Problem 1

Find the length of the longest common string alignment, non-continuous, in two DNA sequences. An empty string has no match.

a) Write a top-down function dna_match_topdown(DNA1, DNA2) in external file DNAMatch.py.

b) Write a bottom-up function dna_match_bottomup(DNA1, DNA2) in external file DNAMatch.py

c) Explain in this file how the two approaches differ.

d) Compute the top-down time and space complexities.

e) Compute the bottom-up time and space complexities.

f) Write the sub-problem and recurrence formula twice if two different formulas, else once.

a) Top-Down Approach

See external file DNAMatch.py.

b) Bottom-Up Approach

See external file DNAMatch.py.

c) Differences Explained

Top-down with memoization starts with the entire problem and recursively saves each subproblem solution in a dictionary, where lookup is $O(1)$ time. It checks if a subproblem is already solved and returns the answer, instead computing the solution if not.[1]

Bottom-up with tabulation instead solves the smallest subproblem first, tabulating up to the whole. When the problem is solved, each preceding subproblem is already solved, with answers stored in a 2D array.[1]

Top-down and bottom-up have similar time and space complexities. They prevent excessive resource consumption by a recursive naive approach of time O^2n, where each call results in two additional recursive calls. However, even if both dynamic programming processes still compute every subproblem at worst, bottom-up still has less overhead than top-down. The latter, with recursion, has more overhead from method calls.[3]

Problem 1 (Continued)

d) Top-Down Time, Space Complexities

Time Complexity: $O(mn)$

In external file DNAMatch.py, function dna_match_topdown() takes two input sequences.

$$DNA1 = (DNA1_1, DNA1_2, ..., DNA1_m), \text{ with } m \text{ the size of sequence DNA1}$$

$$DNA2 = (DNA2_1, DNA2_2, ..., DNA2_n), \text{ with } n \text{ the size of sequence DNA2}$$

Top-down, like the naive solution, is recursive but memoizes results, with no repeat subproblems:

```
1 memo = {}
2 if (pos1, pos2) in memo:
3     return memo[(pos1, pos2)]
4 if pos1 < 0 or pos2 < 0:
5     memo[(pos1, pos2)] = 0
6 elif seq1[pos1] == seq2[pos2]:
7     memo[(pos1, pos2)] = 1 + lcs(seq1, seq2, pos1 - 1, pos2 - 1)
8 else:
9     memo[(pos1, pos2)] = max(lcs(seq1, seq2, pos1 - 1, pos2), lcs(seq1, seq2, p
```

Memoization above prevents a $O(2^n)$ runtime. This code is $O(mn)$, where m and n are the input sizes.

Space Complexity: $O(mn)$

Recursive calls are limited, but top-down is $O(mn)$ space. The code solves every subproblem from base case to size m and n.

Problem 1 (Continued)

e) Bottom-Up Time, Space Complexities

Time Complexity: $O(mn)$

In external file DNAMatch.py, function dna_match_bottomup() takes two input sequences.

$$DNA1 = (DNA1_1, DNA1_2, ..., DNA1_m), \text{ with } m \text{ the size of sequence DNA1}$$

$$DNA2 = (DNA2_1, DNA2_2, ..., DNA2_n), \text{ with } n \text{ the size of sequence DNA2}$$

It initializes a table $c[0...m, 0...n]$ from $c[i, j]$, where i and j are that subproblem's coordinates.[4]

Lines 2, 4, and 6 below calculate each table entry in $O(1)$ time. Runtime is $O(mn)$, where m and n are the two sequence lengths.

```
1 if col == 0 or row == 0:
2     cache[col][row] = 0
3 elif DNA1[col - 1] == DNA2[row - 1]:
4     cache[col][row] = cache[col - 1][row - 1] + 1
5 else:
6     cache[col][row] = max(cache[col - 1][row], cache[col][row - 1])
```

