**Peers I discussed with:**

1. Dallin
2. Andrew
3. Ben
4. Takun
5. Koby

**Conclusion:**

The first thing I learned from this assignment is that LeetCode makes you pay to use their debugger, and their web based IDE offers no coding insights whatsoever, and is completely useless. Dynamic programming was super helpful in solving problems. On a more serious note, I really learned how to depend on myself for projects. I learned that sometimes the simple solution is the best, and that often when a complex problem needs to be solved, you can either spend a lot of time coding and debugging and making a big section of code that’s convoluted but gets the job done, OR, you can spend a lot of time thinking about the problem, writing out pseudocode, looking at possible solutions and trying poke holes in them, and then write 12 lines of code that work without question. Coding nonsense is hard. Thinking critically and using algorithms is hard. **Choose your hard**.

**Analysis (Questions 2 & 3)**

| **#** | **Problem** | **My analysis** | **Peer discussion** |
| --- | --- | --- | --- |
| 1 | N-th Tribonacci Number | Time   * Time complexity is O(n), (where n is the number you’re interested in) since all you have to do is go through each element up to n and compare the last 3 elements. The largest chunk of code is iterating over the array of size n, so O(n)   Space   * Space complexity is O(n), because you only need a single array of size n. Simply as that.   Tools used   * I used dynamic programming for this one, by storing the past results in an array, and using those past elements to calculate future results | Discussed with: Dallin  Dallin and I used the same method to solve this problem, and had nearly the same source code. The main difference between our code was that in creating the array, Dallin created an array of max int size, while I created an array only of the size of the number + 1. This made it so that his space complexity was larger than it needed to be. I suggested he change it to just be an array of size n + 1, and he agreed that was a good idea |
| 2 | Two Sum | Time   * O(n2) for time. This is because for each number in the list, I iterate over each number in the list (besides those behind it in order) and see if the two added together equals the target number. Despite my attempt to lower the number by not looking again at elements behind the current element, the complexity is still O(n2)   Space   * Space complexity is just the size of the list passed in, so O(n). This is because no elements are added to the array, and no other variables or values are created with space greater than the array of size n   Tools used   * I used the naive solution in this case, because it worked very well and was easy to code. I just iterated over all possible combinations and returned the one that added to the right value | Discussed with: None |
| 3 | Combination Sum | Time   * Time complexity is O(n \* m), where n is the size of the candidates list and m is the target number. This is because for each element in the list of candidates, each element could be selected any number of times up to m times. This is because the most number of times an element could be selected would be a 1, up to m times. This is of course worst case though.   Space   * The space complexity would be O(n \* m2) where n is the size of the candidates list and m is the target number. This is because we create a tree of size n \* m, and each node’s value is an array up to size m to hold the combination   Tools used   * For combination sum, I used recursion to create a sequence tree (to save space) and added each element that added to the right target to an array of combinations, and then returned that list of combinations. This didn’t really satisfy any of the problem solving tools we learned in class, but I supposed you could call it naive of greedy | Discussed with: None |
| 4 | Min Cost to Connect All Points | Time   * Time complexity is O(n2) where n is the number of points given. This is because we have to create a list of all possible edges between each node. So each node connected to each other node is O(n2)   Space   * Space complexity is O(n2), because we have a list of size n2. We also have two other lists each of at worst size n. But these are dwarfed by the n2 size array, so they’re not considered   Tools used   * I used Kruskal’s algorithm to solve this since Kruskal’s is great for creating a minimum spanning tree | Discussed with: Andrew  Andrew decided to use Prim’s algorithm rather than Kruskal’s like I used. This was a great idea though, because Prim’s is better suited for more dense graphs with more edges than nodes. Since each node technically could have an edge to each other node, there were n2 edges, compared to n nodes. This made the graph very dense, and made Kruskal’s slower, because Kruskal’s sorts ALL edges, and has to go through them one by one, while Prim’s looks at a smaller subset of edges and decides amongst those |
