1)

a) Unrestricted Algorithm

def unrestricted(self, seq1, seq2):

*self*.score = 0

*self*.matrix[(-1,-1)] = 0

        if(*self*.MaxCharactersToAlign > len(seq1)):

*self*.min\_length1 = len(seq1)

        else:

*self*.min\_length1 = *self*.MaxCharactersToAlign

        if(*self*.MaxCharactersToAlign > len(seq2)):

*self*.min\_length2 = len(seq2)

        else:

*self*.min\_length2 = *self*.MaxCharactersToAlign

        for i in range(-1,*self*.min\_length1):

            for j in range(-1, *self*.min\_length2):

                if(i==-1 and j==-1):

                    continue

                if(i == -1 and j >-1) :

*self*.matrix[-1,j] = *self*.matrix[-1,j-1] + INDEL

*self*.back\_pointers[-1,j] = (-1,j-1,'left')

                    continue

                if(i > -1 and j == -1):

*self*.matrix[i,-1] = *self*.matrix[i-1, -1] + INDEL

*self*.back\_pointers[i,-1] = (i-1, -1, 'top')

                    continue

                if(seq1[i] == seq2[j]):

                    diagonal = *self*.matrix[i-1,j-1] + MATCH

                    direction = *self*.getMin(*self*.matrix[i,j-1]+INDEL, *self*.matrix[i-1,j]+INDEL,diagonal)

                else:

                    diagonal = *self*.matrix[i-1,j-1] + SUB

                    direction = *self*.getMin(*self*.matrix[i,j-1]+INDEL, *self*.matrix[i-1,j]+INDEL,diagonal)

                if(direction == 'left'):

*self*.matrix[i,j] = *self*.matrix[i,j-1] + INDEL

*self*.back\_pointers[i,j] = (i,j-1,'left')

                    continue

                if(direction == 'top'):

*self*.matrix[i,j] = *self*.matrix[i-1,j] + INDEL

*self*.back\_pointers[i,j] = (i-1,j, 'top')

                    continue

                if(direction == 'diagonal'):

*self*.matrix[i,j] = diagonal

*self*.back\_pointers[i,j] = (i-1,j-1,'diagonal')

                    continue

*self*.score = *self*.matrix[*self*.min\_length1-1,*self*.min\_length2-1]

The above code is one of the function executed for unrestricted algorithm. Unrestricted function takes up majority of the runtime and space. The function has a loop inside a loop and it iterates over the length of the seq1 and length of seq2 (with the maximum of the align length given by the user). The matrix dictionary has a length of the number of iterations done by the for loops. Therefore the time complexity and space complexity of this algorithm is O(mn), where m is the size of seq1 and n is the size of seq2 with the maximum of align length.

b) banded algorithm

*self*.score = 0

*self*.matrix[(-1,-1)] = 0

        if(*self*.MaxCharactersToAlign > len(seq1)):

*self*.min\_length1 = len(seq1)

        else:

*self*.min\_length1 = *self*.MaxCharactersToAlign

        if(*self*.MaxCharactersToAlign > len(seq2)):

*self*.min\_length2 = len(seq2)

        else:

*self*.min\_length2 = *self*.MaxCharactersToAlign

        for i in range(-1,*self*.min\_length1):

            if(-1 >= i - MAXINDELS):

                j\_start = -1

            else:

                j\_start = i - MAXINDELS

            if(*self*.min\_length2 <= i + MAXINDELS):

                j\_end = *self*.min\_length2

            else:

                j\_end = i + MAXINDELS + 1

            for j in range(j\_start, j\_end):

                if(i==-1 and j==-1):

                    continue

                if(i == -1 and j >-1) :

*self*.matrix[-1,j] = *self*.matrix[-1,j-1] + INDEL

*self*.back\_pointers[-1,j] = (-1,j-1,'left')

                    continue

                if(i > -1 and j == -1):

*self*.matrix[i,-1] = *self*.matrix[i-1, -1] + INDEL

*self*.back\_pointers[i,-1] = (i-1, -1, 'top')

                    continue

*self*.matrix.setdefault((i,j-1), float(inf))

