Leetcode

I discussed all of the problems with Spencer Hodson and Cameron Coltrin

Tribonacci

```
namespace LeetCode.Tribonacci;

public class TribonacciSolution {

    public int Tribonacci(int n) {

        List<int> solution = new List<int> { 0, 1, 1};
        for (int i = 3; i <= n; i++) {
            // Console.WriteLine(i);
            solution.Add(solution[i-1] + solution[i-2] + solution[i-3]);
        }

        return solution[n];
    }
}
```

My Tribonacci solution uses a dynamic programming approach that works in linear time; O(n). The simple for loop iterates through the past solutions to get the next answer. This is basically the same algorithm that we discussed in class for fibbonacci months ago. The space complexity is also O(n) as it stores the solution for each step in the sequence.

Both of my classmates used this dynamic programming approach and had the same complexities. The only difference is that spencer didn’t initialize the array with 0,1 and 1 and used a helper method based upon the index to determine which value should be returned initially.

Two Sum

```
namespace LeetCode.TwoSum;

public class TwoSumSolution
{
    public int[] TwoSum(int[] nums, int target)
    {
        Dictionary<int, int> map = new();
        for (int i = 0; i < nums.Length; i++)
        {
            int op = target - nums[i];
            if (map.ContainsKey(op))
            {
                return new int[] { map[op], i };
            }
            map[nums[i]] = i;
        }
        return new int[] { -1, -1 };
    }
}
```

For the two sum solution, I used a dictionary that recorded the opposite of the current number being tested. I then checked to see if a solution had already been found for the number and if it did, then returned that solution. This runs in an O(n) time by only iterating over the array once. It has a space complexity of O(n) as it also records the dictionary of each index.

Spencer and Cameron had very similar solutions with a dictionary being used to key the index to its compliment.

Combination Sum

```
namespace LeetCode.CombinationSum
{
    public class CombinationSumSolution
    {
        public IList<IList<int>> CombinationSum(int[] candidates, int target)
        {
            IList<IList<int>> result = new List<IList<int>>();
            IList<int> current = new List<int>();
            Backtrack(candidates, target, 0, current, result);
            return result;
        }

        private void Backtrack(int[] candidates, int target, int start,
IList<int> current, IList<IList<int>> result)
        {
            if (target == 0)
            {
                result.Add(new List<int>(current));
                return;
            }

            for (int i = start; i < candidates.Length; i++)
            {
                if (candidates[i] <= target)
                {
                    current.Add(candidates[i]);
                    Backtrack(candidates, target - candidates[i], i, current,
result);
                    current.RemoveAt(current.Count - 1);
                }
            }
        }
    }
}
```

The combination sum is a similar problem to the two sum, but it can have repetition and multiple values. In discussing with Spencer and Cameron, this is basically a spin off of the knapsack problem with repetition. My solution uses a branch and bound algorithm to recurse through solution paths and prune them if they end up exceeding the sum (going below zero in my case). Since this is a branch and bound, the time complexity can vary widely depending on how the data is initially structured, but at a worst case of having to recurse through all the decision paths, it would be O(2^n). As for space complexity, the only thing being used is the list of candidates, but there is also the recursive call stack. O(n).

All of our solutions used the same general principle but we implemented them differently. Spencer and Cameron both used a heap, whereas I used a List that I kept adding and removing elements to.

Additionally, I used a recursive call to search through the paths, whereas the others used an iterative approach. Spencer also went backwards and added the sums instead of subtracting from the target.

Triangle

```
namespace LeetCode.Triangle;

public class TriangleSolution
{
    public int MinimumTotal(IList<IList<int>> triangle)
    {
        List<List<int>> minPaths = new List<List<int>>();
        for (int i = 0; i < triangle.Count; i++)
        {
            minPaths.Add(new List<int>());
            for (int j = 0; j < triangle[i].Count; j++)
            {
                if (i == 0)
                {
                    minPaths[i].Add(triangle[i][j]);
                }
                else
                {
                    if (j == 0)
                    {
                        minPaths[i].Add(triangle[i][j] + minPaths[i - 1][j]);
                    }
                    else if (j == triangle[i].Count - 1)
                    {
                        minPaths[i].Add(triangle[i][j] + minPaths[i - 1][j - 1]);
                    }
                    else
                    {
                        minPaths[i].Add(triangle[i][j] + Math.Min(minPaths[i -
1][j - 1], minPaths[i - 1][j]));
                    }
                }
            }
        }

        return minPaths[triangle.Count - 1].Min();
    }
}
```

For the triangle problem, I used a solution that calculated the cumulative cost of each path down the triangle. This algorithm calculates the cost of all possibilities which is a complexity of O(n^2). I store the individual path cost decisions in another array which for each branch also ends up being O(n^2)

We discussed how even though this feels like a greedy algorithm, it isn't and gets an optimal solution every time. We all had similar implementations of the cumulative sum algorithm.

