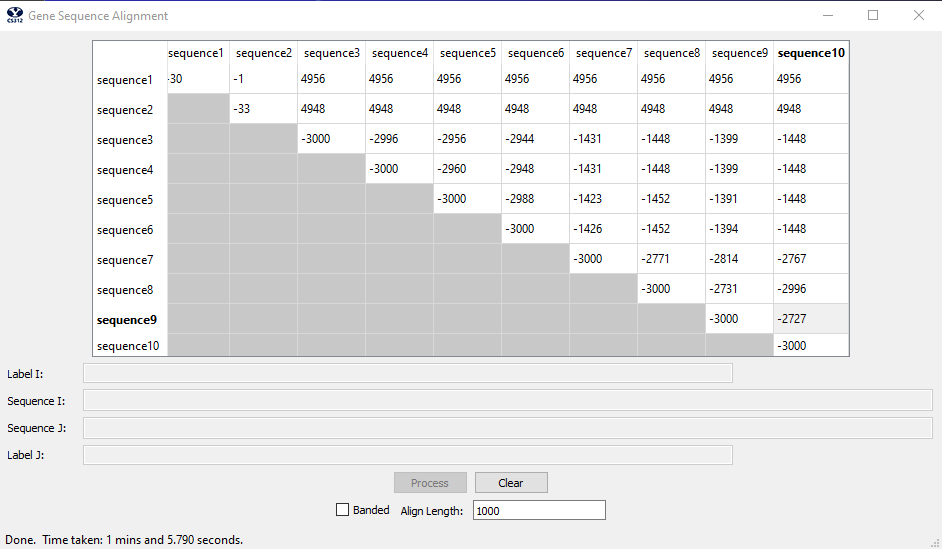
Matthew R. Christensen

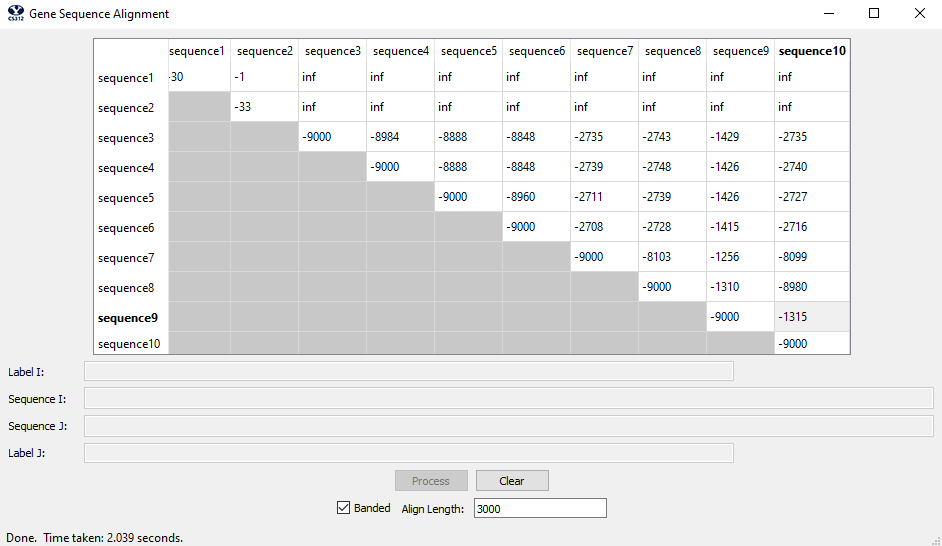
CS 312 (001) – Dr Martinez, Tony R

March 23, 2021

Gene Sequencing Algorithm Report

**Screenshots**



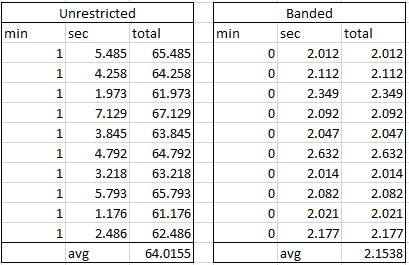


**Analysis**

First, we will talk about the performance requirements of both the unrestricted and banded algorithms. We will then talk generally about the alignment extraction algorithm, including the backtrace. Next, we will talk about the time and space complexity of the program generally and abstractly, then followed by a specific review of the two implementations of the algorithm: the unrestricted and banded implementations.

