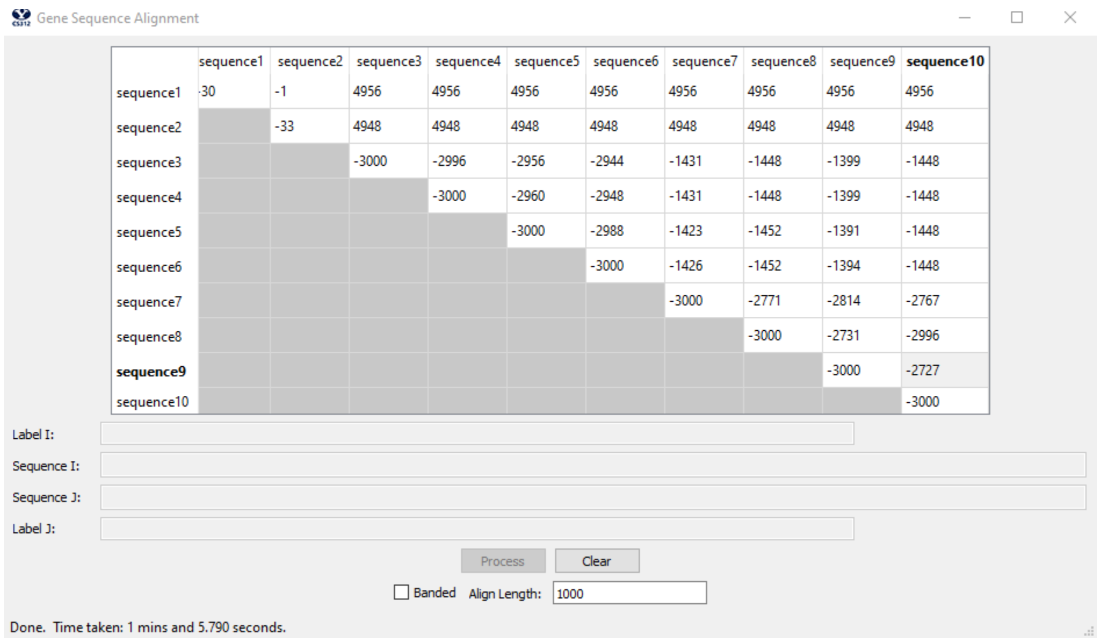
Matthew R. Christensen

CS 312 (001) – Dr Martinez, Tony R

March 23, 2021

Gene Sequencing Algorithm Report

Screenshots



	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	**sequence10**
sequence1	-30	-1	4956	4956	4956	4956	4956	4956	4956	4956
sequence2		-33	4948	4948	4948	4948	4948	4948	4948	4948
sequence3			-3000	-2996	-2956	-2944	-1431	-1448	-1399	-1448
sequence4				-3000	-2960	-2948	-1431	-1448	-1399	-1448
sequence5					-3000	-2988	-1423	-1452	-1391	-1448
sequence6						-3000	-1426	-1452	-1394	-1448
sequence7							-3000	-2771	-2814	-2767
sequence8								-3000	-2731	-2996
sequence9									-3000	-2727
sequence10										-3000

Label I:

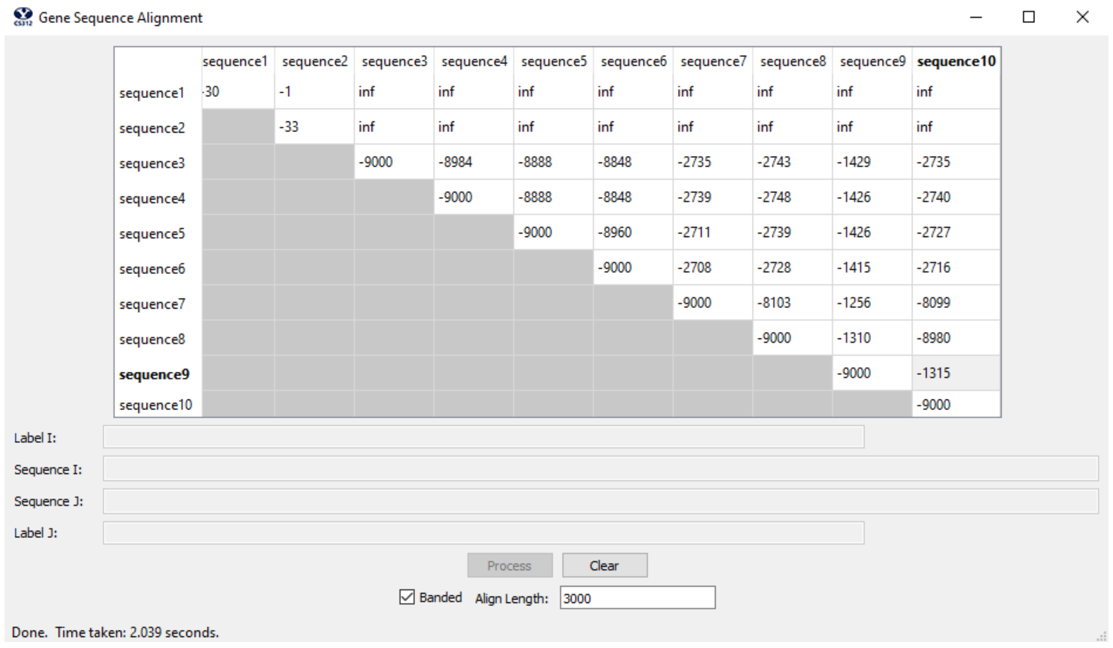
Sequence I:

