Source Code:

```
def tribonacci(n: int) -> int:

        sums:list[int] = []

        for i in range(n+1):
            if len(sums) == 0:
                sums.append(0)
            elif len(sums) == 1 or len(sums) == 2:
                sums.append(1)
            else:
                sums.append(sums[i-3] + sums[i-2] + sums[i-1])

        return sums[n]
```

```
def twoSum(nums: list[int], target: int) -> list[int]:
    ans = []
    for i in range(len(nums)):
        for j in range(len(nums)):

            if i == j:
                continue

            else:
                if nums[i] + nums [j] == target:
                    ans.append(i)
                    ans.append(j)
                    return ans
    return ans
```

```
def minimumTotal(triangle: list[list[int]]) -> int:
    for row in range(len(triangle) - 2, -1, -1):
        for col in range(len(triangle[row])):

            triangle[row][col] += min(triangle[row + 1][col], triangle[row + 1][col +
1])

    return triangle[0][0]
```

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

    ans:list[list[int]] = []
    queue:deque = deque([root])

    while queue:

        lvl_len:int = len(queue)
        current_lvl:list = []

        for _ in range(lvl_len):
            node:TreeNode = queue.popleft()
            current_lvl.append(node.val)

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

        ans.append(current_lvl)

    return ans
```

```
def combinationSum(candidates, target):
    result = []
    divide_and_conquer(0, target, [], candidates, result)
    return result

def divide_and_conquer(start, target, path:list, candidates, result:list):

        if target == 0:
            result.append(path[:])
            return

        for i in range(start, len(candidates)):

            if candidates[i] > target:
                continue

            path.append(candidates[i])
            divide_and_conquer(i, target - candidates[i], path, candidates, result)

            path.pop()
```

```python
def findCircleNum(isConnected: list[list[int]]) -> int:

    def dfs(city:int)->None:
        for neighbor in range(len(isConnected)):

            if isConnected[city][neighbor] == 1 and neighbor not in visited:

                visited.add(neighbor)
                dfs(neighbor)

    visited:set = set()
    ans:int = 0

    for city in range(len(isConnected)):
        if city not in visited:
            ans +=1
            visited.add(city)
            dfs(city)

    return ans
```

Student Discussion and Complexity Analysis:

- Nth Tribonacci (Talked with Micah)
    - We all had very similar code and found that the approach to use was implementing an array to hold the previous sums. This had very good time and space complexity
    - My time complexity is O(n) because I only have to visit each item in the input string once. I do this by manipulating a list. I add each sum into a new list and just index into it. My space complexity is also O(n) because I don't store more than the input within my algorithm
- Two Sum (I met with Kaisha)
    - My partner and I both struggled this one. We ended up just using a brute force algorithm to find the best solution.
    - Due to the brute for nature of my algorithm, the time ended up being O(n^2) because I have the double for loop going through the input data. The space

complexity could have been better. My initial thought of the algorithm was such that I thought I was going to have to keep track of a list of answers, but as I was coding I realized I just needed the first answer that would work. My initial thought led me to appending to a list, which increased my space complexity a little; however, the Big-O is still O(1) because I will only add a constant number of data points (i.e. 1) to a list before I return if an answer is found.

- Combination Sum (I met with Carson)
    - The best approach is to treat it like a divide and conquer algorithm. We both split up the problem using a DSF algorithm and was able to have really good time complexity.
- Binary Tree Level Order Traversal (I met with Kade)
    - We both implemented a queue and used a while loop to iterate through all the parent children's combinations until our queue ran out.
    - The space and time complexity are both O(n). The time because I just had to iterate through all the points in the input and add them and their children to the appropriate list. This was done through a queue, so I ran until it was out. The queue was the length of the input, thus making it the only additional data structure I had to build, and it was the size of my input.
- Number of Provinces (Met with Kaisha again)
    - We both implemented a DFS algorithm. My partner was genius and used a binary list that tracked if a city was connected. I used a set, and her memory and time complexity were better than mine.
    - The time complexity is just O(n^2) because you have to visit each node and then check all the index length of the input to check if the other nodes are connected
- Triangle (Didn't catch her name)
    - We used two different approaches; I used a linear programming approach, and she used a DFS approach. I believe that the linear programming was less efficient because I had to look at more memory than her code.
    - My time complexity is O(n^2) because I have two for loops that look at everything in input string. My space complexity is actually really low because I don't have any extra data structures holding the data. I just modify the list of lists inside its self and move on to the next rows and columns.