1.
Unrestricted Algorithm

Minimum Function:
This function is just performing lookups in the 2d trackback array which all happen in constant time. It also compares the currMin to a new number which is also constant time, so **Time complexity is O(1).**
The **space complexity here is O(1)** since we are not making any new space in the array we are just overwriting the (i, j) space in the trackback_array.

CalculateSequences Function:
The **time complexity here is O(nm)** because of the while loop can, in the worst case, go through the traceback_array the length of the topSequence * the length of the sideSequence. This is n * m since it keeps looping all the way to the first cell at (0,0) since that is where the None is stored.
The space complexity is O(n + m) since we are making two new strings for the sequences and they may have "-" in them. **Thus the overall space complexity would be O(d) where d is equal to which ever length is greater between n (topSequence) and m (sideSequenece)**.

Unrestricted Function:
The overall time complexity here is O(nm) where m is the size of the sideSequence and n is the size of the topSequence. This comes from making the 2d arrays as well as the for loop that runs mn time, and the calculateSequences. Thus it equals O(4nm) but the 4 is dropped since it is a constant and it just comes out to be O(nm).
The overall space complexity here is O(nm) where m is the size of the sideSequence and n is the size of the topSequence. This comes from making the two 2d arrays so it would be O(2nm) but the 2 is dropped since it is a constant and it just comes out to be O(nm).

Thus my unrestricted algorithm has an overall time and space complexity of O(nm).

Banded Algorithm

bandedMin Function:
This function is just performing lookups in the 2d array which all happen in constant time. It also compares the currMin to a new number which is also constant time, so **Time complexity is O(1).**
The space complexity here is O(1) since we are not making any new space in the array we are just overwriting the (i, j) space in the traceback_array.

BandedCalculateSequences Function:
The **time complexity here is O(kn)** because of the while loop can, in the worst case, go through the trackback_array n times which is the amount of rows and k

which is the amount of columns. This is n * k since it keeps looping all the way to the first cell at (0,3) since that is where the None is stored.
The space complexity is O(n + m) since we are making two new strings for the sequences and they will may have "-" in them. **Thus the overall space complexity would be O(d) where d is equal to which ever length is greater between n (topSequence) and m (sideSequenece).**

<u>bandedAlg Function:</u>
The overall time complexity here is O(kn) where k is 7 and n is the size of the topSequence or sideSequence whichever is shorter, or if there is a tie the topSequence is chosen as n. The number of rows (n) can at most be the shorter sequence + 3 since d = 3 which comes to O((n+3)k). But the 3 is dropped in Big-O since it is a constant making it O(nk). This comes from making the 2d arrays as well as the for loop that runs kn time, and the bandedCalculateSequences. Thus it equals O(4kn) but the 4 is dropped since it is a constant and it just comes out to be O(kn).
The overall space complexity here is O(kn) where k is 7 and n is the size of the topSequence or sideSequence whichever is shorter, or if there is a tie the topSequence is chosen as n. The number of rows (n) can at most be the shorter sequence + 3 since d = 3 which comes to O((n+3)k). But the 3 is dropped in Big-O since it is a constant making it O(nk). This comes from making the two 2d arrays so it would be O(2kn) but the 2 is dropped since it is a constant and it just comes out to be O(nk).

Thus my banded algorithm has an overall time and space complexity of O(kn).

2.
<u>How string alignment extraction works with my unrestricted algorithm:</u>
When I made my alignment cost array I also made a traceback array. This traceback array is updated along side of my alignment cost array with the correct value of where the minimum value came from. A “D” (diagonal) means that the value came from the cell up one and over to the left one from the current cell. An “A” (above) means that the value came from the cell up one from the current cell. A “L” (left) means that the value came from the cell to the left one from the current cell. Once the minimum value is found the correct letter is set in the trace array. All ties store the trace back as a “D”. Once all costs are found calculateSequences is called which starts at the last row, last column cell in the traceback array. It works its way through the traceback array till it gets to the (0,0) cell where “None” is found. If the traceback array’s cell has a “D” then the next location is up one and over to the left one. It adds the topSequence’s letter at that location to tempSeqI as well as the sideSequence’s letter at that location to tempSeqJ. If the traceback array’s cell has a “L” then the next location is one over to the left. It adds the topSequence’s letter at that location to tempSeqI. It also adds a “-“ to tempSeqJ since a deletion occurred. If the traceback array’s cell has an “A” then the next location is one up. It adds a “-“ to tempSeqI since an insertion occurred. It also adds the sideSequence’s letter at that location to tempSeqJ. It continues through the matrix till (0,0) is reached at the top.