Space Complexity: $O(mn)$

```
1 n, m = len(DNA1), len(DNA2)
2 cache = [[0 for col in range(m + 1)] for row in range(n + 1)]
```

Above, a table of lengths m and n store the subproblem answers. The space is $O(mn)$, as the 2D array is no larger than the sequence lengths to store its subproblems.

f) Recurrence Formula

The top-down and bottom-up approaches have the same recurrence formula.[4] What's different between the two approaches and the naive method is how subproblems are optimized and stored.

$$T(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ T(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } DNA1_i = DNA2_j \\ \max(T(i-1, j), T(i, j-1)) & \text{otherwise (if } i, j > 0 \text{ and } DNA1_i \neq DNA2_j) \end{cases}$$

Problem 2

A random number N is given in a puzzle. You have one- and two-unit-length blocks; arrange them back-to-back for a total N-unit length. How many ways can you arrange the blocks for a given N?

a) Write pseudocode for a dynamic-programming approach, top-down or bottom-up.

b) Write pseudocode for a brute-force approach.

c) Compare the time complexity of both approaches.

d) Write the problem's recurrence formula.

a) Bottom-Up Pseudocode

A bottom-up approach is similar to the Fibonacci sequence using tabulation.[2]

1. Initialize a table $Arr[row][col]$ of size N.
2. Set $Arr[1][1]$ (i.e., the first entry) as 1. There's only one way to get zero — with no blocks.
3. Set $Arr[1][2]$ (i.e., the second entry) as 1. There's only one way to get 1 — with one block of 1.
4. From slot 3 to number N:

   Each slot is the sum of $N-1$ and $N-2$, which uses the previous slots that have already been computed.
5. This tabulates each subproblem until the entire problem is solved.

b) Brute-Force Pseudocode

A brute-force approach is more succinct but intensive. With each recursive call, every subproblem is again solved.

1. If N is less than or equal to 1, return N (i.e., like before, zero and 1 are calculated only one way).
2. Else, recursively call this function as the sum of $N-1$ and $N-2$.

Problem 2 (Continued)

c) Time-Complexity Comparison

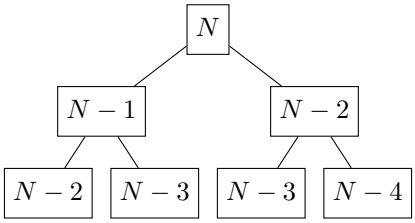
Bottom-Up: $O(n)$

Each subproblem is solved at $O(1)$ time, never rising above size N.

$$T(N) = (N-1) + (N-2)$$

Brute-Force: $O(2^n)$

Two recursive calls are called each time, so the brute-force method is $O(2^n)$.[2]



d) Recurrence Formula

The recurrence formula is similar to the one for Fibonacci.[2]

$$T(n) = \begin{cases} 1 & \text{if } N = 0 \\ 1 & \text{if } N = 1 \\ T(n-1) + T(n-2) & \text{if } N > 1 \end{cases}$$

Sources

[1] "Exploration 3.3: Dynamic Programming - Longest Common Subsequence Problem." Oregon State University: CS 325 Canvas. Canvas.OregonState.edu/courses/1914700/pages/Exploration-3-dot-3-Dynamic-Programming-Longest-common-subsequence-problem (accessed April 24, 2023).

[2] "Exploration 3.1: Dynamic Programming Fundamentals." Oregon State University: CS 325 Canvas. Canvas.OregonState.edu/courses/1914700/pages/Exploration-3-dot-1-Dynamic-Programming-Fundamentals (accessed April 24, 2023).

[3] R. Morelli and R. Walde, "Chapter 12, Section 7," in Java, Java, Java: Object-Oriented Problem Solving, 3rd ed., Minneapolis, MN: Open Textbook Library, 2016, pp. 574–575.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Chapter 15, Section 4, Introduction to Algorithms, 3rd ed., Cambridge (Inglaterra): MIT Press, 2009.