Leetcode

December 5, 2023

Gage Moore

1. N-th Tribonacci Number (#1137)

My solution in LeetCode

Code:

```python
class Solution:
    '''
    Memoize tribonacci sequence to n digits
    Time Complexity: O(n)
    Space Complexity: O(n)
    '''
    def memo(self, n: int) -> list[int]:
        # Initialize state list
        state = [0, 1, 1]

        # Iterate through tribonacci
        for _ in range(n): # O(n)
            state.append(state[-1]+state[-2]+state[-3])

        return state

    '''
    Call memo(n) (O(n)) and return nth element
    Time Complexity: O(n)
    Space Complexity: O(n)
    '''
    def tribonacci(self, n: int) -> int:
        return self.memo(n)[n] # O(n)
```

Brief Analysis:

The N-th tribonacci number was calculated by first memoizing the entire tribonacci sequence up to n digits and simply returning the last, (i.e. the nth), element. The memoization kept track of the necessary state to calculate each successive digit so it was able to run in $O(n)$ time. The main `tribonacci()` function which returns the solution simply accesses the nth digit, a constant time operation, therefore the overall time complexity of the solution is $O(n)$. Due to the memoization of the tribonacci sequence up to n digits, the space complexity is also $O(n)$, because the list to store the sequence will grow to be n elements long.

Group Discussion:

For this problem we both used an approach of generating the entire tribonacci sequence first and foremost, and pulling from the sequence to get the nth element. My partner and I had both tried using a moving window strategy to reduce the space that was used in the algorithm however this proved too complicated for what it was asking for. We both had the same time complexity, linear time, and space complexity, O(n)s.

2. Queue Reconsutruction by Height (#406)

My solution in LeetCode

Code:

```
class Solution:
    '''
    Time Complexity: O(n^2)
    Space Complexity: O(n)
    '''
    def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:
        numAhead = {}
        heights = []
        result = []

        # Generate dictionary of height to number of people ahead for each
person.
        for height, peopleAhead in people: # O(n)
            if height in numAhead:
                # Height is already in the dictionary, insert peopleAhead for
given height.
                numAhead[height].append(peopleAhead)
                numAhead[height] = sorted(numAhead[height])
            else:
                numAhead[height] = [peopleAhead]
                heights.append(height)

        # Ensure heights is in descending order.
        heights.sort(reverse=True)

        index = 0
        length = len(heights)

        # Go through every person, fine tune their final position.
        while index < length: # O(n)
            height = heights[index]
            if not numAhead[height]:
                # Nobody with current height.
                index+=1
            else:
                temp = numAhead[height].pop(0)
                peopleAhead = temp

                # Move through people until index, (j), is in correct position.
                j = 0
                while j < len(result): # O(n)
                    if peopleAhead == 0:
                        break
                    if result[j][0] >= height:
                        peopleAhead-=1
                    j+=1

                # Insert person at j in the result list.
                result.insert(j, [height, temp])
```

```
        return result
```

Brief Analysis:

I approached this problem by initializing a dictionary to match heights with lists of the number of people ahead of the given height as well as a separate list to hold heights. I then looped through every person, ($O(n)$), and added their height and number of people ahead to the previously mentioned dictionary and list. I then went through every person, (at most), once more by iterating through the heights list, ($O(n)$). For every height with some number of people ahead, I callibrated a new index `j` by iterating through every person, (at most), ($O(n)$). This puts the time complexity as a whole at $O(n^2)$. The space complexity is dependent on the dictionary and list, both of which grow to handle n elements at most, (as both are based on the individual heights of the people), thus the space complexity is $O(n)$.

Group Discussion:

For this problem, my partner used the Python `sort` function to `sort` the number of people according to their height and the number of people ahead and putting this into a new list. He then used this list to insert each person into a result this accordingly. His solution was only 8 lines of code, which is a huge simplification from my implementation. His time complexity was $O(n^2)$, since his Python `sort` function sorted by two criteria. His space complexity was only $O(n)$. I noticed how much faster his solution was and learned the importance of using the Python standard library.

