# Noah Schill -- Lab 4 Submission

## Complexity analysis

### Unrestricted algorithm

My implementation (inspired by Chester) consists of two parts, the table compilation, and the path-finding matrix which traces itself back to produce a string.

First, we create two matrices, one with edit distances, and another with pointers informing where the previous cell was. The initialization process consists of two for loops to initialize the top row and first column of the edit distance matrix, one taking $n$ time and the other taking $m$ time.

The next section is somewhat concerning , as there are two nested for loops, but this runs at $O(mn)$ time for two sequences of $n, m$ length. These nested loops will populate both the edit distance matrix and the pointer matrix. Each cell is calculated by three of it's neighbors, which takes constant time. Running this for every cell in an $m \times n$ table will yield $O(mn)$.

The path-finding section in a worst-case scenario run at $O(m + n)$ and at best $\min(O(m), O(n))$ as it has to trace it's way from the optimal point along the pointer matrix back to the origin. These are all dominated by the multiplicative $O(mn)$ and therefore, the entire algorithm has this complexity.

### Banded algorithm

I was unfortunately not able to implement the banded algorithm, but I understand conceptually how it works. In the unrestricted algorithm, we are filling the matrix entirely without respect to if any of those entires are along the best path or not from the goal to the origin. When we use a bounded algorithm, we are setting a restriction $k$ dictating that we cannot continue in a single direction (if it is not diagonal). For example, if $k = 3$, then if my cell equation yielded an "up" three times, then if it yielded up again, then the banded algorithm would force the matrix development to continue down the diagonal. It prevents the meandering nature of the unrestricted algorithm. If we have a lower $k$, that creates a narrower bandwidth for our divide-and-conquer to go explore. It is an area of cells being filled with some $n$-length from origin to goal, and with a "width" of k. Resulting in $kn$ cells being solved each in constant time, $O(kn)$.

## Extraction

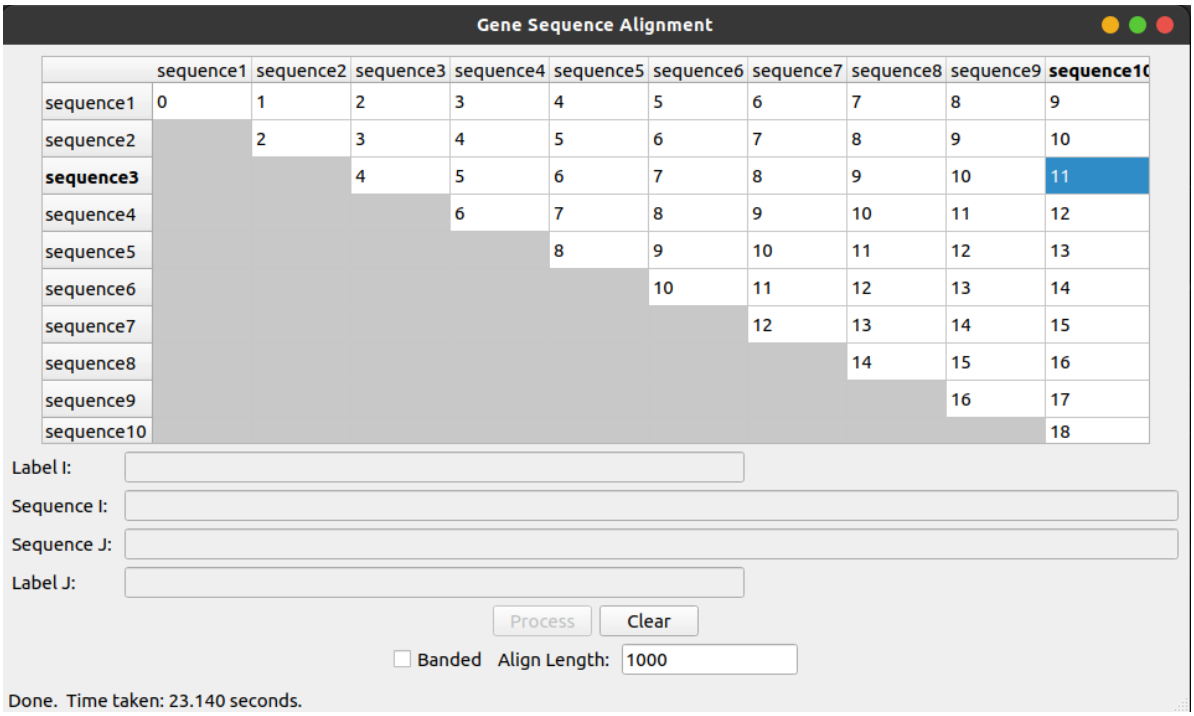
The extraction process is very simple. When creating our matrix of weights to determine the best path, we simultaneously create a partner matrix containing all the pointers to where their previous one came from. Each cell in the matrix maps to a specific section of each sequence and can be represented as a potential edit. For example, if a cell's previous pointer is diagonal, then we can assume that these sequences in this specific iteration both contain the same letter at that index. If the cell was to the left, or above, then it means there was an insertion at one sequence, and a deletion at another, yielding the '-' in the sequence.