**Performance Requirements**

The defined performance requirements for this lab were to fill the 10x10 result matrix for 1000 base pairs with the unrestricted algorithm in 120seconds or less. Additionally, the banded algorithm should fill in the 10x10 result matrix, for 3000 base pairs, in 10 seconds or less. These performance requirements were met in this lab and implementation.



Above we see that the average time for the unrestricted algorithm was 64.0155 seconds – which is around half of the 120 second time requirement for performance. Additionally, the banded algorithm took, on average, 2.1538 seconds to complete (which is less than the 10 second requirement for the performance review). Thus, both algorthims satisfy the timed performance requirements.

Additionally, it is important to note that each test for the unrestricted was completed for a 10x10 result matrix with an align length of 1,000 (meaning the first 1,000 characters were used in finding the alignment). And the banded algorithm used an align length of 3,000 on a 10x10 result matrix for the corresponding tests.

Furthermore, the specific performance requirements in regards to time complexity and space complexity will be discussed in the relevant section below.

**Alignment Extraction Algorithm**

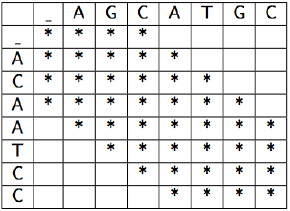
We will now analyze the alignment extraction algorithm used for the gene sequencing of this lab. Generally speaking, we use the Needleman–Wunsch algorithm to align gene codes between two sequences. This is done using dynamic programming, meaning that the problem is divided into subproblems where each smaller subproblem is used to find an optimal solution for the larger problem. Thus, we build a dynamic programming table to complete the Needleman–Wunsch algorithm over a given pair of gene sequences. In each entry of the dynamic table we store the result of the comparison for a given part of a comparison between genes (thus, comparing two letters and deciding if it’s a match, mismatch or an insertion or deletion) – thus storing the score. Additionally, we also store the most optimal direction from that point forward – known as backtraces or previous pointers. Then, to then build a path or extract the alignment, we use the dynamic table and traverse the backtraces to know which box to navigate to while collecting the scores and sequences along the way as we traverse through the dynamic table. In the case of this lab, indels (insertions or deletions) have a penalization of 5, substitutions (where a single character does not match) have a penalization of 1, and matches are rewarded with 3 units. Finally, it is important to note that we create a smaller table that reflects the smaller dynamic space necessary while using the banded algorithm such that we have a table that meets the demands for a O(kn) time complexity. This will be further discussed in the next subsection, including how the table is created, the O(kn) complexity, as well as the definition of the constant k.

**Time and Space Complexity**

In regards to the time complexity and the space complexity, we will first talk about the specific requirements and goals for both time and space complexity in addition to defining some premises that are important to this analysis. Then we will analyze pseudo code that acts as an accurate model for how we implemented the algorithm (additionally, the actual code will be included in the appendix with specific comments on where the time and space complexity are coming from in a much more granular fashion). This will be done on a function by function level for the unrestricted algorithm first, which will then be contrasted by the differences in the banded algorithm that make the improvements possible. Then the functions’ big O complexity, for both time and space, will be added together to prove the requirements for time and space complexity are met by our implementation of this lab.

If the level of detail is not granular enough in this section (from time and space complexity) please reference the code in the appendix for more detail and more specifics concerning the big O analysis.

Additionally, it is important to define the constant k. The constant k represents the bandwidth of the scoring matrix (dynamic table), which is important for reasoning about the performance of the banded algorithm’s implementation. We can find the value of k by solving for 2d + 1. For the purposes of this project d is equal to 3, meaning k = 2d + 1 = 2(3) + 1 = 6 + 1. Thus, k is equal to 7. A visual representation of what this looks like generally on a score matrix/dynamic table can be seen below:



Notice how only the bands around the diagonal of the table/scoring matrix are filled (while the very corners are left untouched).

Additionally, there are performance requirements concerning the time and space complexity for this lab. The unrestricted algorithm (the first one we are going to look at) must have a time and space complexity of O(nm). For the purpose of this lab, we are going to assume that n and m are similar enough to say that we must have a O(n^2). For the banded algorithm (the second algorithm we are going to analyze) we expect an optimization and speedup such that we achieve a time and space complexity of O(kn) (where k is the previously define constant representing bandwidth within the score matrix and denoting the entries filled in said matrix/table).

