Bioinformatics Homework 2.2

Local Alignment Problem

Problem Statement: Given a match reward, a mismatch penalty, an indel penalty, and two nucleotide strings, find the maximum alignment of the two strings where the score of the alignment is maximized. The score is computed based on the rewards and penalties given.

Algorithm Description:

```
LocalAlignment(match, mismatch, penalty, s, t)
      // Initialization (storing sizes and dp table)
      N <- |s|, M <- |t|
      DP <- a two-dimensional table that stores the size of the maximum
                local matching for s and t
      // Iterating through each pair of characters
      For each i in range of 1 to N
      For each j in range of 1 to M
      // Finding calculating the best alignment using dp
            DP_i,j <- maximum value for all:
                  DP_i-1,j - penalty,
                  DP_i,j-1 - penalty,
                  if s_i == t_j, then DP_i,j = DP_i-1,j-1 + match
                  else DP_i,j = DP_i-1,j-1 - mismatch

      // build-back from the results in DP
      X, Y <- Local Alignment strings built-back from the DP table
```

This algorithm takes in two strings *s* and *t* and finds the maximum local alignment between both strings. Integers *match, mismatch,* and *penalty*, are given to score the strings based on the characters that match, mismatch, and get deleted/skipped respectively. For each possible pair of characters in *s* and *t*, we select the maximum matching by either matching the two characters, deleting one of the characters from one of the strings, or deleting both characters. The maximum score in DP will contain the best, final match for any local alignment in *s* and *t*. We can then traverse the DP table and build back the final alignment for each string.

Time Analysis: This algorithm uses two nested loops, each of them iterating through all the characters in *s* and *t* which runs in O(|s|*|t|). The build-back finds the maximum local matching for both of the strings. Because the build-back is limited by the size of the smaller string since the maximum alignment is limited by the size of the smaller string, the runtime for the build-back is min(|s|, |t|). The total runtime then is: min(|s|, |t|) + |s|*|t| **or** min(n, m)) + O(nm) **which evaluates to** O(nm)

The space complexity is also O(nm) since we create a table of n*m.

Implementation: **#1115233**

https://cogniterra.org/submissions/202340/1115233?unit=45358

Discussion: For this algorithm, I wanted to prioritize runtime. Because of this, I implemented the algorithm iteratively. Another way I could have implemented this algorithm efficiently was to prioritize memory. Although the memory efficient implementation uses linear space, the runtime is doubled by a factor of two. Both implementations are valid depending on what the application is prioritizing.