Neha Maddali

Problem 1:

```
rod-cut-modification(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-i] - c)
        r[j] = q
    return r[n]
```

The change that was made was q = max(q, p[i] + r[j-i] - c) which reflects the fixed cost of making the cut. This is deducted from the revenue. When there are no cuts, the total revenue is p[j] so then the inner for loop will run from i to j-1 instead of to j. Without this modification, in the case of no cuts, there would be a deduction of c from the total revenue.

Problem 2:

For an input string $s = s_1, s_2, \ldots s_n$, let f(i,j) be the length of the longest palindrome subsequence on the substring $s_i s_{i+1} \ldots s_j$. It would have such a recurrence

$$f(i,j) = \begin{cases} 0 & \text{if } j-i < 0 \\ 1 & \text{if } j-i = 0 \\ \max(f(i+1,j), f(i, j-1)) & \text{if } j-i > 0 \text{ and } s_i \neq s_j \\ 2 + f(i+1, j-1) & \text{if } j-i > 0 \text{ and } s_i = s_j \end{cases}$$

```
longest-palindrome-subsequence(s)
    let n = |s|
    let array dp[n][n] = 0
    for i =1 to n
        dp[i][i] = 1
    for l =2 to n
        for i=1 to n-l+1
            j = i + l -1
            if s_i ≠ s_j then
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])
            else
                dp[i][j] = 2 + dp[i+1][j-1]
    return dp[1][n]
```

The base cases correspond to the empty string and a single character. If the string is at least of length 2 then check if the first and last characters are equal. If true, take both characters to be part of the palindrome and recurse on the inner substring. Otherwise, take the max of the two subproblems that ignore the first and last character. So the length of the longest palindrome subsequence of s is f(1,n). The running time of this algorithm is O(n^2)

Problem 3:

A recursive definition of P[i] which is the max expected profit at location i, based on the constraints

$$P[i] = \max\{ \max_{j<i} \{P|j| + \alpha(m_i, m_j) * p_i\}$$
$$P_i$$

And the function α is:

$$\alpha(m_i, m_j) = \{ 0 \text{ if } m_i - m_j < k$$
$$1 \text{ if } m_i - m_j \geq k$$

```
profit-expected(N, P)
    //input: N locations, P[1…N] where P[i] denotes profit at location i
    //output: max expected profit P_max
    Profit[i]
    for i = 1 to N
        Profit[i] = 0
    for i =2 to N
        for j=1 to i-1
            temp = Profit[j] + α(m_i, m_j) * P[i]
            if temp > Profit[i]
                temp = Profit[i]
        if Profit[i] < P[i]
            Profit[i] = P[i]
```

The maximum expected profit of location i comes from the maximum of expected profits of location j and whether we can open a location i. The profit from opening a restaurant at location i is p_i. The cases when there is no restaurant at location j will be considered by case k where k < j. It's likely that profit at location i, p_i is higher than the $P[j] + \alpha(m_i, m_j) * p_i$. In those cases, we need to do a comparison with p_i as well. The running time of this algorithm is O(n^2).

Problem 4:

a. Consider the sequence 2, 100, 1, 1. If the first player is greedy and takes the first card with the value of 2 then the second player can take the card with the value of 100 and win. The optimal strategy for the first player is to take the last card with the value of 1. Then the second player will take either the 2 or the other 1 and the first player can take the 100 value card.

b. Let opt(i,j) be the difference between:
   - The largest total the first player can obtain
   - Ant the corresponding score of the second player when playing on sequence $s_i…s_j$

   At any stage of the game, there are 2 possible moves for the first player:
   - Either choose the first card, in which case he will gain v_i and score -opt(i+1, j) in the rest of the game
   - Or the last card, gaining v_j and -opt(i, j-1) from the remaining cards

   We are interested in the largest total the first player can gain over the second, so take the max of these 2 values (for i < j):

   $$opt(i,j) = \max\{v_i - opt(i+1, j), v_j - opt(i, j-1)\}$$

The base cases are: opt(i,i) = v_i
for j=1…n
    for i = j…1
        opt[i,j] = max(v[i] - opt(i+1,j), v[j] - opt(i,j-1))
return opt[1,n]
The run time of this algorithm is O(n^2)

Problem 5:
First sort the set of n points {x_1, x_2…, x_n} to get the set Y = {y_1, y_2…, y_n} such that $y_1 \leq y_2 \ldots \leq y_n$. Then do a linear scan on {y_1, y_2…, y_n} started from y_1. Every time while encountering y_i, for some $i \in \{1 \ldots n\}$, we put the closed interval [y_i, y_i+1] in our optimal solution set S, and remove all the points in Y covered by [y_i, y_i+1]. Repeat the above procedure, finally, output S while Y becomes empty. Next show that S is an optimal solution.

Make the claim that there is an optimal solution that contains the unit length interval [y_i, y_i+1]. Suppose that there exists an optimal solution S* such that y_1 is covered by [x', x'+1] $\in$ S* where x' < 1. Since y_1 is the leftmost element in the given set, there is no other point in [x',y_1). So if we replace [x', x'+1] in S* by [y_i, y_i+1], we will get another optimal solution that explains the greedy choice property.

Thus, by solving the remaining subproblem after removing all the points in [y_i, y_i+1] to find an optimal set of intervals, denoted as S', which cover the points to the right of y_1+1, we will get an optimal solution to the original problem by taking the union of [y_1, y_1+1] and S'. So the running time of the algorithm is O(nlogn).