HW5

Problem 1

Kleinberg and Tardos, Chapter 6, Exercise 5.

Solution: Let's denote Y(I, k) as the substring that goes from y[i] to y[k]. Let Opt(k) represent the maximum quality of an optimal segmentation of the substring Y(1, k). The maximum quality of an optimal segmentation for this substring Y(1, k) can be determined by adding the quality of the last word in the segmentation (for example, the substring y[i] . . . y[k]) to the quality of an optimal segmentation of the substring Y(1, i). If there was a better solution to Y(1, i) , we could use it to improve the maximum quality for Opt(k), which would lead to a contradiction.

Therefore, we can express Opt(k) as the maximum value of Opt(i) + quality(Y(I+1, k)) for all i where 0<i<k.

The starting condition for our problem is Opt(0) = 0, and we can compute Opt(k) incrementally for each k from 1 to n. As we do this, we'll keep track of where we make the segmentation at each step.

The optimal segmentation will correspond to Opt(n). We can find this segmentation in Θ(n^2) time complexity. This represents an efficient solution to the problem.

Problem 2

Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are equivalent if they correspond to the same account.

It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than n/2 of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only O(n log n) invocations of the equivalence tester.

Solution: Let's denote OPT(k,w) as the maximum value attainable by using 'k' types of items ($1 \leq k \leq n$), and a knapsack of capacity 'w' ($0 \leq w \leq W$). Given that we have an infinite number of items for each type, we have two options to consider:

1. We can include another item of type 'k' and then resolve the sub-problem OPT(k, w - wk), where wk is the weight of the 'k' type item.
2. We don't include any item of type 'k', move on to the next type of item, and solve the sub-problem OPT(k - 1, w).

Hence, we can express OPT(k, w) as the maximum between OPT(k - 1, w) and OPT(k, w - wk) + vk, where vk is the value of the 'k' type item.

The base case for this problem is OPT(0, 0) = 0, which represents an empty knapsack with no items to choose from.

Regarding the time complexity of this problem: Since we are computing OPT for every combination of 'k' (from 1 to n) and 'w' (from 0 to W), the time complexity would be O(n*W). This time complexity assumes that each individual computation of OPT(k,w) takes constant time. In this problem, the value of OPT(k,w) is computed using only previously computed values of OPT, and each computation involves a constant amount of work (two comparisons and possibly an addition). Hence, the total time complexity is O(nW), representing a polynomial time solution when 'W' is considered as part of the input size.

Problem 3

Tommy and Bruiny are playing a turn-based game together. This game involves N marbles placed in a row. The marbles are numbered 1 to N from the left to the right. Marble i has a positive value m_i. On each player's turn, they can remove either the leftmost marble or the rightmost marble from the row and receive points equal to the sum of the remaining marbles' values in the row. The winner is the one with the higher score when there are no marbles left to remove.

Tommy always goes first in this game. Both players wish to maximize their score by the end of the game.

Assuming that both players play optimally, devise a Dynamic Program- ming algorithm to return the difference in Tommy and Bruiny's score once the game has been played for any given input.

Your algorithm must run in $O(N^2)$ time.

Solution: First, we compute a prefix sum for the array of marbles. This allows us to find the sum of a continuous range of values in constant time. For instance, if we have an array like [5,3,1,4,2], then our prefix sum array would become [0, 5, 8, 9, 13, 15].

Afterwards, we define OPT(i, j) as the greatest possible score difference for the current player, considering that the marbles from index i to j (inclusive) are still available.

Here is the pseudo-code for this algorithm, with arrays being 0-indexed:

Algorithm: Max-Difference-Scores(marbles)
- First, let n be the size of the marbles array.
- Calculate the prefix sum array for the marbles array. This step will take O(n) time.
- Next, create a new n by n array, OPT, with all values initialized to 0.
- Start a loop from i = n - 2 to 0. Within this loop:
  - Start another loop from j = i + 1 to n - 1. Within this inner loop:
    - Calculate the score if you take the i-th marble: prefix_sum[j+1] - prefix_sum[i+1] - OPT[i+1][j]
    - Calculate the score if you take the j-th marble: prefix_sum[j] - prefix_sum[i] - OPT[i][j-1]
    - OPT[i][j] is then the maximum of these two scores.
- Finally, return OPT[0][n-1].

This algorithm has a time complexity of O(n^2) because it needs to compute n*n subproblems. This time complexity already accounts for the initial calculation of the prefix sum array.

Problem 4
Kleinberg and Tardos, Chapter 6, Exercise 6.