*self*.matrix.setdefault((i-1,j),float(inf))

                if(seq1[i] == seq2[j]):

                    diagonal = *self*.matrix[i-1,j-1] + MATCH

                    direction = *self*.getMin(*self*.matrix[i,j-1]+INDEL, *self*.matrix[i-1,j]+INDEL,diagonal)

                else:

                    diagonal = *self*.matrix[i-1,j-1] + SUB

                    direction = *self*.getMin(*self*.matrix[i,j-1]+INDEL, *self*.matrix[i-1,j]+INDEL,diagonal)

                if(direction == 'left'):

*self*.matrix[i,j] = *self*.matrix[i,j-1] + INDEL

*self*.back\_pointers[i,j] = (i,j-1,'left')

                    continue

                if(direction == 'top'):

*self*.matrix[i,j] = *self*.matrix[i-1,j] + INDEL

*self*.back\_pointers[i,j] = (i-1,j, 'top')

                    continue

                if(direction == 'diagonal'):

*self*.matrix[i,j] = diagonal

*self*.back\_pointers[i,j] = (i-1,j-1,'diagonal')

                    continue

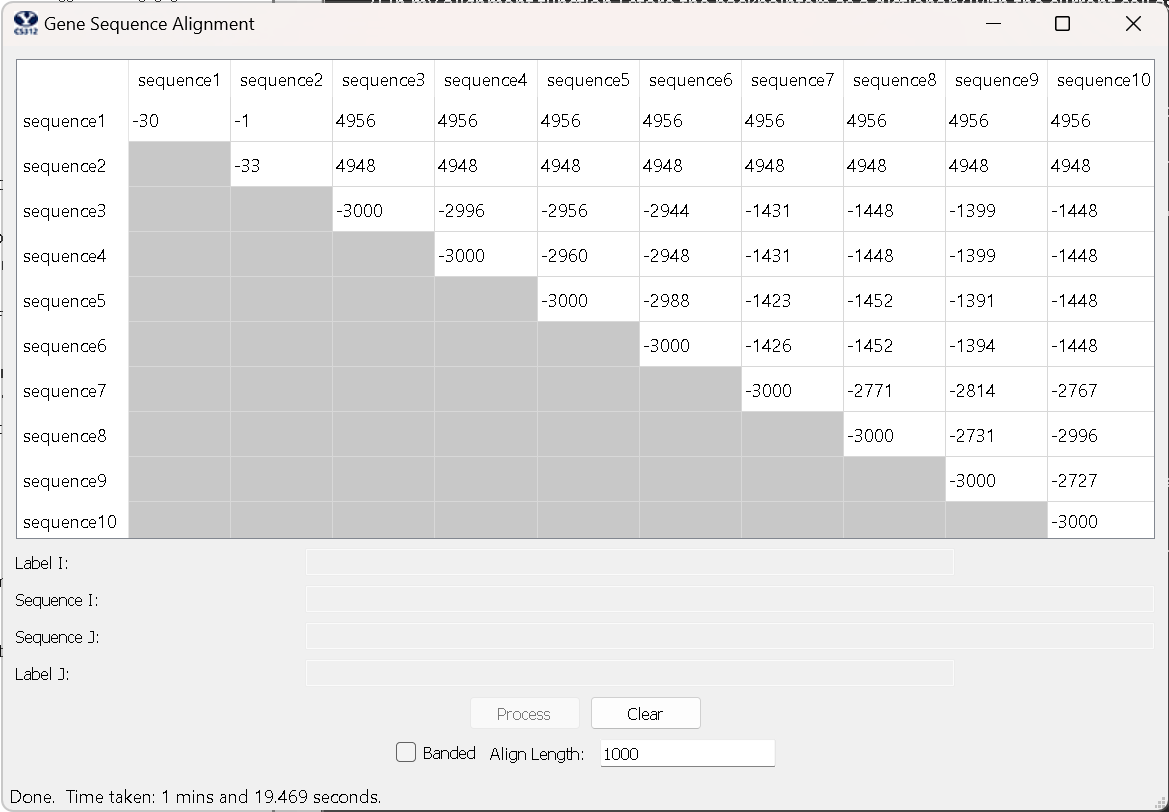
*self*.matrix.setdefault((*self*.min\_length1-1,*self*.min\_length2-1), float(inf))