## Results

```
ataagagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatctaatctaaactttataaacg
gcacttcctgtgtgtccatgcccgtgggcttggtcttgtcatagtgctgacatttgtggttccttggtttttgttctctgc
cagtgacgtgtccattcggcgccagcagcccacccataggttgcataatggcaaagatgggcaaatacggtctcggcttca
aatgggccccagaatttccatggatgcttccgaacgcatcggagaagttgggtagccctgagaggtcagaggaggatgggt
tttgccccctctgctgcgcaagaaccaaaaactaaaggaaaaactttgattaatcacgtgagggtggattgtagccggcttc
cagcattggagtgctgtgttcagtctgccataatccgtgatatttttgttgatgaggatcccttgaatgtggaggcctcaa
ctatgatggcattgcagttcggtagtgctgtcttggtcaagccatccaagcgcttgtctattcaggcatgggctaagttgg
gtgtgctgcctaaaactccagccatggggttgttcaagcgcttctgcctgtgtaacaccagggagtgcgtttgtgacgccc
acgtggcttttcaacttttacggtccagcctgatggtgtatgcctgggcaatggccgttttataggctggtttgtgccag
tcacagccataccggcgtatgcgaagcagtggttgcaaccctggtccatccttcttcgtaagggtggtaacaaaggttctg
taacatccggccatttccgccgcgctgttaccatgcctgtgtatgactttaatgtggaggatgcttgtgaggaggttcatc
ttaacccgaagggtaagtactcccgcaaggcgtatgctcttcttaagggctatcgcggtgttaaatccatcctattcttgg
accagtatggttgtgactatactgggcg
```

```
ataagagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatctaatctaaactttataaacg
gcacttcctgtgtgtccatgcccgtgggcttggtcttgtcatagtgctgacatttgtggttccttggtttttgttctctgc
cagtgacgtgtccattcggcgccagcagcccacccataggttgcataatggcaaagatgggcaaatacggtctcggcttca
aatgggccccagaatttccatggatgcttccgaacgcatcggagaagttgggtagccctgagaggtcagaggaggatgggt
tttgccccctctgctgcgcaagaaccaaaaactaaaggaaaaactttgattaatcacgtgagggtggattgtagccggcttc
cagcattggagtgctgtgttcagtctgccataatccgtgatatttttgttgatgaggatcccttgaatgtggaggcctcaa
ctatgatggcattgcagttcggtagtgctgtcttggtcaagccatccaagcgcttgtctattcaggcatgggctaagttgg
gtgtgctgcctaaaactccagccatggggttgttcaagcgcttctgcctgtgtaacaccagggagtgcgtttgtgacgccc
acgtggcttttcaacttttacggtccagcctgatggtgtatgcctgggcaatggccgttttataggctggtttgtgccag
tcacagccataccggcgtatgcgaagcagtggttgcaaccctggtccatccttcttcgtaagggtggtaacaaaggttctg
taacatccggccatttccgccgcgctgttaccatgcctgtgtatgactttaatgtggaggatgcttgtgaggaggttcatc
ttaacccgaagggtaagtactcccgcaaggcgtatgctcttcttaagggctatcgcggtgttaaatccatcctattcttgg
accagtatggttgtgactatactgggcg
```

I ran these through a diff tool, and they are identical. There's a bug somewhere. I struggled with passing data around between the GUI and my code.




