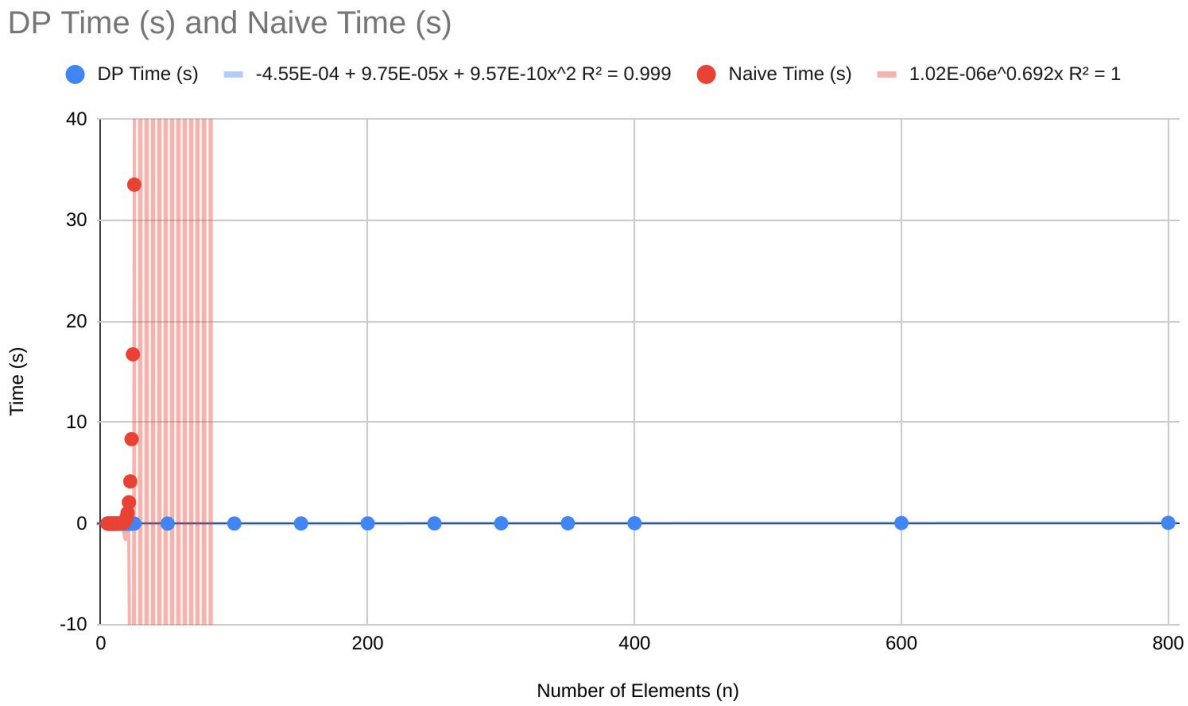
```
w = weight array
v = value array
n = number of elements
c = capacity

knapsack_naive(w, v, n, c):
     if c == 0 or n == 0:
          return 0
     if w[n] > c:
          return knapsack_naive(w, v, (n - 1), c)

     leave = knapsack_naive(w, v, (n - 1), c)
     take = knapsack_naive(w, v, (n - 1), (c - w[n])) +
v[n]

     return max(leave, take)
```

b. This algorithm improves on the last one as it also includes a 2D array (n by c), where it can store the result of certain computations, so they don't need to be redone. This algorithm starts with the last element in the list and first checks if it has a value in the 2D array that corresponds to that pair of (n,c). If so, it will return that immediately, saving time. Else, it will check to see if it is done with the list, and if it can fit the item in the bag. If it cannot, it will move on to the next item down the list (n-1). If it can fit the item, it will call both a recursive call to itself with and without having selected that item, and it will return the max wealth gained between those two.

```
w = weight array
v = value array
n = number of elements
c = capacity
o = blank 2d array, n X c

knapsack_dp(w, v, n, c, o):
     if o[n][c] != None:
```

```
            return o[n][c]

        if c == 0 or n == 0:
            return 0

        if w[n] > c:
            return knapsack_dp(w, v, (n - 1), c, o)

        leave = knapsack_dp(w, v, (n - 1), c, o)
        take = knapsack_dp(w, v, (n - 1), (c - w[n]), o) +
v[n]

        o[n][c] = max(leave, take)
        return max(leave, take)
```

B. Done



C.

	Run 1	Run 2	Run 3	Run 4	Run 5
DP	0.00005459785461	0.00005316734314	0.00005483627319	0.00005197525024	0.00005197525024
	0.00008273124695	0.00006890296936	0.00009751319885	0.00008749961853	0.00008749961853
	0.0001788139343	0.0001766681671	0.0001900196075	0.0001816749573	0.0001816749573
	0.0002906322479	0.0002474784851	0.0002956390381	0.0002727508545	0.0002727508545