| 5 | Binary Tree Level-order Traversal | Time   * Time complexity is O(n) where n is the number of nodes in the tree. This is because both when creating the augmented tree, and when iterating over it to get the traversal order, iteration is done recursively, so each node is visited only once   Space   * Space complexity is O(n) where n is the number of nodes in the tree. This is because two trees of size n are created, and each node has a constant number of elements   Tools used   * I used an interesting one for this problem. I recursively went through the graph and added a depth variable to each node. This made it very simple to go through each level of the graph by comparing the depth value of each node, and ensure I didn’t get any out of place. This didn’t really satisfy any of the problem solving tools we learned in class, but I supposed you could call it naive of greedy | Discussed with: None |
| 6 | Number of Provinces | Time   * Time complexity is (where n is the number of cities in the given matrix) O(n2) at worst, though the average case is closer to O(n). This is because for each city, it could have to add each other city to the priority queue.   Space   * Space complexity of this problem is O(n2). This is because we are given a matrix of size n \* n, and the queue and array that we make as we solve this problem are only of size n at most. This means that the matrix outweighs everything else, and the space complexity is O(n2)   Tools used   * On this problem, I used a queue to go through each city and all the cities it was connected to. As soon as it could visit no other cities, I incremented the number of provinces and moved to the next unvisited node. This didn’t really satisfy any of the problem solving tools we learned in class, but I supposed you could call it naive of greedy | Discussed with: None |
| 7 | Triangle | Time   * Time complexity is O(n2), where n is the depth of the triangle. This is because we have to create an n\*n matrix to hold all possible paths down the triangle.   Space   * Space complexity is also O(n2), for the same reason. We have a n\*n matrix where n is the depth of the triangle. This matrix is used to hold all path values, which are just ints.   Tools used   * I used dynamic programming so hard on this one. Literally almost copied the gene sequencing lab. Created a 2d matrix and solved the problems line by line, then iterated over the bottom row and return ed the lowest value | Discussed with: Andrew  Andrew used dynamic programming as well, but he did it slightly differently than me. He chose to start at the bottom and for each node, look at its two children, and add the lowest cost one to the value of that node, and work his way up from there, arriving at the lowest value by the time he got to the top. Realistically, this is slightly faster than mine, since he didn’t have to take a pass of time O(n) to look at the bottom row and select the lowest value. It was also easier to code, as he didn’t have to convert his edges to an n x n matrix, and better space complexity of n instead of n2 |
| 8 | Perfect Squares | Time   * Time complexity of this one is O(n2). This is because we have to iterate over each number in our stored list of squared numbers for each number leading up to n, the target number. So we iterate over a list of size n, n times.   Space   * Space is O(n), because we store two lists, both of size n. One is the list of squared numbers up to n, and the other is a list of the optimized sizes for each number up to n. Both of size n, so O(n)   Tools used   * I also used dynamic programming on this one. This was essentially repeating knapsack, so I just stored the value of each best combination in each index of an array, and then checked back farther in the array each time, and either put that in for the element at the current index, or improved it | Discussed with: Takun  Takun loves his recursion. He built a tree of all possible solutions recursively and then returned the path with the lowest number of nodes. This was pretty big in time and space complexity, but ultimately resulted in the right answer. We discussed how my solution was a bit faster as well and why |
