Joyce Chen Dis 1F 2:00PM Parshan Teimouri

CS180 Homework #6

1. **Exercise 19 on page 329**

   Algorithm
   - Define a string as an interleaving only if it is made up of strings x and y where the last character of the string is from either x or y.
   - Let O(i, j) be the optimal solution for a string of length i+j, where i = last index of our proposed string x' and j = last index of our proposed string y'.
   - Let O(0, 0) be a valid interleaved string.
   - For string length L = 1, 2, …, n
     - For all pairs of (i, j) such that i+j = k:
       - If O(i-1, j) is a valid string and the last character is x'[i]:
         - O(i, j) is a valid string
       - Else if O(i, j-1) is a valid string and the last character is y'[j]
         - O(i, j) is a valid string
       - Otherwise, O(i, j) isn't a valid string
   - If there is a pair (i, j) where i+j = n and O(i, j) is a valid string, then there exists a valid interleaving.

   Proof of Correctness
   - Base case:
     - String length = 0 signifies an empty string, which is a valid interleaving
   - Inductive hypothesis:
     - Suppose we know the validity for strings of length 1 to n-1, given different prefixes of x and y – x' and y'.
     - To find the validity of a string with length n, we consider the last character, which could either come from x' or y'.
     - If the last character came from x', then we check whether the string of length n-1 is a valid interleaving of x' without its last character and y' – if so, then our current string is valid.
     - If the last character came from y', check whether the string n-1 is a valid interleaving of y' without its last character and x' – if so, our current string is valid.
     - Therefore, our algorithm works to find the validity of a string of length n.

   Time Complexity
   - We are considering all possible string lengths, which is O(N).
   - The inner loop for considering all pairs of (i, j) for a set string length also takes O(N). Therefore, the 2 loops combined takes $O(N^2)$ time.
   - Total runtime = $O(N^2)$.

2. **Exercise 22 on page 330**

   Algorithm

- Create a 2D array costArr, where the rows i represent the # of edges in a path and the columns j represent the nodes in the graph. costArr[i][j] = cost of the shortest path from v → j of length i.
- Create a 2D array numArr, where the rows i represent the nodes and the columns j represent the # of edges. numArr[i][j] = number of shortest paths from v → i of length j.
- Set the costArr[0][0…n] = 0 and costArr[0…n][0] = 0, for v=0 as the starting node.
- For # of edges e = 1 to |E|:
    - For nodes x = 1 to |V|:
        - Consider all neighboring nodes of x, say {$x_1 \ldots x_n$}
            - Let C = costArr[x_i][e-1] + cost of the edge (x, x_i).
        - Set costArr[x][e] = minimum value of C.
        - Let S = sum of numArr[y_i][e-1], where {$y_1 \ldots y_n$} are all neighboring nodes that yield the optimal cost for path length e-1.
        - Set numArr[x][e] = S.
- To find the # of shortest paths from v→w:
    - Iterate through costArr[1…n][w] and find the path lengths that give the shortest costs.
    - Sum up numArr[w][p] for each optimal path length {$p_i \ldots p_n$}
    - Return this sum.

Proof of Correctness

- Base case:
    - There is only 1 node in the graph, the starting node.
    - There are 0 shortest paths to the starting node.
    - Our algorithm establishes that the starting node will have 0 shortest paths to it by setting costArr[0][0…n] = 0.
- Inductive hypothesis:
    - Suppose we know the optimal (shortest) path length to all preceding vertices {$v_i \ldots v_k$} of vertex w. That is, ($v_i \ldots v_k$ → w) are valid directed edges in the graph.
    - We want to know the optimal path length to vertex w.
    - The optimal path length to w will be the shortest path length we can find by adding [s → a preceding vertex v] and [edge cost between v and w].
    - Because we have optimized the path length from s → v, we know that it will be minimized.
    - Therefore, we just have to find the minimum of (s → v path length + edge cost), which our algorithm does.
    - By tracking the # of shortest paths to the vertex in a 2D array, we can effectively return the # of shortest path lengths to any vertex in the graph.

Time Complexity

- Iterating through all the edges in the graph in the outer loop takes O(E) time.
- Iterating through each node in the inner loop takes O(V) time.