Appendix:

```
import math
import time
```

```
import numpy as np

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

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

TOP = 0
DIAG = 1
LEFT = 2


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

                    if not banded:
                        not_banded_result = self.get_not_banded(sequences[i]
[:align_length],
                                                                sequences[j]
[:align_length])
                        score = not_banded_result[0]
                        alignment1 = not_banded_result[1]
                        alignment2 = not_banded_result[2]
                    else:
                        banded_result = self.get_banded(sequences[i]
[:align_length], sequences[j][:align_length])

                    score = i + j
                    alignment1 = 'abc-easy  DEBUG:(seq{}, {} chars,align_len={}
{})'.format(i + 1,
```

```
          len(sequences[i]),

          align_length,

          ',BANDED' if banded else '')
                    alignment2 = 'as-123--  DEBUG:(seq{}, {} chars,align_len={}
{})'.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

    def get_not_banded(self, sequence1, sequence2):

        # matrix = np.ones((len(sequence1) + 1, len(sequence2) + 1)) * np.inf
        # back_pointer_matrix = np.ones((len(sequence1) + 1, len(sequence2) + 1))
* np.inf

        matrix = [[math.inf for x in range(len(sequence2)+1)]for y in
range(len(sequence1)+1)]
        back_pointer_matrix = [[math.inf for x in range(len(sequence2)+1)]for y
in range(len(sequence1)+1)]


        # initialize matrix stuff
        for i in range(len(sequence1) + 1):
            matrix[i][0] = i
            back_pointer_matrix[i][0] = TOP

        for j in range(len(sequence2) + 1):
            matrix[0][j] = j
            back_pointer_matrix[0][j] = LEFT

        back_pointer_matrix[0][0] = -1

        for i in range(1, len(sequence1)+1):
            for j in range(1, len(sequence2)+1):
                #  find out what the top is, what diag is, and what left is
                #  compare them all to get the shortest while accounting for
INDEL, sub and match
                #  get a value that goes into the
                #  update back matrices accordingly -> back_pointer_matrix[i][j]
                if i == 1 and j ==1 :
                    matrix[i][j] = 1
```

```
                    back_pointer_matrix[i][j] = DIAG
                else:
                    if sequence1[i-1] == sequence2[j-1]:
                        diff = 0
                    else:
                        diff = 1

                    corners = [matrix[i-1][j] + 1, matrix[i - 1][j - 1]+diff, 1 +
matrix[i][j-1]]##top, diag, left
                    E = min(corners)

                    matrix[i][j] = E
                    back_pointer_matrix[i][j] = corners.index(E)  #either TOP,
DIAG, LEFT

        not_banded_path = self.get_path(back_pointer_matrix, sequence1,
sequence2)

        return matrix[1][1], not_banded_path[0], not_banded_path[1]

    def get_path(self, back_pointer_matrix, sequence1, sequence2):

        alignment1 = ""
        alignment2 = ""
        i, j = len(sequence1), len(sequence2)  # our back pointer matrix can only
go backwards
        while i != 0 or j != 0:
            if back_pointer_matrix[i][j] == TOP:
                # do the zero stuff, it came from top
                sequence1_char = sequence1[i-1]
                alignment1 = sequence1_char + alignment1  # add it to the front
of alignment one
                alignment2 = "-" + alignment2  # add it to the front of alignment
2
                i -= 1

            elif back_pointer_matrix[i][j] == DIAG:
                # came from diagonal
                sequence1_char = sequence1[i - 1]
                sequence2_char = sequence2[j - 1]
                alignment1 = sequence1_char + alignment1  # add it to the front
of alignment 1
                alignment2 = sequence2_char + alignment2  # add it to the front
of alignment 2
                i -= 1
                j -= 1

            elif back_pointer_matrix[i][j] == LEFT:
                sequence2_char = sequence2[j-1]
                alignment1 = "-" + alignment1  # add it to the front of alignment
1
                alignment2 = sequence2_char + alignment2  # add it to the front
of alignment 2
                j -= 1
            else:
                raise ValueError("Invalid pointer {} in array at ({},
{})".format(back_pointer_matrix[i][j], i, j))
```

```
        return [alignment1, alignment2]
```