3. Combination Sum (#39)

My solution in LeetCode

Code:

```
class Solution:
    '''
    BFS tree implementation:
    1. Split candidates into branches, e.g. [1,2] -> [[1],[2]]
    2. While branches exist:
        - pop the first branch
        - add to solutions if sum == target
        - create every permutation of branch + each candidate
        - for every permutation, if sum == target add to solutions
    3. Return solutions
    Time Complexity: O(n) + O(bn) + O(b) = O(bn)
    Space Complexity: O(bn)
    '''
    def combinationSum(self, candidates: List[int], target: int) ->
List[List[int]]:
        # Generate initial branches (1 for each candidate).
        branches = [[candidate] for candidate in candidates] # O(n)

        # Store solutions in set so Python takes care of duplicates for me!
        solutions = set()

        # Loop while there exists some branch such that
        #   sum(branch) < target
        while branches: # O(b)
            current_branch = branches.pop(0)

            # Branch as it exists adds up to target.
            if sum(current_branch) == target:
                solutions.add(tuple(current_branch))

            # Generate every permutation of current branches + candidate.
            for candidate in candidates: # O(n)
                new_branch = current_branch.copy()
                new_branch.append(candidate)
                new_branch.sort()

                # Permutated branch adds up to target.
                if sum(new_branch) <= target:
                    branches.append(new_branch)

        return [list(branch) for branch in solutions] # O(b)
```

Brief Analysis:

To find the combination sum solution I took a breadth first search approach. I started from the root node and iterated through each possible level of branches with 1 node, branches with 2 nodes, branches with 3 nodes, and so forth until the sum of all branches exceeded the target. To split the candidates list into the first level of n branches each with 1 node, I used a list comprehension which

ran in $O(n)$. I then iterated through every level of the tree, which I called $O(b)$. To create every new branch at each level, I looped through the candidates list, ($O(n)$), and found every combination of new branches. With this in mind, the overall time complexity of my solution is $O(n) + O(b \times n) + O(b)$, which can be reduced to $O(b \times n)$. Considering how n represents the number of candidates and b represents the number of levels the space complexity will come out to be $O(b \times n)$ because at most the problem will result in a tree of b levels each containing n elements.

Group Discussion:

My partner found the combination sum in a similar fashion by generating every row of node permutations by length. They pruned according to the sum of each list, and while mine started from the first list of candidates theirs checked the candidates individually first before creating combinations of them. This allowed their solution to exit sooner than mine for small targets. Their solution ran in $O(n^2)$ time and had a space complexity of $O(n^2)$.

4. Min Cost to Connect All Points (#1584)

My solution in LeetCode

Code:

```
from math import inf

class Solution:
    '''
    Return the manhattan distance between two points p1 and p2.
    Returns |x1 - x2| + |y1 - y2|
    Time Complexity: O(1)
    Space Complexity: O(1)
    '''
    def dist(self, p1, p2):
        return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

    '''
    Return the index of UNVISITED point with the smallest distance.
    Time Complexity: O(n)
    Space Complexity: O(1)
    '''
    def closestPoint(self, dists, visited):
        min = inf
        min_index = -1

        for u in range(len(dists)): # O(n)
            if visited[u] == False and dists[u] < min:
                # Point has not been visited and its distance is smaller than the
                # current minimum.
                min = dists[u]
                min_index = u

        return min_index

    '''
    Calculate cost to connect all points.
    Used Prim's algorithm:
        1. Initialize distance to each point, (aside from first point), to
infinity
        2. Keep track of which points I've visisted, (used list of bools, length
n)
        3. For every point:
            - visit the closest point
            - update the distances with distances to all neighbors of closest
point:
    Time Complexity: O(n^2)
    Space Complexity: O(2n) = O(n)
    '''
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        n = len(points)

        # Initialize list for distances of each point and whether each point has
```

```
been visited.
        dists = [inf] * n
        dists[0] = 0
        visited = [False] * n

        for _ in range(n): # O(n)
            # Visit the nearest point.
            u = self.closestPoint(dists, visited) # O(n)
            visited[u] = True

            for v in range(n): # O(n)
                if visited[v] == False and self.dist(points[u], points[v]) <
dists[v]:
                    # v hasn't been visited and going from u to v is shorter than
last
                    # recorded distance. Update distances list with better route.
                    dists[v] = self.dist(points[u], points[v])

        return sum(dists)
```

Brief Analysis:

To find the minimum cost to connect all points, I used Prim's algorithm as discussed in class this semester. I kept a list for the distance to every point, (which I initialized to ∞ for every point aside from the first), and I kept a list of booleans to record which points I visit. I then looped through every point, ($O(n)$), and considered the distance from that point to every other point, ($O(n)$), updating distances whenever a faster route is found. Because of this, my time complexity for this solution is $O(n^2)$. The space complexity is influenced by the `dists` list and the `visited` list, both of which will hold n elements at all times. Because of this, the space complexity is $O(2n)$, or more simply just $O(n)$.

