Tribonacci Number

```
class Solution:
    def tribonacci(self, n: int) -> int:
        seq = [0, 1, 1]
        curr_pos = 3

        while curr_pos <= n:
            seq.append(
                seq[curr_pos - 3] +
                seq[curr_pos - 2] +
                seq[curr_pos - 1]
            )
            curr_pos += 1
        return seq[n]
```

The way I solved the problem was very simple, I had a list that I started out with putting values `[0, 1, 1]` then I used append to add the sum of the previous 3 values onto the end of my list until the N'th value was reached, then returned. The time and space complexity for my algorithm would be O(n). The

When discussing this problem with my classmate Tristan, we spoke about how he did a very similar setup as me, but instead he used 3 variables that were shuffled around and overwritten as the computation proceeded. This had the advantage of being much more memory efficient at the slight expense of computational efficiency due to the swapping around of values.

Triangle

```
class Solution:
    def minimumTotal(self, triangle: list[list[int]]) -> int:
        self.triangle = triangle
        n = len(triangle)
        for i in range(n - 2, -1, -1):
            for j in range(len(triangle[i])):
                triangle[i][j] += min(triangle[i + 1][j], triangle[i + 1][j
+ 1])
        return triangle[0][0]
```

I solved this problem by starting at the bottom level and determining the lowest possible cost for each of the items in the level above the bottom level. I then overwrote those values and used them for the calculation for the row above that one. The determining of the time complexity is a

bit complicated for this one due to the bottom rows taking O(n) calculations and the top taking closer to O(1), if n is defined as the depth/height of the triangle. If we instead define n to be the total number of input items, then the time and space complexity of this algorithm are both O(n).

I talked to my classmate Porter, and he solved it in a very similar way, starting at the bottom and going up, but his implementation used a set of 2 rows that were modified starting at the bottom, then going upwards. In a situation if the input is passed in by reference and needed later (ie. I can't change the original input because it is being used elsewhere), then for my implementation there would be the requirement of copying the entire 2d array. That would make my implementation be slightly less computationally and memory efficient then his.

Combination Sum

```
class Solution:
    def combinationSum(self, candidates: list[int], target: int) ->
list[list[int]]:
        nums = set(candidates)

        # Make an entry for a solution at each number leading up to the
solution
        solutions: list[list[int]] = [[] for _ in range(target + 1)]
        solutions[0] = [[]]

        for currtarget in range(1, target + 1):
            for num in candidates:
                if currtarget - num >= 0: # If the current try is greater
than half of num
                    for comb in solutions[currtarget - num]:
                        if not comb or num >= comb[-1]:
                            solutions[currtarget].append(comb + [num])

        return solutions[target]
```

For the combination sum problem, I went with a dynamic programming approach where I have a list of solutions for each number from 1 to the target. Each iteration I go through all of the possible combinations up to the target number, then when I am calculating combinations, I consider all of the sub numbers and their combinations as well. Because of the dynamic programming aspect of the algorithm, calculating the time complexity is a little difficult, but since there is a triple nested for loop, the worst case complexity time is $O(n^3)$. For the space complexity, since the results are being stored in a 2d array, the space complexity, depending on the problem and the sub problems, should be roughly $O(n^2)$.

Course Schedule

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: list[list[int]]) ->
bool:
        self.matrix = self.buildTree(numCourses, prerequisites)
        self.coursesPossible = set()
        isPossible = True
        indexes = list(range(len(self.matrix)))
        random.shuffle(indexes)

        for currCourseIndex in indexes:
            print(f"Now checking {currCourseIndex}")
            isPossible = self.iterative_dfs(currCourseIndex)
            if not isPossible:
                return False
            self.coursesPossible.add(currCourseIndex)
        return True

    def buildTree(self, numCourses: int, prerequisites: list[list[int]]) ->
list[list[int]]:
        matrix = []

        row = []
        for i in range(numCourses):
            row.append(0)

        for i in range(numCourses):
            matrix.append(row.copy())

        for prereqSet in prerequisites:
            course = prereqSet[0]
            prereq = prereqSet[1]
            matrix[course][prereq] = 1

        return matrix

    def iterative_dfs(self, startCourseIndex: int) -> bool:
        stack = [(startCourseIndex, 0, {startCourseIndex})]

        while stack:
            currCourseIndex, child_index, currentPath = stack.pop()

            for prereqIndex in range(child_index,
len(self.matrix[currCourseIndex])):
                if self.matrix[currCourseIndex][prereqIndex] == 0:
                    continue
                if currCourseIndex == prereqIndex:
```

```
                    return False # this course requires itself as a prereq

                # That means that if one of the prerequsites is
                # required in the above tree, this will fail
                # because it is a circular dependencie
                if prereqIndex in currentPath:
                    return False