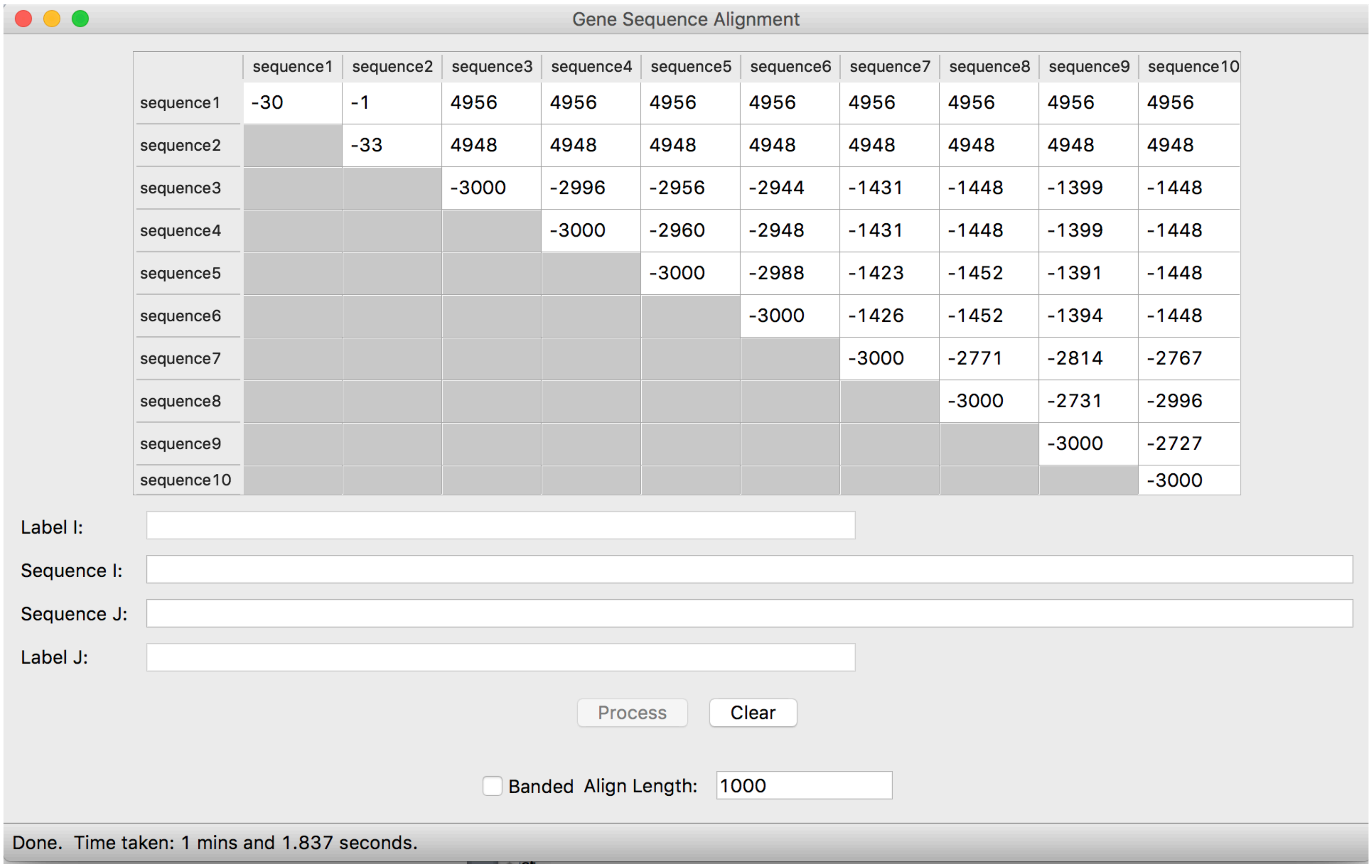
Then the string is reversed to be in the correct order and then cut down to 100 characters.

