Brendan Jang
CS 325
Homework 3

1. 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.

   See below for a counterexample for the "greedy" strategy:

i	1	2	3	4
p_i	1	20	33	36
p_i / i	1	10	11	9

   Let the given rod length be 4. Following a "greedy" strategy, the first cut would be a rod of length 3 for price of 33. This leaves us with rod length 1 of price 1. The total price for the rod would be 34. The most optimal way would be to cut it into two rods of length 2 each fetching us 40.

2. 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.

   See modified algorithm psuedocode below:

```
modifiedRodCut(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 – 1] – c)
                r[j] = q
        return r[n]
```

The inner for loop was modified to show the fixed cost of making the cut which is deducted from the revenue. We also have to account for the case that may arise where no cuts are made. In this case, the total revenue would be p[j]. Therefore, the inner loop has been designed to loop to run from i to j – 1. The line where we declared q = p[j] handles the case where there are no cuts.

3. A. The program can be designed with two arrays. We use the first array for total coins and the second as a tracker array. The function will go coin by coin and iterate through the first array to

see if a coin can make the total. This first array would go from [0… total change requested]. If the selected coin can make the total, the second array will save that coin's position. If there is a coin that can make the total in a smaller count, then the second array will hold the position of that coin. The goal is to take the last element in the first array subtract that with the coin's location from the second array. This will be the min amount of coins we need to make the total.

Psuedocode:

```
minCoins(total, coins[])
        totalArr of (total + 1 ) = ∞
        trackerArr of (total + 1 ) = -1
for i = 0 to n
        for j = 1 to coins.length
        if ( j > = coins of i)
                if (totalArr of (j – coins of i) + 1 < totalArr of j)
                        totalArr of j = 1 + totalArr of (j – coins of i)
                        trackerArr of j = j
return totalArr of total
```

B. $T\llcorner i \lrcorner = \{\min \{ T\llcorner i \lrcorner, T(i – coins[j] + 1)$

Therefore, the theoretical running time for the algorithm using bottom-up is $\Theta(n)$.

4. A. This algorithm would be designed as follows. First, we would read the number of test cases (T) from the input file. This would tell us to run the loop T times. In each iteration of the loop, we would read number of items (N). Each loop would run and save the prices and weights of N items into two separate arrays. Then we would read the number of family members (F). For each family members, we would calculate the max price of items that can be carried by that family member. We would also keep track of the total max prices. Finally, we would write the maximum total price to output file.

Pseudocode would be as follows:

```
read T from input file
// process T number of cases
for k = 0 to T - 1
    // read the number of items from input file
    read N from input file
    // read price and weight of each item into respective arrays
    for i = 0 to N - 1
        read P[i] from input file
        read W[i] from input file
    maxprice = 0
    // read the number of family members
    read F
```

```
// find the max price of items that can be carried by each family member and keep track of the
// total of the max prices
for j = 0 to F - 1
    read M from input file
    create a table K of size(N + 1) * (M + 1)
    // build table
    for i = 0 to N
        for w = 0 to M
            if (i == 0 or w == 0)
                K[i][w] = 0
            else if W[i - 1] <= w
                K[i][w] = max{P[i -1] + K[i - 1] [w - W[i -1]], K[i - 1][w])
            else
                K[i][w] = K[i - 1][w]
    // K[N][M] represents the max price of items that can be carried by the family member so
    // add it to max price
    maxprice = maxprice + k[N][M]
// write the max total to output file
write maxprice to outputfile
```

B. Running time of a knapsack using dynamic programming takes a running time of Θ(NM). N is the number of items and M is the max weight that can be carried by each family member. If there are F number of family members and the ith member can carry M_i pounds:

   Θ(NM_1 + NM_2 + … + NM_F) = Θ(N * $\sum_{i=1}^{F} M_i$)

C. Implementation in shopping.cpp