Project 4 Report

1. Time and Space complexity

Unrestricted Algorithm Pseudocode

```
Width = length of string1 or align length                    decrement i and j
Height = length of string2 or align length                    Time = 1        Space = m*n
Table[width][height]                                          Time = 1        Space = m*n
prevTable[width][height]
Table[0][0] = 0                                               Time = n
Table[i][0] = table[i-1][0] + INDEL for all I                 Time = m
Table[0][j] = table[0][j-1] + INDEL for all j                 N repetitions
For i in width:                                               M repetitions
  For j in height:                                            Constant time
    Diagonal = MATCH if str1[i] == str2[j], else SUB          “”
    Table[i][j] = min(top + INDEL, left + INDEL, diagonal)    “”
    prevTable = ‘u’ if top, ‘l’ if left, or ‘m’ if diagonal
i = width                                                     Getting alignment
j = height                                                    Repeats at worst n + m times
while I or j are greater than 0:                              Constant time
  if prevTable[i][j] = ‘u’:
    alignA = “-“ + alignA
    alignB = B[j] + alignB
    j -= 1
  same but opposite for ‘l’
  if prevTable[i][j] = ‘m’:
    alignA and alignB plus the letter at i and j
```

Total time complexity is N*M for filling out the table. While loop for extracting the alignment is at worst case N + M time so is dominated by filling out the table. Total time complexity isO(nm)

Storing the strings and alignments are not significant as they are just n or m in length. Space Complexity is dominated by the two tables for the score values and the previous pointers which are both N*M tables so total space complexity is O(nm).

Banded Algorithm Pseudocode

```
If absolute value( length of str1 – length str2 ) > 100: skip and return nothing    Time = 1
Width = 1 + MAXINDEL*2                                                              width = 3*2 + 1 = 7 (this is k)
Height = length of string2
Table[width][height]                                                                Space = kn
prevTable[width][height]                                                            Space = kn
Table[0][j] = table[0][j-1] + INDEL for all j, prevTable = ‘u’
for j in width:                                                                     k repeats
  for i in height:                                                                  n repeats
    if I > MAXINDEL:                                                                Calculate table value (constant time)
      upper = table[i-1][j+1] + INDEL
      left = table[i][j-1] + INDEL
```

```
        diagonal = table[i-1][j] + MATCH if same or SUB if different
        table[i][j] = min(upper, left, diagonal)
        set prevPointer
    else I <= MAXINDEL:
        do same as unrestricted for first 3 rows
i = length of string1
j = length of string2
while I or j are greater than 0:                                    worst case n + m time and space
  adjust = j – i + MAXINDEL
  same as for restricted but use
      adjust instead of j in accessing
      the tables.
```

Total time complexity is N*k for filling out the table. While loop for extracting the alignment is at worst case N + M time so is dominated by filling out the table. Total time complexity isO(nk)

Storing the strings and alignments are not significant as they are just n or m in length. Space Complexity is dominated by the two tables for the score values and the previous pointers which are both N*k tables so total space complexity is O(nk).

2. Alignment Extraction Algorithm

I implemented the back trace algorithm using a previous pointer table that saved for each index, a letter signifying which direction that number was calculated from. For example, ‘l’ meant that the number at that index was based on the number to the left plus the value of an INDEL. ‘m’ signified a match or substitution.

Once the table was filled, I extracted the alignment strings by starting at the final goal index and checking the previous pointer. A previous pointer of ‘u’ for upper or ‘l’ for left meant and INDEL was used to reach that value so I added the character from one string to it’s alignment and a ‘-‘ to the other string. I then decremented the i (or j) value to represent the upper or left cell. A previous pointer of ‘m’ signified a match or substitution so I added both strings characters at that index to both alingments and decremented both i and j.

The alignment strings were constructed thus in reverse direction from the left to the right. Each character was appended to the beginning of the current alignment string until the index [0, 0] was reached. Since this was the start index, nothing is added and both alignments are returned.