- Therefore, the total runtime is O(E*V), where E = # edges and V = # vertices.
- Space complexity is $O(V^2)$, because we are using 2D arrays.

3. **Exercise 24 on page 331**

   Algorithm
    - We can divide this problem into subproblems of gerrymandering precincts from 1-k, where k = {1…n}. For every precinct we add, we can either place it in district 1 or district 2. We save the result of placing it in both district 1 and district 2, and continue onwards. This is because there's a possibility that the precinct is in either district for the optimal solution, so we must consider both cases.
    - Create a 4D array arr[i, j, k, l] with I representing the precinct #, j representing the # of precincts in district 1, k representing the # of A-votes in district 1, and l representing the # of A-votes in district 2.
    - Initialize arr[0, 0, 0, 0] = true
    - For i = 1 to n precincts:
        - For j = 1 to n precincts in district 1:
            - For x = 0 to m*n votes:
                - For y = 0 to m*n votes:
                    - If either arr[i-1, j-1, x-A[j], y] or arr[i-1, j-1, x, y-A[j]] is true, then arr[i, j, x, y] = true. Otherwise, it's false.
    - If there exists a true entry in arr where arr[i, j, k, l] = arr[n, n/2, x, y] where x and y are both greater than m*n/4, then we return true.
    - Otherwise, return false.

   Proof of Correctness
    - Base case:
        - There are 2 precincts.
        - There will be a majority unless the sum of the 2 parties' votes are tied.
    - Inductive hypothesis:
        - Suppose we know whether gerrymandering would work for the first 1-n precincts.
        - We want to know whether it would work for the nth precinct.
        - Given that we can either place this precinct in district 1 or 2, we can see which placement will yield a majority.
        - If placing it in district 1 or 2 yields a majority, then there would be a valid gerrymandering. If neither yields a majority, there is no valid gerrymandering.
        - Our algorithm keeps track of the results of these choices through the 4D array, and therefore is able to figure out whether gerrymandering works for the nth precinct.

   Time Complexity
    - We are dividing the problem down into N^2M^2 subproblems, because there are N precincts and M total voters in each precinct.

- Because each computation in the innermost loop takes constant time, the total runtime of the algorithm is $O(N^2M^2)$.
- The space complexity is $O(N^4)$ because we are using a 4D array.

4. **Exercise 7 on page 417**

Algorithm
- Let n = # of clients and k = # of base stations.
- Create a network graph G with source node S, sink node T, and internal nodes that represent the clients and base stations.
- Add edges of capacity 1 between all client nodes v_c and the source node S.
- Add edges of capacity L (load parameter) between all station nodes v_s and the sink node T.
- Add an edge of capacity 1 between a client and station only if the client is within distance r of the station.
- If there exists a valid S-T flow with a value of n in G, then there is a way to connect all clients to a base station subject to the given requirements.

Proof of Correctness
- We've already proved the maxflow algorithm works.
- Consider a cut consisting of source S combined with all the edges it touches, and the rest of the nodes in the other partition.
- Because each internal edge has capacity 1, the number of edges (and therefore the number of clients) over this cut would equal the flow.
- Because each outgoing flow from the station nodes to the sink is L, each station node can be connected to at most L client nodes.
- Therefore, if there is an outgoing flow from the station nodes to the sink of L, that means that there is a total of L clients over the cut.
- That means that the max flow is n, and each client is matched to a station; our algorithm thus works to figure out whether all clients can be matched to a station.

Time Complexity
- The network graph G will have n+k nodes and at most n*k edges, so the runtime will be the time to solve such a max-flow problem.
- The max possible flow through G will be n, given that each client is connected to all stations with an edge of capacity 1.
- Finding a S-T path in the max-flow problem costs O(V+E), or O(n+k + n*k)
- Therefore, the total runtime would be O(n * (n+k + n*k)), or roughly $O(n^2k)$.

5. **Exercise 9 on page 419**

Algorithm
- Let n = # of injured people and k = # of hospitals.
- Create a network graph G with source node S, sink node T, and internal nodes that represent the injured people and hospitals.
- Add edges of capacity 1 between all injured people nodes v_i and source node S.

