Analysis – Unrestricted Algorithm:

String size n, string size m

The unrestricted algorithm starts by initializing an array of size n+1 * m+1, (which is O(n+1 * m+1), but as n and m get bigger and bigger the +1 becomes so small in comparison that we can simplify to O(nm)) which takes O(nm) time and O(nm) space. This is the only substantial space that we initialize in the algorithm, as all other parts of our algorithm only take O(1) space when making a single variable. Later, we concatenate the sequences as strings, which takes O(m+n) space, which simplifies to either O(n) or O(m) space depending on which input is longer as our ending strings will be length of the longest string. This is small in comparison to O(nm) space of the 2d array, so we can ignore this.

Apart from initializing the 2d array, we then loop through each element, each taking O(1) time to determine the minimum value from its neighbors, which leaves us with n*m elements each with O(1) time. This leaves us with another O(nm) time chunk, but when added together to get O(2nm), we simplify to O(nm). When concatenating the strings at the end, our time complexity is O(m+n), which simplifies to either O(n) or O(m) as we traverse all the way to the origin of our 2d matrix of size n*m. By comparison, these times are small when compared to O(nm), so we are left with only O(nm) time complexity and O(nm) space complexity.

See code for more analysis on each function's individual time and complexity

Analysis – Banded Algorithm:

String size n, string size m, size k bandwidth

The unrestricted algorithm starts by initializing an array of size k*n, where n is the longest string which takes O(kn) time and O(kn) space. This is the only substantial space that we initialize in the algorithm, as all other parts of our algorithm only take O(1) space when making a single variable. Later, we concatenate the sequences as strings, which takes O(m+n) space, which simplifies to either O(n) or O(m) space depending on which input is longer as our ending strings will be length of the longest string. This is small in comparison to O(kn) space of the 2d array, so we can ignore this.

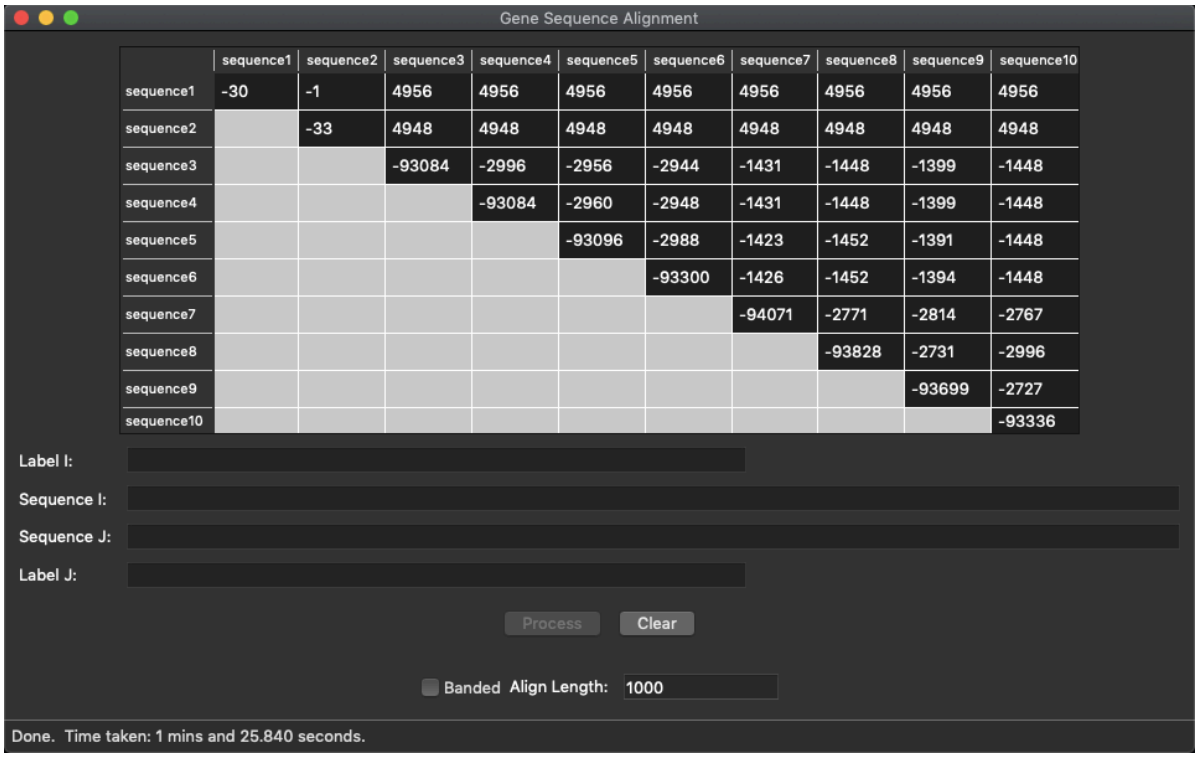
Apart from initializing the 2d array, we then loop through each element, each taking O(1) time to determine the minimum value from its neighbors, which leaves us with k*n elements each with O(1) time. Because we only care about the elements that are within d=3 elements from the diagonal, we only need to iterate through 7 elements on each row of the matrix, so even though we are comparing each element of n and m, we only do this when m is +- 3 from the diagonal, which leaves us iterating through k*n elements. This leaves us with another O(kn) time chunk, but when added together to get O(2kn), we simplify to O(kn). When concatenating the strings at the end, our time complexity is O(m+n), which simplifies to either O(n) or O(m) as we are traversing our 2d matrix of size k*n all the way back to the origin point. By comparison, these times are small when compared to O(kn), so we are left with only O(kn) time complexity and O(kn) space complexity.