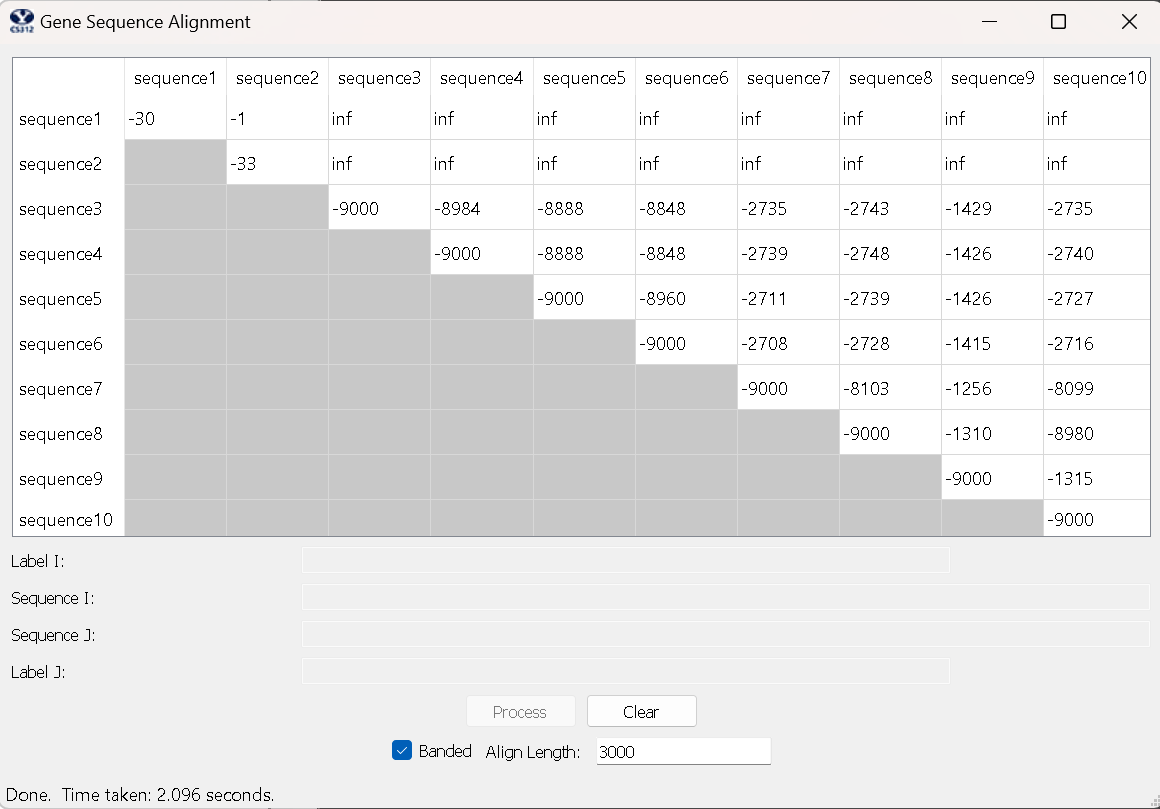
*self*.score = *self*.matrix[*self*.min\_length1-1,*self*.min\_length2-1]

banded\_alignment function has the majority of time and space complexity of the algorithm when the user asks for banded alignment. The function has for loop but the number of iterations is the banded length times the length of the first sequence. The algorithm is very similar to the unrestricted with the only exception of the no of iteration in the inner loop which is banded length. The matrix dictionary has the length of the number of executions of the inner code. Therefore this algorithm has a time and space complexity of O(kn), where k is the banded length and n is the length of the seq1 and with maximum of align length.

2) In my alignment function I store the backpointers as a dictionary with the current cell as a tuple of I,j and with the value of 3-tuple of i,j of previous cell, and the direction the cell points to the current cell. For example, key = (2,2) and value = (1,1,’diagonal’). I use my alignment function to extract my alignments. I start with the last cell and go back and reach the final cell while checking if the tuple value is left or top. If I get left, I would add ‘-‘ to the first sequence and if I get top I would add ‘-‘ to the second sequence. This is how my extraction algorithm works.

