Krish Patel
Homework 6
CS 180 Disc 1D

Question 1)
For this question, we are going to use the following algorithm:

- We start with three strings s,x and y
- Initialize a dp table where dp[i][j] will be true if the first *i* characters of *s* can be formed by interleaving the first *i*+*j* characters of *x* and *y*. Initialize dp[0][0] to true as you can form an empty string by interweaving empty strings
- Fill out the first row of dp by checking if first I characters of s match I characters of x.
- Similarly fill out the first column by doing the same with y.
- For each other remaining cell dp[i][j], check if *dp*[*i*−1][*j*] is true and *s*[*i*+*j*−1] matches *x*[*i*−1], or if *dp*[*i*][*j*−1] is true and *s*[*i*+*j*−1] matches *y*[*j*−1].
- The value of dp[len(x)][len(y] would indicate if it is a proper interleaving

This would have a time complexity of O(n*m) which is the time complexity of a dynamic prrgramming problem, where m and n are number of rows respectively. N is length of s and m is length of x or y(whichever is longer)

Proof of correctness:
Base case initialized to true, thus indicating that empty string s and can be formed by interleaving two empty strings x and y.
The algorithm fills out the DP table, ensuring that dp[i][j] is true only if first I characters of s a can be formed by interleaving the first i+j characters of x and y.
The algorithm considers two cases for each cell dp[i][j]. If it is true true and s[i+j-1] matches x[i-1], it sets dp[i][j] to true. This is because if the first i-1 characters of s can be formed by interleaving the first i-1+j characters of x and y, and s[i+j-1] matches x[i-1], then the first i characters of s can be formed.
Similarly, if dp[i][j-1] is true and s[i+j-1] matches y[j-1], it also sets dp[i][j] to true. This is because if the first i characters of s can be formed by interleaving the first i+j-1 characters of x and y, and s[i+j-1] matches y[j-1], then the first i characters of s can be formed.
Finally, the algorithm checks dp[len(x)][len(y)] to determine if it's possible to form s by interleaving x and y. If it's true, then the algorithm returns true; otherwise, it returns false.

Question 2)

For this question, we would use the following algorithm:

- Define Opt(i, s) to represent the optimal cost to reach node s using exactly i edges. Define N(i, s) to represent the number of shortest paths to node s using exactly i edges.
- Initialize Opt(i, v) = infinity for all nodes v except the source node, where Opt(0, v) = 0. Initialize N(i, v) = 0 for all nodes v except the source node, where N(0,v) = 1.
- For each possible number of edges i from 1 to the maximum number of edges:
    - For each node s in the graph, we set Opt(i, s) to the minimum of Opt(i - 1, t) + c_ts for all predecessor nodes t where there is an edge from t to s:
- After computing Opt(i, s) for all nodes: For each node s in the graph, we set N(i, s) to the sum of N(i - 1, t) for all predecessor nodes t where Opt(i, s) = Opt(i - 1, t) + c_ts. Then for the destination node w, find Opt(w) which is the minimum of Opt(i, w) over all possible i.
- we calculate N(w) which is the sum of N(i, w) for all i where Opt(i, w) = Opt(w).
- Return Opt(w) as the optimal cost to reach node w. Return N(w) as the number of optimal shortest paths to node w.

The algorithm iterates through all possible numbers of edges from 1 to the maximum number of edges, which is at most V - 1. For each number of edges, it iterates through all nodes (V) to compute Opt(i, s) and N(i, s). For each node, it considers all predecessor nodes, which can be at most E. Therefore, the time complexity of the algorithm is O(V * E * (V - 1)) = O(V^2 * E)
The algorithm has a time complexity of O(V^2 * E) and a space complexity of O(V^2). This complexity is polynomial in the number of nodes and edges, making it efficient.

As we consider edge counts from 1 to the maximum, Opt(i, s) gets updated to reflect the minimum cost of reaching node s with exactly i edges. This is accomplished by evaluating all predecessor nodes t connected to s and selecting the minimum cost. This step accurately represents the optimal cost of reaching node s with i edges.
After computing Opt(i, s) for all nodes, N(i, s) is updated to count the number of shortest paths to node s with i edges. This is achieved by summing N(i - 1, t) for all predecessor nodes t where Opt(i, s) matches Opt(i - 1, t) plus the cost c_ts. This step correctly tallies the number of optimal shortest paths to node s.
Finally, Opt(w) is determined by finding the minimum Opt(i, w) over all i, signifying the optimal cost of reaching the destination node w. N(w) is calculated as the sum of N(i, w) for all i where Opt(i, w) equals Opt(w), representing the count of optimal shortest paths to node w.