| 9 | Flower Planting With No Adjacent | Time   * Time complexity is O(n), where n is the number of gardens we need to connect. This is because for each garden, there are at most 3 neighbors it is connected to. Because this is a constant number, we don’t consider it when we loop over each garden and assign it a different flower than any of its neighbors. Therefore the largest complexity is going over all gardens, making this problem’s time complexity O(n)   Space   * Space complexity is the same as time complexity, O(n). This is because we have a list of gardens, each with a constant number of elements. This is the largest thing we create, so O(n)   Tools used   * This one was all the way greedy. I went over each flower garden and assigned the first flower that wasn’t the same as one of its neighbors. Surprisingly, it worked really well, was quick, and easy to code! | Discussed with: None |
| 10 | Course Schedule | Time   * The time complexity of this one is O(n \* m), where n is the number of courses, and m is the length of the list of preRequisites. This is because for each class, we have to go over each preRequisite for that class, and in the WORST case scenario, each class could have each other class as a preRequisite. This would lead to a O(n \* m)   Space   * Space complexity is the same for the same reason. O(n \* m) because you have an array with numCourses elements, which could each have an array of size len(preRequisites) as an element.   Tools used   * This one was tough. I tried to use DFS, but couldn’t get that to work. I worked for a really long time before needing more than a hint. I deliberately didn’t want to just copy someone else’s answer, and instead went looking at algorithms that were optimized for finding cycles in graphs (because I knew that was the problem I was really trying to solve). I found Kahn’s algorithm and made sure to learn it fully myself before implementing it. It involves finding the node with the least number of incoming edges and adding it to a priority queue, and then raking those nodes off one by one | Discussed with: Koby  Koby used a really cool method to solve this problem. Originally, I was trying to do it the same way by creating a tree and then doing a DFS on the tree, and if you reached a visited node, then return that there’s a cycle. But I could never get mine to work, so I switched to something else. Koby solved the problem by adding a “visited” Boolean, and a “stillBeingVisited” Boolean, to track his progress through different trees in the forest. This meant that if he reached a node that had been visited, it was okay, it had been visited on a previous loop. But if he reached a node that had been visited AND was stillBeingVisited was true, then he was in a loop, and the courses could not be completed. Very clever |
| 11 | Queue Reconstruction by Height | Time   * The time complexity of this algorithm is O(n2), where n is the number of people in the queue. This is because you have to iterate over each position in the queue (sort of, actually less than that) for each person in the queue. The sorting algorithm used is O(nlogn), so it’s dwarfed by the n2   Space   * Space complexity is O(n) where n is the number of people in the queue. This is because you have two arrays of size n. One to hold the people (each with a constant number of elements), and one to hold the reconstructed queue   Tools used   * This one was super cool! I didn’t use a tool from class, other than the mindset that I’ve been taught to think in by this class! I recognized a pattern. The shortest person will always be at the index equal to the number of people taller than them. So by sorting the people by height, and then negatively by number of people ahead of them, I could count the number of empty indices ahead of them, and insert them at the next available empty index | Discussed with: None |
