1. **Time and Space Complexities**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **TIME** | **unrestrictedAlignment/bandedAlignment** | **generateAlignment**  **/generateBandedAlignment** | **findMin** | **checkBoundaries** | sum | **Align**  **(calls others)** |
| **Unrestricted** | O(nm) | O(n + m) | O(1) | O(1) | O(nm + n + m + 1 + 1) | O(nm) |
| **Banded** | O(kn) | O(n) | O(1) | O(1) | O(kn + n + 1 + 1) | O(kn) |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SPACE** | **unrestrictedAlignment/bandedAlignment** | **generateAlignment**  **/generateBandedAlignment** | **findMin** | **checkBoundaries** | **sum** | **Align**  **(calls others)** |
| **Unrestricted** | O(nm) | O(n + m) | O(1) | O(1) | O(nm + n + m + 1 + 1) | O(nm) |
| **Banded** | O(kn) | O(n) | O(1) | O(1) | O(kn + n + 1 + 1) | O(kn) |

* **unretrictedAlignment:**

        Time:

            O(nm) it will repeat at max n\*m times as there are m cols and n rows max

            the cols and rows are the min of lengths of the strings and align\_length

            In most cases the align\_length < the length of the strings so in most cases it

            would be O(a^2) where a is the align\_length

        Space:

            O(nm) the two arrays used for storage are cols = m and rows = n

            and so the storage needed is 2\*m\*n (2 arrays)

            In most cases the align\_length < the length of the strings so in most cases it

            would be O(a^2) where a is the align\_length

* **bandedAlignment**

Time:

O(kn) it will repeat at max k times as the cols are set to be 2\*MAXINDELS + 1 \*

n which is how many rows there are.

In most cases the align\_length < the length of the strings so in most cases it

would be O(ka) where a is the align\_length

Space:

O(kn) the two arrays used for storage are cols = k and rows = n

and so the storage needed is 2\*k\*n (2 arrays)

In most cases the align\_length < the length of the strings so in most cases it

would be O(ka) where a is the align\_length

* **generateAlignment**

       Time:

            O(n + m) as the max legnth of the alignment which would be the max repeats.

            This would be the case if it were all indels the length of one string and

            then the entire other string and vice versa. Because align\_length is normally

            less than the legnths the O(a) where a is the align\_length in most cases

        Space:

            O(n + m) as the max legnth of the alignment.

            This would be the case if it were all indels the length of one string and

            then the entire other string and vice versa. Because align\_length is normally

            less than the legnths the O(a) where a is the align\_length in most cases

* **generateBandedAlignment**

        Time:

            O(n) as the max legnth of the alignment which would be the max repeats.

            This would be because the legnth is bound by the legnth of the smaller string

            + some indels which are restricted by the banding algorithm so it would be

            O(n + i) (i is number of indels) i << n  --> O(n)

            align\_legnth is normally less than the legnths so it can be simplified to

            O(a) where a is the align\_length in most cases

        Space:

            O(n) as the max legnth of the alignment.

            This would be because the legnth is bound by the legnth of the smaller string

            + some indels which are restricted by the banding algorithm so it would be

            O(n + i) (i is number of indels) i << n  --> O(n)

            align\_legnth is normally less than the legnths so it can be simplified to

            O(a) where a is the align\_length in most cases

* **findMin**

Time:

O(1) just 3 steps max

Space:

O(1) doesn't store anything

* **checkBoundaries**

Time:

O(1) just 3 steps max

Space:

O(1) doesn't store anything

1. **Explanation of Algorithm**

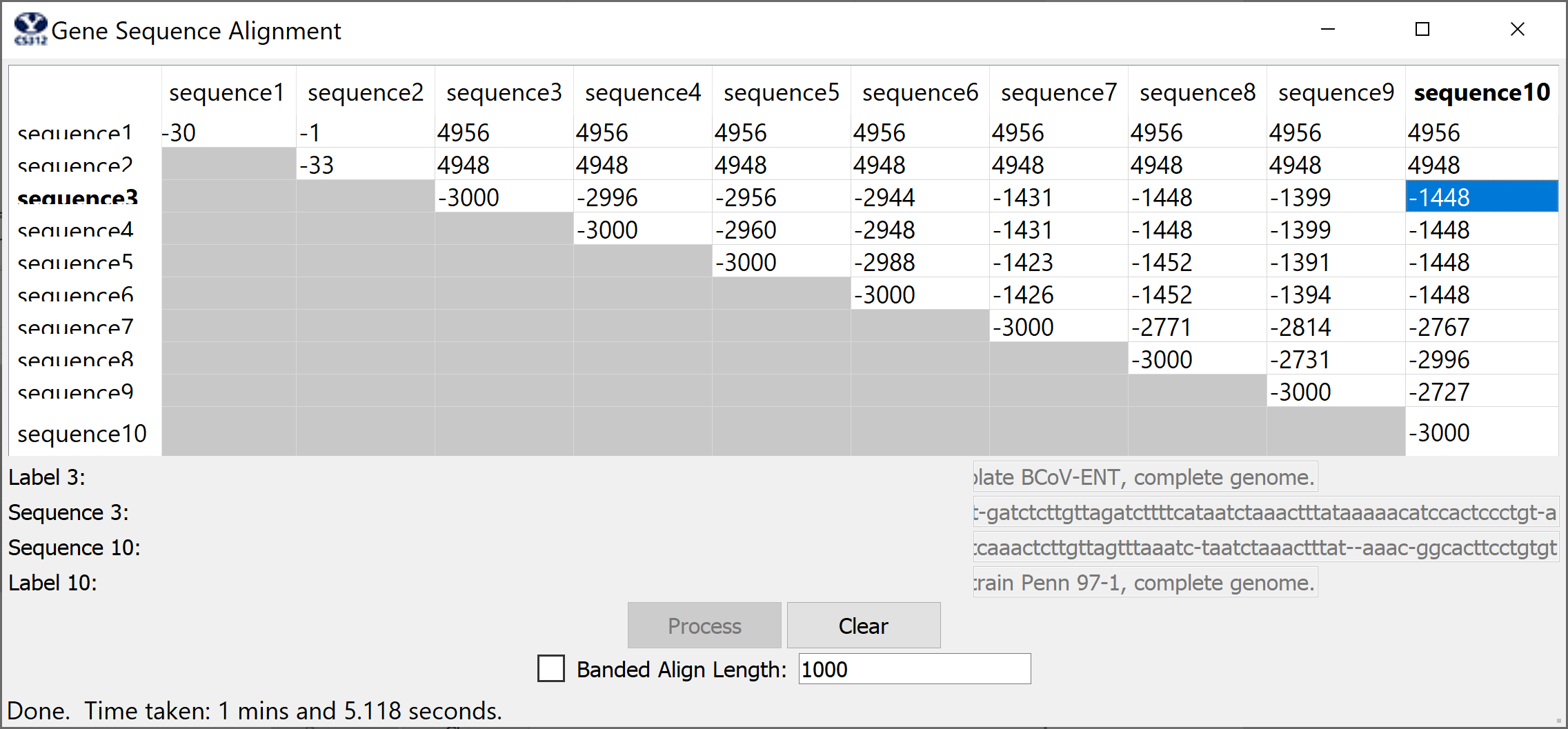
This algorithm works by going through the strings and seeing how they best align in a dynamic way. At each step it will determine what is the best step at this point (going right, down, or diagonal). It will then store the best result in two arrays, cost and fromArray (fromArray is for backtracking later). The cost stored in the cost array is the sum of the previous solution plus the cost to get to the new solution. The algorithm is continued till we have analyzed the entire alignment length.

The algorithm determines the best solution at each step by taking the value in the cells above, to the left, and diagonal and adding 5 if its from the left or above (insertion or deletions), 1 if the letters are different when traveling diagonal and -3 if they match. It then finds the min and adds this value to the cost array and the direction to the fromArray. The solution to the entire problem is in the bottom right corner in the unrestricted algorithm and the last cell in the last row with a solution in it. This is read and returned to the GUI to show the cost.

