Project 4 - Gene Sequencing

In code, `Fn` refers to function complexity

In code, comments with `T O(...)` are time complexities, then comments with `S O(...)` are space complexities.

If no time complexity is provided next to a line of code, it is `O(1)` constant time.

If no space complexity is provided next to a line of code, it doesn't cost any space.

1. Time and Space Complexity

See section 5 for most in-depth explanation of how the overall time complexity came to be. In this section the focus is what each algorithm's complexity is based on the functions it calls. As far as knowing the whys behind the complexity of the functions called, see section 5.

a) Unrestricted Algorithm

Time Complexity: $O(nm)$

Space Complexity: $O(nm)$

See #5 for full unrestricted code, but the code block from the `solve_unrestricted()` function has been copied below for convenience.

```
def solve_unrestricted(self, seq1, seq2):     # Fn: T O(n*m), S O(n*m) - Added up T and S results from functions called
        # Initialize 2D arrays
        num_rows = len(seq1) + 1  # Need +1 to account for empty string
        num_cols = len(seq2) + 1  # Need +1 to account for empty string

        val_table, back_table = self.u_init_tables(num_rows, num_cols)   # See Fn - T O(n), S O(n*m)

        val_table, back_table = self.u_fill_tables(  # See Fn - T O(n*m), S O(1)
            seq1, seq2, val_table, back_table, num_rows, num_cols
        )

        # Figure out score (it will be the value in the bottom right corner of the value table)
        score = val_table[num_rows - 1][num_cols - 1]

        alignment1, alignment2 = self.u_find_alignments(  # See Fn - T O(n+m), S O(1)
            seq1, seq2, back_table, num_rows, num_cols
        )

        return score, alignment1, alignment2
```

The time complexity for the Unrestricted is $O(nm)$ because it calls the fill tables function (this one dominates since it has the largest time complexity of all the called functions). The fill tables function is O(n*m) because in order to fill all the cells in the table, you have to traverse every row and col. So if the rows are n and the cols are m, it would make sense that time-wise, my algorithm would at least take n*m.

b) Banded algorithm

Time Complexity: $O(kn)$

Space Complexity: $O(kn)$

See #5 for full banded code, but the code block from the `solve_banded()` function has been copied below for convenience.

```
def solve_banded(self, seq1, seq2):           # Fn: T O(k*n), S O(k*n) - Added up T and S results from functions called
        # Immediately return if there's significant length discrepancies between seq1 and seq2
        if abs(len(seq1) - len(seq2)) > MAXINDELS:
            return math.inf, "No Alignment Possible", "No Alignment Possible"

        # Initialize 2D arrays
        num_rows = len(seq1) + 1
        num_cols = 2 * MAXINDELS + 1  # For this project, banded will have 7 columns
```

```
        val_table, back_table = self.b_init_tables(num_rows, num_cols)  # See Fn - T O(n), S O(k*n)

        val_table, back_table = self.b_fill_tables(    # See Fn - T O(k*n), S O(1)
            seq1, seq2, val_table, back_table, num_rows, num_cols
        )

        # Figure out score -- the cell to pull it from depends on the sequence lengths (dimensions of table)
        score = 0
        score_i = 0
        score_j = 0
        if len(seq1) == len(seq2):
            score_i = num_rows - 1
            score_j = 3
            score = val_table[score_i][score_j]
        if (len(seq1) + 1) == len(seq2):
            score_i = num_rows - 1
            score_j = 4
            score = val_table[score_i][score_j]

        alignment1, alignment2 = self.b_find_alignments(    # See Fn - T O(n+k), S O(1)
            seq1, seq2, back_table, score_i, score_j
        )

        return score, alignment1, alignment2
```

The time complexity for the banded is $O(kn)$ instead of O(n*m). This improvement is because this algorithm uses the banded method, meaning that it stores an array with a smaller amount of columns (only keeps the maximum amount of columns that would correspond with a reasonable amount of inserts and deletes — any more inserts or deletes outside of that are discarded). As far as the details, it's O(k*n) because it calls the fill tables function (this one dominates since it has the largest time complexity of all the called functions). The fill tables function is O(k*n) because in order to fill all the cells in the table, you have to traverse every row and col. So if the rows are n and the cols are k (7), it would make sense that time-wise, my algorithm would at least take k*n.

