Project 4 Report

```python
def get_banded_range(i, row_length, banded_width):
    if banded_width == -1:
        return range(0, row_length)
    else:
        lower = i - banded_width
        upper = i + banded_width + 1
        if lower < 0:
            lower = 0
        if upper > row_length:
            upper = row_length
        return range(lower, upper)
```

Get Banded Range

This is one of the helper functions that is able to calculate the specific range of cells in a given row that should be included when using a banded width. It is able to do this in O(1) time and space complexity.

```python
def align(seq1: str, seq2: str, match_award=-3, indel_penalty=5, sub_penalty=1,  3 usages
          banded_width=-1, gap='-') -> tuple[float, str | None, str | None]:
    mod_seq1 = "_" + seq1
    mod_seq2 = "_" + seq2
    len1 = len(mod_seq1)
    len2 = len(mod_seq2)
    matrix = {(0, 0): (0, None, None)}
    for i in range(len1):
        for j in get_banded_range(i, len2, banded_width):
            if i == 0 and j == 0:
                pass
            elif i == 0:
                matrix[(i,j)] = (j * indel_penalty, (i, j-1), 'left')
            elif j == 0:
                matrix[(i,j)] = (i * indel_penalty, (i-1, 0), 'up')
            else:
                diag_cost = sub_penalty + matrix[(i-1, j-1)][0]
                if mod_seq1[i] == mod_seq2[j]:
                    diag_cost = match_award + matrix[(i-1, j-1)][0]
                left_cost = math.inf
                if (i,j-1) in matrix:
                    left_cost = indel_penalty + matrix[(i, j-1)][0]
                up_cost = math.inf
                if (i-1, j) in matrix:
                    up_cost = indel_penalty + matrix[(i-1, j)][0]
                if min(diag_cost, left_cost, up_cost) == diag_cost:
                    matrix[(i,j)] = (diag_cost, (i-1, j-1), 'diagonal')
                elif min(left_cost, up_cost) == left_cost:
                    matrix[(i,j)] = (left_cost, (i, j-1), 'left')
                else:
                    matrix[(i,j)] = (up_cost, (i-1, j), 'up')
    alignment_cost = matrix[(len1-1, len2-1)][0]
    left_alignment_string, right_alignment_string = find_alignment_strings(matrix, seq1, seq2, gap)
    return alignment_cost, left_alignment_string, right_alignment_string
```

Align

In the “align” function, I chose to use a dictionary to store the matrix of score’s. That way, I was able to store the scores for the banded width without wasting space. The bulk of this function takes place in a double for loop. The first for loop loops through a range of the length of the first sequence and the second for loop loops through the range that is returned by “get_banded_range”. All the logic inside the double for loop has a constant time and space complexity. Because of this, the total space and time complexity of the function is O(m*n) if it is unbanded or O(n*k) if it is banded. The difference lies in the range that is returned by “get_banded_range”, allowing the inner for loop to be much quicker when there is a banded width.

```
def find_alignment_strings(matrix, left_string, right_string, gap):  1 usage
    left_alignment_string = ''
    right_alignment_string = ''
    current_cell = matrix[len(left_string), len(right_string)]
    while current_cell[2] is not None:
        if current_cell[2] == 'diagonal':
            left_alignment_string = left_string[-1] + left_alignment_string
            left_string = left_string[:-1]
            right_alignment_string = right_string[-1] + right_alignment_string
            right_string = right_string[:-1]
        if current_cell[2] == 'left':
            left_alignment_string = gap + left_alignment_string
            right_alignment_string = right_string[-1] + right_alignment_string
            right_string = right_string[:-1]
        if current_cell[2] == 'up':
            left_alignment_string = left_string[-1] + left_alignment_string
            left_string = left_string[:-1]
            right_alignment_string = gap + right_alignment_string
        current_cell = matrix[current_cell[1]]
    return left_alignment_string, right_alignment_string
```

Find Alignment Strings

The “align” function calls “find_alignment_strings” at the very end. This function starts at the end of the matrix and traces its way to the origin in order to calculate the two alignment strings. The worst case time complexity of this function would be O(n+m) or O(n+k). Because of this, the function does not add to the complexity of the “align” function.