<u>How string alignment extraction works with my banded algorithm:</u>
When I made my alignment cost array I also made a traceback array. This traceback array is updated along side of my alignment cost array with the correct value of where the minimum value came from. A “D” (diagonal) means that the value came from the cell up one from the current cell. This occurs because the columns became squished in the banded traceback array so that diagonals in the normal array are now directly above a cell. An “A” (above) means that the value came from the cell up one and over to the right one from the current cell. Again this happens because the columns became squished in the banded traceback array so that aboves in the normal array are now up one and over to the right one. A “L” (left) means that the value came from the cell to the left one from the current cell. Once the minimum value is found the correct letter is set in the trace array. All ties store the trace back as a “D”. Since the banded array has some areas that would be out of range in the normal array it has areas in it that have stored in their cells math.inf within the cost array. This means that the index is now out of range for the topSequence and thus the character cannot be looked up. It fills in any areas that are out of range with this math.inf value. It then finds in the last row which column does not have a math.inf stored in its cell and hands that column number to the bandedCalculateSequences. Using the last row and the column handed to it. It goes through the traceback array until it hits None which is found at the top of the array at (0,3). If the traceback array’s cell has a “D” then the next location is up one. It adds the topSequence’s letter at that location to tempSeqI as well as the sideSequence’s letter at that location to tempSeqQ. If the traceback array’s cell has a “L” then the next location is one over to the left. It adds the topSequence’s letter at that location to tempSeqI. It also adds a “-“ to tempSeqQ since a deletion occurred. If the traceback array’s cell has an “A” then the next location is one up and to the right one. It adds a “-“ to tempSeqI since an insertion occurred. It also adds the sideSequence’s letter at that location to tempSeqQ. It continues through the matrix till (0,3) is reached at the top. Then the string is reversed to be in the correct order and then cut down to 100 characters.

<u>Results:</u>

3.
Unrestricted k = 1000

Jonathan Armknecht Project 4 Report CS 312 Sec2



	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

Label I:

Sequence I:

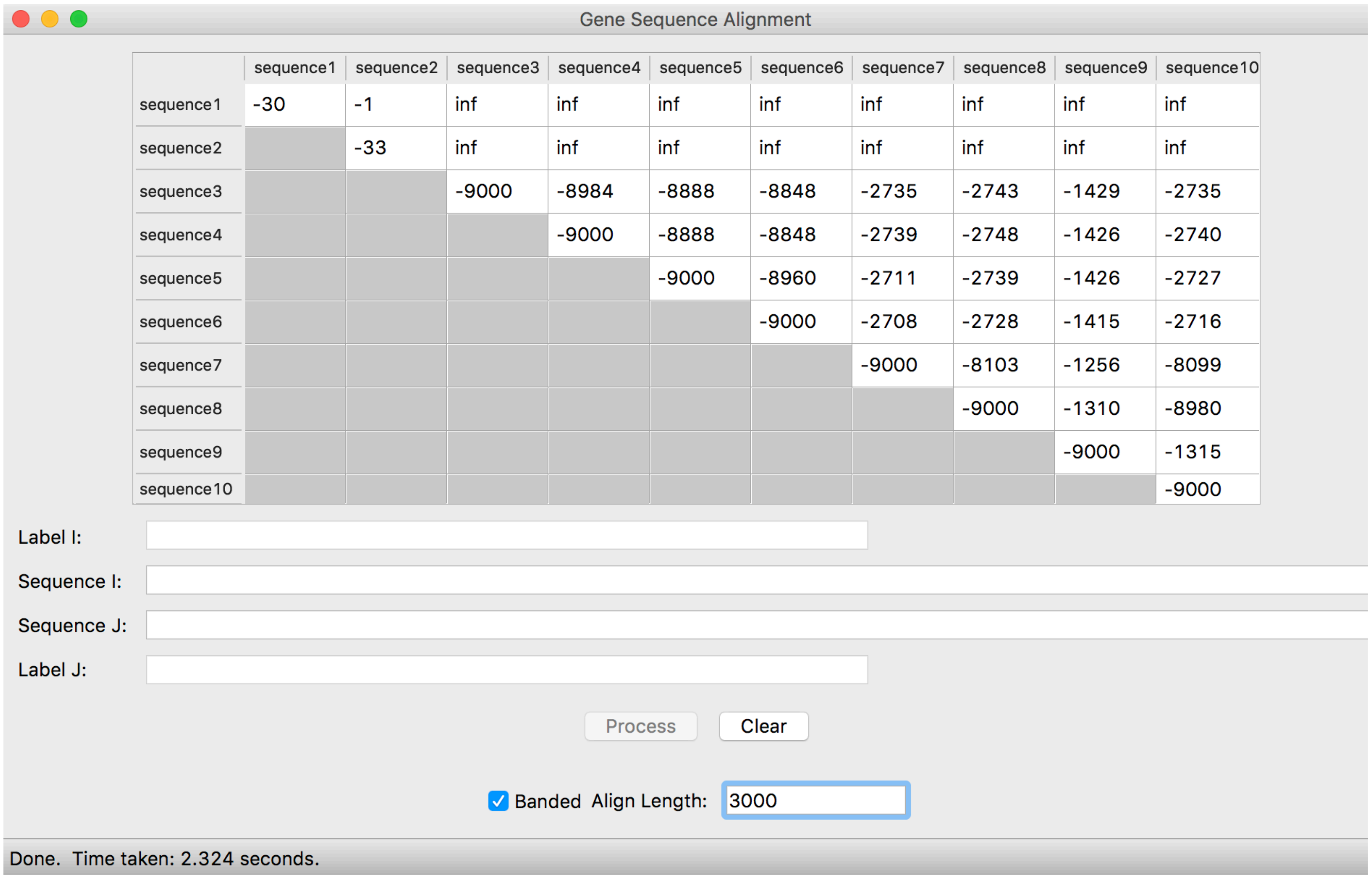
Sequence J:

Label J:

Banded Align Length: 1000

Done. Time taken: 1 mins and 1.837 seconds.

Banded k = 3000



	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

Label I:

Sequence I:

Sequence J:

Label J:

[✓] Banded Align Length: 3000

Done. Time taken: 2.324 seconds.

4.

Extracted Alignment, Unrestricted, k = 1000 #3, #10

gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a
-a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt

Extracted Alignment, Banded, k = 3000 #3, #10

gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a
-a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt

5.

```
#!/usr/bin/python3

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))
```

```python
import math
import time

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

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


class GeneSequencing:

    def __init__(self):
        self.alignment1 = None
        self.alignment2 = None

    # This is the method called by the GUI.  _sequences_ is a list of the ten sequences, _table_ is a
    # handle to the GUI so it can be updated as you find results, _banded_ is a boolean that tells
    # you whether you should compute a banded alignment or full alignment, and _align_length_ tells you
    # how many base pairs to use in computing the alignment

    def align(self, sequences, table, banded, align_length):
        self.banded = banded
        self.MaxCharactersToAlign = align_length
        results = []

        for i in range(len(sequences)):  # Rows O(10) so constant O(1)
            jresults = []
            for j in range(len(sequences)):  # Columns O(10) so constant O(1)

                if self.banded:
                    cost = self.bandedAlg(sequences[i], sequences[j])  # O(kn) time and space

                else:
                    # Compare each row to every column and then move down a row
                    cost = self.unrestricted(sequences[i], sequences[j])  # O(mn) time and space

                if (j < i):
                    s = {}
                else:
###############################################################################################
###
                    # your code should replace these three statements and populate the three variables: score, alignment1 and
alignment2
                    score = cost  # O(1)
                    alignment1 = self.alignment1.format(i + 1, len(sequences[i]), align_length,
                                                        ',BANDED' if banded else '')
                    alignment2 = self.alignment2.format(j + 1, len(sequences[j]), align_length,
                                                        ',BANDED' if banded else '')

###############################################################################################
###
                    s = {'align_cost': score, 'seqi_first100': alignment1, 'seqj_first100': alignment2}
                    table.item(i, j).setText('{}'.format(int(score) if score != math.inf else score))
                    table.repaint()
                jresults.append(s)
            results.append(jresults)
        return results
```

```
  # The overall time complexity here is O(nm) where m is the size of the sideSequence and n is the size of the
topSequence.
  #   This comes from making the 2d arrays as well as the for loop that runs mn time, and the calculateSequences. Thus
  #   it equals O(4nm) but the 4 is dropped since it is a constant and it just comes out to be O(nm)
  # The overall space complexity here is O(nm) where m is the size of the sideSequence and n is the size of the
topSequence.
  #   This comes from making the two 2d arrays so it would be O(2mn) but the 2 is dropped since it is a constant and it
  #   just comes out to be O(nm)
  def unrestricted(self, sequence1, sequence2):
      maxAlignment = self.MaxCharactersToAlign  # O(1)
      topSequence = list(sequence1)  # O(1)
      sideSequence = list(sequence2)  # O(1)

      # Checks to see how big the amount of columns n in the 2d array there will be
      if (len(sequence1) > maxAlignment):  # O(1)
          n = maxAlignment  # O(1)
          topSequence = list(sequence1[:maxAlignment])  # O(1) grabs from start to maxAlignment of the list
