```
D:\SER501Assignment4>flake8 --max-complexity 10 D:\SER501Assignment4

D:\SER501Assignment4>python assignment_4.py
passed
passed

D:\SER501Assignment4>
```

Problem 1 Code:

```
def longest_ordered_subsequence(L):
    n = len(L)
    sortedArray = np.sort(L)
    rows = n + 1
    cols = n + 1
    table = []
    for i in range(rows):
        col = []
        for j in range(cols):
            col.append(0)
        table.append(col)
    for i in range(1, rows):
        for j in range(1, cols):
            if L[i - 1] == sortedArray[j - 1]:
                table[i][j] = 1 + table[i - 1][j - 1]
            else:
                table[i][j] = max(table[i - 1][j], table[i][j - 1])
    return table[n][n]
```

To find the longest ordered subsequence, one can use the longest common subsequence algorithm with the first list being the given list of numbers and the second list being an ordered version of the given list of numbers. The resulting common subsequence will be the same as the longest ordered subsequence.

Problem 2 Code:

```
def visitAdjacentSquare(i, j, waterIndices, m, n):
    if i < 0 or j < 0 or i > m - 1 or j > n - 1:
        return
    if waterIndices[i][j][0] == -1 or waterIndices[i][j][1]:
        return
    waterIndices[i][j][1] = True
    visitAdjacentSquare(i + 1, j - 1, waterIndices, m, n)
    visitAdjacentSquare(i, j - 1, waterIndices, m, n)
    visitAdjacentSquare(i - 1, j - 1, waterIndices, m, n)
    visitAdjacentSquare(i - 1, j, waterIndices, m, n)
    visitAdjacentSquare(i - 1, j + 1, waterIndices, m, n)
    visitAdjacentSquare(i, j + 1, waterIndices, m, n)
    visitAdjacentSquare(i + 1, j + 1, waterIndices, m, n)
    visitAdjacentSquare(i + 1, j, waterIndices, m, n)


def count_ponds(G):
    m = len(G)
    n = len(G[0])
    pondCount = 0
    waterIndices = []
    for i in range(m):
        col = []
        for j in range(n):
            if G[i][j] == "#":
                col.append([1, False])
            else:
                col.append([-1, False])
        waterIndices.append(col)
    for i in range(m):
        for j in range(n):
            if waterIndices[i][j][0] == 1 and not waterIndices[i][j][1]:
                visitAdjacentSquare(i, j, waterIndices, m, n)
                pondCount = pondCount + 1
    return pondCount
```

This solution creates a table that replicates the given NxM rectangle of water and land. This table indicates whether water elements have been examined. A double for loop then iterates over this table

to discover unvisited water elements. When an unvisited water element is discovered, a recursive function is activated. This recursive function visits adjacent unvisited water elements by calling itself eight times. Upon visiting all adjacent water elements, a pond is identified, so the pond count increases by 1.

Problem 3:

```
def supermarket(Items):
    n = len(Items)
    time = 0
    for i in range(n):
        if Items[i][1] > time:
            time = Items[i][1]
    rows = time + 1
    cols = n + 1
    table = []
    for i in range(rows):
        col = []
        for j in range(cols):
            col.append([0, i])
        table.append(col)
    for j in range(1, cols):
        for i in range(1, rows):
            if table[i][j][1] <= i:
                if (Items[j - 1][0] +
                        table[i - table[i][j][1]][j - 1][0]
                        > table[i][j - 1][0]):
                    table[i][j][0] = (Items[j - 1][0] +
                                      table[i - table[i][j][1]][j - 1][0])
                else:
                    table[i][j][0] = table[i][j - 1][0]
            else:
                table[i][j][0] = table[i][j - 1][0]
    return table[rows - 1][cols - 1][0]
```

Unfortunately, this implementation is not entirely finished. This problem is very similar to the 0-1 Knapsack problem in that certain benefits result after considering a constraint that affects the entire solution. While the Knapsack problem featured weight as its constraint, the Supermarket problem

features time as its constraint. To make this algorithm work correctly, the implementation must consider not only the amount of time available, but what items are unable to be sold as time elapses. A greedy approach is also capable of correctly providing optimal solutions to the sample inputs in the assignment. If each product were ordered in ascending order by the earliest deadline and in descending order by most profitable product, then an algorithm could iterate over this ordered list, pick the first instance of the most profitable item with the shortest deadline, and add that to the total profit. This would produce the sample outputs of 80 with the first input and 185 with the second.