The Backtracking is done by getting the index of the final solution in the fromArrray and then going backwards through the array. This is where the string of the alignment is calculated. If it was from above, one letter from the vertical string is added and a ‘-‘ for an insertion or deletion is added for the horizontal string. The opposite is done if it was from the left ad a letter from each string is added from diagonal. The indexes are then adjusted to go to the cell that is being pointed to by the current cell. In unrestricted this is literally above, to the left, or diagonal, but with banded it is different as diagonal is above, above is diagonal to the right and left is still to the left. This is how the banding algorithm fits everything into a smaller array.

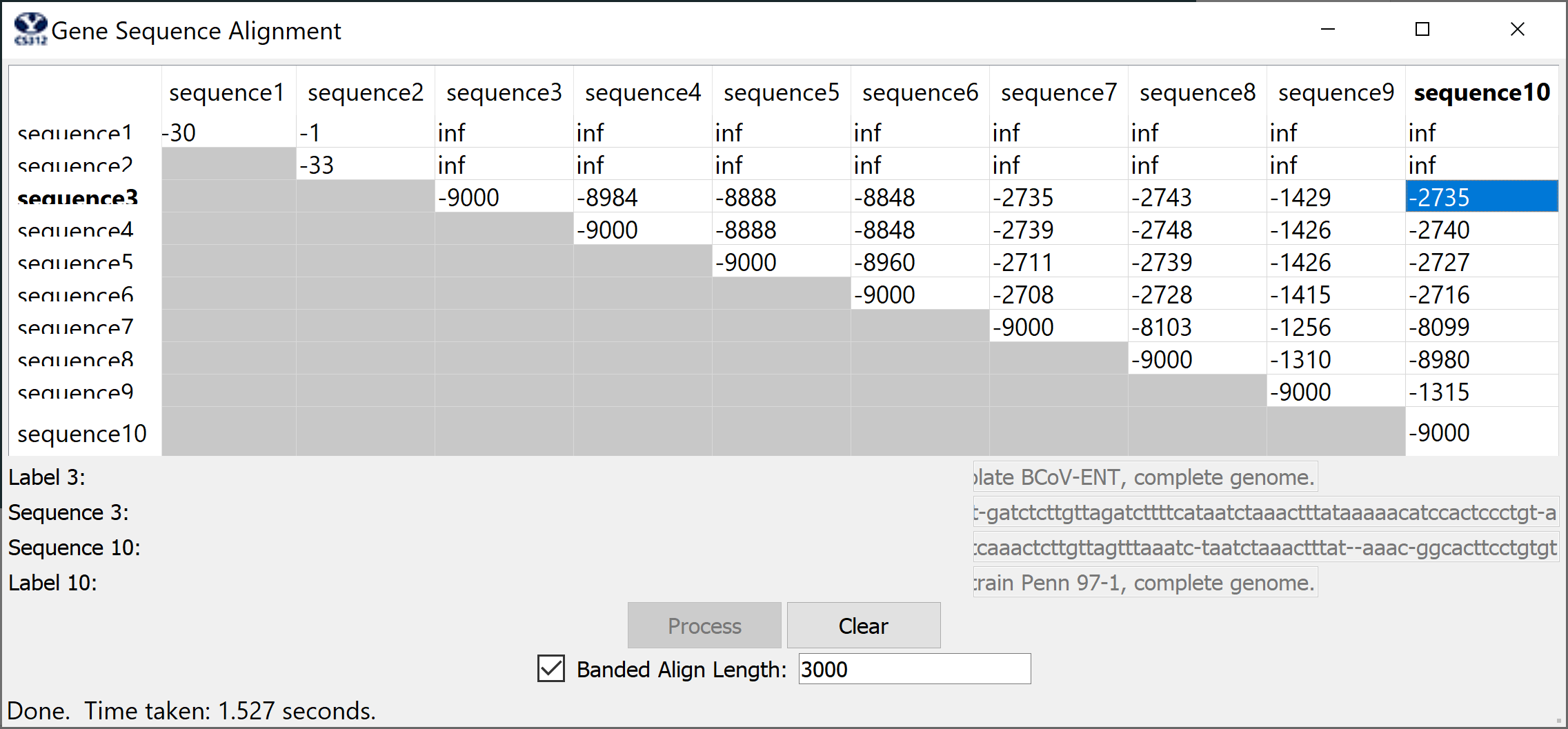
**3. Results:**

n =1000 Unrestricted



**Alignments:**

n =3000 Banded d=3



**4. Alignments:**

n =1000 Unrestricted

3 gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a 10 -a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt

n =3000 Banded d=3

3 gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a 10 -a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt

**5. Code:**

*#!/usr/bin/python3*

*# from PyQt5.QtCore import QLineF, QPointF*

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):

        pass

    '''

        Checks the boundaries and returns True or False

        Time:

            O(1) just 3 steps max

        Space:

            O(1) doesn't store anything

    '''

**def** checkBoundaries(self, i, j, rows, cols):

        if i < 0 or j < 0:

            return False

        if i >= rows or j >= cols:

            return False

        else:

            return True

    '''

        Find min and returns string of that is the min

        Time:

            O(1) just 3 steps max

        Space:

            O(1) doesn't store anything

    '''

**def** findMin(self, leftCost, aboveCost, diagonalCost):

        if diagonalCost <= leftCost and diagonalCost <= aboveCost:

            return "diagonal"

        elif aboveCost <= leftCost and aboveCost <= diagonalCost:

            return "above"

        elif leftCost <= diagonalCost and leftCost <= aboveCost:

            return "left"

    '''

        Generates the Alignment String

n the length of the smaller string

        Time:

            O(n) as the max legnth of the alignment which would be the max repeats.

            This would be because the legnth is bound by the legnth of the smaller string

            + some indels which are restricted by the banding algorithm so it would be

            O(n + i) (i is number of indels) i << n  --> O(n)

            align\_legnth is normally less than the legnths so it can be simplified to

            O(a) where a is the align\_length in most cases

        Space:

            O(n) as the max legnth of the alignment.

            This would be because the legnth is bound by the legnth of the smaller string

            + some indels which are restricted by the banding algorithm so it would be

            O(n + i) (i is number of indels) i << n  --> O(n)

            align\_legnth is normally less than the legnths so it can be simplified to

            O(a) where a is the align\_length in most cases

    '''

**def** generateBandedAlignment(self, horizontalSeq, verticalSeq, fromArray, align\_length,

retJ):

        horizontalAlignment = []

        verticalAlignment = []

        v = len(fromArray) - 1

        h = retJ - 1  *# 7 in our case*

        while fromArray[v][h] != "START":

            adjustH = v + h - MAXINDELS

            if fromArray[v][h] == "LEFT":

                verticalAlignment.append("-")

                horizontalAlignment.append(horizontalSeq[adjustH-1])

                h -= 1

            elif fromArray[v][h] == "ABOVE":

                verticalAlignment.append(verticalSeq[v-1])

                horizontalAlignment.append("-")

                v -= 1

                h += 1

            elif fromArray[v][h] == "DIAGONAL":

                verticalAlignment.append(verticalSeq[v-1])

                v -= 1

                horizontalAlignment.append(horizontalSeq[adjustH - 1])

            else:

                print(fromArray[v][h])

                raise ValueError("UNKNOWN VALUE")

        return "".join(horizontalAlignment[::-1]), "".join(verticalAlignment[::-1])

    '''

        Generates the Alignment String

       n and m are the lengths of the strings

        Time:

            O(n + m) as the max legnth of the alignment which would be the max repeats.

            This would be the case if it were all indels the length of one string and

            then the entire other string and vice versa. Because align\_length is normally

            less than the legnths the O(a) where a is the align\_length in most cases

        Space:

            O(n + m) as the max legnth of the alignment.

            This would be the case if it were all indels the length of one string and

            then the entire other string and vice versa. Because align\_length is normally

            less than the legnths the O(a) where a is the align\_length in most cases

    '''

**def** generateAlignment(self, horizontalSeq, verticalSeq, fromArray, align\_length):

        horizontalAlignment = []

        verticalAlignment = []

        v = len(fromArray) - 1

        h = len(fromArray[0]) - 1

        while fromArray[v][h] != "START":

            if fromArray[v][h] == "LEFT":

                verticalAlignment.append("-")

                horizontalAlignment.append(horizontalSeq[h-1])

                h -= 1

            elif fromArray[v][h] == "ABOVE":

                verticalAlignment.append(verticalSeq[v-1])

                horizontalAlignment.append("-")

                v -= 1

            elif fromArray[v][h] == "DIAGONAL":

                verticalAlignment.append(verticalSeq[v-1])

                v -= 1

                horizontalAlignment.append(horizontalSeq[h-1])

                h -= 1

            else:

                print(fromArray[v][h])

                raise ValueError("UNKNOWN VALUE")

        return "".join(horizontalAlignment[::-1]), "".join(verticalAlignment[::-1])