2. Alignment Extraction Algorithm Explained

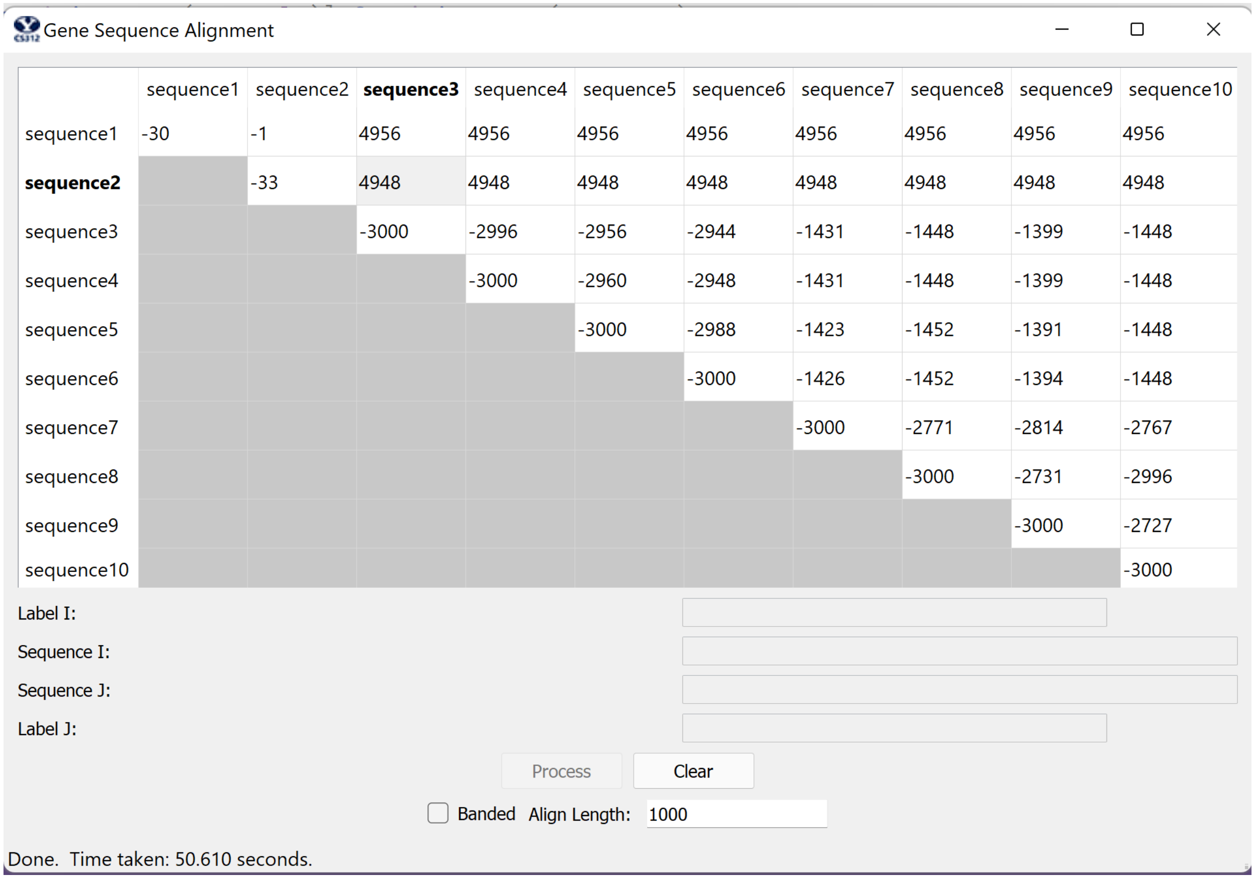
In order to find the optimal alignments (alignment1 and alignment2) for sequence1 and sequence2, we first fill out the values table and the back pointer table with appropriate values. The values table has the costs to get to each cell, then the back pointer table shows all the paths where each answer came from. To figure out optimal alignments, you essentially go from the bottom right corner of your back pointer table (back_table) and trace back the arrows in that table until you get back to the starting node in the top left corner. That said, we don't just want the path. We also want the 2 alignments. To get those, for every cell in the path, we look at how seq1 and seq2 should be manipulated given the operation. For instance, if the back pointer has a diagonal arrow, that means we'll do a substitution, so the alignment1 and alignment2 can just keep the same values as seq1 and seq2 at that spot. If it's an insertion or deletion, the one sequence keeps its value while the other adds a dash in its place. You keep on building the strings backward until you reach the top left corner of your tree, then you're done!