Group Discussion:

For this problem my partner used Kruskal's algorithm because that's what he was most familiar with. It worked out well for him, though he did run into problems with how he was using a Python set. This was a unique perspective from what I did, since I used Prim's algorithms. We both used a cost-based search algorithm for doing this. My partner's time complexity was $O(n^2 \log n)$, which differed from mine. Both of our solutions were $O(n)$.

5. Binary Tree Level-order Traversal (#102)

My solution in LeetCode

Code:

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:

    '''
    Explore the current level, (left & right nodes).
    If the node exists, add it to a list.
    If either left or right exists, run function on the next level.
    Time Complexity: O(n)
    Space Complexity: O(n)
    '''
    def exploreLevel(self, traversal, left, right, level):
        if len(traversal) <= level:
            # Current level has been explored yet, initialize list.
            traversal.append([])

        # Record children that are not null on current level.
        if left: traversal[level].append(left.val)
        if right: traversal[level].append(right.val)

        # Recurse for children that exist.
        if left: self.exploreLevel(traversal, left.left, left.right, level+1)
        if right: self.exploreLevel(traversal, right.left, right.right, level+1)

    '''
    Recursive approach: split between left & right each recursion.
    Time Complexity: O(n)
    Space Complexity: O(n)
    '''
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if root:
            traversal = [[root.val]]
            # Begin recursion. Start from level 1 since root is 0.
            self.exploreLevel(traversal, root.left, root.right, 1)
            return traversal[:-1]
        else:
            # Root is null.
            return []
```

Brief Analysis:

My algorithm performed a level-order traversal of a given tree through recursion. I maintained a list, `traversal`, and called the `exploreLevel()` function recursively for every level of the tree, calling the

function twice for the left and right children at every split. Ultimately, the function, which performs constant time operations, would run once for every node in the tree, resulting in a time complexity of $O(n)$. Because the traversal list would grow to ultimately hold every node in the tree as well, the space complexity was also $O(n)$.

Group Discussion:

For this problem, my partner used a depth first search approach in which she visited each node to the bottom and generated a dictionary according to which level each node is found, and then converting this dictionary to a list and returning it. She chose this because it seemed the most intuitive. Mine was the opposite, I used breadth first search. We both ran into issues when nodes did not exist and had to include lots of `if` statements to combat this. We both had the same time complexity, linear time, and space complexity, $O(n)$.

6. Number of Provinces (#547)

My solution in LeetCode

Code:

```
class Solution:
    '''
    DFS search approach
    Search recursively as far down as you can go with each city.
    Every time we have to backtrack to top of DFS, increment the
    province count.
    Time Complexity: O(n^2)
    Space Complexity: O(n)
    '''
    def findCircleNum(self, isConnected: List[List[int]]) -> int:
        n = len(isConnected)
        visited = []

        def dfs(city):
            visited.append(city)
            for neighbor in range(n): # O(n)
                if isConnected[city][neighbor] == 1 and neighbor not in visited:
                    # New unvisited neighbor, must be within the current
province.
                    dfs(neighbor)

        count = 0
        for city in range(n): # O(n)
            if city not in visited:
                # New city not discovered by dfs yet,
                # must be a new province.
                count += 1
                dfs(city)

        return count
```

Brief Analysis:

My approach for calculating the number of provinces amongst a list of cities was to run a depth first search algorithm on every unvisited city and increment the number of provinces according to the number of times I need to backtrack to the start of each search. I implemented a depth first search function `dfs()` which recursively called itself for every unvisited neighbor of the current city, ($O(n)$). I then ran this search function for every city in the list of cities, ($O(n)$), tallying up the number of provinces that I find according to every time the program returns to the root invocation of `dfs()`. Because of this, the time complexity ended up being $O(n^2)$. The space complexity for the solution is $O(n)$ which represents the list `visited` which holds every city, (n), that gets visited by the search function.

Group Discussion:

For the number of provinces problem, I met with two people and both of them had the same approach that I did: using a depth first search. Our implementations were the same in the sense that we ran depth first search, incrementing the province number for each new cycle discovered, however the specifics of our implementations all differed. While I maintained a list of visited cities, others used boolean lists for this. Our runtime and space complexity all matched.