- Add edges of capacity $\lceil n/k \rceil$ between hospital nodes v_h and sink node T.
- Add an edge of capacity 1 between an injured person v_i and hospital v_h only if the hospital is within half-hour's driving time of the person's current location.
- If there is a valid S-T flow with a value of n, then there exists a way to send all n people to a hospital without overloading any of the hospitals.

Proof of Correctness

- We've already proved the maxflow algorithm works.
- Consider a cut consisting of source S combined with all the edges it touches, and the rest of the nodes in the other partition.
- Because each internal edge has capacity 1, the number of edges (and therefore the number of injured people) over this cut would equal the flow.
- Because each outgoing flow from the hospital nodes to the sink is n, each hospital node can be connected to at most n people nodes.
- Therefore, if there is an outgoing flow from the hospital nodes to the sink of n, that means that there is a total of n injured people over the cut.
- That means that the max flow is n, and each person is matched to a hospital; our algorithm thus works to figure out whether everyone can be matched to hospitals.

Time Complexity

- The network graph G will have n+k nodes and at most n*k edges, so the runtime will be the time to solve such a max-flow problem.
- The max possible flow through G will be n, given that each person is connected to all hospitals with an edge of capacity 1.
- Finding a S-T path in the max-flow problem costs O(V+E), or O(n+k + n*k)
- Therefore, the total runtime would be O(n * (n+k + n*k)), or roughly $O(n^2k)$.

6. **Given a sequence of numbers, find a subsequence of alternating order, find the length 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 >)
   Algorithm
   - For this problem, we want to check the longest alternating subsequence for every possible ending element (index 0 to n-1). The ending element of the subsequence will either be a high (greater than the previous element) or a low (less than the previous element). We must consider both cases.
   - Create a 2D array arr with rows 0-n that represent the ending index of the subsequence, and columns 0-1 that specify whether it's a low or high, respectively. arr[i][0] holds the length of the longest alternating subsequence that ends at index i and is a "low." arr[i][1] holds the length if the element is a "high."
   - Set arr[0][0] and arr[0][1] to 1.

- Create a variable maxLen that tracks the maximum subsequence length.
- For each index i from 0 to n-1:
    - Keep highMax and lowMax variables that track the longest subsequence length if we end on a high or low, respectively.
    - For every preceding index j from 0 to i:
        - if arr[j][1] + 1 > lowMax (case if we end on low)
            - set lowMax = arr[j][1] + 1
        - if arr[j][0] + 1 < highMax (case if we end on low)
            - set highMax = arr[j][0] + 1
    - arr[i][0] = lowMax
    - arr[i][1] = highMax
    - Set maxLen = max of lowMax and highMax if it is greater than maxLen
- Return maxLen

Proof of Correctness

- The key to this algorithm is that the optimal alternating subsequence that ends on high = optimal preceding subsequence that ends on low + the current element. Likewise for the optimal subsequence that ends on low.
- Base case:
    - The alternating subsequence ends on index 0.
    - The max length out of all possible alternating subsequences = 1, which is just the element itself.
- Inductive hypothesis:
    - Suppose we know the length of the optimal alternating subsequences that end on or before indices n-1, for both whether the ending element is a low or high.
    - We want to figure out the length of the optimal alternating subsequence for the sequence 0-n (ends on index n).
    - If we have the max length of a preceding subsequence that ends on a low, then its length + 1 = length of optimal subsequence for sequence 0-n that ends on high.
    - If we have the max length of a preceding subsequence that ends on a high, then its length + 1 = length of optimal subsequence for sequence 0-n that ends on low.
    - The length of the optimal subsequence for sequence 0-n would therefore be the maximum out of these 2 values.
    - Our algorithm correctly finds the length of the alternating subsequence that ends on index n!

Time Complexity

- The outer loop goes through indices from 0-n, which takes O(N) time. We are looping through every possible ending index.

- The inner loop goes through indices from 0-i, which takes worst case O(N) time. We are looping through the preceding indices of i to find the most optimal length so far.
- Therefore, the total runtime is $O(N^2)$. Space complexity is O(N) because we are using a 2D array that has a fixed column length of 2 but variable row length that depends on the input size.