CS 180 Homework 6

Prithvi Kannan

405110096

1. Exercise 19 on page 329

We create an array to store whether i and j form a correct interleaved structure of strings x and y. In order for dp[i,j] to be true is s(1,i+j) is an interleaving of x'(1,i) and y'(1,j). Now we can simplify into an optimal substructure: $dp[i,j] = true\ if\ dp[i-1,j] = true\ and\ s[i+j]\ =\ x'[i]\ or\ s[i+j]\ =\ y'[j]$.

The algorithm looks as follows:

```
Initialize a 2d array of size $n$
Loop from k=1 up to n
    For all i and j such that i+j=n
    If dp[i-1,j]=true
        If s[i+j] = x'[i], then dp[i,j]=true
        If s[i+j] = y'[j], then dp[i,j]=true
    Else dp[i,j]=false
Return yes if there is a pair of i and j such that i+j-n so that dp[i,j]=true
```

This algorithm runs in time $O(n^2)$ since the array is initialized to size $n\ x\ n$ and it takes constant time to fill each entry of the array, since it involves checking at most 3 steps: if the past entry is true, and if the last letter is part of x or y.

2. Exercise 22 on page 330

First, we consider the shortest paths to each vertex using i edges and then consider the number of such paths.

As a base case, we set $opt(i, v) = 0$ and $opt(i, v') = \infty$ since the we assume the maximum distances for all paths. We also set $num(i, v) = 1$ and $num(i, v') = 0$.

To compute the shortest path, $opt(i, s)$, we take the minimum of $opt(i-1, t) + ct$ where ct is the weight of a path to t. This recurrence makes sense as it states that to travel to node s using i edges we must first travel to a node t using $i-1$ edges, and then go to node s.

The $num(i, s)$ is computed by taking the sum of all counts $num(i-1, t)$ where the cost is minimized, aka $opt(i, s)$ is minimized.

Now the optimal path to w is the minimum of all paths of different length to w, $opt(w) = \min(opt(i, w))$, and the number of such paths is equal to the sum of the counters for all paths that are minimal cost, $num(w) = sum\ of\ N(i, w)\ for\ all\ i\ such\ that\ opt(i, w) = opt(w)$.

3. Exercise 24 on page 331

To consider gerrymandering n districts, we first consider how to gerrymander j districts for $1 \leq j \leq n$. We initialize a counter for votes for party A and party B, and also need to keep track of how many precincts have already been assigned to each party (since it must end with $n/2$ for each party), but since there are only two parties and each precinct is assigned to exactly one, we must only store number of precincts assigned to party A.

We set up an array dp where $dp[j, k, x, y] = true$ if it is possible for A to earn at least x A-votes in their precincts and y A-votes in their precincts.

At each step, we consider precinct $j + 1$. We can either put the precinct into A's group or B's group, so we consider the previously solved problem of $dp[j, k - 1, x - z, y]$ or $dp[j, k, x, y - z]$.

There is a solution to the gerrymandering problem if any entry in the array in the form $dp[n, n/2, x, y]$ is true where both x and y are greater than $mn/4$.

This algorithm runs in time $O(n^2m^2)$ since the array is size $n\ x\ n\ x\ m\ x\ m$, as the max values of x and y are m and the max values of j and k is n. At each step, the algorithm considers two previously computed entries, which takes constant time.

4. Exercise 7 on page 417

We construct a node for each client in our network and another node for each base station. There is an edge from a client to a node of capacity 1 if the client is within the range of the base station. If the client is too far away from the base station, there is no edge. Since we have multiple clients and base stations, we simplify our problem by creating a super source connected to each client with an edge of capacity 1 and a super sink connected to each station with an edge of capacity L.

Call the super source s and super sink t. If the max flow path from s to t is equal to n, then that means that every client can be connected to a base station, which means there exists a feasible connection. It is impossible for the max flow path to be greater than n since the super source, s, is connected to n nodes, each with capacity 1, meaning a cut there would be value n, and the max flow cannot exceed a cut in the graph. If the max flow path is less than n, that means that at least one client was not able to be connected all the way to the super sink, so there cannot be a feasible connection.