Sequence J:

Label J:

☐ Banded Align Length: 1000

Done. Time taken: 1 mins and 5.790 seconds.



	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	**sequence10**
sequence1	-30	-1	inf	inf	inf	inf	inf	inf	inf	inf
sequence2		-33	inf	inf	inf	inf	inf	inf	inf	inf
sequence3			-9000	-8984	-8888	-8848	-2735	-2743	-1429	-2735
sequence4				-9000	-8888	-8848	-2739	-2748	-1426	-2740
sequence5					-9000	-8960	-2711	-2739	-1426	-2727
sequence6						-9000	-2708	-2728	-1415	-2716
sequence7							-9000	-8103	-1256	-8099
sequence8								-9000	-1310	-8980
sequence9									-9000	-1315
sequence10										-9000

Label I:

Sequence I:

Sequence J:

Label J:

☑ Banded Align Length: 3000

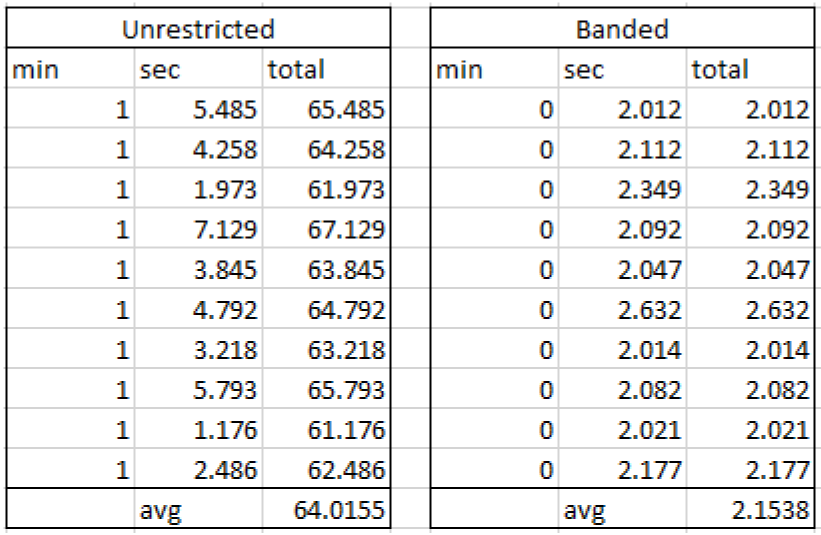
Done. Time taken: 2.039 seconds.

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 120 seconds 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.



Unrestricted		
min	sec	total
1	5.485	65.485
1	4.258	64.258
1	1.973	61.973
1	7.129	67.129
1	3.845	63.845
1	4.792	64.792
1	3.218	63.218
1	5.793	65.793
1	1.176	61.176
1	2.486	62.486
	avg	64.0155

Banded		
min	sec	total
0	2.012	2.012
0	2.112	2.112
0	2.349	2.349
0	2.092	2.092
0	2.047	2.047
0	2.632	2.632
0	2.014	2.014
0	2.082	2.082
0	2.021	2.021
0	2.177	2.177
	avg	2.1538

