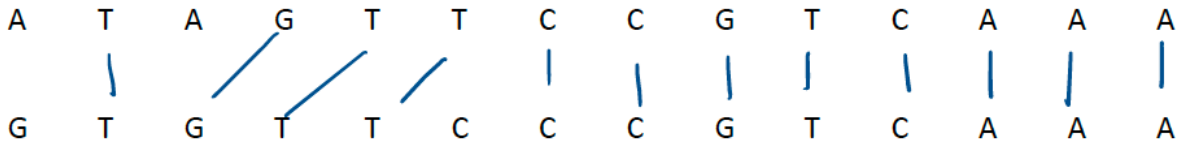
Assignment: Dynamic Programming

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 of 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):**

```python
def dna_match_topdown(DNA1, DNA2):
    # initialize cache
    cache = [[-1 for j in range(len(DNA2))] for i in range(len(DNA1))] #only filling up cells of cache we need
    # call recursive helper function
    return lcs_topdown(DNA1, DNA2, len(DNA1)-1, len(DNA2)-1, cache)

def lcs_topdown(DNA1, DNA2, i, j, cache):
    # base case: one of the strings is empty
    if i < 0 or j < 0:
        return 0
    # check if already computed and return value from cache
    if cache[i][j] != -1:
        return cache[i][j]
    # if last characters match, add 1 to LCS and recur for remaining strings
    if DNA1[i] == DNA2[j]:
        cache[i][j] = 1 + lcs_topdown(DNA1, DNA2, i-1, j-1, cache)
    else:
        # find LCS by excluding last character of DNA1 and last character of DNA2
        # take maximum of two possible ways
        cache[i][j] = max(lcs_topdown(DNA1, DNA2, i-1, j, cache), lcs_topdown(DNA1, DNA2, i, j-1, cache))
    return cache[i][j]
```

b. Implement a solution to this problem using Bottom-up Approach of Dynamic Programming, name your function **dna_match_bottomup(DNA1, DNA2)**
Write implementation of a & b in a single python file, name your file DNAMatch.py
Do not write the functions in a Python class.

```
def dna_match_bottomup(DNA1, DNA2):
    # initialize cache
    cache = [[0 for j in range(len(DNA2)+1)] for i in range(len(DNA1)+1)] #filling up all cells of
cache
    # fill cache bottom-up
    for i in range(1, len(DNA1)+1):
        for j in range(1, len(DNA2)+1):
            if DNA1[i-1] == DNA2[j-1]:
                cache[i][j] = 1 + cache[i-1][j-1]
            else:
                cache[i][j] = max(cache[i-1][j], cache[i][j-1])
    # return length of LCS
    return cache[len(DNA1)][len(DNA2)]
```

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

1. Approach: The top-down approach uses recursion and memoization while the bottom-up approach uses iteration and dynamic programming.
2. Implementation: The top-down approach requires a recursive function to compute the LCS, while the bottom-up approach uses a nested loop (iteration) to fill a table with the computed values.
3. Cache initialization: The top-down approach initializes the cache table with -1 to represent values that have not yet been computed, while the bottom-up approach initializes the table with 0 to represent the base case.
4. Subproblem Solved: The subproblem definition is the same for both approaches - finding the LCS between two substrings of the original strings. However, the specific subproblems that are solved may differ depending on the approach.
5. **Main difference:** The main difference between the two approaches is their approach to solving subproblems. The top-down approach starts by solving the larger subproblems and recursively breaks them down into smaller subproblems until the base case is reached. The bottom-up approach starts with the smallest subproblems and iteratively builds up to the larger subproblems.

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

Time complexity: O(mn)

Space complexity: : O(mn)

The time complexity of the top-down approach is O(m*n), where m and n are the lengths of the input strings, since each subproblem is solved only once and the results are cached.*

*The space complexity is also O(m*n), as the cache needs to store the results of *all possible(n*m)* subproblems.

e. What is the time complexity and Space complexity using Bottom-up Approach

Time complexity: : O(mn)

Space complexity: : O(mn)

The time complexity of the bottom-up approach is also O(*(nm), as each subproblem is solved only once.*

The space complexity is O(mn), as the cache needs to store the results of *all possible(n*m)* subproblems.

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: Given two strings of lengths i and j of DNA1 and DNA2 respectively, what is the length of the longest common subsequence between them?

Recurrence formula:

LCS(i,j) = 0	if i=0 or j=0
LCS (i,j) = 1 + LCS (i-1,j-1)	if 1st char of DNA1(len i) & DNA2(len j) match
LCS (i,j) = max { LCS (i,j-1), LCS (i-1,j)}	if 1st char of DNA1(len i) & DNA2(len 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 & 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.

Example 1:

Input: N=2, Result: 2
Explanation: There are two ways. (1+1 , 2)

Example 2:

Input: N=3, Result: 3
Explanation: There are three ways (1+1+1, 1+2, 2+1)

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

Approach to solve using Dynamic Programming (bottom-up approach):

```
function arrangeBlocksDP(N):
#create an array cache of size (N+1) & initialize it w/ zeros
#cache[i] represents the number of ways to arrange the blocks for len i

#base cases to handle lengths 0-2:
    cache[0] = 1 #base case – only 1 way to arrange blocks for len 0
    cache[1] = 1 #base case - only 1 way to arrange blocks for len 0
    cache[2] = 2 #base case – 2 ways: (1+1 , 2)

#handling sizes 3 or bigger

    for i from 3 to N:
        cache[i] = cache[i-1] + cache[i-2]
       #cache[i-1] represents the num of ways to arrange the blocks of
       len 1 at the end
       #cache[i-2] represents the num of ways to arrange the blocks of
       len 2 at the end

    return cache[N]
```

b. Write pseudocode/description for the brute force approach

```
function countWays(N):
    if N == 0:    #base case
        return 1
    elif N == 1:  #base case
        return 1:
    elif N == 2:  #base case
        return 1:
    else:         #recursion
        return countWays(N-1) + countWays(N-2)
```

c. Compare the time complexity of both the approaches

The brute force approach has a time complexity of $O(2^N)$, since it recursively computes all possible arrangements of the blocks, which doubles for every increment in N. This is because for each block, there are two options - it can be placed horizontally (1 unit) or vertically (2 units), and we need to explore all possible combinations of these options.

On the other hand, the dynamic programming approach has a time complexity of O(N), since it fills up the cache[] array with precomputed values for each value of i up to N, using the recurrence formula `cache [i] = cache [i-1] + cache [i-2]`. This means that we only need to compute each value of `cache [i]` once, and can use the precomputed values for all subsequent calculations.

Therefore, the dynamic programming approach is much more efficient than the brute force approach for larger values of N.

d. Write the recurrence formula for the problem

```
cache[i]                                          if i = 0 or i = 1
cache[i]                                          if i = 2
cache[i] = cache[i-1] + cache[i-2]                if i =>3

countWays(N) = 1                                  if N = 0 or N =1
countWays(N) = 2                                  if N = 2
countWays(N) = countWays(N-1) + countWays(N-2)    if N =>3
```

We can simply write it as:

Recurrence formula:

$F(N) = 1$ if $N = 0$ or $N=1$

$F(N) = 2$ if $N = 2$

$F(N) = F(N-1) + F(N-2)$ if $N =>3$

Here, `F(N)` represents the number of distinct ways to arrange the blocks to get a length of `N`.

We can compute `F(N)` using the values of `F(N-1)` and `F(N-2)`, which represent the number of distinct ways to arrange the blocks to get a length of `N-1` and `N-2`, respectively.