Khrystian Clark

Winter 2022, CS 325

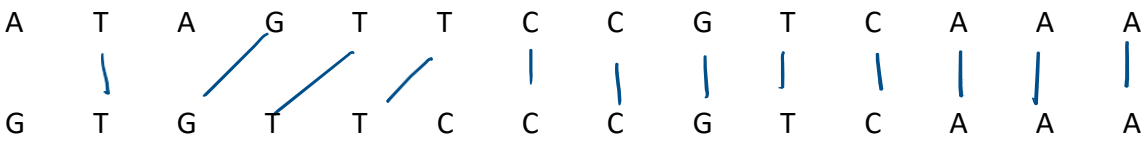
Assignment: Dynamic Programming

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):
    m = len(DNA1)
    n = len(DNA2)

    cache = [[-1 for x in range(n+1)] for x in range(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i==0 or j==0:
                cache[i][j] = 0
            elif 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])

    return cache[m][n]

DNA1 = "ATAGTTCCGTCAAA"
DNA2 = "GTGTTCCCGTCAAA"
print(dna_match_topdown(DNA1, DNA2))
```

   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):
    m = len(DNA1)
```

```
    n = len(DNA2)

    cache = [[0 for x in range(n+1)] for x in range(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i==0 or j==0:
                cache[i][j] = 0
            elif 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])

    return cache[m][n]

DNA1 = "ATAGTTCCGTCAAA"
DNA2 = "GTGTTCCCGTCAAA"
print(dna_match_bottomup(DNA1, DNA2))
```

c. Explain how your top-down approach different from the bottom-up approach?
   - My top down approach attempts to solve for an end value, and then solves the sub-problems while finding the solution for the main value.
   - My bottom up approach starts solving the problem from the lowest possibility and works its way up to finding a solution for a larger problem.

d. What is the time complexity and Space complexity using Top-down Approach
   - Had some trouble understanding this concept in terms of putting into a program function. So I mimicked my bottom up approach and started iteration from the end of each string.
   - O(m*n)

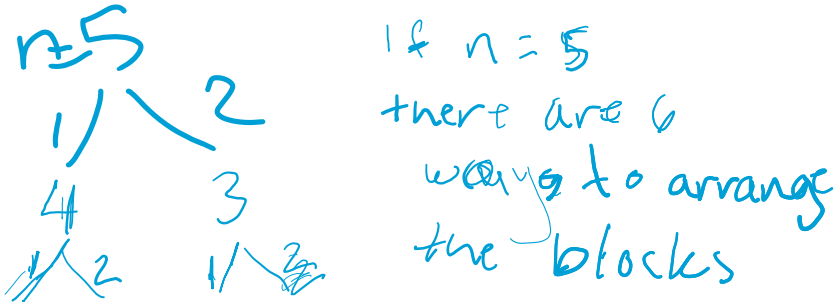
e. What is the time complexity and Space complexity using Bottom-up Approach
   - Dimensional array size = m*n
   - O(m*n) for the stored 2D array

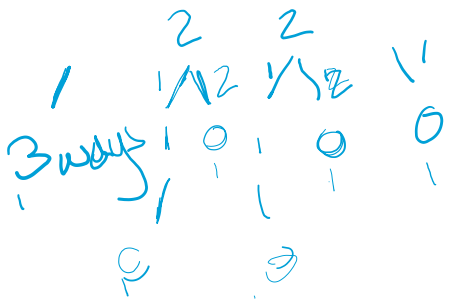
f. Write the subproblem and recurrence formula for your approach. If the top down and bottom-up approaches have the subproblem recurrence formula you may write it only once, if not write for each one separately.
   - f(n) = 0 if i or j is 0
   - f(n) = f(i,j)+1 for every i that is equal to j.
   - it is (i-1) or (j-1) if the items we are comparing 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)





b. Write pseudocode/description for the brute force approach
    - Bf(n)
        - If n == 0, ret 0
        - For all 2 ≥ n > 0:
            - If n is 1 or 2, return n
        - Else:
            - If n % 2 = 0
                - n-2(add one to the amount of blocks) and iterate until n=0
            - else:
                - n-1(add 1 to the amount of blocks) and iterate through the twos until n=0
c. Compare the time complexity of both the approaches
    - DP = O(n) on the tree, only increases at any level if there are any values left.
    - BF = $O(2^n)$ it increases each level at 2^n rate.
    - DP is faster.
d. Write the recurrence formula for the problem
    - F(n) = F(n+1)+F(n+2) +c for all n ≥ 0

e. 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)