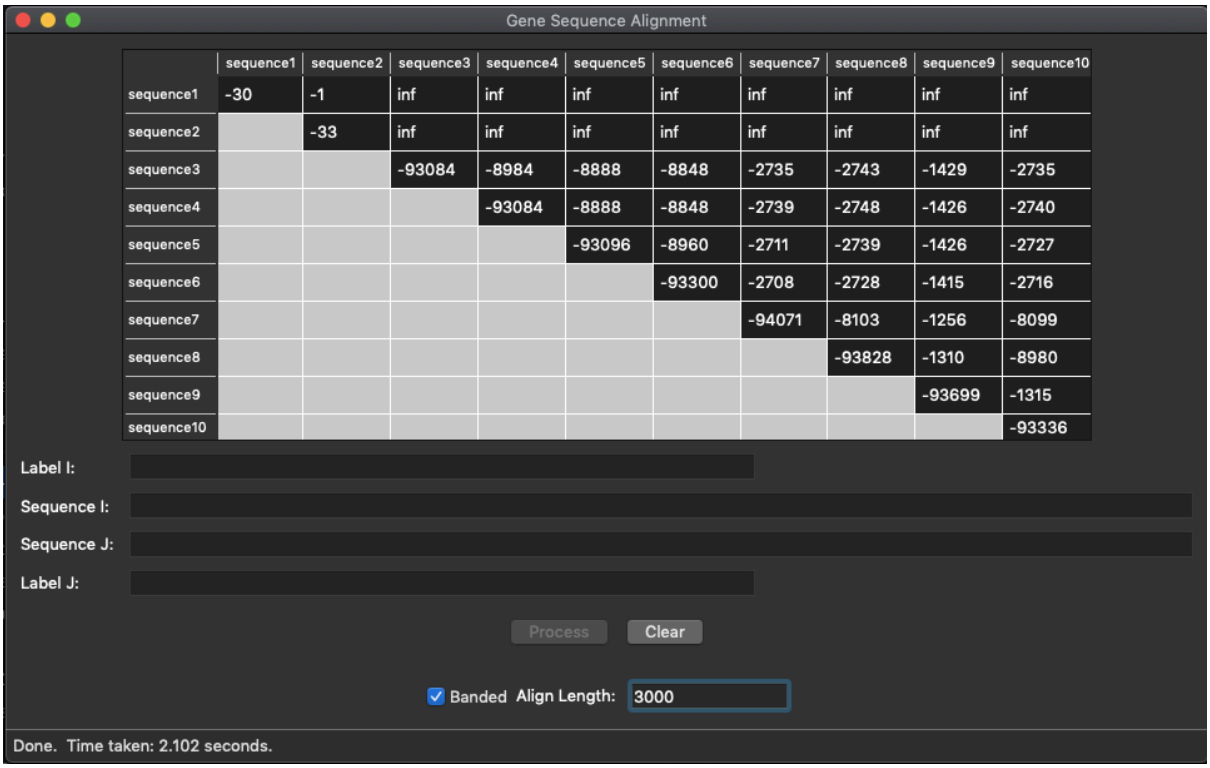
See code for more analysis on each function's individual time and complexity