Question 3:
GerryMandering problem:

- We start with an array precincts representing the number of A-votes for each precinct and two integers A and B representing the minimum number of A-votes needed in district 1 and district 2 respectively.
- Initialize a 4-dimensional dp table M[j][p][x][y] where M[j][p][x][y] will be true if it is possible to assign p precincts to district 1 with at least x A-votes and the rest to district 2

with at least y A-votes, after considering the first j precincts. Initialize M[0][0][0][0] to true as the base case.

- Fill out the first layer of dp for j = 1 by considering the first precinct and updating M[1][p][x][y] based on whether the precinct is assigned to district 1 or district 2, and if the A-vote requirements x and y can still be met.
- Similarly, fill out the subsequent layers for j = 2, 3, ..., n, where n is the total number of precincts.
- For each cell M[j][p][x][y] in the dp table, check if either M[j-1][p-1][x-precincts[j]][y] is true (assigning the j-th precinct to district 1) or M[j-1][p][x][y-precincts[j]] is true (assigning the j-th precinct to district 2), where precincts[j] is the number of A-votes in precinct j.
- The value of M[n][n/2][A][B] will indicate if it is possible to achieve the gerrymandering goal with at least A A-votes in district 1 and B A-votes in district 2.

The time complexity of the above algorithm would be O(n^2 * m^2). This is because we use a 4 dimensional dynamic programming table to store these values.

Proof of correctness: **Base Case (j = 0):** In the base case, we have no precincts to consider. The only valid configuration is when p = 0, x = 0, and y = 0, which corresponds to no precincts assigned to any district. This configuration is correctly initialized in the dp table, so P(0, 0, 0, 0) is true.
**Inductive Hypothesis:** Assume that P(j', p', x', y') is true for all j' ≤ j, where j is some arbitrary but fixed positive integer.
To update P(j + 1, p, x, y), we consider two possibilities for the (j + 1)-th precinct:

Precinct j+1 is assigned to district 1: In this case, we check if P(j, p - 1, x - precincts[j+1], y) is true. This means that we have assigned one more precinct to district 1, with reduced p, x, and y values, and the remaining precincts can still satisfy the requirements. By the inductive hypothesis, if P(j, p - 1, x - precincts[j+1], y) is true, then P(j + 1, p, x, y) will also be true.
Precinct j+1 is assigned to district 2: In this case, we check if P(j, p, x, y - precincts[j+1]) is true. This means that we have assigned one more precinct to district 2, with reduced y value, and the remaining precincts can still satisfy the requirements. By the inductive hypothesis, if P(j, p, x, y - precincts[j+1]) is true, then P(j + 1, p, x, y) will also be true.
Since we consider all possible assignments of precinct j+1 (either to district 1 or district 2) and check if any of them result in P(j + 1, p, x, y) being true, the algorithm correctly computes the value of M[j+1][p][x][y]

Question 4:
We can solve this problem using the network flow problem. The steps for the algorithm are as follows:

- For each of the clients, we create a node v(i) and for each base station, we create a node w(j).

- For every client, if they are within range parameter r of a base station j, create an edge between the v(i) and w(j) with a capacity of 1.
- We then create a super source node S which we would use as our origin node. This would be connected to each client node v(i) with a capacity of 1. We also create a super sink node which is connected to all the stations. Let each of these edges have a capacity of L, and thus when more than L clients are connected, it no longer can be used for further connections.
- We then run the a network flow algorithm for finding the maximum capacity(Ford Fulkerson) to find the maximum flow from the super source node to the super sink node. We store the maximum flow in a variable max_flow

After running Ford Fulkerson's, we then check this value max_flow with the value n. If it is equal, then there is a way to connect all the clients to the stations. Else if the value is less than n, that means there isn't a way to connect all the stations. This is because max flow calculated the maximum flow we could carry through the network, and this would automatically account for the best connection possible. A capacity of L to the supersink means that when L clients are connected, there can't be any more connections made from the source node to the supersink using that particular station.

Time complexity: For creating nodes and edges:  O(n+k), for building the graph: O(n*k), and Fork Fulkerson has a time complexity of O(E*n), where E is total number of edges which is a maximum of (n+k+n*k).Thus time complexity is O(n^2 * k)

Proof of correctness: The algorithm above makes sure clients are connected to the right base stations while keeping in mind how far they can be from each other and how many clients each station can handle. This then uses the correctness of the max flow algo to optimize assignments.
First, we create nodes for each client and base station and connect them with edges. These connections represent valid links between clients and stations within a certain distance, and each connection can handle just one client. We also add special super source and sink nodes to account for the problem's rules. Then, we use Ford-Fulkerson to figure out the best way to make these connections while following the rules.
The algorithm is correct because it does a good job of making sure each client goes to just one base station and doesn't overload any station beyond its capacity. It also tries to connect as many clients as possible while keeping everything within the limits set by the rules. Thus this a reliable, correct, and efficient algorithm.

