Molly Hayward: **kgxj22**
Mini Technical Report

Background information on the problem area:

Aligning DNA sequences allows us to quantify the similarities between certain species. This is useful when researching the origins of diseases, however a lot of computation is required to analyse DNA sequence data due to its large volume. As a result of this, we must create efficient means of analysing this data in an accurate way - and maintain databases big enough to store it.

Comparison of techniques for computing the optimal alignment of two sequences:

The recursive algorithm takes two sequences as input and is recursively called on 3 variations of these sequences, after their last letters are aligned uniquely. With the alignment increasing and the sequences reducing in length at each step, we inevitably end when one sequence is empty - the base case. The recursive algorithm checks and scores every alignment to find the optimal. The number of alignments of two sequences of length n and m can be computed using the formula: $al(n, m) = al(n-1, m-1) + al(n-1, m) + al(n, m-1)$

We can see that the number of alignments increases rapidly with sequence length, therefore this technique is inefficient when working with longer sequences.

In order to overcome this problem I implemented dynamic programming; a technique in which a scoring matrix and a backtracking matrix are used to compute the optimal alignment of two sequences. It can be used effectively to solve problems with both optimal substructure and overlapping subproblems, as it only needs to compute each sub-alignment once - unlike recursion which resolves the same subproblems repeatedly.

Python code and design choices:

Initially, the length of sequences 2 and 1 are stored in variables i and j respectively. Both the scoring and backtracking matrices are initialised and the function populateMatrices(i,j) is called.
I used a while loop to follow the populated backtracking matrix, beginning in the bottom right corner - exiting the while loop when cell [0,0] is reached. Variables seq1A and seq2A begin as empty strings and are updated based on the letter in each cell - ultimately producing two strings composed of letters in the original sequences and gaps. These sequences are initially in reverse order due to the direction followed through the backtracking matrix, therefore they are reversed using string splicing and assigned to best_alignment.

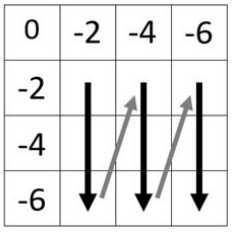
populateMatrices(i,j):

Starts by assigning U and L to the left-most column and top row cells of the backtracking matrix, respectively. Multiples of -2 are assigned to the same positions in the scoring matrix.

END	L	L	L
U			
U			
U			

0	-2	-4	-6
-2			
-4			
-6			

The remaining cells are scored in columns, beginning at cell [1,1] - using a nested for loop. As the top row and left-most column are already scored, we know that cells [x-1, y-1], [x-1,y] and [x,y-1] of the scoring matrix already contain values. We use the values in these cells to calculate the score of the current cell. Hence, scoring cells in this pattern means that we needn't use recursion which would drastically increase the running time. I would also need to amend the default recursion limit for larger sequences.



The corresponding cell in the backtracking matrix is scored simultaneously, and if-statements are used to determine which letter is appropriate for each cell (the variable matching maxScore). If more than one term produces the maximum score, then priority is given to 'D'. This was a design choice as this tends to produce an alignment with fewer gaps.

Molly Hayward: **kgxj22**
Mini Technical Report

Applying python code to the test sequences:

Length 3:

```
Time taken: 0.0
Best (score -3):
Alignment
String1: GAA

String2: TGT
```

Length 4:

```
Time taken: 0.0
Best (score 6):
Alignment
String1: GAAT
           || |
String2: GATT
```

Length 5:

```
Time taken: 0.0
Best (score 5):
Alignment
String1: GAATT
            || |
String2: GATTC
```

Length 6:

```
Time taken: 0.0
Best (score 6):
Alignment
String1: GA-ATTC
            | | ||
String2: -AGACTC
```

Length 7:

```
Time taken: 0.0
Best (score 12):
Alignment
String1: GAATTCA-
              ||||||
String2: -AATTCAC
```

Length 8:

```
Time taken: 0.0
Best (score 10):
Alignment
String1: GAATTCAA-
               | |||||
String2: G-TTTCAAT
```

Length 100:

```
Time taken: 0.019050121307373047
Best (score 146):
Alignment
String1: GAATTCAATAC-TCCACTTTCCATTCTGTTCAAAGGTCACGTATAGTCCTGGGAATACTCAGGGTTCTCA-CTTCATGGCTATGCAGGTATTTGTTC-C-C-AC------A--
         |||||| || | | ||| || |  ||| |||| ||||| |||| |||||| ||||| |||||||||||| | | | || ||||||||||||||| ||| | | ||      |
String2: GAATTC-AT-CGTGCAC-TT-C--TCTTTTCATAGGTCCCGTA-AGTCCT-GGAATTCTCAGGGTTCT-AGC-T-AT-GCTATGCAGGTATTT-TTCACACTACTTAAATACT
```

Analysis of the test sequences:

The dynamic programming approach has time complexity $O(n^2)$ when aligning two sequences of length n, therefore it is realistic that the shorter sequences shown are aligned in a negligible amount of time. Output from my program supported that running time is proportional to the square of the sequence length, hence why length 5000 sequences are aligned approximately 100x slower than length 500 sequences.

```
Length 500:  Time taken: 0.581017017364502
             Best (score 796):
Length 5000: Time taken: 58.10610294342041
             Best (score 7766):
```

$$\frac{5000^2}{500^2} = 100$$

A vital aspect of the dynamic programming approach is that we store the scores of sub-alignments in a matrix so that we can efficiently align sequences with overlapping subproblems. However, in terms of memory this can be costly. When aligning two sequences of length n, we require a matrix of size $(n+1)^2$, therefore using dynamic programming on larger sequences is impractical.

I found that gaps were **mostly** positioned between letters - as opposed to on the ends of sequences. This arises because alignments with multiple gaps on either side score badly under our current scoring system therefore are rarely optimal. Only when we see long identical substrings (e.g. length 7 sequences) do we tend to see alignments that would normally arise under ends-free scoring instead. The alignments produced conform to what I would expect to see with dynamic programming in terms of the positioning of gaps. Additionally, I decided to give priority (in a greedy fashion) to diagonal movement through the backtracking matrix. This selects the alignment with the fewest gaps, in cases where more than one alignment produce the optimal score. This affects the quality of the alignments in a positive way as it minimises the number of gaps, hence, the number of potential insertions or deletions.

Improvements or alternative algorithms to tackle any problems:

Dynamic programming may ignore potentially significant local alignments, such as substrings that represent a protein found in two different genes. A side-effect of our current scoring system is that an important local alignment (contained within a low-scoring alignment) would not be highlighted as such. In order to rectify this, we must not allow negative entries in the scoring matrix, replacing these with 0, and we must take the highest score in the matrix to be the optimal score, rather than the score in the bottom right. This would improve my results by flagging up local alignments which could be of use.

The time complexity of dynamic programming could make aligning much larger sequences incredibly slow. To improve upon this we must use heuristic algorithms such as BLAST, although we must compromise on the quality of the alignment - which is not guaranteed to be optimal. With BLAST, only some alignments are reported, as it is first decided if the match is significant or not - making it quicker to align longer sequences.