3) Results:





4)

Unrestricted:

gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-

-aataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtg

Banded:

gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-

-aataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtg

5) Source Code:

class GeneSequencing:

    def \_\_init\_\_( self ):

        pass

*# Generates alignment using the back\_pointers dictionary generated while executing either banded or unrestricted.*

    def alignment(self, seq1,seq2):

*self*.alignment1 = []

*self*.alignment2 = []

        current = *self*.back\_pointers[*self*.min\_length1-1, *self*.min\_length2-1]

        i = *self*.min\_length1-1

        j = *self*.min\_length2 -1

        while(current[0] > -1 or current[1] > -1):

            if(current[2] == 'top'):

*self*.alignment2.insert(0,'-')

*self*.alignment1.insert(0,seq1[i])

            if(current[2] == 'left'):

*self*.alignment1.insert(0, '-')

*self*.alignment2.insert(0, seq2[j])

            if(current[2] == 'diagonal'):

*self*.alignment1.insert(0,seq1[i])

*self*.alignment2.insert(0,seq2[j])

            i = current[0]

            j = current[1]

            current = *self*.back\_pointers[current[0], current[1]]

*self*.alignment1.insert(0,seq1[i])

*self*.alignment2.insert(0,seq2[j])

        if(current[2] == 'top'):

*self*.alignment2.insert(0,'-')

        if(current[1] == 'left'):

*self*.alignment1.insert(0,'-')

        return

*#Finds the min of the three values and return from which direction the cell would go to current cell.*

*# Used in banded or unrestricted*

    def getMin(self, a, b, c):

        minimum = min(a,b,c)

        if(minimum == a):

            return 'left'

        if(minimum == b):

            return 'top'

        if(minimum == c):

            return 'diagonal'

*# Algorithm for implementing the unrestricted alignment.*

    def unrestricted(self, seq1, seq2):

*self*.score = 0

*self*.matrix[(-1,-1)] = 0

        if(*self*.MaxCharactersToAlign > len(seq1)):

*self*.min\_length1 = len(seq1)

        else:

*self*.min\_length1 = *self*.MaxCharactersToAlign

        if(*self*.MaxCharactersToAlign > len(seq2)):

*self*.min\_length2 = len(seq2)

        else:

*self*.min\_length2 = *self*.MaxCharactersToAlign

        for i in range(-1,*self*.min\_length1): *#Time complexity of O(m) where m is size of seq1*

            for j in range(-1, *self*.min\_length2): *#Time complexity of O(n) where n is size of seq2.*

                if(i==-1 and j==-1):

                    continue

                if(i == -1 and j >-1) :

*self*.matrix[-1,j] = *self*.matrix[-1,j-1] + INDEL

*self*.back\_pointers[-1,j] = (-1,j-1,'left')

                    continue

                if(i > -1 and j == -1):

*self*.matrix[i,-1] = *self*.matrix[i-1, -1] + INDEL

*self*.back\_pointers[i,-1] = (i-1, -1, 'top')

                    continue

                if(seq1[i] == seq2[j]):

                    diagonal = *self*.matrix[i-1,j-1] + MATCH

                    direction = *self*.getMin(*self*.matrix[i,j-1]+INDEL, *self*.matrix[i-1,j]+INDEL,diagonal)

                else:

                    diagonal = *self*.matrix[i-1,j-1] + SUB

                    direction = *self*.getMin(*self*.matrix[i,j-1]+INDEL, *self*.matrix[i-1,j]+INDEL,diagonal)

                if(direction == 'left'):

*self*.matrix[i,j] = *self*.matrix[i,j-1] + INDEL

*self*.back\_pointers[i,j] = (i,j-1,'left')

                    continue

                if(direction == 'top'):

*self*.matrix[i,j] = *self*.matrix[i-1,j] + INDEL

*self*.back\_pointers[i,j] = (i-1,j, 'top')

                    continue

                if(direction == 'diagonal'):

*self*.matrix[i,j] = diagonal

*self*.back\_pointers[i,j] = (i-1,j-1,'diagonal')

                    continue

*self*.score = *self*.matrix[*self*.min\_length1-1,*self*.min\_length2-1]

*# Algorithm for implementing the banded alignment.*

    def banded\_alignment(self, seq1, seq2):