💡 See `u_find_alignments()` and `b_find_alignments()` for the code blocks implementing this algorithm.

3 & 4. Results (Tables and Alignments)

Unrestricted (10x10, n=1000)

Screenshot



	sequence1	sequence2	**sequence3**	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	sequence10
sequence1	-30	-1	4956	4956	4956	4956	4956	4956	4956	4956
sequence2		-33	4948	4948	4948	4948	4948	4948	4948	4948
sequence3			-3000	-2996	-2956	-2944	-1431	-1448	-1399	-1448
sequence4				-3000	-2960	-2948	-1431	-1448	-1399	-1448
sequence5					-3000	-2988	-1423	-1452	-1391	-1448
sequence6						-3000	-1426	-1452	-1394	-1448
sequence7							-3000	-2771	-2814	-2767
sequence8								-3000	-2731	-2996
sequence9									-3000	-2727
sequence10										-3000

Extracted alignment of first 100 characters

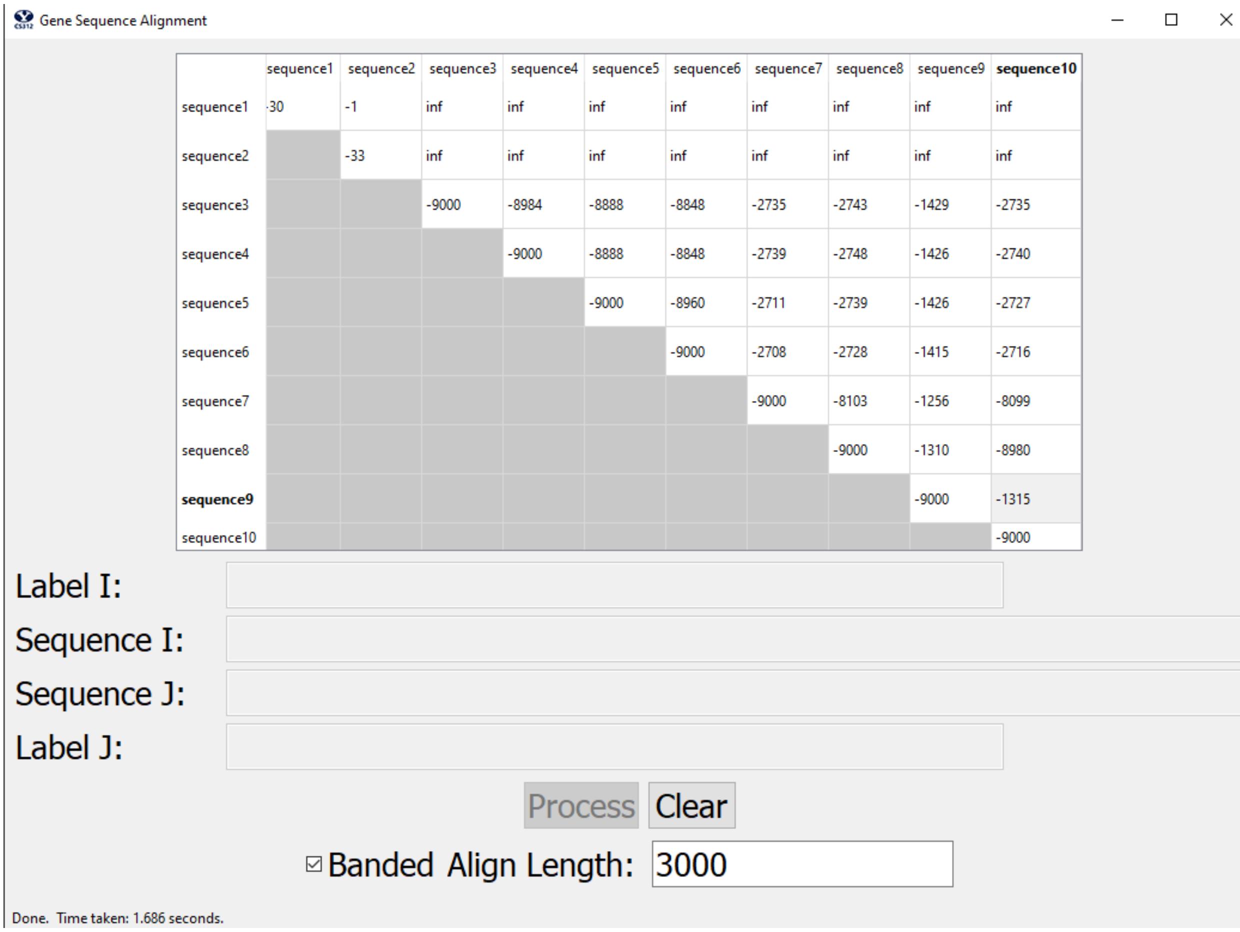
(Sequence 3 first, then Sequence 10)

`gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-`

`ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgt`

Banded (10x10, n=3000)

Screenshot



	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	**sequence10**
sequence1	-30	-1	inf	inf	inf	inf	inf	inf	inf	inf
sequence2		-33	inf	inf	inf	inf	inf	inf	inf	inf
sequence3			-9000	-8984	-8888	-8848	-2735	-2743	-1429	-2735
sequence4				-9000	-8888	-8848	-2739	-2748	-1426	-2740
sequence5					-9000	-8960	-2711	-2739	-1426	-2727
sequence6						-9000	-2708	-2728	-1415	-2716
sequence7							-9000	-8103	-1256	-8099
sequence8								-9000	-1310	-8980
sequence9									-9000	-1315
sequence10										-9000

Extracted alignment of first 100 characters

(Sequence 3 first, then Sequence 10)

`gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-`

`gttt-tgctttacttaataggttggcagaagattatgtctacctctttgatgagggaggcga-gataagagtgattggcgt--gtac-taccctttcta`

5. Source Code

Functions with `u_` or `unrestricted` correspond with the unrestricted algorithm. To be explicit, these are those functions:

- `solve_unrestricted()` - First function invoked in algorithm. Calls the functions below so that it initializes the tables, fills them (with vals and back pointers), then finds their alignments.
- `u_init_tables()` - Initializes a value table and a backpointer table that are both `n x m`, if n is length of sequence 1 and m is the length of sequence 2 (they can have a max number of chars as the align length entered). Sets most values to 0 or NONE, except for first row and col.
- `u_fill_tables()` - Fills in the rest of the values in the table by looking at left, top, and diagonal neighbors and finding the one with the best value to add on to the current cell.
- `u_find_alignments()` - Goes from the bottom right most cell (where score is) in the array and traces the path all the way to the top left cell in the array, denoting the shortest edit distance. Along the way it uses each step in the path to build the alignment strings for sequence 1 and 2.

Functions with `b_` or `banded` correspond with the banded algorithm. To be explicit these are those functions:

- `solve_banded()` - First function invoked in algorithm. Calls the functions below so that it initializes the tables, fills them (with vals and back pointers), then finds their alignments.
- `b_init_tables()` - Initializes a value table and a backpointer table that are both `k x n`, if k is 7 for the MAXINDELS and n is length of sequence 1 (sequence 1 can have a max number of chars as the align length entered). Sets most values to 0 or NONE, except for first row and col (in this case the first column is diagonal)
- `b_fill_tables()` - Fills in the rest of the values in the table by looking at left, top, and diagonal neighbors and finding the one with the best value to add on to the current cell. Does extra work with indices since our array is banded.
- `b_find_alignments()` - Goes from the final score cell in the array (this spot depends on the dimensions), and traverses until it hits the starting cell. denoting the shortest edit distance. Along the way it uses each step in the path to build the alignment strings for sequence 1 and 2. This one is a bit more difficult since the array is banded, so more index adjustment is involved.

Functions shared between the two

- `compare_chars()` - Used for diagonal comparisons to see if two characters within the sequences are a MATCH or need a SUBSTITUTION, then returns the appropriate cost/reward.
- `align()` - The built-in function that the GUI calls where everything starts. It calls `solve_unrestricted()` or `solve_banded()`, then returns the result.

All Source Code

> 💡 The code below is just how it's stored in my .py file, in the same exact order. I've just separated code blocks so that it's easier to digest.

