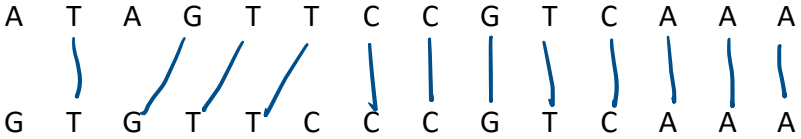
Assignment: Dynamic Programming

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

   DNA sequence is made of characters A, C, G and T, which represent nucleotides. A sample DNA string can be given as 'ACCGTTTAAAG'. Finding similarities between two DNA sequences is a critical computation problem that is solved in bioinformatics.
   Given two DNA strings find the length of the longest common string alignment between them (it need not be continuous). Assume empty string does not match with anything.
   Example:
   DNA string1: ATAGTTCCGTCAAA
   DNA string2: GTGTTCCCGTCAAA



   Length the best continuous length of the DNA string alignment: 12 (TGTTCCGTCAAA)

   a. Implement a solution to this problem using Top-down Approach of Dynamic Programming, name your function **dna_match_topdown(DNA1, DNA2)**

```
def dna_match_topdown(DNA1, DNA2):
    str1 = DNA1
    str2 = DNA2
    m = len(str1)
    n = len(str2)
    cache = [[-1 for x in range(n+1)] for x in range(m+1)]
    return topdown_helper(str1,str2, m, n, cache)

def topdown_helper(s1, s2, m, n, cache):
    #base case
    if(m == 0 or n == 0):
        return 0

    #If the subproblem already computed return it
    if(cache[m][n] != -1): return cache[m][n]

    #if the chars match, store result
    if(s1[m-1] == s2[n-1]):
        cache[m][n] = 1 + topdown_helper(s1, s2, m-1, n-1, cache)
        return cache[m][n]

    else:
        cache[m][n] = max(topdown_helper(s1, s2, m-1, n ,cache) ,
topdown_helper(s1, s2, m, n-1, cache))
        return cache[m][n]

print(dna_match_topdown("ATAGTTCCGTCAAA", "GTGTTCCCGTCAAA"))
```

b. Implement a solution to this problem using Bottom-up Approach of Dynamic Programming, name your function **dna_match_bottomup(DNA1, DNA2)**

```
def dna_match_bottomup(DNA1, DNA2):
    str1 = DNA1
    str2 = DNA2
    m = len(str1)
    n = len(str2)

    # create a 2-D dynamic programming table of size m+1 X n+1
    cache = [[0 for x in range(n+1)] for x in range(m+1)]

    # building the matrix
    for i in range(m+1):
        for j in range(n+1):
            if i==0 or j==0:
                cache[i][j] = 0
            elif str1[i-1] == str2[j-1]:
                cache[i][j] = cache[i-1][j-1] + 1
            else:
                cache[i][j]= max(cache[i-1][j] , cache[i][j-1])

    return cache[m][n]


print(dna_match_bottomup("ATAGTTCCGTCAAA", "GTGTTCCCGTCAAA"))
```

c. Explain how your top-down approach different from the bottom-up approach?
The top down approach first tries to solve the problem of sized n and then solves for dependent subproblems until the base case is reached. The bottom-up approach solves the problem starting at size 1 and then leads to the solution of the full problem in the end.

d. What is the time complexity and Space complexity using Top-down Approach.
Here we are accessing 2-D table, hence the time complexity will be O(m*n), where m = len(str1), n = len(str2). This is the worst-case time complexity.The program needs an additional space of O(m*n) to store the 2-dimensional array.

e. What is the time complexity and Space complexity using Bottom-up Approach
Here we are working with a matrix of size len(str1) * len(str2). Hence the time complexity will be O(m*n), where m = len(str1), n = len(str2).