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

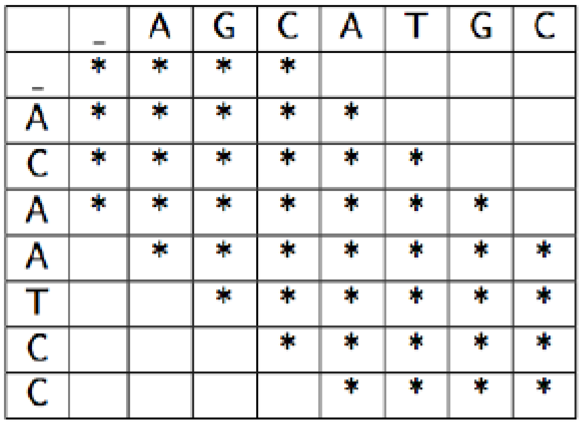
We will now analyze the alignment extraction algorithm used for the gene sequencing of this lab. Generally speaking, we use the Needleman–Wunsch algorithm to align gene codes between two sequences. This is done using dynamic programming, meaning that the problem is divided into subproblems where each 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:



	_	A	G	C	A	T	G	C
_	*	*	*	*				
A	*	*	*	*	*			
C	*	*	*	*	*	*		
A	*	*	*	*	*	*	*	
A		*	*	*	*	*	*	*
T			*	*	*	*	*	*
C				*	*	*	*	*
C					*	*	*	*

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)

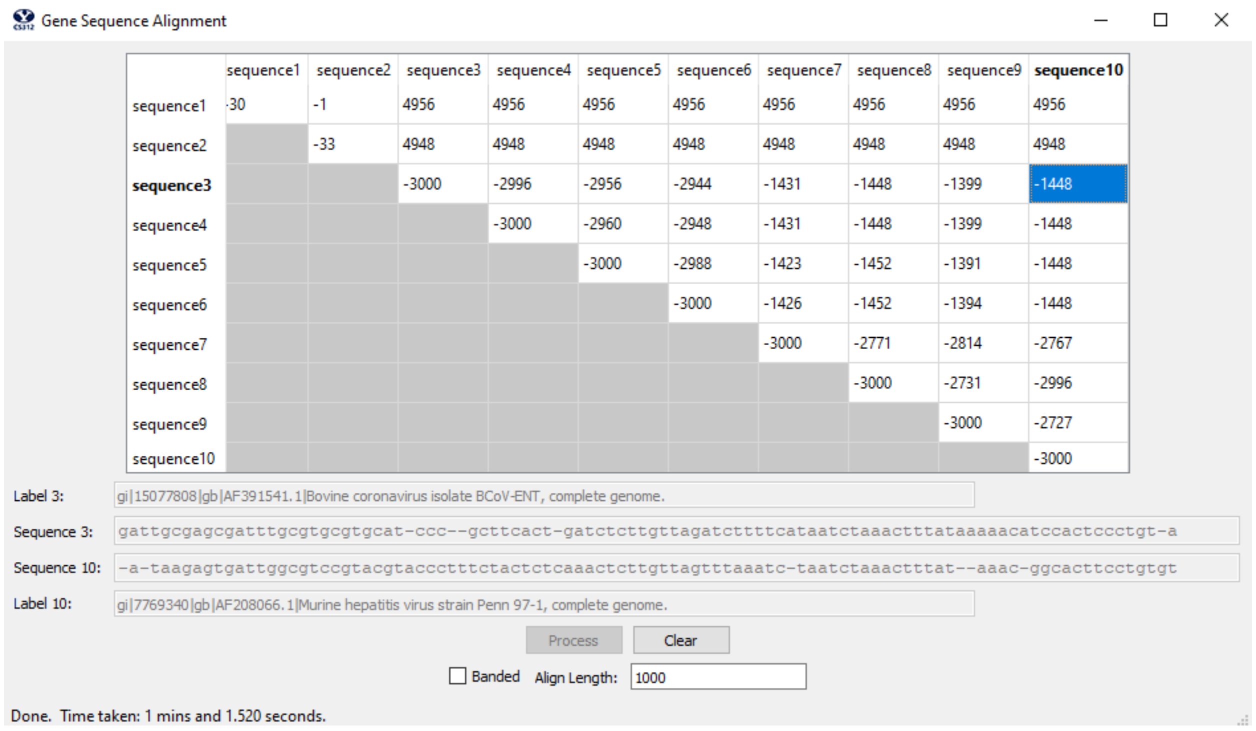
While space complexity is:

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



	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	sequence10
sequence1	-30	-1	4956	4956	4956	4956	4956	4956	4956	4956
sequence2		-33	4948	4948	4948	4948	4948	4948	4948	4948
sequence3			-3000	-2996	-2956	-2944	-1431	-1448	-1399	-1448
sequence4				-3000	-2960	-2948	-1431	-1448	-1399	-1448
sequence5					-3000	-2988	-1423	-1452	-1391	-1448
sequence6						-3000	-1426	-1452	-1394	-1448
sequence7							-3000	-2771	-2814	-2767
sequence8								-3000	-2731	-2996
sequence9									-3000	-2727
sequence10										-3000

Label 3: gi|15077808|gb|AF391541.1|Bovine coronavirus isolate BCoV-ENT, complete genome.