                # If the course has already been checked and verified,
                # then skip checking it again
                if prereqIndex in self.coursesPossible:
                    continue

                stack.append((currCourseIndex, prereqIndex + 1,
currentPath))

                newPath = currentPath.copy()
                newPath.add(prereqIndex)
                stack.append((prereqIndex, 0, newPath))
                break
            self.coursesPossible.add(currCourseIndex)
        return True
```

This problem took me many hours to complete, and many different iterations and optimizations to pass all of the test cases. My final iteration is shown above. My solution was to first convert the input information into a useful format, and chose a 2d array because I used a very similar system for Project 5 and felt the most familiar in this format. I then use a recursive DFS search using a stack to solve for each of the prerequisites. Then for extra speed, I used dynamic programming by caching an already solved for value into the `coursesPossible` set.

Because of the dynamic programming, the exact time and space complexity are not very easy to calculate. The `iterative_dfs` function in the worst case should have a complexity of no more than $O(n^2)$, and that will have to run in a loop n times, so the overall time complexity should be $O(n^3)$. The space complexity is determined by my storage method, and since that is a 2d array, the space complexity is $O(n^2)$.

For the Course Schedule problem I talked to Jed, who did a similar approach to me by using a DFS to be able to search all the possibilities. The one main difference though is that he used a dictionary to hold the input values rather than a 2d array. This should have resulted in a much faster lookup time in certain instances due to the hash table as opposed to a normal list/array.

Lowest Common Ancestor of a Binary Tree

```python
class Solution:
    def lowestCommonAncestor(
        self,
        root: 'TreeNode',
        p: 'TreeNode',
        q: 'TreeNode'
    ) -> 'TreeNode':

        p_path = self.get_node_path(root, p.val)
        q_path = self.get_node_path(root, q.val)

        if q.val in p_path:
            return q

        if p.val in q_path:
            return p

        if len(p_path) >= len(q_path):
            for i in range(len(q_path)):
                if q_path[i] in p_path:
                    return TreeNode(q_path[i])
        else:
            for i in range(len(p_path)):
                if p_path[i] in q_path:
                    return TreeNode(p_path[i])

    def get_node_path(self, curr_tree: TreeNode, val: int) -> list[int]:

        if curr_tree.val == val:
            return [val]

        if curr_tree.left == None:
            if curr_tree.right == None: # Handle if is leaf
                return None

            if curr_tree.right.val == val:
                return [val, curr_tree.val]

            right = self.recurse(curr_tree.right, val)
            if right == None:
                return None
            right.append(curr_tree.val)
            return right

        if curr_tree.right == None:
            if curr_tree.left.val == val:
```

```
                return [val, curr_tree.val]

            left = self.recurse(curr_tree.left, val)
            if left == None:
                return None
            left.append(curr_tree.val)
            return left

        if curr_tree.left.val == val or curr_tree.right.val == val:
            return [val, curr_tree.val]

        left = self.recurse(curr_tree.left, val)
        right = self.recurse(curr_tree.right, val)

        if left == None:
            if right == None:
                return None
            right.append(curr_tree.val)
            return right
        left.append(curr_tree.val)
        return left
```

This problem I solved by doing a depth first search to find the path for each node, then doing an iterative comparison to efficiently find which node both of the selected nodes shared. The most complex part of the algorithm, the DFS search means that this algorithm has a time complexity of O(n) with a worst case scenario space complexity of O(n) as well.

One other way to solve this problem would be a flatting of the tree data structure to an array representation of the binary tree. In this case, the tree could be traversed a single time, and each time it would be O(n/2) complexity to find the path it takes to get from the root node to the desired node. That could have slightly decreased the time complexity at the expense of increasing the space complexity by having to store the entire tree separately.

Two Sum

```
class Solution:
    def twoSum(self, nums: list[int], target: int) -> list[int]:
        for i in range(len(nums)):
            for j in range(len(nums)):
                if i == j:
                    continue
                if nums[i] + nums[j] == target:
                    return [i, j]
```

My solution for two sum is quite simple, I use a nested for loop to be able to try every possible combination of two numbers. This has the advantage of being an exhaustive search, but the disadvantage of potentially double-checking already checked combinations. The time complexity of my algorithm is $O(n^2)$ and the space complexity is O(1) because there is no data being stored, only being compared until it is returned.

When talking this over with a classmate, I can't remember exactly what they did, but I believe that they used subtraction, similar to the combination sum problem above. This way for their algorithm, the list `nums` only needs to be looped over once, and the first time one of the numbers subtracted from the target is another number in the list, then the algorithm is able to return a value. With this system, the time complexity would be improved from my $O(n^2)$ to O(n).