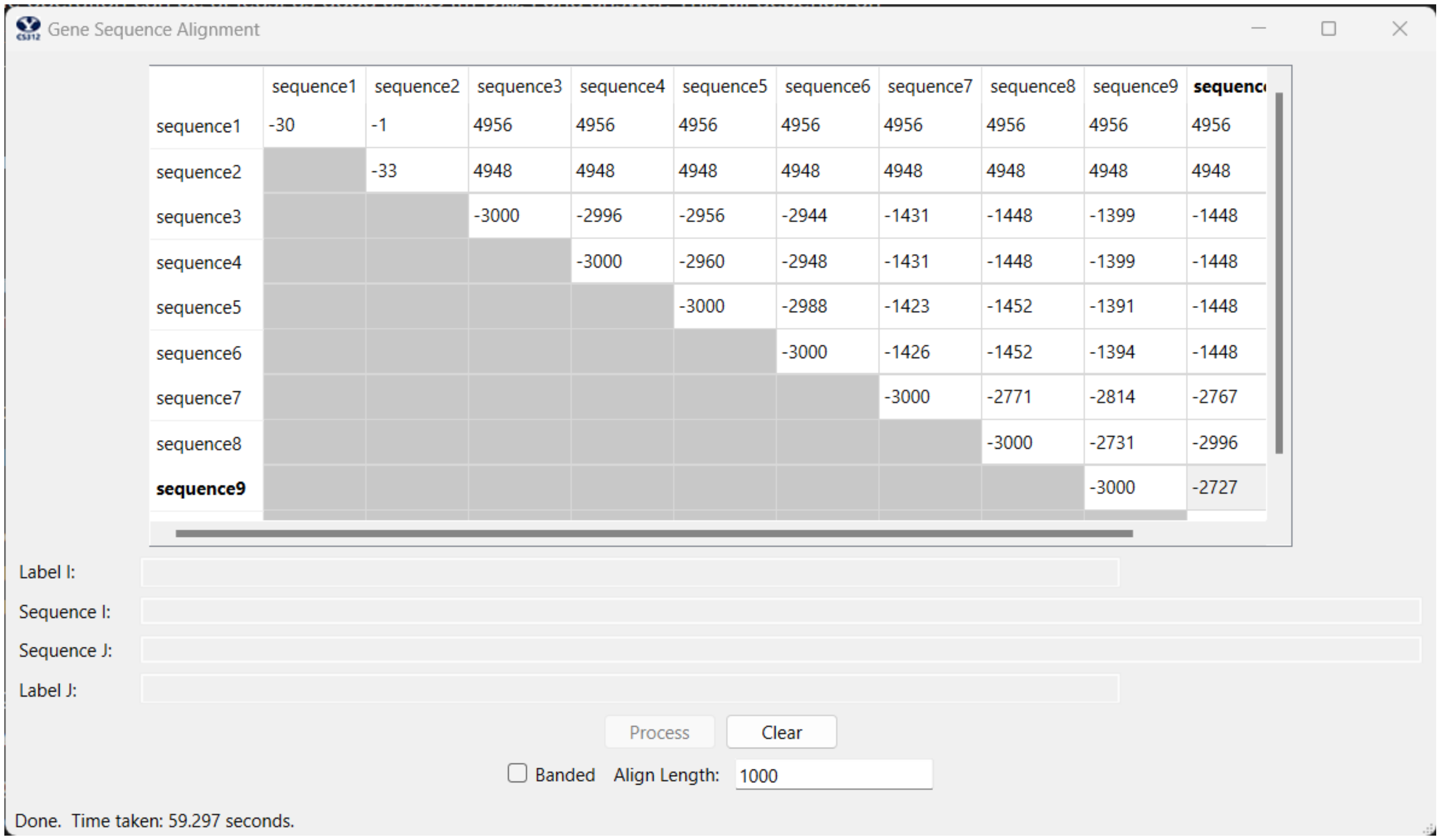
CS 312 Project 4 – Gene Sequence Alignment

Braden Webb

1. My commented source code for both algorithms is included at the end of this document in the appendix.
2. My discussion of the time and space complexity of each algorithm is quite extensive within the comments of the source code.
3. I've included the two required screenshots below.



