1. 1137. N-th Tribonacci Number

```
class Solution(object):
    def tribonacci(self, n):
        """
        :type n: int
        :rtype: int
        """

        # Dynamic programming solution, # O(n)
        if n==0:
            return 0
        elif n <= 2:
            return 1

        trib = [0 for i in range(n+1)] # intialize the list
        trib[1] = 1
        trib[2] = 1
        for i in range(3, n+1):
            trib[i] = trib[i-1] + trib[i-2] + trib[i-3]
        return trib[n]
```

Explanation:

- The time complexity of this function is O(n) because it is iterating from 3 to ‘n’ which is the nth number in the tribonacci sequence. Since its a single loop that runs from 3 to ‘n’, and it performs a constant amount of work inside the loop so this makes the overall time complexity O(n)
- The space complexity of this function is also O(n). The array is of size n + 1 where n is the nth number in the tribonacci sequence. This makes the overall space complexity also O(n)
- The technique I used was Dynamic Programming. I store the solutions to subproblems (the Tribonacci numbers for smaller values of n) in an array called trib. Each entry of trib[i] depends on the three preceding values (trib[i-1], trib[i-2], and trib[i-3]), which are already computed and stored in the array. This allows this function to avoid redundant calculations building up to the solution.

Discussion with another student:

I discussed my solution with Rachel George and we both had the same technique of dynamic programming with slightly different approaches. She didn’t have an array in her solution, but still had the same conceptual idea with the loop always computing the 3 numbers before the actual number we want to get the solution for a nth number in the tribonacci sequence. Instead of using an array she used 3 variables in her loop that were set to default values outside of the loop. So instead of calculating all the previous values like I am with this array she just overwrites those variables in each iteration updating them as the function goes. So our time complexities are still the same at O(n), but her space complexity is better than mine at O(1) since her function doesn’t keep track of all previous solutions, instead just keeping track

of the previous three.

2. 39. Combination Sum

```
class Solution(object):
    def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        # Backtracking solution (depth first search)
        result = []
        self.helper(candidates, target, 0, [], result)
        return result

    def helper(self, candidates, target, start_index, current_combination, result):
        # Base cases
        if target < 0: # means its a invalid solution, go back up the tree
            return
        if target == 0: # means we found a valid combination
            result.append(current_combination)
            return

        for i in range(start_index, len(candidates)):
            self.helper(candidates, target - candidates[i], i, current_combination +
[candidates[i]],   result)
```

Explanation:

- The time complexity of this function is O(N^(T/M+1), where N is the number of candidates, ‘T’ is the target value, and ‘M’ is the minimal value among the candidates.In the worst case the candidates array is filled with 1s and the target is ‘T’. In this case, the depth of the recursion tree can go up to ‘T’, and each level can have at most ‘N’ nodes.
- The space complexity of this function is O(T/M) for the recursive call stack. In the worst case, we could have a combination that equals the target with all elements being the smallest number in candidates, leading to a depth of T/M. There could be some additional space complexity taken into account as well with the results array having to be stored which could be in the worst case O(2^T), but usually this isn’t taken into account.
- The technique I used here was backtracking or Depth First Search. This means this solution explores all potential solutions and “backtracking” once it determines the current path does not lead to a valid solution or the end of this decision tree is reached. So basically I just treat it like a maze and keep going till I hit a dead end and return back and keep going adding all the solutions to the results array.

Discussion with another student:

I discussed this problem also with Rachel George. For this problem we took literally the

exact same approach and almost the exact same code. Instead of defining a function outside of the combinationsum method like I did, she instead defined a function called backtrack that was defined within the method. I actually found that this enhanced the readability when originally I thought while I was coding myself that it might detract from the readability. I thought it enhanced it though because after declaring the backtrack she then called it right below and it overall just made it quicker to understand while I think mine might take a little longer even though our code is pretty much the same.

3. 1584. Min Cost to Connect All Points

```
class Solution(object):
    def minCostConnectPoints(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """

        # Greedy for MST using prims algorithm
        n = len(points)
        mst = set()
        cost = [float('inf')] * n # inital distances to INF
        cost[0] = 0 # starting point (could be any point)
        total_cost = 0
        min_heap = [(0, 0)] # cost, point_index

        while min_heap and len(mst) < n:
            current_cost, u = heapq.heappop(min_heap)
            if u in mst:
                continue

            # Add the selected vertex to the MST
            total_cost += current_cost
            mst.add(u)

            # update the min distances for the rest of the points
            for v in range(n):
                if v not in mst:
                    dist = manhattan_dist(points[u], points[v])
                    if dist < cost[v]:
                        cost[v] = dist
                        heapq.heappush(min_heap, (dist, v))

        return total_cost
```

Explanation:

- The time complexity of this function is O(N^2logN), where N is the number of points. This is because, in the worst case, each point is extracted from the min-heap, which takes O(logN), and for each point it potentially needs to look at all other N points to update the

cost, lead to N * O(NlogN) making the overall time complexity O(N^2logN).