First we analyze the non-banded, unrestricted, full algorithm. The performance goal is O(n^2) for both time complexity as well as space complexity. The unrestricted algorithm is comprised of the following parts/functions:

generate\_unrestricted\_map(s1, s2):

x\_value = len(s2)

y\_value = len(s1)

while True:

direction = (self.matrix[y\_value][x\_value])[1]

if direction == LEFT:

alignment\_1, alignment\_2, x\_value = left\_unrestricted(alignment\_1, alignment\_2, seq\_2, x\_value)

elif direction == DIAGONAL:

alignment\_1, alignment\_2, x\_value, y\_value = diagonal\_unrestricted(alignment\_1, alignment\_2, seq\_1, seq\_2, x\_value, y\_value)

else:

alignment\_1, alignment\_2, y\_value = down\_unrestricted(alignment\_1, alignment\_2, seq\_1, y\_value)

if y\_value == 0 and x\_value == 0:

break

return alignment\_1, alignment\_2

This function has a time complexity of O(2n) = O(n), as the while loop will continue until the x and y are at 0. Thus, in the worst was scenario we traverse all the way up and over through the score matrix before reaching 0. Furthermore, the functions for left, right, and down are all constant time in both space and time complexity and thus won’t change the overall class of O(n) for time complexity (additionally, they are constant time for space complexity also and will thus be omitted from the discussion concerning space complexity). The space complexity is constant because of the only space need being the return of the two alignments.

Next is the alignment calculations for the unrestricted algorithm:

unrestricted\_alignment(seq\_1, seq\_2):  
 self.matrix = [[0 for j in range(0, len(seq\_2))] for i in range(0, len(seq\_1))]  
 for i in range(0, len\_seq\_2):  
 self.matrix[0][i] = (i \* 5, LEFT)  
 for i in range(0, len\_seq\_1):  
 self.matrix[i][0] = (i \* 5, DOWN)  
 for i in range(1, len\_seq\_1):  
 for j in range(1, len\_seq\_2):  
 match = False  
 if seq\_2[j] == seq\_1[i]:  
 match = True  
 direction = LEFT  
 left\_val = self.matrix[i][j - 1]  
 best\_distance = left\_val[0] + 5  
 top\_val = self.matrix[i - 1][j]  
 top = top\_val[0] + 5 # TOP  
 if top <= best\_distance:  
 best\_distance = top  
 direction = DOWN  
 top\_left\_val = self.matrix[i - 1][j - 1]  
 if match:  
 top\_left = top\_left\_val[0] - D  
 else:  
 top\_left = top\_left\_val[0] + 1  
 if top\_left > best\_distance:  
 pass  
 else:  
 best\_distance = top\_left  
 direction = DIAGONAL  
 self.matrix[i][j] = (best\_distance, direction)  
 return self.matrix[len(seq\_1) - 1][len(seq\_2) - 1][0]

For this function we store a matrix of n by m, where n and m are considered equivalent. Thus our space complexity will be O(n^2) to hold such a matrix. In regards to time complexity, we need to iterate through the entire n x m (or n x n) matrix to populate the table with scores and pointers in addition to general population of O(n) and O(m). Deciding the score and pointer are all constant time, thus the time complexity will be O(n + m + n^2) = O(n + n + n^2) = O(2n + n^2) = O(n^2).

Finally, for the unrestricted algorithm we simply call the generate and alignment functions in sequentially:

align(s1, s2):

align1, align2 = generate\_unrestricted\_map(s1, s2)

score = unrestricted\_alignment(s1, s2)

return {'align\_cost': score, 'seqi\_first100': alignment1, 'seqj\_first100': alignment2}

Thus, the overall big O for time complexity would be:

**O(unrestricted\_map) + O(unrestricted\_align) = O(n) + O(n^2) = O(n + n^2) = O(n^2)**

While space complexity is:

**O(unrestricted\_map) + O(unrestricted\_align) = O(1) + O(n^2) = O(1 + n^2) = O(n^2)**

Thus, we show that our unrestricted algorithm complies with the performance demands of O(n^2) for time and space complexity.