Queue Reconstruction

```
namespace LeetCode.QueueReconstruction;

public class QueueReconstructionSolution
{
    public int[][] ReconstructQueue(int[][] people)
    {
        Array.Sort(people, (a, b) =>
        {
            if (a[0] == b[0])
            {
                return a[1] - b[1];
            }
            return b[0] - a[0];
        });

        List<int[]> result = new List<int[]>();
        foreach (int[] p in people)
        {
            result.Insert(p[1], p);
        }

        return result.ToArray();
    }
}
```

This was quite a fun problem and had an interesting solution. In a way, the array gets sorted twice. At first we sort the array in descending order in height and ascending of people in front that are taller. After the initial sort, we iterate through that sorted list and insert each person into their sorted spot by the calculation of where they are sorted by height and the amount of people in front of them. The C# sort algorithm uses a combination of a quicksort, heapsort, and insertion sort, to achieve a complexity of O(nlogn). Additionally, we add the second iterative sort at the end which gives a final complexity of O(n^2logn).

While I used a list for this solution, Cameron used a heap queue that allocated space in the line for taller people and as taller people were found during sorting, knocked down the allocated space.

Min Cost Points:

```
namespace LeetCode.MinCostPoints;

public class MinCostPointsSolution
{
    public int MinCostConnectPoints(int[][] points)
    {
        int n = points.Length;
        int[] dist = new int[n];
        Array.Fill(dist, int.MaxValue);
        dist[0] = 0;
        bool[] visited = new bool[n];
        int result = 0;

        for (int i = 0; i < n; i++)
        {
            int minIndex = -1;
            int minDist = int.MaxValue;
            for (int j = 0; j < n; j++)
            {
                if (!visited[j] && dist[j] < minDist)
                {
                    minDist = dist[j];
                    minIndex = j;
                }
            }

            visited[minIndex] = true;
            result += minDist;

            for (int j = 0; j < n; j++)
            {
                int d = Math.Abs(points[minIndex][0] - points[j][0]) +
Math.Abs(points[minIndex][1] - points[j][1]);
                if (!visited[j] && d < dist[j])
                {
                    dist[j] = d;
                }
            }
        }

        return result;
    }
}
```

The premise of the min cost points problem is creating a minimum spanning tree of an undirected graph. For this problem, I used Prim’s algorithm to iteratively add nodes to the graph in the most efficient manner. This gives a time complexity of O(n^2). As for space, we are using an adjacency matrix to store the distance values which also gives a space complexity of O(n^2).

Cameron used the different approach of using Kruskal's algorithm to find the MST instead of prims. He found his time complexity to be better, but actually ran slower due to the computational power to union all of the sets for each iteration. Because this is a dense graph, Prim's turned out to be the most efficient.

Perfect Squares:

```
namespace LeetCode.PerfectSquares;

public class PerfectSquaresSolution
{
    public int NumSquares(int n)
    {
        List<int> squares = new();
        for (int i = 1; i * i <= n; i++)
        {
            squares.Add(i * i);
        }

        List<int> subProblems = new() { 0 };

        for (int i = 1; i <= n; i++)
        {
            int min = int.MaxValue;
            foreach (int square in squares)
            {
                if (i - square >= 0)
                {
                    min = Math.Min(min, subProblems[i - square] + 1);
                }
            }
            subProblems.Add(min);
        }

        return subProblems[n];
    }
}
```

This is also a dynamic programming problem. To start, I initialized an array with all the squares leading up to n. Then for each value of n, I subtract the next biggest square and use previous solution to find that solution and so on. This has a complexity of O(n). The space complexity is simply O(n) as well to store the values of the subproblems.