Setup

```
#!/usr/bin/python3

from enum import Enum
from which_pyqt import PYQT_VER

if PYQT_VER == "PYQT5":
    from PyQt5.QtCore import QLineF, QPointF
elif PYQT_VER == "PYQT4":
    from PyQt4.QtCore import QLineF, QPointF
else:
    raise Exception("Unsupported Version of PyQt: {}".format(PYQT_VER))

import math
import time
import random

# Used to compute the bandwidth for banded version
MAXINDELS = 3

# Used to implement Needleman-Wunsch scoring
MATCH = -3
INDEL = 5
SUB = 1

# Enum for values in back_table since it only stores ints
class Arrow(Enum):
    NONE = 0  # Current cell doesn't have value
    START = 1  # Starting cell (useful when finding path back to it)
    LEFT = 2  # Current cell got its value from the left cell
    DIAG = 3  # Current cell got its value from the diagonal cell
    UP = 4  # Current cell got its value from the cell above it


class GeneSequencing:
```

```
    def __init__(self):
        pass
```

```
def u_init_tables(self, num_rows, num_cols):  # Fn: T O(n) - 2 for loops, S O(n*m) - 2 2D arrays of size n*m
        """Initializes the value and back pointer tables (0s everywhere, except the value table has i in the first row and col."""

        val_table = [       # S O(n*m) - 1 array of size n*m
            [0 for i in range(num_cols)] for j in range(num_rows)
        ]  # Table that holds edit distance values

        back_table = [        # S O(n*m) - 1 array of size n*m
            [Arrow.NONE for i in range(num_cols)] for j in range(num_rows)
        ]  # Table that holds back pointers

        # Set up first col
        for i in range(num_rows):     # O(n) - for loop over rows (seq1)
            val_table[i][0] = (
                i * INDEL
            )  # Initialize first col of each row to be i * INDEL
            back_table[i][0] = Arrow.UP  # Make up back pointers across left col

        # Set up first row
        for j in range(num_cols):     # O(n) - for loop over cols (seq2)
            val_table[0][j] = j * INDEL  # Initialize first row of each col to be j
            back_table[0][j] = Arrow.LEFT  # Make left back pointers across top row

        back_table[0][0] = Arrow.START  # Make sure start has its own value

        return val_table, back_table
```

```
def compare_chars(self, char1, char2):  # Fn: T O(1) - just if statement and return, S O(1) - No data structures stored
        """Checks to see if two characters match, then returns the appropriate reward/cost accordingly."""
        if char1 == char2:
            return MATCH
        return SUB
```

```
def u_fill_tables(self, seq1, seq2, val_table, back_table, num_rows, num_cols):  # Fn: T O(n*m) - 2D for loop, S O(1)
                                                                                         # - Declares ints, but arrays pre-declared
        """Starting at [1,1], fill out the dynamic programming tables that hold values and back pointers."""

        for i in range(1, num_rows):            # T O(n*m) - 2D for loop over rows (seq1) and cols (seq2)
            for j in range(1, num_cols):
                left_ins_cost = INDEL + val_table[i][j - 1]
                diag_sub_cost = (
                    self.compare_chars(seq1[i - 1], seq2[j - 1])
                    + val_table[i - 1][j - 1]
                )  # Checks current chars for a match, adds that to diagonal value
                up_del_cost = INDEL + val_table[i - 1][j]

                # Figure out smallest cost

                # Left first - first tiebreaker
                if left_ins_cost <= diag_sub_cost and left_ins_cost <= up_del_cost:
                    val_table[i][j] = left_ins_cost
                    back_table[i][j] = Arrow.LEFT

                # Up second - second tiebreaker
                elif up_del_cost < left_ins_cost and up_del_cost <= diag_sub_cost:
                    val_table[i][j] = up_del_cost
                    back_table[i][j] = Arrow.UP

                # 3rd case - diagonal
                else:
                    back_table[i][j] = Arrow.DIAG
                    val_table[i][j] = diag_sub_cost
```

```
        return val_table, back_table
```

```
# FOR UNRESTRICTED
def u_find_alignments(self, seq1, seq2, back_table, num_rows, num_cols):  # Fn: T O(n+m) - Worst case traversal is along edge
    cur_row = num_rows - 1                                                    #  S O(1) - Just ints, arrays pre-declared
    cur_col = num_cols - 1
    back_ptr = back_table[cur_row][cur_col]  # Start at last cell (bottom right)
    alignment1 = ""
    alignment2 = ""

    while back_ptr != Arrow.START:   # T O(n + m) - Worst case you'd have to trace it back to the left then all the way up