    '''

        Generates the Alignment String

        k is the bandwidth and n is the length of the smaller string

        nested for loop dominates so the other function calls and the generateAlignment

        does not make an impact on big Oh

        Time:

            O(kn) it will repeat at max k times as the cols are set to be 2\*MAXINDELS + 1 \*

            n which is how many rows there are.

            In most cases the align\_length < the length of the strings so in most cases it

            would be O(ka) where a is the align\_length

        Space:

            O(kn) the two arrays used for storage are cols = k and rows = n

            and so the storage needed is 2\*k\*n (2 arrays)

            In most cases the align\_length < the length of the strings so in most cases it

            would be O(ka) where a is the align\_length

    '''

**def** bandedAlignment(self, horizontalSeq, verticalSeq, align\_length):

        maxJ = min(align\_length, len(horizontalSeq))

        cols = min((align\_length + 1), len(horizontalSeq) + 1)

*# n is legnth of smaller string which is always the 2nd argument*

        rows = min((align\_length + 1), len(verticalSeq) + 1)

        if cols - rows > MAXINDELS:

            return float('inf'), "No Alignment Possible", "No Alignment Possible"

        cols = 2\*MAXINDELS + 1  *# 2\*MAXINDELS + 1 == k*

        cost = [[float('inf')]\*(cols) for \_ in range(rows)]

        fromArray = [["NONE"]\*(cols) for \_ in range(rows)]

        cost[0][MAXINDELS] = 0

        fromArray[0][MAXINDELS] = "START"

        retJ = 0

        for i in range(rows):

            for j in range(cols):

                if i == 0 and j <= MAXINDELS:

                    continue

                adjustJ = j + i - MAXINDELS

                if adjustJ > maxJ or adjustJ < 0:

                    continue

                if i == rows - 1:

                    retJ += 1

*# print(jEnd, adjustJ, i, j, len(horizontalSeq))*

                if horizontalSeq[adjustJ - 1] == verticalSeq[i-1]:

                    diagonalCost = (

                        cost[i - 1][j] + MATCH) if self.checkBoundaries(i - 1, j, rows, cols)

else float('inf')

                else:

                    diagonalCost = (

                        cost[i-1][j] + SUB) if self.checkBoundaries(i-1, j, rows, cols)

else float('inf')

                aboveCost = (

                    cost[i-1][j + 1] + INDEL) if self.checkBoundaries(i-1, j + 1, rows, cols)

else float('inf')

                leftCost = (

                    cost[i][j - 1] + INDEL) if self.checkBoundaries(i, j - 1, rows, cols)

else float('inf')

                minCost = self.findMin(

                    leftCost, aboveCost, diagonalCost)

                if minCost == "left":

                    cost[i][j] = leftCost

                    fromArray[i][j] = "LEFT"

                elif minCost == "above":

                    cost[i][j] = aboveCost

                    fromArray[i][j] = "ABOVE"

                elif minCost == "diagonal":

                    cost[i][j] = diagonalCost

                    fromArray[i][j] = "DIAGONAL"

        assert(cost[0][MAXINDELS] == 0)

        alignment1, alignment2 = self.generateBandedAlignment(

            horizontalSeq, verticalSeq, fromArray, align\_length, retJ)

        return cost[rows-1][retJ - 1], alignment1, alignment2

    '''

        Generates the Alignment String

        n and m are the lengths of the strings

        Time:

            O(nm) it will repeat at max n\*m times as there are m cols and n rows max

            the cols and rows are the min of lengths of the strings and align\_length

            In most cases the align\_length < the length of the strings so in most cases it

            would be O(a^2) where a is the align\_length

        Space:

            O(nm) the two arrays used for storage are cols = m and rows = n

            and so the storage needed is 2\*m\*n (2 arrays)

            In most cases the align\_length < the length of the strings so in most cases it

            would be O(a^2) where a is the align\_length

    '''

**def** unrestrictedAlignment(self, horizontalSeq, verticalSeq, align\_length):

        cols = min((align\_length + 1), len(horizontalSeq) + 1)  *# m*

        rows = min((align\_length + 1), len(verticalSeq) + 1)  *# n*

        cost = [[float('inf')]\*cols for \_ in range(rows)]

        fromArray = [["NONE"]\*cols for \_ in range(rows)]

        cost[0][0] = 0

        fromArray[0][0] = "START"

        for i in range(rows):

            for j in range(cols):

*# left*

                if i != 0 or j != 0:

                    if horizontalSeq[j-1] == verticalSeq[i-1]:

                        diagonalCost = (

                            cost[i - 1][j - 1] + MATCH) if self.checkBoundaries(i - 1, j - 1,

rows, cols) else float('inf')

                    else:

                        diagonalCost = (

                            cost[i-1][j-1] + SUB) if self.checkBoundaries(i-1, j-

1, rows, cols) else float('inf')

                    aboveCost = (

                        cost[i-1][j] + INDEL) if self.checkBoundaries(i-

1, j, rows, cols) else float('inf')

                    leftCost = (

                        cost[i][j-1] + INDEL) if self.checkBoundaries(i, j-

1, rows, cols) else float('inf')

                    minCost = self.findMin(leftCost, aboveCost, diagonalCost)

                    if minCost == "left":

                        cost[i][j] = leftCost

                        fromArray[i][j] = "LEFT"

                    elif minCost == "above":

                        cost[i][j] = aboveCost

                        fromArray[i][j] = "ABOVE"

                    elif minCost == "diagonal":

                        cost[i][j] = diagonalCost

                        fromArray[i][j] = "DIAGONAL"

        alignment1, alignment2 = self.generateAlignment(

            horizontalSeq, verticalSeq, fromArray, align\_length)

*# print(alignment1, alignment2)*

        return cost[rows-1][cols-1], alignment1, alignment2

*# This is the method called by the GUI.  \_sequences\_ is h list of the ten sequences, \_table\_ is h*

*# handle to the GUI so it can be updated as you find results, \_banded\_ is h boolean that tells*

*# you whether you should compute h banded alignment or full alignment, and \_align\_length\_ tells you*

*# how many base pairs to use in computing the alignment*

    '''

        calls either bandedAlignment or unrestrictedAlignment

        k is bandwidth

        n and m are lengths of the strings

        a is align\_length

        time and space:

            if banded:

                O(kn) --> avg large n's O(ka)

            else:

                O(nm) --> avg large n and m O(a^2)

        see other functions for explanation

    '''

**def** align(self, sequences, table, banded, align\_length):

        self.banded = banded

        self.MaxCharactersToAlign = align\_length

        results = []

        for i in range(len(sequences)):

            jresults = []

            for j in range(len(sequences)):

                if j < i:

                    s = {}

                else:

                    if i == j:

                        legnth = min((align\_length + 1),

                                     len(sequences[i]) + 1)

                        score, alignment1, alignment2 = MATCH \* \

                            (legnth-1), sequences[i], sequences[j]

                    elif banded:

                        if len(alignment1) > len(alignment2):

                            score, alignment1, alignment2 = self.bandedAlignment(

                                sequences[i], sequences[j], align\_length)

                        else:

                            score, alignment2, alignment1 = self.bandedAlignment(

                                sequences[j], sequences[i], align\_length)

                    else:

                        score, alignment1, alignment2 = self.unrestrictedAlignment(

                            sequences[i], sequences[j], align\_length)

*# if i == 2 and j == 9:*

*#     print("3", alignment1[:100])*

*#     print("10", alignment2[:100])*

                    s = {'align\_cost': score, 'seqi\_first100': alignment1[:100],

                         'seqj\_first100': alignment2[:100]}

                    table.item(i, j).setText('{}'.format(

                        int(score) if score != math.inf else score))

                    table.repaint()

                jresults.append(s)

            results.append(jresults)

        return results