How it works:

First we make a 2d matrix, size dependent on whether we want it banded or unrestricted (see analysis for difference). Then we loop through each element, and using its neighbors (top, left, and diagonal) we calculate the minimum value of the sequence if we have an indel, a match, or a mismatch. We would then update the current position with the current minimum value from my neighbors, and go to the next element in my sequences. In my matrix, I used a tuple to determine where I was coming from. If I came from a diagonal, my tuple would contain the current value and a value "D" so I knew when I would traverse back from the end of the matrix, I would see the "D" in my tuple and know that the next node I needed to visit was matrix[i-1][j-1]. I would store the same for top ("T") and left ("L"), with the origin point containing a "$" so my algorithm knew when it was done and had reached the upper left hand corner. We would start from the bottom right hand corner of my cost matrix, and follow the back pointers until we reached the start, each time either adding a character from sequence1, sequence2, or both if a diagonal. If we came from the Left, we would add a – to the beginning of sequence1, and a character to the beginning of sequence2, if from the top, we would add a – to the beginning of sequence2 and a character to the beginning of sequence1. If we come from a diagonal, we add a character to both sequences' beginnings.

Results - Screenshots:





Results - Alignment:

K = 1000

```
gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a
-a-taagagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt
```

K = 3000

```
gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a
-a-taagagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt
```

Commented Code:

Unbanded Algorithm:

```python
# Unrestricted Algorithm
# Time: O(nm) time, to fill up the matrix, and to iterate through each element in the 2d matrix
# Space: O(nm) space, as we fill up a 2d matrix and alter those values, but it is the only major object we create
def sequenceAlgorithm(self, sequence1, sequence2, align_length):
    if sequence1 == sequence2:
        return MATCH * len(sequence1), sequence1, sequence2
    else:
        sequence1 = sequence1[:align_length]
        sequence2 = sequence2[:align_length]

        if(len(sequence2) > len(sequence1)):
            temp = sequence2
            sequence2 = sequence1
            sequence1 = temp

        matrix = [[(float("inf"), 'X') for j in range(len(sequence2) + 1)] for i in range(len(sequence1) + 1)] #O(nm) time and space
        matrix[0][0] = (0, '$')

    for i in range(len(sequence1) + 1): #O(nm) time
        for j in range(len(sequence2) + 1):
            newTuple = self.nextValue(i, j, matrix, sequence1, sequence2)
            if newTuple != 0:
                matrix[i][j] = newTuple

    bestFit = matrix[len(sequence1)][len(sequence2)][0]
    alignment1, alignment2 = self.getAlignments(matrix, sequence1, sequence2)  #O(n) time, with O(n) space for the strings

    return bestFit, alignment1, alignment2
```

```python
# Time: O(1), because each lookup is constant, and we have no loops, just lots of
#       simple comparisions
# Space: O(1), no objects of length n are created, each value stored is O(1)
def nextValue(self, i, j, matrix, sequence1, sequence2):
    topValue = float("inf")
    leftValue = float("inf")
    diagonalValue = float("inf")
    validOption = False
    leftValid = False
    topValid = False
    diagonalValid = False

    if i - 1 >= 0:                  # All these comparisons are O(1)
        validOption = True
        topValid = True
        topValue = matrix[i - 1][j][0]
    if j - 1 >= 0:
        validOption = True
        leftValid = True
        leftValue = matrix[i][j - 1][0]
    if i - 1 >= 0 and j - 1 >= 0:
        validOption = True
        diagonalValid = True
        diagonalValue = matrix[i - 1][j - 1][0]

    if validOption:
        if diagonalValid:
            if sequence1[i -1] == sequence2[j-1]:
                diagonalValue = diagonalValue + MATCH
            else:
                diagonalValue = diagonalValue + SUB

        if topValid:
            topValue = topValue + INDEL

        if leftValid:
            leftValue = leftValue + INDEL

        tuple = self.getMin(diagonalValue, topValue, leftValue);    #O(1) time and space
        return tuple
    else:
        return 0
```

```python
# Time: O(1), these are just simple comparisions to find the minimum value
# Space: O(1), we only create and store a single tuple which takes O(1) space
def getMin(self, diagonal, top, left):

    if diagonal < top and diagonal < left:          # All these comparisons are O(1) time and space
        return (diagonal, "D")
    elif top < diagonal and top < left:
        return (top, "T")
    elif left < diagonal and left < top:
        return (left, "L")
    else:
        minValue = min(diagonal, top, left)

        if minValue == diagonal:
            return (diagonal, "D")
        elif minValue == top:
            return (top, "T")
        else:
            return (left, "L")

# Time: O(n) or O(m), whichever is largest, as we travers back through the 2d matrix
#       all the way to the upper right corner
# Space: O(n) or O(m), we store strings of length m and n,
def getAlignments(self, matrix, sequence1, sequence2):
    i = len(sequence1)
    j = len(sequence2)
    letterIndex1 = len(sequence1) - 1
    letterIndex2 = len(sequence2) - 1
    currentAlignment1 = ""
    currentAlignment2 = ""
    currentTuple = matrix[i][j]

    while currentTuple[1] != '$':           # O(m) or O(n) worst case, as it will traverse through the matrix back
                                            # to the origin
        if currentTuple[1] == "T":          # Each comparison is O(1) time
            currentAlignment2 = "-" + currentAlignment2
            currentAlignment1 = sequence1[letterIndex1] + currentAlignment1
            letterIndex1 -= 1
            i = i - 1
        elif currentTuple[1] == "L":
            currentAlignment1 = "-" + currentAlignment1
            currentAlignment2 = sequence2[letterIndex2] + currentAlignment2
            letterIndex2 -= 1
            j = j - 1
        else:
            currentAlignment1 = sequence1[letterIndex1] + currentAlignment1
            currentAlignment2 = sequence2[letterIndex2] + currentAlignment2
            i = i - 1
            j = j - 1
            letterIndex1 -= 1
            letterIndex2 -= 1
        currentTuple = matrix[i][j]

    return currentAlignment1, currentAlignment2
```