        if back_ptr == Arrow.LEFT:
            # Replace seq1 letter with a dash
            alignment1 = "-" + alignment1
            # Keep seq2 letter
            alignment2 = seq2[cur_col - 1] + alignment2
            # Move left 1
            cur_col -= 1
        elif back_ptr == Arrow.DIAG:
            # Keep seq1 letter
            alignment1 = seq1[cur_row - 1] + alignment1
            # Keep seq2 letter
            alignment2 = seq2[cur_col - 1] + alignment2
            # Move up 1, left 1
            cur_row -= 1
            cur_col -= 1
        elif back_ptr == Arrow.UP:
            # Keep seq1 letter
            alignment1 = seq1[cur_row - 1] + alignment1
            # Replace seq2 letter with a dash
            alignment2 = "-" + alignment2
            # Move up 1
            cur_row -= 1

        # Move the back_ptr
        back_ptr = back_table[cur_row][cur_col]

    return alignment1, alignment2
```

```
def solve_unrestricted(self, seq1, seq2):     # Fn: T O(n*m), S O(n*m) - Added up T and S results from functions called
        # Initialize 2D arrays
        num_rows = len(seq1) + 1  # Need +1 to account for empty string
        num_cols = len(seq2) + 1  # Need +1 to account for empty string

        val_table, back_table = self.u_init_tables(num_rows, num_cols)   # See Fn - T O(n), S O(n*m)

        val_table, back_table = self.u_fill_tables(  # See Fn - T O(n*m), S O(1)
            seq1, seq2, val_table, back_table, num_rows, num_cols
        )

        # Figure out score (it will be the value in the bottom right corner of the value table)
        score = val_table[num_rows - 1][num_cols - 1]

        alignment1, alignment2 = self.u_find_alignments(  # See Fn - T O(n+m), S O(1)
            seq1, seq2, back_table, num_rows, num_cols
        )

        return score, alignment1, alignment2
```

```
def b_init_tables(self, num_rows, num_cols):     # Fn: T O(n) - 2 for loops, S O(k*n) - 2 2D arrays of size k*n (k is 7)
        """Initializes the value and back pointer tables (0s everywhere, except the value table has i in the first row and col."""

        val_table = [         # S O(k*n) - 1 array of size k*n
            [0 for i in range(num_cols)] for j in range(num_rows)
        ]  # Table that holds edit distance values
```

```
        back_table = [          # S O(k*n) - 1 array of size k*n
            [Arrow.NONE for i in range(num_cols)] for j in range(num_rows)
        ]  # Table that holds back pointers

        # Set up first col
        for i in range(0, MAXINDELS):         # O(n) - for loop over rows (seq1)
            val_table[MAXINDELS - i][i] = (
                MAXINDELS - i
            ) * INDEL  # Initialize first col of each row to be i * INDEL
            back_table[MAXINDELS - i][
                i
            ] = Arrow.UP  # Make up back pointers across left col

        # Set up first row
        for j in range(MAXINDELS, num_cols):   # O(k) - for loop over cols (7 is max)
            val_table[0][j] = (
                j - MAXINDELS
            ) * INDEL  # Initialize first row of each col to be j
            back_table[0][j] = Arrow.LEFT  # Make left back pointers across top row

        back_table[0][MAXINDELS] = Arrow.START  # Starting point

        return val_table, back_table
```

```
def b_fill_tables(self, seq1, seq2, val_table, back_table, num_rows, num_cols):      # Fn: T O(k*n) - 2D for loop, S O(1)
                                                                                      # - Declares ints, but arrays pre-declared
        MAX_IDX_SUM = (
            len(seq2) + MAXINDELS + 1
        )  # Helps us tell when we gone out of bounds on bottom right

        for i in range(1, num_rows):       # T O(k*n) - 2D for loop over rows (seq1) and cols (7)
            for j in range(0, num_cols):
                # Skip out of bounds
                # - first condition is top left corner
                # - second condition is bottom right corner
                if (i + j) <= MAXINDELS or (i + j) >= MAX_IDX_SUM:
                    continue

                left_ins_cost = math.inf
                if j > 0:
                    left_ins_cost = INDEL + val_table[i][j - 1]
                diag_sub_cost = math.inf
                if i > 0:
                    diag_sub_cost = (
                        self.compare_chars(seq1[i - 1], seq2[i + j + -MAXINDELS - 1])
                        + val_table[i - 1][j]
                    )  # Checks current chars for a match, adds that to diagonal value
                up_del_cost = math.inf
                if (j + 1) < num_cols and i > 0:
                    up_del_cost = INDEL + val_table[i - 1][j + 1]

                # Figure out smallest cost