The runtime of this algorithm is $O(n^2k)$. It takes $O(nk)$ to compute the distance from each client to base station in order to build the edges in our network. In the worst case, our network contains n edges from s to the clients, nk edges between the clients and base stations, and then k edges from the base stations to the super sink, meaning $O(n + k + nk)$ edges, and n clients and k base stations so $O(n + k)$ vertices. Finding the max flow path using the Ford-Fulkerson algorithm takes $O(|f|(e + v))$ so for this problem, $|f| = n, e = n + k + nk$, and $v = n + k$ so $O(n * (n + k + nk + n + k) = O(n^2k)$. In combination with the precomputation step described above, this is $O(n^2k + nk) = O(n^2k)$.

5. Exercise 9 on page 419

Like the above problem, we construct a network flow graph between the set of patients and hospitals. For each of the n patients, if there is a hospital within 30 minutes driving time, we add an edge from that patient to that hospital with capacity 1 to indicate that we can send that patient to that hospital. Since we have a problem with multiple sources and sinks, we construct a super source s which is connected to every patient with an edge of capacity 1 and a super sink t which is connected to every hospital with an edge of capacity $\left\lceil \frac{n}{2} \right\rceil$ to satisfy the condition about distributing the load on each hospital.

Call the super source s and super sink t. If the max flow path from s to t is equal to n, then that means that every patient can be sent to a hospital, which means there exists a feasible assignment. It is impossible for the max flow path to be greater than n since the super source, s, is connected to n nodes, each with capacity 1, meaning a cut there would be value n, and the max flow cannot exceed a cut in the graph. If the max flow path is less than n, that means that at least one patient was not able to be sent to a hospital, so there cannot be a feasible assignment.

The runtime of this algorithm is $O(n^2k)$. It takes $O(nk)$ to compute which patients are within 30 minutes of which hospitals. In the worst case, our network contains n edges from s to the clients, nk edges between the patients and hospitals, and then k edges from the hospitals to the super sink, meaning $O(n + k + nk)$ edges, and n clients and k base stations so $O(n + k)$ vertices. Finding the max flow path using the Ford-Fulkerson algorithm takes $O(|f|(e + v))$ so for this problem, $|f| = n, e = n + k + nk$, and $v = n + k$ so $O(n * (n + k + nk + n + k) = O(n^2k)$. In combination with the precomputation step described above, this is $O(n^2k + nk) = O(n^2k)$.

6. Given a sequence of numbers and a sub-sequence of alternating order, where the sub-sequence is as long as possible. (that is, and a longest sub-sequence with alternate low and high elements). As always, prove the correctness of your algorithm and analyze its time complexity.

Example Input: 8; 9; 6; 4; 5; 7; 3; 2; 4

Output: 8; 9; 6; 7; 3; 4 (of length 6) because- $8 < 9 > 6 < 7 > 3 < 4$

We use a dynamic programming approach to solve this problem where we consider the length of the longest alternating subsequence ending at an index with a greater than and with a less than. We construct a dp array of size n by 2 to represent the 2 cases we have: ending with greater than and ending with less than. The optimal substructure for this problem is twofold

1. $dp[i][0] = max(dp[i][0], dp[j][1] + 1)$ for all $j < i$ and $arr[j] < arr[i]$
2. $dp[i][1] = max(dp[i][1], dp[j][0] + 1)$ for all $j < i$ and $arr[j] > arr[i]$

This recurrence is based on the idea that if we are at position I, we must either find a larger element to satisfy one type of alternating sequence or a smaller element to satisfy the other type of alternating sequence. For example, if our sequence ends with 3, then we must either find a value that is greater 3 to place a '>' in that index of our alternating sequence, or we must find a smaller value to place a '>' in that index.

The reason we consider $dp[j][1] + 1$ instead of $dp[j][0] + 1$ when filling the entry for $dp[i][0]$ is to satisfy alternate property because in $dp[j][0]$ last element is bigger than its previous one and $arr[i]$ is greater than $arr[j]$ which will break the alternating property if we update. By symmetry we show this for the other case.

Pseudocode for this algorithm looks like the following:

```
Initialize an n x 2 array with values 1 at each entry
Loop through values of i from 1 up to n
    Loop through values of j from 1 up to i
        If arr[j]<arr[i] set dp[i][0] to max of dp[i][0] and dp[j][1]+1
        If arr[j]>arr[i] set dp[i][1] to max of dp[i][1] and dp[j][0]+1
Return the largest value in the array
```

This algorithm runs in time $O(n^2)$ since the outer loop runs n times and the inner loop can run up to n times for each iteration of the outer loop. The steps within the inner loop take constant time to run since both steps involve a fixed number of comparisons and looking a previously filled entry of the array.