*self*.score = 0

*self*.matrix[(-1,-1)] = 0

        if(*self*.MaxCharactersToAlign > len(seq1)):

*self*.min\_length1 = len(seq1)

        else:

*self*.min\_length1 = *self*.MaxCharactersToAlign

        if(*self*.MaxCharactersToAlign > len(seq2)):

*self*.min\_length2 = len(seq2)

        else:

*self*.min\_length2 = *self*.MaxCharactersToAlign

        for i in range(-1,*self*.min\_length1): *#Time Complexity of O(n).*

            if(-1 >= i - MAXINDELS):

                j\_start = -1

            else:

                j\_start = i - MAXINDELS

            if(*self*.min\_length2 <= i + MAXINDELS):

                j\_end = *self*.min\_length2

            else:

                j\_end = i + MAXINDELS + 1

            for j in range(j\_start, j\_end): *#Time Complexity of O(K) which is 7 in this case.*

                if(i==-1 and j==-1):

                    continue

                if(i == -1 and j >-1) :

*self*.matrix[-1,j] = *self*.matrix[-1,j-1] + INDEL

*self*.back\_pointers[-1,j] = (-1,j-1,'left')

                    continue

                if(i > -1 and j == -1):

*self*.matrix[i,-1] = *self*.matrix[i-1, -1] + INDEL

*self*.back\_pointers[i,-1] = (i-1, -1, 'top')

                    continue

*self*.matrix.setdefault((i,j-1), float(inf))

*self*.matrix.setdefault((i-1,j),float(inf))

                if(seq1[i] == seq2[j]):

                    diagonal = *self*.matrix[i-1,j-1] + MATCH

                    direction = *self*.getMin(*self*.matrix[i,j-1]+INDEL, *self*.matrix[i-1,j]+INDEL,diagonal)

                else:

                    diagonal = *self*.matrix[i-1,j-1] + SUB

                    direction = *self*.getMin(*self*.matrix[i,j-1]+INDEL, *self*.matrix[i-1,j]+INDEL,diagonal)

                if(direction == 'left'):

*self*.matrix[i,j] = *self*.matrix[i,j-1] + INDEL

*self*.back\_pointers[i,j] = (i,j-1,'left')

                    continue

                if(direction == 'top'):

*self*.matrix[i,j] = *self*.matrix[i-1,j] + INDEL

*self*.back\_pointers[i,j] = (i-1,j, 'top')

                    continue

                if(direction == 'diagonal'):

*self*.matrix[i,j] = diagonal

*self*.back\_pointers[i,j] = (i-1,j-1,'diagonal')

                    continue

*self*.matrix.setdefault((*self*.min\_length1-1,*self*.min\_length2-1), float(inf))

*self*.score = *self*.matrix[*self*.min\_length1-1,*self*.min\_length2-1]

*# This is the method called by the GUI.  \_seq1\_ and \_seq2\_ are two sequences to be aligned, \_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, seq1, seq2, banded, align\_length):

        if(seq1 == 'polynomial' and seq2 == 'polynomial'):

*self*.no\_of\_calls = 0

*self*.no\_of\_calls += 1

*self*.banded = banded

*self*.MaxCharactersToAlign = align\_length

*self*.back\_pointers = {}

*self*.matrix = {}

        if(*self*.banded):

*self*.banded\_alignment(seq1, seq2)

        else:

*self*.unrestricted(seq1, seq2)

        if(*self*.score == float(inf)):

            return {'align\_cost':*self*.score, 'seqi\_first100': 'No Alignment Possible', 'seqj\_first100':'No Alignment Possible'}

*self*.alignment(seq1,seq2)

*# gets the first 100 characters of the alignment string.*

        seqi100 = ''.join(*self*.alignment1[:100])

        seqj100 = ''.join(*self*.alignment2[:100])

*# Prints out the first 100 strings of each alignment as the gui was cutting off the values.*

        print(str(*self*.no\_of\_calls) +': ' + seqi100)

        print(str(*self*.no\_of\_calls) + ': ' + seqj100)

        print('')

        return {'align\_cost':*self*.score, 'seqi\_first100':seqi100, 'seqj\_first100':seqj100}