                # Left first - first tiebreaker
                if left_ins_cost <= diag_sub_cost and left_ins_cost <= up_del_cost:
                    val_table[i][j] = left_ins_cost
                    back_table[i][j] = Arrow.LEFT

                # Up second - second tiebreaker
                elif up_del_cost < left_ins_cost and up_del_cost <= diag_sub_cost:
                    val_table[i][j] = up_del_cost
                    back_table[i][j] = Arrow.UP

                # 3rd case - diagonal
                else:
                    back_table[i][j] = Arrow.DIAG
                    val_table[i][j] = diag_sub_cost

        return val_table, back_table
```

```
# FOR BANDED
def b_find_alignments(self, seq1, seq2, back_table, score_i, score_j): # Fn - T O(n+k) worst case, going around edges
        cur_row = score_i                                                 #  S O(1) - just ints, arrays pre-declared
        cur_col = score_j
        back_ptr = back_table[cur_row][cur_col]  # Start at last cell (bottom right)
        alignment1 = ""
        alignment2 = ""
        seq2_idx = len(seq2) - 1

        while back_ptr != Arrow.START:   # T O(n+k) - Worst case you'd have to trace it back to the left then all the way up
            if back_ptr == Arrow.LEFT:
                # Replace seq1 letter with a dash
                alignment1 = "-" + alignment1
                # Keep seq2 letter
                alignment2 = seq2[seq2_idx] + alignment2
                # Move left 1
                cur_col -= 1
            elif back_ptr == Arrow.DIAG:
                # Keep seq1 letter
                alignment1 = seq1[cur_row - 1] + alignment1
                # Keep seq2 letter
                alignment2 = seq2[seq2_idx] + alignment2
                # Move up 1
                cur_row -= 1
            elif back_ptr == Arrow.UP:
                # Keep seq1 letter
                alignment1 = seq1[cur_row - 1] + alignment1
                # Replace seq2 letter with a dash
                alignment2 = "-" + alignment2
                # Move up 1, right 1
                cur_row -= 1
                cur_col += 1

            # Move the back_ptr
            back_ptr = back_table[cur_row][cur_col]
            seq2_idx -= 1

        return alignment1, alignment2
```

```
def solve_banded(self, seq1, seq2):           # Fn: T O(k*n), S O(k*n) - Added up T and S results from functions called
        # Immediately return if there's significant length discrepancies between seq1 and seq2
        if abs(len(seq1) - len(seq2)) > MAXINDELS:
            return math.inf, "No Alignment Possible", "No Alignment Possible"

        # Initialize 2D arrays
        num_rows = len(seq1) + 1
        num_cols = 2 * MAXINDELS + 1  # For this project, banded will have 7 columns

        val_table, back_table = self.b_init_tables(num_rows, num_cols)  # See Fn - T O(n), S O(k*n)

        val_table, back_table = self.b_fill_tables(    # See Fn - T O(k*n), S O(1)
            seq1, seq2, val_table, back_table, num_rows, num_cols
        )

        # Figure out score -- the cell to pull it from depends on the sequence lengths (dimensions of table)
        score = 0
        score_i = 0
        score_j = 0
        if len(seq1) == len(seq2):
            score_i = num_rows - 1
            score_j = 3
            score = val_table[score_i][score_j]
        if (len(seq1) + 1) == len(seq2):
            score_i = num_rows - 1
            score_j = 4
            score = val_table[score_i][score_j]

        alignment1, alignment2 = self.b_find_alignments(    # See Fn - T O(n+k), S O(1)
            seq1, seq2, back_table, score_i, score_j
        )

        return score, alignment1, alignment2
```

```
def align(self, seq1, seq2, banded, align_length):  # Fn: Worst case is T O(n*m), S O(n*m) if unrestricted
                                                               # Better case is banded - T O(k*n), S O(k*n)
        self.banded = banded
        self.max_chars_to_align = align_length

        # Cut down sequences to be max character length
        if len(seq1) > self.max_chars_to_align:
            seq1 = seq1[: self.max_chars_to_align]
        if len(seq2) > self.max_chars_to_align:
            seq2 = seq2[: self.max_chars_to_align]

        # Solve

        score, alignment1, alignment2 = (
            self.solve_unrestricted(seq1, seq2)      # See Fn - T O(n*m), S O(n*m)
            if not banded
            else self.solve_banded(seq1, seq2)       # See Fn - T O(k*n), S O(k*n)
        )

        return {
            "align_cost": score,
            "seqi_first100": alignment1[:100],
            "seqj_first100": alignment2[:100],
        }
```