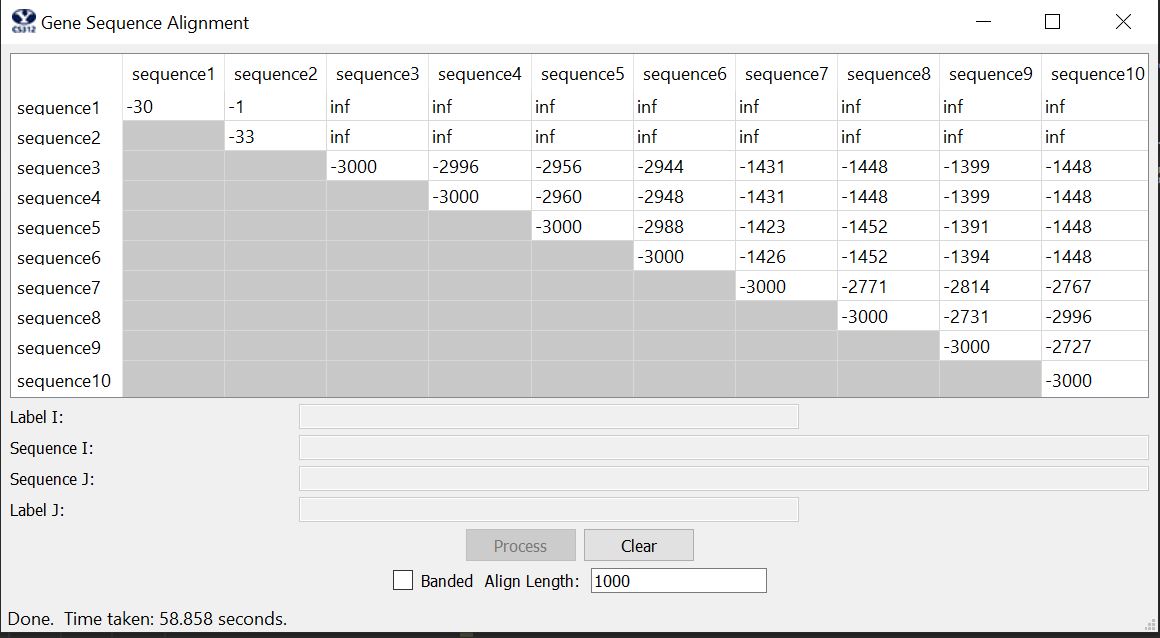
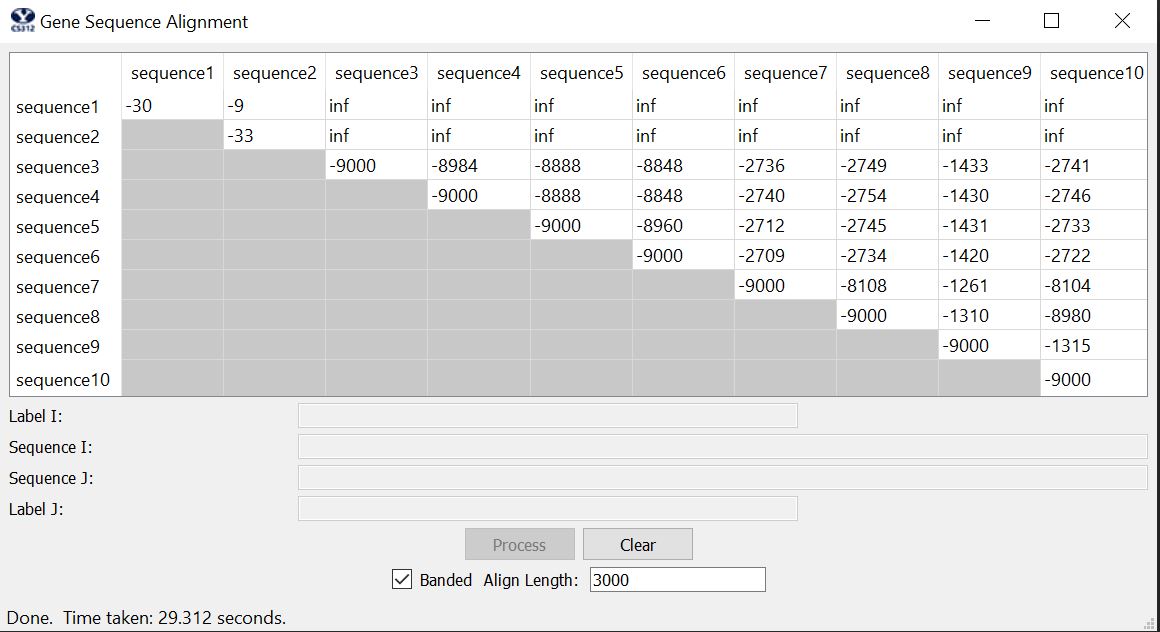
class GeneSequencing:  
  
 def \_\_init\_\_( self ):  
 pass  
  
# This is the method called by the GUI. \_sequences\_ is a list of the ten sequences, \_table\_ is a  
# handle to the GUI so it can be updated as you find results, \_banded\_ is a boolean that tells  
# you whether you should compute 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)):  
 jresults = []  
 for j in range(len(sequences)):  
 if j < i:  
 s = {}  
 else:  
###################################################################################################  
# your code should replace these three statements and populate the three variables: score, alignment1 and alignment2  
  