      else:
          n = len(sequence1)  # O(1)

      # Checks to see how big the amount of rows m in the 2d array there will be
      if (len(sequence2) > maxAlignment):  # O(1)
          m = maxAlignment  # O(1)
          sideSequence = list(sequence2[:maxAlignment])  # O(1) grabs from start to maxAlignment of the list
      else:
          m = len(sequence2)  # O(1)

      columns = n + 1  # Number of columns O(1) zero column has INDELS which are multiples of five
      rows = m + 1  # Number of rows O(1) zero row has INDELS which are multiples of five

      array = [[0 for i in range(columns)] for j in
               range(rows)]  # O(nm) time & space makes extra row and column for "-"
      array[0][0] = 0  # O(1)
      traceback_array = [["-" for i in range(columns)] for j in range(rows)]  # O(nm) time and space
      traceback_array[0][0] = None  # O(1)

      # these are used to fill in the first row and first column by
      # a multiple of five since that is how much an indel costs
      for i in range(1, rows):  # O(m) number of rows or chars in sequence2
          array[i][0] = i * INDEL  # O(1)
          traceback_array[i][0] = "A"  # O(1) A means that the number came from the above one it was an insert

      for j in range(1, columns):  # O(n) number of columns or chars in sequence1
          array[0][j] = j * INDEL
          traceback_array[0][j] = "L"  # O(1) L means that the number came from the left one it was a delete

      # O(mn) overall it loops through the number of rows m times which each
      #  loop through the number of columns n times
      for i in range(1, rows):  # O(m) where m is the number of characters in sequence2
          for j in range(1, columns):  # O(n) where n is the number of characters in sequence1
              char1 = sideSequence[i - 1]  # O(1)
              char2 = topSequence[j - 1]  # O(1)
              sameChar = char1 == char2  # O(1)
              array[i][j] = self.minimum(array, traceback_array, i, j, sameChar)  # O(1)

      self.calculateSequences(topSequence, sideSequence, traceback_array)  # O(nm)

      return array[rows - 1][columns - 1]  # O(1)

  # This function is just performing lookups in the 2d array which happen in constant time and comparing
  #   the currMin to a new number which is also constant time, so Time complexity is O(1)
  # The space complexity here is O(1) since we are not making any new space in the array we are just overwriting
```

```
    #   the (i, j) space in the trackback_array
    def minimum(self, array, trackback_array, i, j, sameChar):
        above = array[i - 1][j]  # O(1)
        left = array[i][j - 1]  # O(1)
        diagonal = array[i - 1][j - 1]  # O(1)

        if sameChar:  # O(1)
            currMin = diagonal + MATCH  # O(1)
            trackback_array[i][j] = "D"  # O(1) D means it came from the diagonal
        else:
            currMin = diagonal + SUB  # O(1)
            trackback_array[i][j] = "D"  # O(1) D means it came from the diagonal

        if currMin > (left + INDEL):  # O(1)
            currMin = left + INDEL  # O(1)
            trackback_array[i][j] = "L"  # O(1) L means it came from the left a delete

        if currMin > (above + INDEL):
            currMin = above + INDEL
            trackback_array[i][j] = "A"  # O(1) A means it came from above an insert

        return currMin

    # The time complexity here is  O(nm) because of the while loop which will at the worst case go through the
    #   traceback_array the lengths of the topSequence * the sideSequence which is n * m since it keeps looping all the
way
    #   to the first cell at (0,0) since that is where the None is stored
    # The space complexity is O(n + m) since we are making two new strings for the sequences they may have "-"
    #   in them. Thus the overall space complexity would be O(d) where d is equal to which ever length is greater
between
    #   n (topSequence) and m (sideSequenece)
    def calculateSequences(self, topSequence, sideSequence, traceback_array):
        tempSeqI = None  # O(1)
        tempSeqJ = None  # O(1)
        currTopChar = len(topSequence)
        currSideChar = len(sideSequence)