| 12 | Lowest Common Ancestor of a Binary Tree | Time   * Time complexity of this one is O(n), where n is the number nodes in the tree. This is because we have to convert the tree into an augmented tree by adding a depth and parent element to each node. We do this recursively, so each node is only ever visited once. This is the greatest complexity in the problem   Space   * The space complexity is the same, because we end with 3 trees of size n. One is the original, the next is the augmented one, and then we have to create a deAugmented tree as well. Since these are all the same complexity (each node has a constant number of elements), the space complexity is O(n)   Tools used   * I solved this problem the same way as Binary Tree Level-order Traversal. Technically I didn’t use any direct algorithms from class, but I used my noggin. I looked and realized that if I knew both the parents and depth of each node, I could flip the graph and start at the lowest given node and go up until it met the same level as the other given node, and then move both up one at a time until they met, and that would be the common ancestor. I guess again it’s greedy or naive if you wanted to put a name on it | Discussed with: Ben  Ben and I did the same thing with this problem. He didn’t have his source code for this problem (LeetCode didn’t let him see his Windows code from his Mac which is weird?), so he couldn’t tell me how he reversed the graph, but I explained to him how I converted each node into an augmented node with a depth counter and a parent node pointer. We both then flipped the graph, and followed the children up their parents until they both met at a shared node, and returned that as the answer. We weren’t able to compare time complexity, but mine was likely less time and space efficient since I had to create a second tree of size n, and use more time converting the tree as well. |

**Source Code (Question 1)**

| **#** | **Problem** | **Code** |
| --- | --- | --- |
| 1 | N-th Tribonacci Number | class Solution {  public:  int tribonacci(int n) {  int solutions[n + 1];  solutions[0] = 0;  solutions[1]= 1;  solutions[2] = 1;  for(int i = 3; i <= n; i++){  solutions[i] = solutions[i - 3] + solutions[i - 2] + solutions[i - 1];  }  return solutions[n];  }  }; |
| 2 | Two Sum | class Solution:  def twoSum(self, nums: List[int], target: int) -> List[int]:  listSize = len(nums)  for i in range(listSize):  for j in range(i + 1, listSize):  currentNum = nums[i] + nums[j]  if currentNum == target:  return [i, j] |
| 3 | Combination Sum | import copy  from typing import List  class TreeNode:  def \_\_init\_\_(self, value, parent):  self.value = value  self.parent = parent  self.children = []  class Solution:  def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:  def createTree():  root = TreeNode([], None)  recursCreateNode(root)  return root  def recursCreateNode(node):  if sum(node.value) == target:  return  for nextNum in candidates:  if sum(node.value) + nextNum <= target:  newValue = copy.copy(node.value)  newValue.append(nextNum)  newNode = TreeNode(newValue, node)  node.children.append(newNode)  recursCreateNode(newNode)  def recursCreateCombos(node):  if sum(node.value) == target:  combinations.add(tuple(sorted(node.value)))  for child in node.children:  recursCreateCombos(child)  def getConvertedSetOfTuples():  convertedCombos = []  for combination in combinations:  combinationList = []  for number in combination:  combinationList.append(number)  convertedCombos.append(combinationList)  return convertedCombos  combinations = set()  root = createTree()  recursCreateCombos(root)  return getConvertedSetOfTuples() |
| 4 | Min Cost to Connect All Points | class Solution:  def minCostConnectPoints(self, points: List[List[int]]) -> int:  edges, connectedPoints = self.createEdgesAndPointsInLists(points)  totalCost = 0  while len(connectedPoints) > 1:  edge = edges.pop(0)  edgesConnected, point1Index, point2Index = self.arePointsConnected(edge, connectedPoints)  if not edgesConnected:  connectedPoints[point1Index].extend(connectedPoints[point2Index])  connectedPoints.pop(point2Index)  totalCost += edge.distance  return totalCost  def arePointsConnected(self, edge, connectedPoints):  point1 = edge.point1  point2 = edge.point2  point1Index = None  point2Index = None    for index in range(len(connectedPoints)):  pointGroup = connectedPoints[index]  if point1 in pointGroup:  point1Index = index  if point2Index != None:  break  if point2 in pointGroup:  point2Index = index  if point1Index != None:  break  return point1Index == point2Index, point1Index, point2Index  def createEdgesAndPointsInLists(self, points: List[List[int]]):  edges = set()  pointsInLists = []  pointsSize = len(points)  for i in range(pointsSize):  pointsInLists.append([points[i]])  for j in range(i + 1, pointsSize):  edges.add(Edge(self.getManhattanDistance(points[i], points[j]), points[i], points[j]))  return sorted(edges, key=lambda x: x.distance), pointsInLists  def getManhattanDistance(self, fromPoint, toPoint):  return abs((fromPoint[0] - toPoint[0])) + abs((fromPoint[1] - toPoint[1]))  class Edge:  def \_\_init\_\_(self, distance, point1, point2):  self.distance = distance  self.point1 = point1  self.point2 = point2 |