 # Comparing itself  
 if i == j:  
 score = max(-3\*align\_length, -3\*len(sequences[i]))  
 alignment1 = 'self'  
 alignment2 = 'self'  
 # Comparing with artificial sequences  
 elif (i == 0 and j != 1) or (i == 1 and j != 0):  
 score = float('inf')  
 alignment1 = 'No alignment possible'  
 alignment2 = 'No alignment possible'  
 else:  
 sequence\_i\_length = len(sequences[i])  
 sequence\_j\_length = len(sequences[j])  
 # Initialize the arrays  
 if align\_length > sequence\_i\_length:  
 if align\_length > sequence\_j\_length:  
 matrix\_distance = [[0 for column in range(sequence\_i\_length + 1)] for row in range(sequence\_j\_length + 1)]  
 matrix\_path = [['' for column in range(sequence\_i\_length + 1)] for row in range(sequence\_j\_length + 1)]  
 else:  
 matrix\_distance = [[0 for column in range(align\_length + 1)] for row in range(align\_length + 1)]  
 matrix\_path = [['' for column in range(align\_length + 1)] for row in range(align\_length + 1)]  
  
 # Filling out the first row  
 for k in range(len(matrix\_distance[0])):  
 if banded:  
 if k > 3:  
 matrix\_distance[0][k] = float('inf')  
 else:  
 matrix\_distance[0][k] = k \* INDEL  
 matrix\_path[0][k] = 'r'  
  
 # Filling out the first column  
 for k in range(len(matrix\_distance)):  
 if banded:  
 if k > 3:  
 matrix\_distance[k][0] = float('inf')  
 else:  
 matrix\_distance[k][0] = k\*5  
 matrix\_path[k][0] = 't'  
 matrix\_path[0][0] = ''  
  
 for count in range(1, len(matrix\_distance)):  
 if banded:  
 if count - 3 > 0:  
 start = count - 3  
 else:  
 start = 0  
 if count + 4 < len(matrix\_distance[0]):  
 finish = count + 4  
 else:  
 finish = len(matrix\_distance[0])  
 else:  
 start = 1  
 finish = len(matrix\_distance[0])  
  
 # loop through and calcuate each value and find the minimum cost, filling out the matrix  
 # path matrix keeps track of which direction it came from  
 # Would take at most O(mn) time and space  
 for index in range(start, finish):  
 diagonal = matrix\_distance[count-1][index-1] + self.get\_difference(sequences[i], sequences[j], count-1, index-1)  
 right = matrix\_distance[count-1][index] + INDEL  
 top = matrix\_distance[count][index-1] + INDEL  
  
 min\_val = min(right, top, diagonal)  
  
 matrix\_distance[count][index] = min\_val  
 if min\_val == diagonal:  
 matrix\_path[count][index] = 'd'  
 elif min\_val == top:  
 matrix\_path[count][index] = 't'  
 elif min\_val == right:  
 matrix\_path[count][index] = 'r'  
  
 # Assign alignments using backtrace  
 # Would take O(mn) time and space  
 alignment1, alignment2 = self.get\_string\_alignment(matrix\_path, sequences[i], sequences[j])  
 score = matrix\_distance[-1][-1]  
 #if i == 2 and j == 9:  
 # print(alignment1)  
 # print(alignment2)  
  
###################################################################################################  
 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  
  
 def get\_string\_alignment(self, matrix\_path, sequence\_i, sequence\_j):  
 path = ''  
 i\_final = ''  
 j\_final = ''  
 j,k = -1,-1  
  
 # Should start at the bottom right corner and backtrace  
 # Find if it was from diagonal, from the above, and from right, and backtrace appropriately  
 while matrix\_path[j][k] != '':  
 path = path + matrix\_path[j][k]  
 if matrix\_path[j][k] == 'd':  
 j = j - 1  
 k = k - 1  
 elif matrix\_path[j][k] == 'r':  
 k = k - 1  
 elif matrix\_path[j][k] == 't':  
 j = j - 1  
   
 path = path[::-1]  
 index\_r, index\_c = 0,0  
 for i in range(len(path)):  
 if path[i] == 't':  
 i\_final += '-'  
 else:  
 i\_final += sequence\_i[index\_r]  
 index\_r = index\_r + 1  
 if path[i] == 'r':  
 j\_final += '-'  
 else:  
 j\_final += sequence\_j[index\_c]  
 index\_c = index\_c + 1  
 return i\_final, j\_final  
  
  
 # return -3 if match, 1 if sub  
 # Would take O(1) time and constant space  
 def get\_difference(self, seqeunce\_i, sequence\_j, count, index):  
 if seqeunce\_i[index] == sequence\_j[count]:  
 return MATCH  
 return SUB

1. The time complexity would be O(m\*n). It’ll take O(mn) time to initialize the matrix, and O(mn) time to backtrace. The Space complexity would be also O(mn) because it’s using two 2-d matrices.

2. For the alignment extraction, the characters are used to know if the minimum cost was from the above, diagonal, or from the right side. The extraction starts from the bottom right corner of the matrix and start backtrack as it reduces the matrix. Once it hits the top left corner, the alignments are made by looping through the path. I put dash when it was INDEL and append it to each alignment 1 and 2.

3. Results





4. 