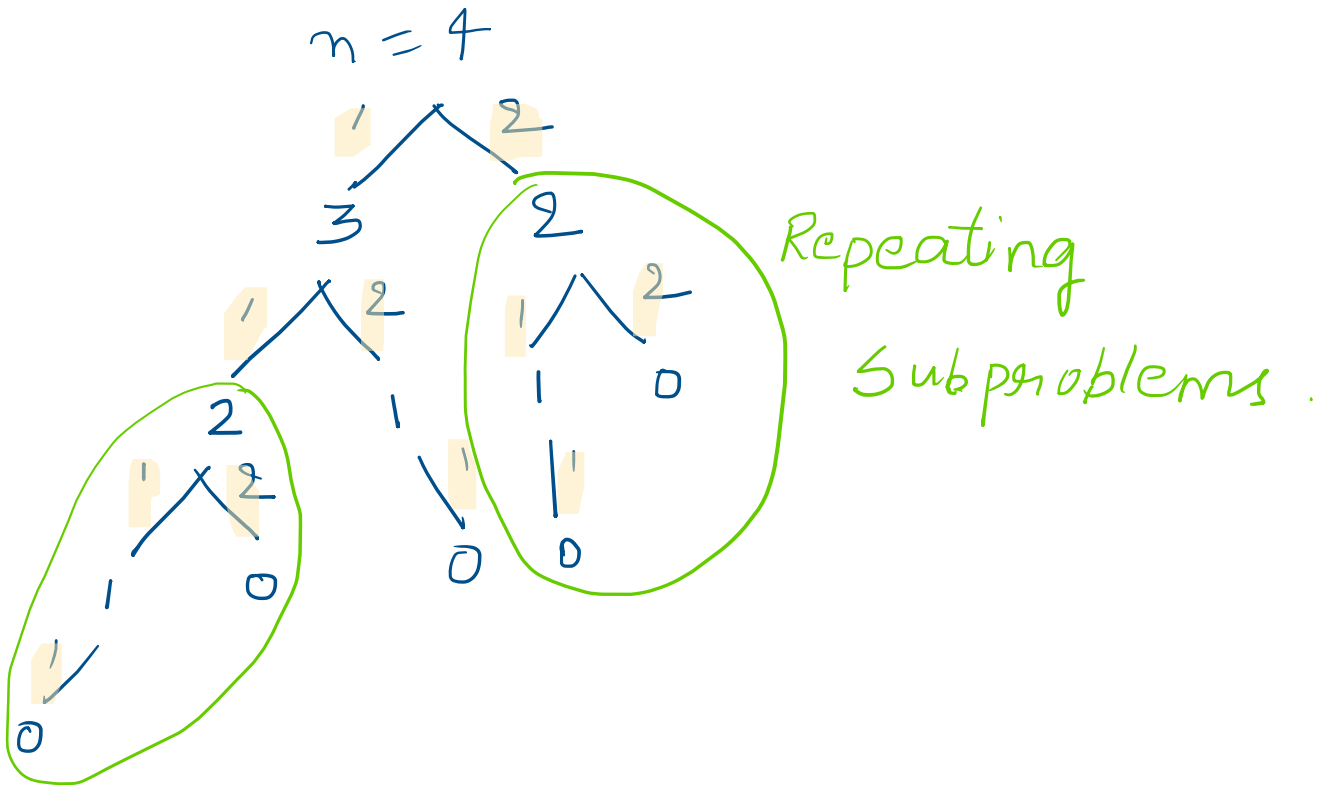
The program needs an additional space of O(m*n) to store the 2-dimensional table.

f. 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: F(i,j): Gives the length of the longest matching continuous sequence between first i characters and the first j characters of DNA1 and DNA2
Recurrence Formula:
F(i,j) = 0 if i=0 or j=0
F(i,j) = 1 + F(i,j) if the first character of DNA1(length i) and DNA2(length j) match

F(i,j) = max { F(i,j-1), F(i-1,j)} if the first character of DNA1(length i) and DNA2(length j) Do not match.

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)

We have repeating subproblems as shown in the figure below. We can cache these use the dynamic programming technique to find the result.



Following is code for bottom-up approach

```
def blockpuzzle_dp(n):
    if n==1: return 1

    memo = [0]*(n+1)
    memo[1] = 1
    memo[2] = 2
    for i in range(3, n+1):
        memo[i] = memo[i-1] + memo[i-2]

    return memo[n]
```

   b. Write pseudocode for the brute force approach

```
def blockpuzzle_bruteforce(n):
    if (n <= 0):
        return 0
    if (n == 1):
        return 1
    if(n ==2):
        return 2
    return blockpuzzle_bruteforce(n-1) + blockpuzzle_bruteforce(n-2)
```

c. Compare the time complexity of both the approaches

Brute force approach has time complexity of theta(2^n) (If we draw recursion tree: it will have n levels and at level k 2^k amount of work will be done)

The dynamic programming approach has time complexity of theta(n).

The dynamic programming approach solves this problem extremely faster compared to the brute force approach.

d. Write the recurrence formula for the problem

F(n) = 0 for n = 0

F(n) = 1 for n = 1

F(n) = 2 for n = 2

F(n) = F(n-1) + F(n-2) for all values of n