Banded Algorithm:

```
#Banded Algorithm
# Time: O(kn), we iterate through a 2d array of n*k size, so if each element is O(1),
#   then our time complexity for this algorithm to make the array and iterate through it
#   is O(kn) time
# Space: O(kn), to hold a 2d matrix of size k*n
def bandwidthSequenceAlgorithm(self, sequence1, sequence2, align_length):
    if sequence1 == sequence2:
        return MATCH * len(sequence1), sequence1, sequence2
    else:
        k = 7;

        sequence1 = sequence1[:align_length]
        sequence2 = sequence2[:align_length]

        if abs(len(sequence1) - len(sequence2)) > MAXINDELS:
            return float("inf"), "No Alignment Possible", "No Alignment Possible"

        sequence1 = sequence1[:align_length]
        sequence2 = sequence2[:align_length]

        if (len(sequence2) > len(sequence1)):
            temp = sequence2
            sequence2 = sequence1
            sequence1 = temp

        matrix = [[(float("inf"), 'X') for j in range(k)] for i in range(len(sequence1) + 1)]   #O(kn) time and space for this 2d array
        matrix[0][0] = (0, '$')
        shift = 0
        for i in range(len(sequence1) + 1):     # Loops through k*n times, O(kn) complexity
            if i - (len(sequence1) + 1) >= -3:
                pass
            elif i > 3:
                shift += 1
            for j in range(k):
                j += shift
                if i - j > 3:               # These comparisions are O(1)
                    continue
                elif j - i > 3:
                    break
                elif j > len(sequence2):
                    break

                if i <= 3:
                    newTuple = self.nextValue(i, j, matrix, sequence1, sequence2)
                    if newTuple != 0:
                        matrix[i][j] = newTuple
                else:
                    newTuple = self.bandedNextValue(i, j, matrix, sequence1, sequence2)
                    if newTuple != 0:
                        matrix[i][j - (i - 3)] = newTuple

        bestFit = float("inf")
        indexOfBestFit = 0

        for j in range(k):          #O(k) time, but is small compared to O(kn)
            if matrix[len(sequence1)][j][0] == float("inf"):
                bestFit = matrix[len(sequence1)][j - 1][0]
                indexOfBestFit = j - 1
                break

        alignment1, alignment2 = self.getBandwidthAlignments(matrix, sequence1, sequence2, indexOfBestFit)  #O(n) or O(m) time and space
        return bestFit, alignment1, alignment2
```

```python
# Time: O(1), because each lookup is constant, and we have no loops, just lots of
#       simple comparisions
# Space: O(1), no objects of length n are created, each value stored is O(1)
def bandedNextValue(self, i, j, matrix, sequence1, sequence2):
    topValue = float("inf")
    leftValue = float("inf")
    diagonalValue = float("inf")
    validOption = False
    leftValid = False
    topValid = False
    diagonalValid = False

    shiftedI = i - 3

    if i - 1 >= 0 and j - shiftedI + 1 < 7:          # comparisions are O(1)
        validOption = True
        topValid = True
        topValue = matrix[i - 1][j + 1 - shiftedI][0]
    if j - shiftedI - 1 >= 0:
        validOption = True
        leftValid = True
        leftValue = matrix[i][j - 1 - shiftedI][0]
    if i - 1 >= 0:
        validOption = True
        diagonalValid = True
        diagonalValue = matrix[i - 1][j - shiftedI][0]

    if validOption:
        if diagonalValid:
            if sequence1[i -1] == sequence2[j - 1]:
                diagonalValue = diagonalValue + MATCH
            else:
                diagonalValue = diagonalValue + SUB

        if topValid:
            topValue = topValue + INDEL

        if leftValid:
            leftValue = leftValue + INDEL

        tuple = self.getMin(diagonalValue, topValue, leftValue); #O(1) time and space
        return tuple
    else:
        return 0
```

```python
# Time: O(n) or O(m), whichever is largest, as we travers back through the 2d matrix
#       all the way to the upper right corner
# Space: O(n) or O(m), we store strings of length m and n,
def getBandwidthAlignments(self, matrix, sequence1, sequence2, indexOfBestFit):
    i = len(sequence1)
    j = indexOfBestFit
    letterIndex1 = len(sequence1) - 1
    letterIndex2 = len(sequence2) - 1
    currentAlignment1 = ""
    currentAlignment2 = ""
    currentTuple = matrix[i][j]

    while currentTuple[1] != '$':           # Will loop through matrix until gets to origin, O(n) or O(m) complexity
        if i <= 3:
            subString1 = sequence1[:(letterIndex1 + 1)]         # Each of these is O(1)
            subString2 = sequence2[:(letterIndex2 + 1)]
            beginningAlignment1, beginningAlignment2 = self.getAlignments(matrix, subString1, subString2)
            currentAlignment1 = beginningAlignment1 + currentAlignment1
            currentAlignment2 = beginningAlignment2 + currentAlignment2
            break
        if currentTuple[1] == "T":
            currentAlignment2 = "-" + currentAlignment2
            currentAlignment1 = sequence1[letterIndex1] + currentAlignment1
            letterIndex1 -= 1
            i = i - 1
            j = j + 1
        elif currentTuple[1] == "L":
            currentAlignment1 = "-" + currentAlignment1
            currentAlignment2 = sequence2[letterIndex2] + currentAlignment2
            letterIndex2 -= 1
            j = j - 1
        else:
            currentAlignment1 = sequence1[letterIndex1] + currentAlignment1
            currentAlignment2 = sequence2[letterIndex2] + currentAlignment2
            i = i - 1
            letterIndex1 -= 1
            letterIndex2 -= 1
        currentTuple = matrix[i][j]

    return currentAlignment1, currentAlignment2
```