| 5 | Binary Tree Level-order Traversal | class TreeNode:  def \_\_init\_\_(self, val=0, left=None, right=None):  self.val = val  self.left = left  self.right = right  class AugmentedTreeNode(TreeNode):  def \_\_init\_\_(self, x, parent, depth):  super().\_\_init\_\_(x)  self.val = x  self.left = None  self.right = None  self.depth = depth  self.parent = parent  class Solution:  def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:  def recursAddToArray(node):  if node == None:  return  if (len(levelOrder) - 1) < node.depth:  levelOrder.append([node.val])  else:  levelOrder[node.depth].append(node.val)  recursAddToArray(node.left)  recursAddToArray(node.right)  levelOrder = []  augRoot = self.createAugementedBinaryTree(root)  recursAddToArray(augRoot)  return levelOrder  def createAugementedBinaryTree(self, root):  augmentedTreeRoot = self.dfsAssignParentAndDepth(root, None, 0)  return augmentedTreeRoot  def dfsAssignParentAndDepth(self, node, parentNode, depth):  if(node == None):  return None  newAugmentedNode = AugmentedTreeNode(node.val, parentNode, depth)  newAugmentedNode.left = self.dfsAssignParentAndDepth(node.left, newAugmentedNode, depth + 1)  newAugmentedNode.right = self.dfsAssignParentAndDepth(node.right, newAugmentedNode, depth + 1)  return newAugmentedNode  def build\_binary\_tree(arr):  if not arr:  return None  nodes = [TreeNode(val) if val is not None else None for val in arr]  for i in range(len(arr)):  if nodes[i] is not None:  left\_child\_idx = 2 \* i + 1  right\_child\_idx = 2 \* i + 2  if left\_child\_idx < len(arr):  nodes[i].left = nodes[left\_child\_idx]  if right\_child\_idx < len(arr):  nodes[i].right = nodes[right\_child\_idx]  return nodes[0]  def find\_node(root, val):  if root is None or root.val == val:  return root  left\_result = find\_node(root.left, val)  if left\_result:  return left\_result  return find\_node(root.right, val) |
| 6 | Number of Provinces | from typing import List  class Solution:  def findCircleNum(self, matrix: List[List[int]]) -> int:  notVisited = [i for i in range(len(matrix))]  q = set()  numProvinces = 0  while len(notVisited) > 0:  q.add(notVisited[0])  while len(q) > 0:  fromCity = q.pop()  notVisited.remove(fromCity)  for toCity in range(len(matrix)):  if fromCity == toCity:  continue  if toCity in notVisited and matrix[fromCity][toCity] == 1:  q.add(toCity)  numProvinces += 1  return numProvinces |
| 7 | Triangle | import math  from typing import List  class Solution:  def minimumTotal(self, triangle: List[List[int]]) -> int:  paths = self.makePathsTable(triangle)  self.dpItUp(paths, triangle)  return self.findLowestCost(paths)  def makePathsTable(self, triangle: List[List[int]]):  maxSize = len(triangle)  paths = [[math.inf for \_ in range(maxSize)] for \_ in range(maxSize)]  paths[0][0] = triangle[0][0]  for index in range(1, maxSize):  # Always left  paths[index][0] = triangle[index][0] + paths[index - 1][0]  #Always right  paths[0][index] = triangle[index][len(triangle[index]) - 1] + paths[0][index - 1]  return paths  def dpItUp(self, paths, triangle):  maxSize = len(triangle)  for i in range(1, maxSize - 1):  for j in range (1, maxSize - i):  if paths[i - 1][j] < paths[i][j - 1]:  previousPath = paths[i - 1][j]  else:  previousPath = paths[i][j - 1]  currentPositionInTriangleCost = triangle[i + j][j]  paths[i][j] = previousPath + currentPositionInTriangleCost # Straight up  def findLowestCost(self, paths):  lowestCost = math.inf  i = len(paths) - 1  j = 0  while j < len(paths):  currentCost = paths[i][j]  if currentCost < lowestCost:  lowestCost = currentCost  j += 1  i -= 1  return lowestCost |
| 8 | Perfect Squares | import math  import sys  from math import sqrt  class Solution:  def numSquares(self, n: int) -> int:  MAX\_INT = sys.maxsize  squaredNumbers = [i\*i for i in range(math.ceil(sqrt(n)) + 1)]  optimized = [MAX\_INT for \_ in range(n + 1)]  for i in range(n + 1):  for number in squaredNumbers:  if number == i:  optimized[i] = 1  elif number < i:  optimizedPastNumber = optimized[i - number] + optimized[number]  if optimizedPastNumber < optimized[i]:  optimized[i] = optimizedPastNumber  return optimized[n] |
