Michael Nutt
CS325.401
HW 3

1.

Length i	1	2	3	4
Price p_i	1	20	33	36

Let the rod be of length 4. According to the greedy algorithm, we must first cut of a piece of length 3. This leaves us with two rods, one of length 3 and one of length 1. Their combined value equates to 34 (33 + 1). However, a more optimal solution would be to cut the rod in half, giving us two rods of length 2 priced at 20 per rod for a total price of 40.

2. Based on the original algorithm for Cut-Rod, we must modify how *q*, the maximum value of the rod(s), is calculated to include the fixed cost of cutting, *c*. We must also account for the case in which no cutting is done (*i = j*) since no cutting cost will be incurred. Thus we have:

```
Cut-Rod(p, n, c)
    Let r[0…n] be a new array
    r[0] = 0
    for j = 1 to n
        q = -∞
        for i = 1 to j – 1
            q = max(q, p[i] + r[j – i] – c)
        r[j] = q
    return r[n]
```

3.

a. Let C(j) define the minimum number of coins required to make j cents. We can assume that if j = 0, then C[j] is also 0 as it takes 0 coins to make 0 cents. Also, for any value of j >= 1, given a coin of denomination d_i, we can determine that C[j] = 1 + (C[j] – d_i). That is, 1 + the number of coins required to make change for C[j] – d_i. Thus we have:
Let d be an array of denominations, k is the number of denominations, and n is the amount of change to be made.

```
make_change(d, k, n)
    Let C[0] = 0
    for j = 1 to n
        C[j] = ∞
        for i = 1 to k
            if j >= d_i && 1 + C[j – d_i] < C[j]
                C[j] = 1 + C[j – d_i]
    return c
```

b. Since we will examine each denomination, for each value [1…n], the run time is O(n*k) where k represents the number od denominations.

Michael Nutt
CS325.401
HW 3

4.

a. Determine all possible subsets of items and calculate their corresponding weights and values considering only those subsets with weights <= M. From all valid subsets, return the subset with the highest dollar value.

   Let W be an array of weights corresponding to each item *i*. Let P[] be an array of prices corresponding to each item *i.* Let N be the number of available items. Let M be the total weight that can be carried by a member of the family.

```
shopping(W, P, N, M)
    let K[0…N+1][0…M+1] be a new array
    for i = 0 through N
        for j = o through M
            if i = 0 or w = 0
                K[i][j] = 0
        Else if W[i – 1] <= j
            K[i][j] = max(P[i – 1] + K [i – 1][j – W[i – 1]], K[i – 1][w]
        Else
            K[i][w] = K[i – 1][w]
    return K[N][M]
```

b. The run time of this algorithm is O(n*W) where n is the number of items and W is the total weight that can be carried.