3. Results

Unrestricted with align length = 1000

Restricted with Align length = 3000 and MAXINDEL = 3

Unrestricted Alignment for Seq #3 and Seq #10

gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-

-ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgt

Restricted Alignment for Seq #3 and Seq #10

ga-ttgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta

--ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtg

Appendix: Commented Source Code

```
#!/usr/bin/python3

from PyQt6.QtCore import QLineF, QPointF


# 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.banded = None

    # 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):
        self.banded = banded
        self.MaxCharactersToAlign = align_length

        seq1 = "-" + seq1 # Add a blank symbol to the beginning of each string so that
beginning is always a match
        seq2 = "-" + seq2

        AlignmentA = ""
        AlignmentB = ""

        ## Unrestricted
        if not banded:
            width = min(align_length + 1, len(seq1))
            height = min(align_length + 1, len(seq2))
            table = [[float('inf') for i in range(height)] for j in range(width)] #
Initialize tables
            prevTable = [[0 for i in range(height)] for j in range(width)]


            table[0][0] = 0 # Set start point values
            prevTable[0][0] = 'stop'
            for i in range(1, width): # Set base cases
                table[i][0] = table[i - 1][0] + INDEL
```

```
                prevTable[i][0] = 'l'
            for j in range(1, height):
                table[0][j] = table[0][j - 1] + INDEL
                prevTable[0][j] = 'u'

            for i in range(1, width): # Fill in entire table from top left
                for j in range(1, height):
                    left = table[i - 1][j] + INDEL
                    top = table[i][j - 1] + INDEL
                    if seq1[i] == seq2[j]:  # If the letters match then set value to
MATCH, else SUB
                        diag = table[i - 1][j - 1] + MATCH
                    else:
                        diag = table[i - 1][j - 1] + SUB
                    minimum = min(left, top, diag)
                    table[i][j] = minimum
                    if minimum == left:  # Determine which input was used and store in
previous pointer
                        prevTable[i][j] = 'l'
                    elif minimum == top:
                        prevTable[i][j] = 'u'
                    elif minimum == diag:
                        prevTable[i][j] = 'm'

            score = table[-1][-1]

            # Get Alignment not Banded
            A = seq1
            B = seq2
            i = min(len(A) - 1, align_length) # Set the start point
            j = min(len(B) - 1, align_length) # Set the start point
            while i > 0 or j > 0: # While i and j are both above 0 meaning we have not
reached the beginning
                if j > 0 and prevTable[i][j] == 'u': # If previous was u then continue
trace in upper
                    AlignmentA = "-" + AlignmentA
                    AlignmentB = B[j] + AlignmentB
                    j -= 1
                elif i > 0 and prevTable[i][j] == 'l': # If previous was l then continue
in left
                    AlignmentA = A[i] + AlignmentA
                    AlignmentB = "-" + AlignmentB
                    i -= 1
                elif i > 0 and j > 0 and prevTable[i][j] == 'm': # If previous was m then
continue on diagonal
                    AlignmentA = A[i] + AlignmentA
                    AlignmentB = B[j] + AlignmentB
                    i -= 1
                    j -= 1
                else:
                    print("Backtrace Failed") # If none matched then the trace was lost
(should never happen)
                    exit(-111)

        # Restricted
        if banded:
            width = min(align_length + 1, len(seq1))
            height = min(align_length + 1, len(seq2))

            if abs(width - height) > 100:  # Significant difference in length, return not
calculable
                print("No alignment possible as sequence differ in length significantly")
                score = float('inf')
```

```
                alignment1 = "No Alignment Possible"
                alignment2 = "No Alignment Possible"
                return {'align_cost': score, 'seqi_first100': alignment1,
'seqj_first100': alignment2}

            width = 1 + MAXINDELS * 2 # Initialize tables
            table = [[float('inf') for i in range(7)] for j in range(height)]
            prevTable = [[0 for i in range(7)] for j in range(height)]

            table[0][0] = 0 # Set start values
            prevTable[0][0] = 'stop'
            for i in range(1, 4): # Set base cases
                table[0][i] = table[0][i - 1] + INDEL
                prevTable[0][i] = 'u'