| 9 | Flower Planting With No Adjacent | from typing import List  class Garden:  def \_\_init\_\_(self):  self.neighbors = []  self.flower = -1  class Solution:  def gardenNoAdj(self, n: int, paths: List[List[int]]) -> List[int]:  graph = self.makeGraph(paths, n)  for garden in graph:  flowers = [f for f in range(4)]  for neighbor in garden.neighbors:  neighborFlower = graph[neighbor].flower  if neighborFlower != -1 and neighborFlower in flowers:  flowers.remove(graph[neighbor].flower)  garden.flower = flowers[0]  return [(graph[i].flower + 1) for i in range(n)]  def makeGraph(self, paths, n):  graph = [Garden() for \_ in range(n)]  for edge in paths:  g1 = edge[0] - 1  g2 = edge[1] - 1  graph[g1].neighbors.append(g2)  graph[g2].neighbors.append(g1)  return graph |
| 10 | Course Schedule | class Solution:  def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:  graph = self.createGraph(numCourses, prerequisites)  inDegreeList = self.createInDegreeList(graph, numCourses)  q = deque()  for index in range(len(inDegreeList)):  if inDegreeList[index] == 0:  q.append(index)  visitedIndex = 0  visited = [None for \_ in range(numCourses)]  while len(q) != 0:  currentCourseIndex = q.pop()  visited[visitedIndex] = currentCourseIndex  visitedIndex += 1  for childIndex in graph[currentCourseIndex]:  inDegreeList[childIndex] -= 1  if inDegreeList[childIndex] == 0:  q.append(childIndex)  if visited[len(visited) - 1] == None:  return False  else:  return True  def createInDegreeList(self, graph, numCourses):  inDegreeList = [0 for \_ in range(numCourses)]  for index in range(numCourses):  for preReq in graph[index]:  inDegreeList[preReq] += 1  return inDegreeList  def createGraph(self, numCourses, prerequisites):  graph = {i: [] for i in range(numCourses)}  for preReq in prerequisites:  graph[preReq[1]].append(preReq[0])  return graph |
| 11 | Queue Reconstruction by Height | class Solution:  def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:  peopleSorted = sorted(people, key=lambda x: (x[0], -x[1]))  reconstructedList = [None for \_ in range(len(peopleSorted))]  for index in range(len(peopleSorted)):  currentPerson = peopleSorted[index]  numEmpty = 0  for spotIndex in range(len(reconstructedList)):  if reconstructedList[spotIndex] == None:  if numEmpty == currentPerson[1]:  reconstructedList[spotIndex] = currentPerson  break  else:  numEmpty += 1  return reconstructedList |
| 12 | Lowest Common Ancestor of a Binary Tree | # Definition for a binary tree node.  class TreeNode:  def \_\_init\_\_(self, x):  self.val = x  self.left = None  self.right = None  class AugmentedTreeNode(TreeNode):  def \_\_init\_\_(self, x, parent, depth):  super().\_\_init\_\_(x)  self.val = x  self.left = None  self.right = None  self.depth = depth  self.parent = parent  class Solution:  def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':  augmentedTreeRoot = self.createAugementedBinaryTree(root)  LCAAugmentedNode = self.findLCA(augmentedTreeRoot, self.find\_node(augmentedTreeRoot, p.val), self.find\_node(augmentedTreeRoot, q.val))  return self.deAugmentNode(LCAAugmentedNode)  def findLCA(self, root: AugmentedTreeNode, pNode: AugmentedTreeNode, qNode: AugmentedTreeNode):  if pNode.depth < qNode.depth:  higherNode = pNode  lowerNode = qNode  else:  higherNode = qNode  lowerNode = pNode  while lowerNode.depth != higherNode.depth:  lowerNode = lowerNode.parent  if lowerNode == higherNode:  return higherNode  while lowerNode != higherNode:  lowerNode = lowerNode.parent  higherNode = higherNode.parent  return higherNode # Doesn't matter which is returned, lower or higher are same here  def createAugementedBinaryTree(self, root):  augmentedTreeRoot = self.dfsAssignParentAndDepth(root, None, 0)  return augmentedTreeRoot  def dfsAssignParentAndDepth(self, node, parentNode, depth):  if(node == None):  return None  newAugmentedNode = AugmentedTreeNode(node.val, parentNode, depth)  newAugmentedNode.left = self.dfsAssignParentAndDepth(node.left, newAugmentedNode, depth + 1)  newAugmentedNode.right = self.dfsAssignParentAndDepth(node.right, newAugmentedNode, depth + 1)  return newAugmentedNode  def deAugmentNode(self, node):  if node == None:  return None  normalNode = TreeNode(node.val)  normalNode.right = self.deAugmentNode(node.right)  normalNode.left = self.deAugmentNode(node.left)  return normalNode  def find\_node(self, root, val):  if root is None or root.val == val:  return root  left\_result = self.find\_node(root.left, val)  if left\_result:  return left\_result  return self.find\_node(root.right, val) |