        while traceback_array[currSideChar][
            currTopChar] is not None:  # O(nm) cause it repeats all the way to (0,0) in traceback array
            source = traceback_array[currSideChar][currTopChar]  # O(1)

            if source is "D":  # O(1) a match or substitution
                if tempSeqI is not None:  # O(1)
                    tempSeqI += topSequence[currTopChar - 1]  # O(1)
                else:
                    tempSeqI = topSequence[currTopChar - 1]  # O(1)

                if tempSeqJ is not None:  # O(1)
                    tempSeqJ += sideSequence[currSideChar - 1]  # O(1)
                else:
                    tempSeqJ = sideSequence[currSideChar - 1]  # O(1)

                currTopChar -= 1  # O(1)
                currSideChar -= 1  # O(1)

            elif source is "L":  # O(1) a deletion
                if tempSeqI is not None:  # O(1)
                    tempSeqI += topSequence[currTopChar - 1]  # O(1)
                else:
                    tempSeqI = topSequence[currTopChar - 1]  # O(1)

                if tempSeqJ is not None:  # O(1)
                    tempSeqJ += "-"  # O(1)
                else:
                    tempSeqJ = "-"  # O(1)

                currTopChar -= 1  # O(1) move over a column only since we moved left

            elif source is "A":  # O(1) an insert
                if tempSeqI is not None:  # O(1)
                    tempSeqI += "-"  # O(1)
                else:
                    tempSeqI = "-"  # O(1)
                if tempSeqJ is not None:  # O(1)
                    tempSeqJ += sideSequence[currSideChar - 1]  # O(1)
                else:
                    tempSeqJ = sideSequence[currSideChar - 1]  # O(1)

                currSideChar -= 1  # O(1) move up a row since we inserted

            else:
                print("Error: Table didn't contain an A, L or D!")

        if tempSeqI is None:
            self.alignment1 = "No Alignment Possible"
        else:
            tempSeqI = tempSeqI[::-1]  # O(n) where n is the size of tempSeqI this reverses the string
            self.alignment1 = tempSeqI[:100]  # O(100) = O(1) takes first 100 chars and puts into alignment1

        if tempSeqJ is None:
            self.alignment2 = "No Alignment Possible"
        else:
            tempSeqJ = tempSeqJ[::-1]  # O(m) where m is the size of tempSeqJ this reverses the string
            self.alignment2 = tempSeqJ[:100]  # O(100) = O(1) takes first 100 chars and puts into alignment2

    # The overall time complexity here is O(nk) where k is 7 and n is the size of the topSequence or sideSequence
whichever is shorter.
    #   This comes from making the 2d arrays as well as the for loop that runs kn time, and the
bandedCalculateSequences. Thus
    #   it equals O(4nk) but the 4 is dropped since it is a constant and it just comes out to be O(nk)
    # The overall space complexity here is O(nk) where k is 7 and n is the size of the topSequence or sideSequence
whichever is shorter.
    #   The number of rows can at most be the shorter sequence + 3 since d = 3. But the 3 is dropped in Big-O since it is
a constant
    #   This comes from making the two 2d arrays so it would be O(2kn) but the 2 is dropped since it is a constant and it
    #   just comes out to be O(nk)
    def bandedAlg(self, sequence1, sequence2):
        maxAlignment = self.MaxCharactersToAlign  # O(1)
        topSequence = list(sequence1)  # O(1)
        sideSequence = list(sequence2)  # O(1)

        # Checks to see how big the amount of columns n in the 2d array there will be
        if (len(sequence1) > maxAlignment):  # O(1)
            n = maxAlignment  # O(1)
            topSequence = list(sequence1[:maxAlignment])  # O(1) grabs from start to maxAlignment of the list
        else:
            n = len(sequence1)  # O(1)

        # Checks to see how big the amount of rows m in the 2d array there will be
        if (len(sequence2) > maxAlignment):  # O(1)
            m = maxAlignment  # O(1)
            sideSequence = list(sequence2[:maxAlignment])  # O(1) grabs from start to maxAlignment of the list
        else:
            m = len(sequence2)  # O(1)
        cols = n + 1  # Number of columns O(1) O column has INDELS