	0.0003714561462	0.0003700256348	0.0003962516785	0.0003871917725	0.0003871917725
	0.0005190372467	0.0003805160522	0.0004444122314	0.0002279281616	0.0002279281616
	0.0006699562073	0.0007009506226	0.0006897449493	0.0006113052368	0.0006113052368
	0.0007884502411	0.0006730556488	0.000640630722	0.0007336139679	0.0007336139679
	0.0008840560913	0.0009074211121	0.0008113384247	0.0007836818695	0.0007836818695
	0.001032352448	0.0008778572083	0.00103020668	0.0008461475372	0.0008461475372
	0.001166820526	0.00106048584	0.001044988632	0.001044750214	0.001044750214
	0.001153707504	0.00110411644	0.001093387604	0.001095056534	0.001095056534
	0.001279592514	0.001342535019	0.00129365921	0.001342058182	0.001342058182
	0.001496315002	0.00130200386	0.001287221909	0.001316070557	0.001316070557
	0.001611471176	0.001507997513	0.001371383667	0.001334905624	0.001334905624
	0.001738786697	0.00179028511	0.001630783081	0.001615524292	0.001615524292
	0.001524686813	0.001531839371	0.001624584198	0.001700401306	0.001700401306
	0.001863479614	0.001711606979	0.001793146133	0.001721858978	0.001721858978
	0.001842260361	0.001837015152	0.001914978027	0.002008914948	0.002008914948
	0.001859426498	0.001903057098	0.002064704895	0.002058506012	0.002058506012
	0.002161502838	0.002154111862	0.002198457718	0.002046585083	0.002046585083
	0.004621505737	0.004779338837	0.004710197449	0.004212617874	0.00460934639
	0.009102582932	0.008702278137	0.009562015533	0.008768558502	0.009050607681
	0.01473593712	0.01283335686	0.01452302933	0.01590704918	0.01493310928
	0.01958250999	0.01851534843	0.01733756065	0.01803207397	0.01627731323
	0.02406692505	0.02324056625	0.02252101898	0.02459955215	0.02266526222
	0.02908372879	0.02886509895	0.03256082535	0.02624797821	0.02962303162
	0.03306031227	0.03421878815	0.03269004822	0.034512043	0.03148770332
	0.03822827339	0.0406472683	0.04029321671	0.03469705582	0.04060029984
	0.0612039566	0.06030797958	0.0579791069	0.05664038658	0.06371045113
	0.07380199432	0.07616281509	0.0744702816	0.08100342751	0.08180212975
Naive	0.00003457069397	0.00003361701965	0.00003457069397	0.00003337860107	0.00003337860107
	0.0000638961792	0.0000638961792	0.00006532669067	0.00006413459778	0.00006413459778
	0.0001282691956	0.0001356601715	0.0001292228699	0.0001266002655	0.0001266002655
	0.0002539157867	0.0002553462982	0.0002539157867	0.000251531601	0.000251531601
	0.0005037784576	0.0005316734314	0.0005114078522	0.0005006790161	0.0005006790161
	0.001055717468	0.001032829285	0.001029253006	0.001002311707	0.001002311707
	0.002079725266	0.002055644989	0.002045154572	0.002030849457	, 0.002030849457
	0.004116535187	0.004014253616	0.004125595093	0.004005670547	0.004005670547
	0.008206129074	0.008121013641	0.008246183395	0.008074045181	0.008074045181
	0.01639914513	0.01620984077	0.0166015625	0.01641106606	0.01641106606

	0.03353786469	0.03271818161	0.03348755836	0.03237986565	0.03237986565
	0.0664088726	0.06628489494	0.06620883942	0.06630253792	0.06630253792
	0.1351926327	0.1331875324	0.1340036392	0.1300783157	0.1300783157
	0.2669000626	0.2670173645	0.2680630684	0.2608463764	0.2608463764
	0.5275964737	0.5367166996	0.5313565731	0.535492897	0.535492897
	1.041431189	1.075752258	1.073551655	1.05612874	1.05612874
	2.102184296	2.133134365	2.11802721	2.104984522	2.104984522
	4.190527201	4.208354235	4.150338888	4.15451932	4.15451932
	8.352419376	8.357708454	8.306628227	8.379060745	8.379060745
	16.70102835	16.71485972	16.72711873	16.77794456	16.77794456
	33.35538769	33.33099818	33.75194144	33.58626008	33.58626008

D. My implementation is in python3. When run it properly demonstrates the advantages of dynamic programming. I collected data by printing out the results in terms of time for each run. As W gets smaller, the algorithm runs faster, as it has fewer items total that can be placed in the bag. As W gets larger, (to a point), the execution gets slower. Once W > sum(w(i)), then there is no slow down for making W larger, as all items will be included. w(i)) is the weight of each item.
NOTE: The trendline for the Naive algorithm was rendered wrong, but the provided equation is correct - this is a bug in Google Sheets as far as I can tell. Sorry!

Question 4

A. I reuse the 0/1 knapsack algorithm for each family member.

B. Given that this function is run at most $N*M_i*2$ times, where N is the number of items, and M_i is the carrying capacity, per family member. Therefore, this algorithm runs in Theta($N*M_i*F$). This assumes that the time to calculate for M_{small} versus M_{max} is less than $N*M_i*F$, which is true in my case. Pseudocode:

```
w = weight array
v = value array
n = number of elements
c = capacity
o = blank 2d array, n X c

     knapsack_dp(w, v, n, c, o):
          if o[n][c] != None:
               return o[n][c]

          if c == 0 or n == 0:
               return 0

          if w[n] > c:
               return knapsack_dp(w, v, (n - 1), c, o)
```

```
        leave = knapsack_dp(w, v, (n - 1), c, o)
        take = knapsack_dp(w, v, (n - 1), (c - w[n]), o) +
v[n]

        o[n][c] = max(leave, take)
        return max(leave, take)
```

C. Done.