- The space complexity of this function is O(N). All of the data structures like the set and the queue used in this algorithm store a linear amount of elements relative to the input.
- The technique used here is a Greedy algorithm for MST(Minimum Spanning Tree) using Prim's algorithm. Prims algorithm is a greedy approach that build the MST by selecting edges with the least weight that add a new vertex to the existing MST.

Discussion with another student:

I discussed my solution with Max Forsey. He took the same approach as I did using prim's algorithm so we have the same time and space complexity. We do have a slight difference though which is that he used a edge_dict to keep track of all of the edges which may provide a slight optimization by ensuring each potential edge is only considered once for the priority queue. Overall though not really that much of difference between our algorithms except that slight optimization and perhaps he has some slightly additional space complexity since he is storing that extra dictionary for the edges taken.

4. 120. Triangle

```
class Solution(object):
    def minimumTotal(self, triangle):
        """
        :type triangle: List[List[int]]
        :rtype: int
        """

        # dynamic programming solution
        dp = [row[:] for row in triangle]
        n = len(dp)

        # start from the second to last row and move upwards
        for row in range(n - 2, -1, -1):
            for col in range(len(dp[row])):
                # choose the smaller of the two adjacent numbers on the row below
                # add it to the current cell
                dp[row][col] += min(dp[row+1][col], dp[row+1][col+1])

        print(dp)
        return dp[0][0]
```

Explanation:

- The time complexity of this function is O(n^2), where n is the number of rows in the triangle. This is because we have a double loop with the outer loop running for n iterations which is the number of rows, and the inner loop can run up to n iterations as well which is the number of elements in a row. This gives the function an overall time complexity of O(n^2).
- The space complexity of this function is also O(n^2), because it is creating a deep copy of the triangle with the same dimensions to store the intermediate sums in 'dp'. The

triangle itself has a total of (n+1) / 2 elements simplifying to O(n^2) space complexity.
Discussion with another student:

I discussed my solution to this problem with Kaiden Williams. He had the same dynamic programming approach but instead of making a deep copy of the triangle like I did, he directly modifies the triangle. I chose to make a deep copy because I often get paranoid about editing the original thing passed to me (too many times where that got me trouble in a class) but overall we basically had the same algorithm. Our time complexity is the same, but our space complexity differs since Kaiden is directly editing the triangle while I make a deep copy. So his space complexity is better than mine. So our solution mostly comes down to what the best practice is whether that's directly editing the triangle or not.

5. 279. Perfect Squares

```
class Solution(object):
    def numSquares(self, n):
        """
        :type n: int
        :rtype: int
        """
        # Dynamic programming solution
        # Time Complexity: O(n * sqrt(n))
        # Space Complexity: O(n)

        #intialization
        dp = [float('inf')] * (n+1)
        dp[0] = 0

        # calc all perfect squares less than or equal to n
        perfect_squares = []
        for i in range(1, int(n**0.5)+1):
            perfect_squares.append(i**2)

        # Fill in the dp
        for i in range(1, n+1):
            for square in perfect_squares:
                if i >= square:
                    dp[i] = min(dp[i], dp[i - square]+ 1)
                else:
                    break
        return dp[n]

        # example
        # Index:  0  1  2  3  4  5  6  7  8  9  10 11 12
#         dp:    [0, 1, 2, 3, 1, 2, 3, 4, 2, 1,  2, 3, 3]
```

Explanation:

- The time complexity of this function is O(n * sqrt(n)). This is because the first loop calculates all possible perfect squares that are less than or equal to square root of i. Then the next set of loops is bounded by the square root of i since we only consider the square numbers. This makes it so the inner loop runs at most sqrt(n) times for each i. Making the overall time complexity of this algorithm O(n * sqrt(n)).
- The space complexity of this function is O(n) because of the dp array which contains n+1 elements, where n is the number of squares.
- The technique used here is dynamic programming. This employs a bottom-up approach to fill out the dp array where each d[i] represents the minimum numbers of perfect squares that sum up to i. This makes it so it incrementally builds the solution for n using previously computed results for smaller numbers. In the end dp[n] contains the minimum number of perfect squares that sum to 'n'

Discussion with another student:

I discussed my solution with Collin Jordan. He took a different approach to this problem than I did. Instead of taking a dynamic programming approach, he took a backtracking recursive approach. He had a helper function that was called recusiveHelper which took care of the all the calculations and recursion. Basically each recursive call his function would try and see how many perfect squares it could fit in that sum up to the target but do not exceed it. Everytime it found a shorter combination the current min would be updated. After recursing through all possible combinations the shortest combination is returned as the solution. We discussed how this solution works but for larger values of n that go beyond the limit of this problem which was 1001 the efficiency would lack. The dynamic programming approach ensures that even for large values of n that time complexity stays polynomial. We also discussed how the recursion can lead to a large call stack potentially leading to a stack overflow for larger values of n.