        rows = m + 1  # Number of rows O(1) O row has INDELS

        diff = cols - rows  # O(1)

        # If the differance is big then it will have much more insertions and deletes then are allowed in a row
        if diff > MAXINDELS or diff < -MAXINDELS:
            self.alignment1 = "No Alignment Possible"  # O(1)
            self.alignment2 = "No Alignment Possible"  # O(1)
            return math.inf

        if len(topSequence) > len(sideSequence):  # O(1)
            topSequenceBigger = True  # O(1)
        else:
            topSequenceBigger = False  # O(1)

        # O(1) checks to see how many more rows are needed to add to the smaller sequence if a sequence is bigger than
the other one
        differentOfLengths = abs(len(topSequence) - len(sideSequence))

        if (topSequenceBigger):
            rows = len(sideSequence) + 1 + differentOfLengths  # O(1) this is n as seen below, sideSequence is smaller
            temp = topSequence
            topSequence = sideSequence
            sideSequence = temp

        else:
            rows = len(topSequence) + 1 + differentOfLengths  # O(1) this is n as seen below, topSequence is smaller

        columns = (2 * MAXINDELS) + 1  # O(1) this is k = 7

        array = [["nil" for i in range(columns)] for j in range(rows)]  # O(kn) time & space makes extra row for -
        traceback_array = [["-" for i in range(columns)] for j in range(rows)]  # O(kn) time and space

        # Sets up the base cases of array
        array[0][3] = 0  # O(1)
        array[0][4] = 5  # O(1)
        array[0][5] = 10  # O(1)
        array[0][6] = 15  # O(1)

        array[1][2] = 5  # O(1)
        array[2][1] = 10  # O(1)
        array[3][0] = 15  # O(1)

        traceback_array[0][3] = None  # O(1)
        traceback_array[0][4] = "L"  # O(1)
        traceback_array[0][5] = "L"  # O(1)
        traceback_array[0][6] = "L"  # O(1)

        traceback_array[1][2] = "A"  # O(1)
        traceback_array[2][1] = "A"  # O(1)
        traceback_array[3][0] = "A"  # O(1)

        # Sets up upper and lower triangle that isn't in the array
        array[0][0] = math.inf  # O(1)
        array[0][1] = math.inf  # O(1)
        array[0][2] = math.inf  # O(1)
        array[1][0] = math.inf  # O(1)
        array[1][1] = math.inf  # O(1)
        array[2][0] = math.inf  # O(1)
        array[rows - 1][6] = math.inf  # O(1)
        array[rows - 1][5] = math.inf  # O(1)
        array[rows - 1][4] = math.inf  # O(1)
        array[rows - 2][6] = math.inf  # O(1)
        array[rows - 2][5] = math.inf  # O(1)
        array[rows - 3][6] = math.inf  # O(1)

        # O(kn) overall it loops through the number of rows n times which each
        #   loop through the number of columns k (7)
        for i in range(0, rows):  # O(n) n is length of which ever sequence is shorter
            for j in range(0, columns):  # O(k) k is 7
                if array[i][j] == "nil":
                    array[i][j] = self.bandedMin(array, traceback_array, topSequence, sideSequence, i, j)  # O(1)

        column = MAXINDELS  # O(1)
        while array[rows - 1][column] is math.inf:  # O(7) since that is the most columns so it just comes down to O(1)
            column -= 1
        self.bandedCalculateSequences(topSequence, sideSequence, traceback_array, rows, column)  # O(kn)
        return array[rows - 1][column]

    # This function is just performing lookups in the 2d array which happen in constant time and comparing
    #    the currMin to a new number which is also constant time, so Time complexity is O(1)
    # The space complexity here is O(1) since we are not making any new space in the array we are just overwriting
    #     the (i, j) space in the traceback_array
    def bandedMin(self, array, traceback_array, topSequence, sideSequence, i, j):
        currMin = math.inf  # O(1)
        if j is 0:  # O(1)
            left = math.inf  # O(1)
            diagonal = array[i - 1][j]  # O(1)
            above = array[i - 1][j + 1]  # O(1)
        elif j is 6:  # O(1)
            above = math.inf  # O(1)
            diagonal = array[i - 1][j]  # O(1)
            left = array[i][j - 1]  # O(1)
        else:
            diagonal = array[i - 1][j]  # O(1)
