Solomon B. Pobee **Project 6 LeetCode report**

Nth Fibonacci comparison

Time and space complexity

The time complexity is O(n) because the function iterates from 3 to n because the base case for it is 0 to 2. The iteration performs a constant amount of work in each iteration. The time take grows linearly.

The space complexity is O(1) because it uses a fixed amount of space for each input of n. the amount of space does not change from the different variable.

```python
class Solution:
    def tribonacci(self, n: int) -> int:
        if n == 0:
            return 0
        elif n == 1 or n == 2:
            return 1
        t0, t1, t2 = 0, 1, 1
        for i in range(3, n + 1):
            t3 = t0 + t1 + t2
            t0, t1, t2 = t1, t2, t3
        return t3
```

My partner and I addressed the solution differently. In his, he iterated through his steps but we both implemented a dictionary in our code with helped with the time and space complexity. We had the same time complexity but different space complexity. Despite, his steps looking great and understandable, my space complexity was better than his.

The 2nd person I spoke with had the same solution as mine. The only difference was the syntax and variables. We both have the same time and space complexity.

Two sum comparison

Time and space complexity

Time complexity is O(n) where 'n' is the number of element. The time complexity is (n) because the function iterates through the list once, performing constant time operations for each element.

Space complexity is O(n) because the function uses a dictionary to store the indices of the element. In a worse case scenario, all the 'n' elements are stored in the dictionary.

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        num_dict = {}
        # Iterate over the list
        for i, num in enumerate(nums):
            # Calculate the difference
            diff = target - num
            # Check if the difference is in the dictionary
            if diff in num_dict:
                # Return the indices of the two numbers
                return [num_dict[diff], i]
            # Store the index of the current number
            num_dict[num] = i


#time complexity: O(n)
#space complexity: O(n)
```

The person I discussed this solution with had a simple solution as compared to mine, his approach was to iterate through each item and when they are equal, it just returns. His time and space complexity were 0(n^2), O(1) respectively. This is different from mine in the sense that I used a dictionary and iterated over the list, then calculate the difference. I then use an if statement to check the difference in the dictionary . My time and space complexity were 0(n) & 0(n) respectively.

The second person I spoke to used a for loop and compared indices. This time and space complexity was O(n^2) and O(n) respectively. We think mine was faster because of the use of the dictionary.

Combination sum comparison

Time and space complexity

Time complexity is O(n^target) because there is a recursion call. In the function , there has to be an exploration of all the possible combinations of candidates. For each candidate, there are two choices that is either to include it in the combination or exclude it. Doing this means that there will be a branching of 'n' and a dept of target which is the value of the target.

Space complexity is O(target) because the maximum dept of the recursion is target.

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        # Helper function to backtrack
        def backtrack(remain, combo, start):
            # Check if the remaining value is 0
            if remain == 0:
                result.append(list(combo))
                return
            # Check if the remaining value is negative
            elif remain < 0:
                return

            # Iterate through the candidates
            for i in range(start, len(candidates)):
                # Add the candidate to the combination
                combo.append(candidates[i])
                # Recursively call the backtrack function
                backtrack(remain - candidates[i], combo, i)
                combo.pop()

        result = []
        backtrack(target, [], 0)
        return result


#time complexity: O(n^target)
#space complexity: O(target)
```

My partner, had time complexity 0(2^nxn) and his code differed from how I wrote mine. In his code, he created a duplicate which appeared in his output. His style of writing his code had a lot of meaning and it was easy to understand. Mine had a time complexity of 0(n^target) which had a bit of an easy way of running through an if statement and an iteration.

Min cost comparison

Time and space complexity

Time complexity is O(n^2 log n) because the while loop runs 'n' times and in each iterations, there is a heap operations which takes O(log n) times. So in each point, there is a calculation of the Manhattan distance to every other point which also takes O(n). so the total complexity is O(n^2 * log n)

Space complexity is O(n^2) because of the space required to store the heap and the set. The heap might take (n^2) elements.

```
class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        def manhattan_distance(p1, p2):
            return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

        n = len(points)
        if n == 0:
            return 0

        min_heap = [(0, 0)]  # (cost, point_index)
        total_cost = 0
        in_mst = set()

        while len(in_mst) < n:
            cost, i = heapq.heappop(min_heap)
            if i in in_mst:
                continue
            total_cost += cost
            in_mst.add(i)
            for j in range(n):
                if j not in in_mst:
                    heapq.heappush(min_heap, (manhattan_distance(points[i],
points[j]), j))

        return total_cost


#time complexity: O(n^2)
#space complexity: O(n)
```

My partner I discussed this problem with had similar way as mine in solution this question. The only difference between the two codes there how we named each part. I used just letters and he used names and I find that wise to do especially when someone else is reading your codes.

Triangle

Time and space complexity

Time complexity O(n^2) where n = the number of rows in the triangle. The complexity is O(n^2) because the function iterates over each element of the triangle once and it performs a constant amount of work. The out for loop takes a time of o(n) and the inner for loop also takes time O(n).

Space complexity is O(n) because it does not use any space when the function takes and input.

```
class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        for i in range(len(triangle) - 2, -1, -1):
            for j in range(len(triangle[i])):
                # Update the current element to the sum of itself and the minimum of
the two elements directly below it
                triangle[i][j] += min(triangle[i + 1][j], triangle[i + 1][j + 1])
        # The top element now contains the minimum path sum
        return triangle[0][0]


# Time Complexity: O(n^2)
# Space Complexity: O(n)
```

My partner almost had similar way for approaching this problem. His approach seemed reasonable to me but the only thing I found confusing was how this space complexity was O(n).

With the 2nd person I spoke to, we both used a for loop but the only difference was that he used a double for loop. We both had the same time and space complexity

Binary-tree-level-order

Time and space complexity

Time complexity is O(n) because each node is processed once. n = the number of nodes in the binary tree. The storing node at each level tskes O(n).

Space complexity is O(n) because the queue contains all the nodes at the till the end of the tree. This can go down by n/2 nodes.

```
# 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:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []

        result = []
        queue = deque([root])

        while queue:
            level_size = len(queue)
            level_nodes = []

            for _ in range(level_size):
                node = queue.popleft()
                level_nodes.append(node.val)

                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

            result.append(level_nodes)

        return result

#time complexity: O(n)
#space complexity: O(n)
```

My partner and I had different approaches to solving this problem. He went through series of different steps that took lots to time to run. From his procedure, I learned that we sometimes have to find different ways of getting our codes to run faster.

The 2nd person I spoke to have the same structure as mine. We both explored the left and right child of the parent node. The difference was that I used a nested for loop and he used a recursion in breaking the problem into subproblems. Our time and space complexity are the same.