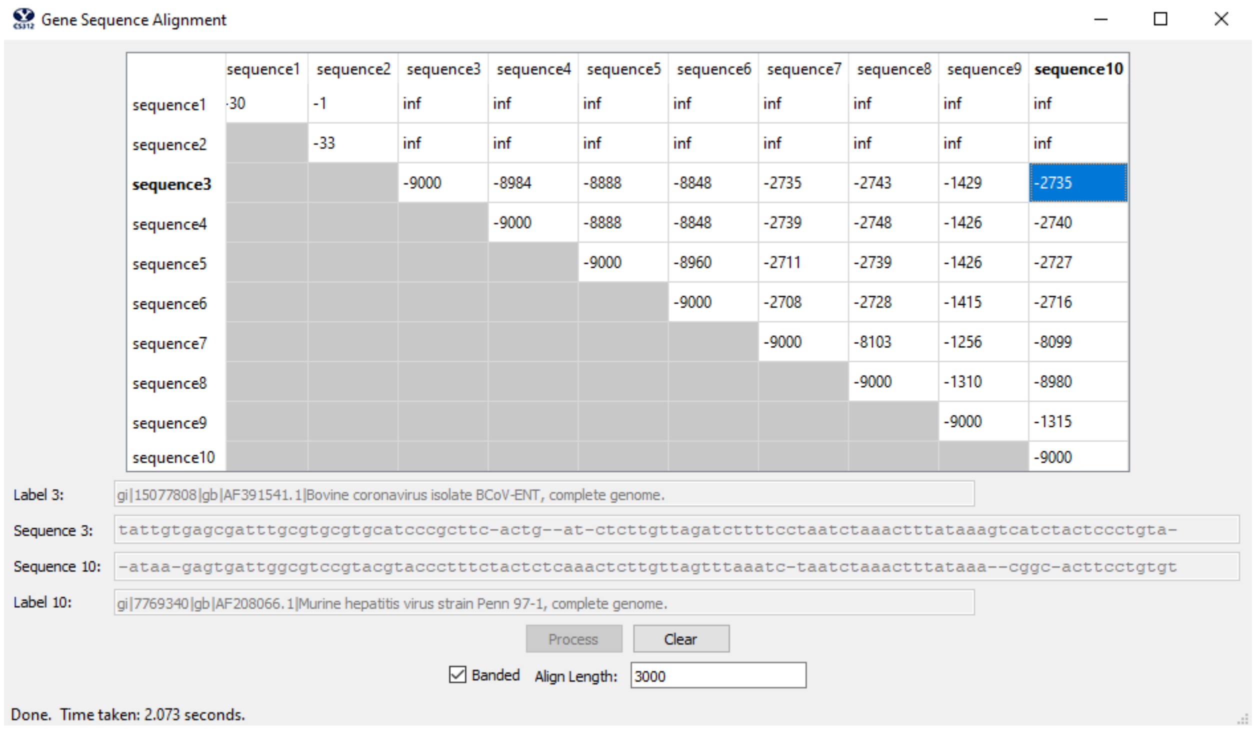
Sequence 3: `gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a`

Sequence 10: `-a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt`

Label 10: gi|7769340|gb|AF208066.1|Murine hepatitis virus strain Penn 97-1, complete genome.

☐ Banded Align Length: 1000

Done. Time taken: 1 mins and 1.520 seconds.



	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	sequence10
sequence1	-30	-1	inf	inf	inf	inf	inf	inf	inf	inf
sequence2		-33	inf	inf	inf	inf	inf	inf	inf	inf
sequence3			-9000	-8984	-8888	-8848	-2735	-2743	-1429	-2735
sequence4				-9000	-8888	-8848	-2739	-2748	-1426	-2740
sequence5					-9000	-8960	-2711	-2739	-1426	-2727
sequence6						-9000	-2708	-2728	-1415	-2716
sequence7							-9000	-8103	-1256	-8099
sequence8								-9000	-1310	-8980
sequence9									-9000	-1315
sequence10										-9000

Label 3: gi|15077808|gb|AF391541.1|Bovine coronavirus isolate BCoV-ENT, complete genome.

Sequence 3: `tattgtgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcctaatctaaactttataaagtcatctactccctgta-`

Sequence 10: `-ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgt`

Label 10: gi|7769340|gb|AF208066.1|Murine hepatitis virus strain Penn 97-1, complete genome.

☑ Banded Align Length: 3000

Done. Time taken: 2.073 seconds.

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]
```