Next, we analyze the time and space complexity for the banded algorithm. We expect a performance speedup such that time and space complexity are of O(kn), where k represents to selected bandwidth for the score matrix/table. First, we analyze the alignment function:

banded\_alignment(seq\_1, seq\_2):  
 self.banded\_matrix = [[0 for j in range(0, k)] for i in range(0, len(sequence\_1))]

r0 = range(1, len(sequence\_1))

r1 = range(0, total\_chars)

r2 = range(0, 6)

for i in r0:

if seq\_to\_beginning + total\_chars + 1 >= len(sequence\_2):

temp\_seq = sequence\_2[seq\_to\_beginning:len(sequence\_2) + 1]

else:

temp\_seq = sequence\_2[seq\_to\_beginning:total\_chars + seq\_to\_beginning]

for j in r1:

dist\_to\_beat = calc\_dist(i, j, sequence\_1, sequence\_2)

self.banded\_matrix[i][index\_j] = [dist\_to\_beat, direction]

if limit\_max:

seq\_to\_beginning = seq\_to\_beginning + 1

if total\_chars < k and not limit\_max:

total\_chars = total\_chars + 1

if total\_chars == k:

limit\_max = True

if i >= len(sequence\_2) - D:

total\_chars = total\_chars - 1

if starting\_index != 0:

starting\_index = starting\_index - 1

sequence\_1\_len = len(sequence\_1) - 1

lowest\_val = (self.banded\_matrix[sequence\_1\_len][0])[0]

chars\_total = total\_chars + 1

if chars\_total > K:

return INF

for i in r1:

temp = (self.banded\_matrix[sequence\_1\_len][i])[0]

if temp <= lowest\_val:

lowest\_val = temp

position = i

for i in r2:

if self.banded\_matrix[sequence\_1\_len][i + 1] == 0:

break

position = i

return position + 1

The time complexity is a little complex in this function. The outer loop of range r0 is of size n. The calls and operations within the loop are of O(100), constant, and O(k) for the worst case scenario – where 100 is the number of character to look at in a given gene sequence (as defined by this lab’s spec). Thus the total time complexity is of O(n(100 + c + k)) = O(kn), which fulfills the time complexity requirement.

The only meaningful metric for time complexity is the matrix of scores that we store. Because the banded constant of k allows us to select width off of a predetermined constant we create a smaller matric of [0 for j in range(0, k)] for i in range(0, n), meaning that our space complexity is O(kn).

Next is the function for the path through the matrix:

generate\_banded\_map(sequence\_1, sequence\_2, current\_position):

alignment\_1 = alignment\_2 = ""

seq\_to\_pos = len(sequence\_2) - 1

x\_val, y\_val = current\_position, len(sequence\_1) - 1

if |len(sequence\_1) - len(sequence\_2)| >= D:

return NO\_ALIGNMENT, NO\_ALIGNMENT

while True:

cur\_val = self.banded\_matrix[y\_val][x\_val]

direction = cur\_val[1]

if direction == DOWN:

alignment\_1, alignment\_2, x\_val, y\_val = down\_banded(alignment\_1, alignment\_2, sequence\_1, x\_val, y\_val)

elif direction == LEFT:

alignment\_1, alignment\_2, seq\_to\_pos, x\_val = left\_banded(alignment\_1, alignment\_2, sequence\_2, seq\_to\_pos, val)

else:

alignment\_1, alignment\_2, seq\_to\_pos, y\_val = diagonal\_banded(alignment\_1, alignment\_2, sequence\_1, sequence\_2, seq\_to\_pos, y\_val)

if y\_val == 0 and x\_val == D:

break

return alignment\_1, alignment\_2

The time and space complexity comprises of this function are very similar to the unrestricted algorithms implementation, as such this section will be a bit more brief to avoid redundant information/discussion. The time complexity is made up of a while loop that moves an x and y position through our matrix. Because the edges of the matrix will always be infinity we are guaranteed that at maximum we will move 2n times in the worst case scenario (from the bottom corner to the top). Thus, because everything in the loop constant time, we have a O(2n) = O(n). For the space complexity, just as the unrestricted algorithm, the space complexity is constant because of the only space need being the return of the two alignments. Thus we have a constant space complexity.

Finally, for the banned algorithm we simply call the generate and alignment functions in sequentially with some interpretation of data:

align(s1, s2):

pos = banded\_alignment(s1, s2, k)

if pos == infinity:

return {'align\_cost': pos, 'seqi\_first100': NO\_ALIGNMENT, 'seqj\_first100': NO\_ALIGNMENT}

align1, align2 = generate\_unrestricted\_map(s1, s2, pos)

return {'align\_cost': pos, 'seqi\_first100': alignment1, 'seqj\_first100': alignment2}

Thus, the overall big O for time complexity would be:

**O(banded\_map) + O(banded\_align) = O(n) + O(kn) = O(n + kn) = O(kn)**

While space complexity is:

**O(banded\_map) + O(banded\_align) = O(1) + O(kn) = O(1 + kn) = O(kn)**

Thus, we show that our banded algorithm complies with the performance demands of O(kn) for time and space complexity.