            for i in range(1, height): # Fill table from top left
                for j in range(width):
                    if i > MAXINDELS: # Use these comparisons once values are being
shifted in table
                        if j + 1 > width - 1:  # Get north value
                            north = float('inf')
                        else:
                            north = table[i - 1][j + 1] + INDEL
                        if j - 1 < 0:  # Get west value
                            west = float('inf')
                        else:
                            west = table[i][j - 1] + INDEL
                        adjust = min(MAXINDELS - i, 0)
                        if j - adjust < len(seq1) and seq1[j - adjust] == seq2[i]:  #
Check for match
                            nw = table[i - 1][j] + MATCH
                        else:
                            nw = table[i - 1][j] + SUB
                        minimum = min(north, west, nw)
                        table[i][j] = minimum

                        if minimum == north: # Determine which value was previous
                            prevTable[i][j] = 'u'
                        elif minimum == west:
                            prevTable[i][j] = 'l'
                        else:
                            prevTable[i][j] = 'm'

                    else:  # For the first 3 rows that don't need adjustment, run like
unbanded
                        north = table[i - 1][j] + INDEL
                        west = table[i][j - 1] + INDEL
                        if seq1[i] == seq2[j]:
                            diag = table[i - 1][j - 1] + MATCH
                        else:
                            diag = table[i - 1][j - 1] + SUB
                        minimum = min(north, west, diag)
                        table[i][j] = minimum
                        if minimum == north:
                            prevTable[i][j] = 'u'
                        elif minimum == west:
                            prevTable[i][j] = 'l'
                        else:
                            prevTable[i][j] = 'm'

            # Get Alignment Banded
            A = seq2
            B = seq1
            i = min(len(A) - 1, align_length)
```

```
            j = min(len(B) - 1, align_length)
            while i > 0 or j > 0:
                if i > MAXINDELS: # For when table is shifted, use these comparisons
                    adjust = j - i + MAXINDELS
                    if i > 0 and j > 0 and prevTable[i][adjust] == 'l': # Check left
                        AlignmentA = "-" + AlignmentA
                        AlignmentB = B[j] + AlignmentB
                        j -= 1
                    elif i > 0 and prevTable[i][adjust] == 'u':  # Check top
                        AlignmentA = A[i] + AlignmentA
                        AlignmentB = "-" + AlignmentB
                        i -= 1
                    elif j > 0 and prevTable[i][adjust] == 'm':  # Check diag
                        AlignmentA = A[i] + AlignmentA
                        AlignmentB = B[j] + AlignmentB
                        i -= 1
                        j -= 1
                    else:
                        print("Backtrace Failed")
                        exit(-111)
                else: # When table is not shifted (first 3 rows)
                    if i > 0 and prevTable[i][j] == 'l':
                        AlignmentA = A[i] + AlignmentA
                        AlignmentB = "-" + AlignmentB
                        i -= 1
                    elif j > 0 and prevTable[i][j] == 'u':
                        AlignmentA = "-" + AlignmentA
                        AlignmentB = B[j] + AlignmentB
                        j -= 1
                    elif i > 0 and j > 0 and prevTable[i][j] == 'm':
                        AlignmentA = A[i] + AlignmentA
                        AlignmentB = B[j] + AlignmentB
                        i -= 1
                        j -= 1
                    else:
                        print("Backtrace Failed")
                        exit(-111)


            # This is easier than changing the above code to switch the strings
            sub = AlignmentA
            AlignmentA = AlignmentB
            AlignmentB = sub

            i = min(len(A) - 1, align_length) # Get the length of str1
            j = min(len(B) - 1, align_length) # Get length of str2
            score = table[i][-abs(j-i)+MAXINDELS] # Get location of final score (Will
depend on offset between i and j)

        alignment1 = AlignmentA[:100]
        alignment2 = AlignmentB[:100]

##########################################################################################
##########

        return {'align_cost': score, 'seqi_first100': alignment1, 'seqj_first100':
alignment2}
```