Lowest Common Ancestor

```csharp
namespace LeetCode.LowestCommonAncestor;

public class TreeNode
{
    public int val { get; set; }
    public TreeNode? left { get; set; }
    public TreeNode? right { get; set; }
    public TreeNode(int val = 0, TreeNode? left = null, TreeNode? right = null)
    {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
public class LowestCommonAncestorSolution
{
    public TreeNode? LowestCommonAncestor(TreeNode? root, TreeNode p, TreeNode q)
    {
        if (root == null)
        {
            return null;
        }
        if (root == p || root == q)
        {
            return root;
        }

        TreeNode? left = LowestCommonAncestor(root.left, p, q);
        TreeNode? right = LowestCommonAncestor(root.right, p, q);

        if (left != null && right != null)
        {
            return root;
        }

        return left ?? right;
    }
}
```

To find the lowest common ancestor of nodes I did a depth first search. Starting at the root node, I traverse down the tree and test whether p and q are in the same subtree or different subtrees. From those base cases, I can determine which node is the lowest common ancestor. Each node is visited only once and so the complexity is O(n). The space used is dependent on the height of the tree because each height layer creates a new call on the stack.

I used a recursive approach for this search, but Cameron used an iterative approach to go through all of the nodes and record the paths of p and q and used the comparisons to find the solution.

Course Schedule

```
namespace LeetCode.CourseSchedule;

public class CourseScheduleSolution
{
    public bool CanFinish(int numCourses, int[][] prerequisites)
    {
        Dictionary<int, List<int>> graph = new();
        for (int i = 0; i < numCourses; i++)
        {
            graph[i] = new List<int>();
        }

        for (int i = 0; i < prerequisites.Length; i++)
        {
            graph[prerequisites[i][0]].Add(prerequisites[i][1]);
        }

        bool[] completed = new bool[numCourses];
        bool[] recStack = new bool[numCourses];

        for (int i = 0; i < numCourses; i++)
        {
            if (IsCyclic(graph, completed, recStack, i))
            {
                return false;
            }
        }

        return true;
    }

    private bool IsCyclic(Dictionary<int, List<int>> graph, bool[] completed,
bool[] recStack, int i)
    {
        if (recStack[i])
        {
            return true;
        }

        if (completed[i])
        {
            return false;
        }

        recStack[i] = true;
        completed[i] = true;

        foreach (int neighbor in graph[i])
        {
            if (IsCyclic(graph, completed, recStack, neighbor))
            {
                return true;
            }
        }

        recStack[i] = false;
        return false;
    }
}
```

Flower Planting

Provinces

```
namespace LeetCode.NumberOfProvinces;

public class NumberOfProvincesSolution
{
    public int FindCircleNum(int[][] isConnected)
    {
        int n = isConnected.Length;
        bool[] visited = new bool[n];
        int result = 0;

        for (int i = 0; i < n; i++)
        {
            if (!visited[i])
            {
                DFS(isConnected, visited, i);
                result++;
            }
        }

        return result;
    }

    private void DFS(int[][] isConnected, bool[] visited, int i)
    {
        visited[i] = true;
        for (int j = 0; j < isConnected.Length; j++)
        {
            if (isConnected[i][j] == 1 && !visited[j])
            {
                DFS(isConnected, visited, j);
            }
        }
    }
}
```

Binary Tree Traversal

```
namespace LeetCode.BinaryTreeTraversal;

public class TreeNode
{
    public int val;
    public TreeNode? left;
    public TreeNode? right;
    public TreeNode(int val = 0, TreeNode? left = null, TreeNode? right = null)
    {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

public class BinaryTreeOrderTraversalSolution
{
    public IList<IList<int>> LevelOrder(TreeNode root)
    {
        var result = new List<IList<int>>();
        if (root == null)
        {
            return result;
        }

        Queue<TreeNode> queue = new Queue<TreeNode>();
        queue.Enqueue(root);
        while (queue.Count > 0)
        {
            int levelSize = queue.Count;
            var currentLevel = new List<int>();
            for (int i = 0; i < levelSize; i++)
            {
                TreeNode current = queue.Dequeue();
                currentLevel.Add(current.val);
                if (current.left != null)
                {
                    queue.Enqueue(current.left);
                }
                if (current.right != null)
                {
                    queue.Enqueue(current.right);
                }
            }
            result.Add(currentLevel);
        }

        return result;
    }
}
```

Similar to the Lowest Common ancestor problem, the binary tree traversal uses a Breadth first search for a level order. I used a queue to keep track of each level and as I processed a node, added the subnodes to the queue. The time complexity is O(n) because each node is visited only once. The space complexity is O(n) as we create a copy of the tree as we traverse it.

Spencer and Cameron had similar answers and complexities. We analyzed why using a queue was the best option opposed to a list. A queue allows for easier management of the heights and order.