            left = array[i][j - 1]  # O(1)
            above = array[i - 1][j + 1]  # O(1)

        if diagonal is not math.inf:  # O(1)
            if i - (4 - j) > len(topSequence) - 1:
                return math.inf
            else:
                if topSequence[i - (4 - j)] == sideSequence[i - 1]:  # O(1)
                    currMin = diagonal + MATCH  # O(1)
                    traceback_array[i][j] = "D"  # O(1)
                else:
                    currMin = diagonal + SUB  # O(1)
                    traceback_array[i][j] = "D"  # O(1)

        if left is not math.inf:  # O(1)
            if currMin > (left + INDEL):  # O(1)
                currMin = left + INDEL  # O(1)
                traceback_array[i][j] = "L"  # O(1) L means it came from the left a delete

        if above is not math.inf:  # O(1)
            if currMin > (above + INDEL):  # O(1)
                currMin = above + INDEL  # O(1)
                traceback_array[i][j] = "A"  # O(1) A means it came from above an insert

        return currMin

    # The time complexity here is  O(nk) because of the while loop which will at the worst case go through the
    #   trackback_array the n times which is the amount of rows and k which is the amount of columns
```

```
  #   which is n * k since it keeps looping all the way to the first cell at (0,3) since that is where the None is stored
  # The space complexity is O(n + m) since we are making two new strings for the sequences they will may have "-"
  #   in them. Thus the overall space complexity would be O(d) where d is equal to which ever length is greater
between
  #   n (topSequence) and m (sideSequenece)
  def bandedCalculateSequences(self, topSequence, sideSequence, traceback_array, rows, column):
      tempSeqI = None  # O(1)
      tempSeqQ = None  # O(1)

      while traceback_array[rows - 1][column] is not None:  # O(nk) cause it repeats all the way to (0,3) in traceback
array
          source = traceback_array[rows - 1][column]  # O(1)

          if source is "D":  # O(1) a match or substitution
              if tempSeqI is not None:  # O(1)
                  tempSeqI += topSequence[rows - (4 - column) - 1]  # O(1)
              else:
                  tempSeqI = topSequence[rows - (4 - column) - 1]  # O(1)

              if tempSeqQ is not None:  # O(1)
                  tempSeqQ += sideSequence[rows - 2]  # O(1)
              else:
                  tempSeqQ = sideSequence[rows - 2]  # O(1)

              rows -= 1  # O(1)

          elif source is "L":  # O(1) a deletion
              if tempSeqI is not None:  # O(1)
                  tempSeqI += topSequence[rows - (4 - column) - 1]  # O(1)
              else:
                  tempSeqI = topSequence[rows - (4 - column) - 1]  # O(1)

              if tempSeqQ is not None:  # O(1)
                  tempSeqQ += "-"  # O(1)
              else:
                  tempSeqQ = "-"  # O(1)

              column -= 1  # O(1) move over a column only since we moved left

          elif source is "A":  # O(1) an insert
              if tempSeqI is not None:  # O(1)
                  tempSeqI += "-"  # O(1)
              else:
                  tempSeqI = "-"  # O(1)
              if tempSeqQ is not None:  # O(1)
                  tempSeqQ += sideSequence[rows - 2]  # O(1)
              else:
                  tempSeqQ = sideSequence[rows - 2]  # O(1)

              rows -= 1  # O(1) move up a row since we inserted
              column += 1  # O(1) move up a column since we inserted

          else:
              print("Error: Table didn't contain an A, L or D!")

      if tempSeqI is None:
          self.alignment1 = "No Alignment Possible"
      else:
          tempSeqI = tempSeqI[::-1]  # O(n) where n is the size of tempSeqI this reverses the string
          self.alignment1 = tempSeqI[:100]  # O(100) = O(1) takes first 100 chars and puts into alignment1

      if tempSeqQ is None:
          self.alignment2 = "No Alignment Possible"
      else:
          tempSeqQ = tempSeqQ[::-1]  # O(m) where m is the size of tempSeqJ this reverses the string
          self.alignment2 = tempSeqQ[:100]  # O(100) = O(1) takes first 100 chars and puts into alignment2
```