Again, please see to code in the appendix for further detail and explanation of complexity if desired/required.

**Results**





Also see the screenshots included at the beginning of this lab report.

**Conclusion**

In conclusion, we have shown that our algorithm is correctly running the gene sequencing algorithm by implementing the Needleman–Wunsch algorithm effectively both in the unrestricted and banded implementations. Furthermore, we have shown that both the unrestricted and banded algorithm implementations run within the 120 and 10 second time requirements respectively as well as meeting the respective **O(n^2)** and **O(kn)** time and space complexity requirements. Thus, we conclude that the analysis of the gene sequencing show completeness and compliance.

**Appendix**

def align(self, sequence\_1, sequence\_2, banded, align\_length):

self.banded = banded

self.MaxCharactersToAlign = align\_length

if not banded: # Unrestricted

align1, align2 = self.generate\_unrestricted\_map(sequence\_1, sequence\_2) # O(n) time, O(1) space

score = self.unrestricted\_alignment(sequence\_1, sequence\_2) # O(n^2) for time and space

else: # Banded

pos = self.banded\_alignment(sequence\_1, sequence\_2, K) # O(kn) for time and space

if pos == INF:

return {'align\_cost': pos, 'seqi\_first100': NO\_ALIGNMENT, 'seqj\_first100': NO\_ALIGNMENT}

score = (self.banded\_matrix[len(sequence\_1)][pos])[0]

align1, align2 = self.generate\_banded\_map(sequence\_1, sequence\_2, pos) # O(n) time, O(1) space

alignment1, alignment2 = get\_num\_chars(align1, align2) # const

return {'align\_cost': score, 'seqi\_first100': alignment1, 'seqj\_first100': alignment2}

# Time: O(2n) = O(n)

# Space: constant (see unrestricted func for more detail)

def generate\_banded\_map(self, sequence\_1, sequence\_2, current\_position):

alignment\_1 = alignment\_2 = ""

seq\_to\_pos = len(sequence\_2) - 1

x\_val, y\_val = current\_position, len(sequence\_1) - 1

if abs(len(sequence\_1) - len(sequence\_2)) >= D:

return NO\_ALIGNMENT, NO\_ALIGNMENT

while True: # n + n = 2n

cur\_val = self.banded\_matrix[y\_val][x\_val] # everything in here is const time and space

direction = cur\_val[1]

if direction == DOWN:

alignment\_1, alignment\_2, x\_val, y\_val = down\_banded(alignment\_1, alignment\_2, sequence\_1, x\_val, y\_val)

elif direction == LEFT:

alignment\_1, alignment\_2, seq\_to\_pos, x\_val = left\_banded(alignment\_1, alignment\_2, sequence\_2, seq\_to\_pos,

x\_val)

else:

alignment\_1, alignment\_2, seq\_to\_pos, y\_val = diagonal\_banded(alignment\_1, alignment\_2, sequence\_1, sequence\_2,

seq\_to\_pos, y\_val)

if y\_val == 0 and x\_val == D:

break

return alignment\_1, alignment\_2

# Time: O(n(100 + c + k)) = O(n(100 k)) = O(kn(100)) = O(kn)

# Space: O(kn)

def banded\_alignment(self, sequence\_1, sequence\_2, k):

global direction

self.banded\_matrix = [[0 for j in range(0, k)] for i in range(0, len(sequence\_1))]

sequence\_2, sequence\_1 = sequence\_2, ' ' + sequence\_1

for i in range(0, 4):

self.banded\_matrix[i][D - i] = (i \* 5, DOWN)

self.banded\_matrix[0][D + i] = (i \* 5, LEFT)

seq\_to\_beginning = 0

limit\_max = False

total\_chars = 4

starting\_index = D

position = 0

r0 = range(1, len(sequence\_1))

r1 = range(0, total\_chars)

r2 = range(0, 6)

for i in r0: # n

if seq\_to\_beginning + total\_chars + 1 >= len(sequence\_2):

temp\_seq = sequence\_2[seq\_to\_beginning:len(sequence\_2) + 1]

else:

temp\_seq = sequence\_2[seq\_to\_beginning:total\_chars + seq\_to\_beginning]

