Problem 1: Solve a problem using top-down and bottom-up approaches of Dynamic Programming technique

1. Explain how your top-down approach different from the bottom-up approach?

The top-down approach breaks down the larger problem and solves the subproblems recursively; a memoization table is used to store the results of subproblems. The bottom-up approach breaks down the problem but solves each subproblem iteratively; it starts with the smallest subproblem and builds up the solution for the larger problem. A table ('cache') is filled up in a bottom-up manner.

2. What is the time complexity and Space complexity using Top-down Approach

Time complexity: O(m*n)
Space complexity: O(m*n)

3. What is the time complexity and Space complexity using Bottom-up Approach
   Time complexity: O(m*n)
   Space complexity: O(m*n)

4. Write the subproblem and recurrence formula for your approach. If the top down and bottom-up approaches have the subproblem recurrence formula you may write it only once, if not write for each one separately.
   Subproblem: The length of the longest common subsequence (LCS) of the prefixes of the DNA strings, DNA1[0, i] and DNA2[0, j].

   Top-down recurrence formula:
   - if i < 0 or j < 0: return 0
   - if (i, j) in memo: return memo [(i, j)]
   - if DNA1[i] == DNA2[j]: memo[(i, j)] = 1 + lcs(i - 1, j - 1)
   - Else: memo[(i, j)] = max(lcs(i - 1, j), lcs(i, j - 1))

   Bottom-up recurrence formula:
   - If i==0 or j==0: cache[i][j] = 0
   - If DNA1[i-1] == DNA2[j-1]: cache[i][j] = cache[i-1][j-1] + 1
   - Else: cache[i][j]= max(cache[i-1][j] , cache[i][j-1])

Problem 2: Solve Dynamic Programming Problem and Compare with Naïve approach

You are playing a puzzle. A random number N is given, you have blocks of length 1 unit and 2 units. You need to arrange the blocks back to back such that you get a total length of N units. In how many distinct ways can you arrange the blocks for given N.

a. Write a description/pseudocode of approach to solve it using Dynamic Programming paradigm (either top-down or bottom-up approach)

Bottom-up approach:

```
def bottomup_count(n):
cache = 0 for x in range(n + 1)

For i from 0 to n:
        If i == 0 or i ==1:
        Cache[i] = 1
        Else:
        cache[i] = cache[i-1] + cache[i-2]
```

b. Write pseudocode/description for the brute force approach

Brute approach:

```
def brute_count(n):
    If n < 0:
    Return 0
    If n == 0:
        Return 1
    Else:
    Return brute_count(n-1) + brute_count(n-2)
```

c. Compare the time complexity of both the approaches

Bottom-up approach: O(n)

Brute approach: O(2^n)

d. Write the recurrence formula for the problem

- If n = 0, f(n) = 1
- Else: f(n) = f(n - 1) + f(n - 2)