6. 207. Course Schedule

```
class Solution(object):
    def canFinish(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: bool
        """
        # Depth First Seach solution to detect if a graph has any cycles
        # representng the course preqs with a directed graph
        # adjcency list

        graph = [[] for i in range(numCourses)]
        for course, prereq in prerequisites:
            graph[course].append(prereq)

        visited = [0] * numCourses
        # 0: unvisted, 1: currently visting, 2: fully explored
        # depth first search to detect if there is any cycles
```

```
        def dfs(course):
            if visited[course] == 1: # found a cycle
                return False
            if visited[course] == 2: # fully explored this node
                return True

            visited[course] = 1 # Mark our current node as visted(being explored)
            for prereq in graph[course]:
                if not dfs(prereq):
                    return False
            visited[course] = 2 # Mark as Fully explored
            return True

        # if dfs returns false for any course, return false
        for course in range(numCourses):
            if visited[course] == 0: # unvisted
                if not dfs(course):
                    return False
        return True
```

Explanation:

- The time complexity of this function is O(V+E) where V is the number of courses (vertices in the graph) and E is the number of prerequisites(edges in the graph). This is because in the worst case the algorithm needs to visit every vertex and edge once.
- The space complexity of this function is O(V+E). This is due to the storage of the graph with is O(E) for the edges and O(V) for the visited array. The recursive call stack of the DFS can also go up to O(V) in the worst case(if the graph is a deep linear chain).
- The technique used here is DFS for cycle detection in a directed graph. Basically it does a depth first search down the graph and if at any point we end up visiting a node more than once that means we have a cycle so it returns false the moment it detects a cycle.

Discussion with another student:

I discussed my solution with Keenan Faulkner. He introduced me to what several other students did, which is Kahn's algorithm. Khan's algorithm is a method for finding the topological ordering of a directed acyclic graph. Its a way of linearly ordering the vertices such that for every directed edge from vertex 'u' to vertex 'v', 'u' comes before 'v' in the ordering. It uses a queue and checks for cycles thus giving us a free cycle detection in the process. I thought this approach was interesting and could provide more detail for somebody if they were looking for an approach to this problem that also enabled the user to see what prerequisites they need to take in order as well as just telling them if it's possible to even accomplish all these preq's.

7. 207. Two Sum

```
class Solution(object):
    def twoSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        # Enumerate the array to keep track of og indices, then sort it
        # O(nlogn) --> sort
        nums = sorted(enumerate(nums), key=lambda x: x[1])

        # O(n)
        left, right = 0, len(nums) - 1
        while left < right:
            current_sum = nums[left][1] + nums[right][1]
            if current_sum == target:
                return [nums[left][0], nums[right][0]]
            elif current_sum < target:
                left += 1
            else:
                right -= 1

        return []
```

Explanation:

- The time complexity of this function is O(nlogn) where n is the number of numbers in the nums array. This complexity is due to the sorted of the nums array at the beginning of the method. This makes it possible for a sort of greedy approach using two pointers on both ends of the array and returning a solution the moment we see it. This works because the problem says there can only be 1 solution. The loop itself runs in O(n) time since it's just iterating over each element in the array of length n, but due to the sorting its O(nlogn).
- The space complexity of this function is O(1) because it isn't storing any additional data structures. You could say the space complexity might be O(n) since we sort the nums array, but that depends on what we are counting for the space complexity. In this case I am not counting the nums array since we are sorting it in place and we aren't creating any additional data structures.

Overall takeaways

My overall takeaway from this lab is that greedy algorithms and dynamic programming basically can solve anything… No I'm just kidding but the majority of my solutions were either a dynamic programming approach or a greedy approach. I think this encapsulates the entire philosophy of these two problem solving approaches. With greedy algorithms the goal is often just to throw something at it, oftentimes companies and people will "throw" a greedy algorithm at a problem just to get started or even leave the greedy algorithm as it is for the solution. This reminded me of the traveling salesperson lab where we started off with the greedy algorithm to get us started. This is because they are very readable, easy to understand, and can often give us a good starting point for a complicated problem. My greedy algorithm greatly reduced the amount of computations in my branch and bound giving it a good BST to start, and greatly increased the amount of pruning that took place since the BST was already pretty decent if not the optimal solution.

Dynamic programming I think has a similar effect as well where the goal is to break down the problem into trivial sub problems that we can build up to solve the big problem. Its also usually very easy to understand and readable. I've started to notice how readability and understandability are becoming more and more of a priority when it comes to employers looking for programmers. I keep hearing more and more that employers are more focused on people who can achieve simple solutions and be able to explain and back up what they do. Trying to utilize greedy and dynamic approaches in this lab really showed me how these two approaches can be super good tools to try to achieve that.