for j in r1: # 100

match = False # everything in here is constant

if sequence\_1[i] == temp\_seq[j]:

match = True

dist\_to\_beat = INF

if j - 1 >= starting\_index:

left\_val = self.banded\_matrix[i][j - 1]

dist\_to\_beat = left\_val[0] + 5

direction = LEFT

index\_j = starting\_index + j

if index\_j + 1 < K:

top\_val = self.banded\_matrix[i - 1][index\_j + 1]

top = top\_val[0] + 5

if top >= dist\_to\_beat:

continue

dist\_to\_beat = top

direction = DOWN

diag\_val = self.banded\_matrix[i - 1][index\_j]

if match:

top\_left = diag\_val[0] - D

else:

top\_left = diag\_val[0] + 1

if top\_left >= dist\_to\_beat:

pass

else:

dist\_to\_beat = top\_left

direction = DIAGONAL

self.banded\_matrix[i][index\_j] = [dist\_to\_beat, direction]

if limit\_max:

seq\_to\_beginning = seq\_to\_beginning + 1

if total\_chars < k and not limit\_max:

total\_chars = total\_chars + 1

if total\_chars == k:

limit\_max = True

if i >= len(sequence\_2) - D:

total\_chars = total\_chars - 1

if starting\_index != 0:

starting\_index = starting\_index - 1

sequence\_1\_len = len(sequence\_1) - 1

lowest\_val = (self.banded\_matrix[sequence\_1\_len][0])[0]

chars\_total = total\_chars + 1

if chars\_total > K:

return INF

for i in r1: # 100

temp = (self.banded\_matrix[sequence\_1\_len][i])[0]

if temp <= lowest\_val:

lowest\_val = temp

position = i

for i in r2: # k

if self.banded\_matrix[sequence\_1\_len][i + 1] == 0:

break

position = i

return position + 1

# Time: O(2n) = O(n)

# Space: constant

def generate\_unrestricted\_map(self, seq\_1, seq\_2):

alignment\_1 = alignment\_2 = ""

x\_value, y\_value = len(seq\_2), len(seq\_1)

while True: # Worst case scenario is traverse all the way up and over matrix: n + n

direction = (self.matrix[y\_value][x\_value])[1]

if direction == LEFT:

alignment\_1, alignment\_2, x\_value = left\_unrestricted(alignment\_1, alignment\_2, seq\_2, x\_value) # const time

elif direction == DIAGONAL:

alignment\_1, alignment\_2, x\_value, y\_value = diagonal\_unrestricted(alignment\_1, alignment\_2, seq\_1,

seq\_2, x\_value, y\_value) # const time

else:

alignment\_1, alignment\_2, y\_value = down\_unrestricted(alignment\_1, alignment\_2, seq\_1, y\_value) # const time

if y\_value == 0 and x\_value == 0:

break

return alignment\_1, alignment\_2

# Time: O(n + m +n^2 + c) = O(n + n +n^2) = O(2n +n^2) = O(n^2)

# Space: O(n^2)

def unrestricted\_alignment(self, seq\_1, seq\_2):

seq\_2, seq\_1 = ' ' + seq\_2, ' ' + seq\_1

self.matrix = [[0 for j in range(0, len(seq\_2))] for i in range(0, len(seq\_1))]

for i in range(0, len(seq\_2)): # n

i\_ = i \* 5

self.matrix[0][i] = (i\_, LEFT) # const

# print(i\_)

for i in range(0, len(seq\_1)): # m = n

i\_ = i \* 5

self.matrix[i][0] = (i\_, DOWN) # const

# print(i\_)

for i in range(1, len(seq\_1)):

for j in range(1, len(seq\_2)): # n \* m = n \* n = n^2

match = False

if seq\_2[j] == seq\_1[i]:

match = True

direction = LEFT

left\_val = self.matrix[i][j - 1]

best\_distance = left\_val[0] + 5

top\_val = self.matrix[i - 1][j]

top = top\_val[0] + 5 # TOP

if top <= best\_distance:

best\_distance = top

direction = DOWN

top\_left\_val = self.matrix[i - 1][j - 1]

if match:

top\_left = top\_left\_val[0] - D

else:

top\_left = top\_left\_val[0] + 1

if top\_left > best\_distance:

pass

else:

best\_distance = top\_left

direction = DIAGONAL

self.matrix[i][j] = (best\_distance, direction)

return self.matrix[len(seq\_1) - 1][len(seq\_2) - 1][0]