Question 5)

We can solve this problem using the network flow problem. The steps for the algorithm are as follows:

- For each of the patients, we create a node v(i) and for each base hospital, we create a node w(j).
- For every patient, if they are within range parameter of half an hour of a hospital j, create an edge between the v(i) and w(j) with a capacity of 1.
- We then create a super source node S which we would use as our origin node. This would be connected to each patient node v(i) with a capacity of 1. We also create a super sink node which is connected to all the hospitals. Let each of these edges have a capacity of n/k, and thus when more than n/k patients are assigned to a hospital, it no longer can be used for more patients.
- We then run the a network flow algorithm for finding the maximum capacity(Ford Fulkerson) to find the maximum flow from the super source node to the super sink node. We store the maximum flow in a variable max_flow.

After running Ford Fulkerson's, we then check this value max_flow with the value n. If it is equal, then there is a way to assign all the patients to the hospitals. Else if the value is less than n, that means there isn't a way to assign all the hospitals. This is because max flow calculated the maximum flow we could carry through the network, and this would automatically account for the best assignment possible. A capacity of L to the supersink means that when L patients are assigned there can't be any more assignments made from the source node to the supersink to that particular hospital.

Proof of correctness: The Ford-Fulkerson algorithm correctly calculates the maximum flow in the network, which represents the maximum number of valid patient-hospital assignments that can be made within the given constraints.The algorithm ensures that no hospital exceeds its capacity (**n/k** patients) since the flow cannot exceed the capacity of edges connected to hospitals.If max_flow equals n, it demonstrates that it is possible to assign all n patients to hospitals within the constraints, ensuring that no hospital is over capacity.If max_flow is less than n, it indicates that there isn't a valid assignment for all patients, given the constraints. The algorithm respects the hospital capacities and the range parameters for assignments.

Time complexity: For creating nodes and edges: O(n+k), for building the graph: O(n*k), and Fork Fulkerson has a time complexity of O(E*n), where E is total number of edges which is a maximum of (n+k+n*k).Thus time complexity is O(n^2 * k)

6. Given a sequence of numbers find a subsequence of alternating order, where the subsequence is as long as possible. (That is, find a longest subsequence with alternate low and high elements).

Example
Input: 8, 9, 6, 4, 5, 7, 3, 2, 4
Output: 8, 9, 6, 7, 3, 4 (of length 6)
Explanation: 8 < 9 > 6 < 7 > 3 < 4 (alternating < and >)

For this, we will be using a dynamic programming approach in the following manner:

We first initialize a dynamic programming

For this algorithm, we declare an array list[n] with the total length of the array, and initialize the first value to 1 as the minimum subsequence of a non-empty array is 1.

We then run the following algorithm

- Iterate for every element in the array, and initialize list[0] to 1. We initialize a counter to be 0.
- Then, we run another nested loop which iterates over the second(with the loop variable i). We first check whether the count variable is an even or odd. If even, we execute the first step
    i) If even, then we label at the current element in the subsequence as the high. We iterate over all the other elements for the following condition. If arr[i] < arr[j] && lis[i] < lis[j] + 1, we increase the length of the sequence and store this sequence. We then increment the counter variable by one.
    ii) Else if odd, we label the current element in the subsequence as the low. We iterate over all the other elements for the following condition. If arr[i] > arr[j] && lis[i] < lis[j] + 1, we increase the length of the sequence and store this sequence. We then increment the counter variable by one.
- We repeat this for all elements, and if we find a longer list, we change the list to that sequence (where we have two lists, one which hold the longest list up to that point and one that holds the current one. )
- We then run the same algorithm once again, but now instead of initializing counter to zero, we initialize it as 1. This would account for the fact that the sequence might start at a low.

Time complexity: as this uses dynamic programming, this would have a time complexity of O(N^2) and the space use is O(n). This is because we use two nested loops to check for sequences.

Proof of correctness:

The algorithm uses dynamic programming to solve this problem. We initialize the first value to one, and thus the base case is true(as if there is one element in the array, the longest subsequence is by default one.

List[j] is the length of the alternate subsequence up till index j, i.e. the longest subsequence till that locations.. Suppose all List[1] to List[j-1] are all correct.Now at index j, either array[j] is correct with length 1, or another subsequence that is extended by the element at index j. This, we check if the element follows the alternate rule with previous subsequences, and if so, its maximum length is L[i] + 1 because we appended the element at index j. The algorithm takes the maximum over these possibilities, so L[j] is correct. Now if each L[j] is correct, then the algorithm is correct because it returns the maximum of L[j] for all j, one of which must be the LIS of the input.