Solution:
This problem is a classic example of dynamic programming. We define our subproblem in terms of the minimum total slack when considering the first `i` words.

Let's define `Opt(i)` to be the minimum sum of the squares of the slacks for the first `i` words in a valid arrangement. The base case `Opt(0)` is `0`, because no words means no slack.

Now, consider adding the next word to the arrangement. It could be on a new line by itself, or it could be on the same line as the previous `j` words, for any `j` such that the total length of words `i-j+1` through `i` (including spaces) does not exceed `L`. This gives us the following recurrence:

Opt(i) = min { Opt(i-j-1) + slack(i-j+1 to i)^2 } for all valid j

where `slack(i-j+1 to i)` represents the remaining spaces on the line after placing words `i-j+1` through `i`.

After computing the `Opt(i)` for all `i`, `Opt(n)` is the solution to our problem. To retrieve the actual arrangement of words, we would also need to store, for each `i`, the value of `j`

that gave the minimum in the recurrence. We would then trace backwards from `Opt(n)` to get the arrangement.

The time complexity of this algorithm is `O(n^2)`, as for each `i` from `1` to `n`, we potentially need to consider `j` from `0` to `i-1`. The space complexity is `O(n)` for storing the `Opt(i)` values and the chosen `j` values.

Problem 5

You have n+1 rooms. There is a heater in Room 0. It can make Room 0 to Room r warm. (1 ≤ r ≤ n) Now you are a robot with k thermometers. Ev- ery time you bring one thermometer into a room. If the room is warm, the thermometer will work well and can be used again, and if the room is cold, the thermometer will be broken and can not be used again. Design a dynamic program algorithm to find the minimum count of entering a room that you need to determine the exact value of r.

a. Write down the recursive formula, and the meaning of each part. b. What is the time complexity of the algorithm?

Solution:

1. dp(k,n) is the optimal count when there are k thermometers and n floors. dp(k, n) = 1 + $\min_{1\le x\le n}(\max(dp(k-1, x-1), dp(k, n-x)))$
   dp(k-1, x-1) is the state when the thermometer is broken.
   dp(k, n-x) is the state when the thermometer works well. x denotes the Room x.

2. $O(kn^2)$, or O(kn log n) [if use binary search]

Problem 6

The Trojan Band consisting of n band members hurries to lined up in a straight line to start a march. But since band members are not positioned by height the line is looking very messy. The band leader wants to pull out the minimum number of band members that will cause the line to be in a formation (the remaining band members will stay in the line in the same order as they were before). The formation refers to an ordering of band members such that their heightssatisfy$r_1 < r_2 < ... < r_i > ... > r_n$,where$1 \le i \le n$.

For example, if the heights (in inches) are given as R = (67, 65, 72, 75, 73, 70, 70, 68)

the minimum number of band members to pull out to make a formation will be 2, resulting in the following formation:

(67, 72, 75, 73, 70, 68)

Give an algorithm to find the minimum number of band members to pull out of the line.

Note: you do not need to find the actual formation. You only need to find the minimum number of band members to pull out of the line, but you need to find this minimum number in $O(n^2)$ time.

For this question, you must write your algorithm using pseudo-code.

Solution:
This problem is solved by performing the Longest-Increasing-Subsequence operation twice, once from left to right and once from right to left. After calculating the longest subsequences, we iterate over the array to find the minimum number of band members that need to be removed to make the height order valid.

Define OPTleft(i) as the maximum length of the line to the left of band member i (including i) that can be arranged in increasing height order by removing some members. Similarly, we can compute OPTright(i) by flipping the array and finding the same values from the other direction.

The recurrence relations are:

- OPTleft(i) = max(OPTleft(i), OPTleft(j) + 1) for all $1 \leq j < i$ where ri > rj
- OPTright(i) is calculated similarly after flipping the array

Here is the pseudo-code for the problem:

- *Initialize OPTleft(i) and OPTright(i) to 1 for all band members*
- *For each band member i from 2 to n - 1:*
  - For each band member j from 1 to i - 1:
  - If ri > rj, update OPTleft(i) = max(OPTleft(i), OPTleft(j) + 1)
- *Perform similar calculations for OPTright(i), but with the array flipped*
- *Initialize result as negative infinity*
- *For each band member i from 1 to n:*
- *Update result = min(result, n - (OPTleft(i) + OPTright(i) - 1))*
- *Return the result*

The time complexity of this algorithm is dominated by the two nested for loops, each taking O(n^2) time to calculate the Longest Increasing Subsequence in both directions. Hence, the total time complexity of the solution is O(n^2).