	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	**sequenc**
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

Label I:

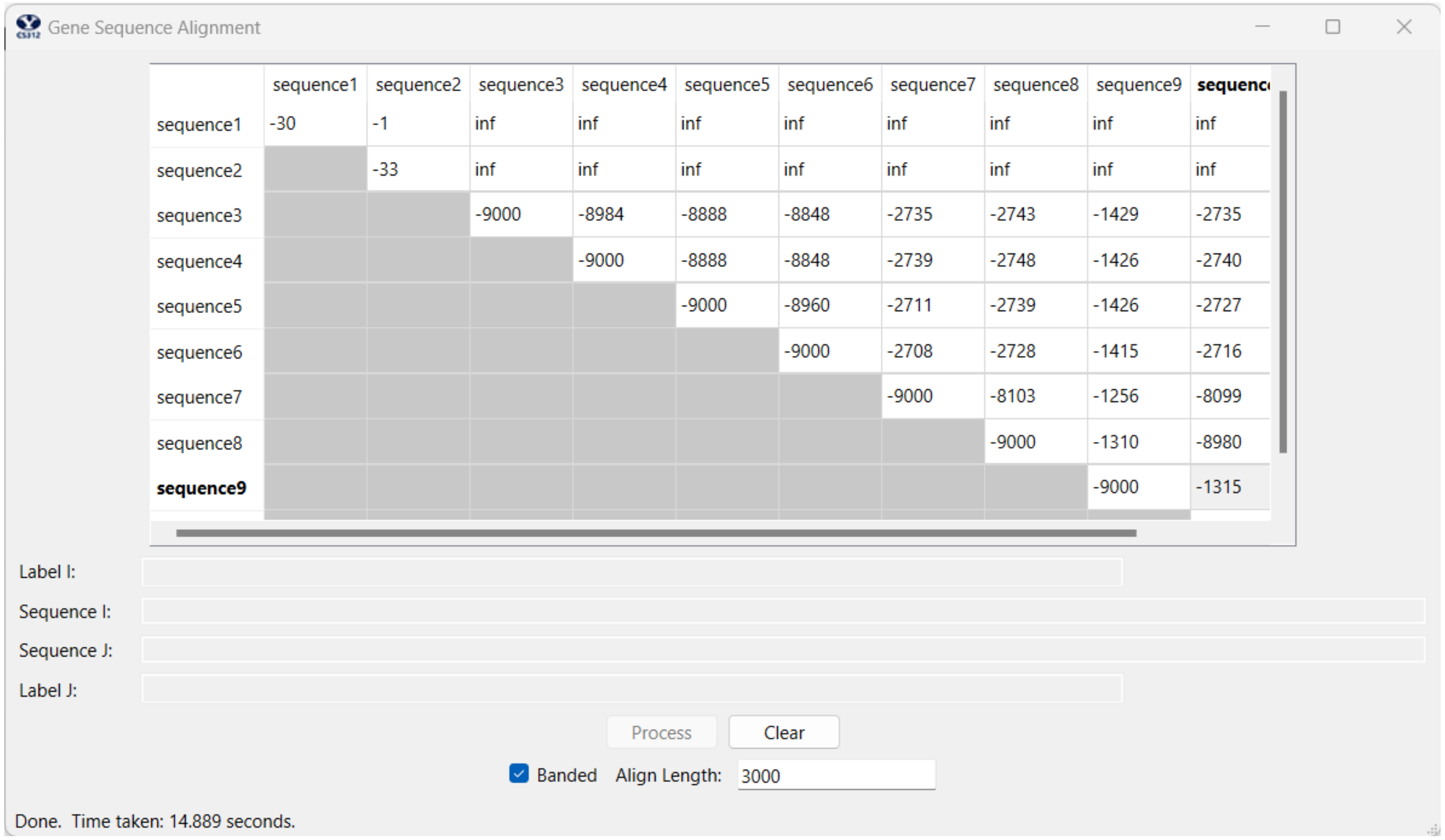
Sequence I:

Sequence J:

Label J:

☐ Banded Align Length: 1000

Done. Time taken: 59.297 seconds.



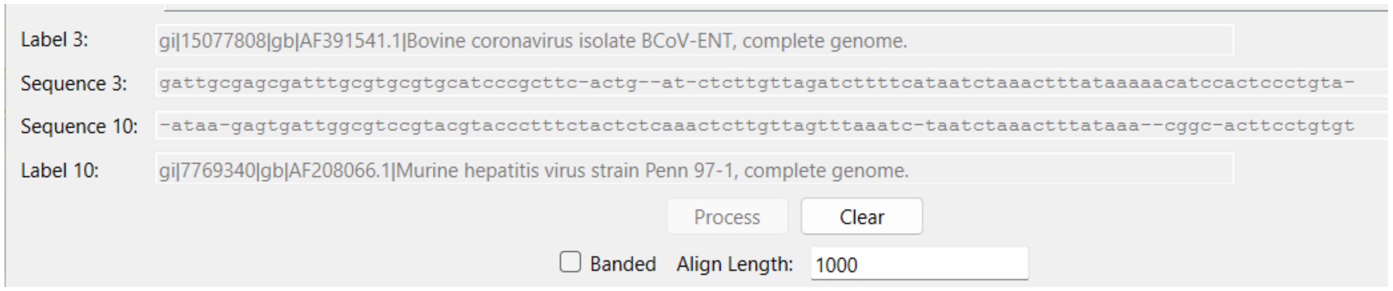
	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	**sequenc**
sequence1	-30	-1	inf	inf	inf	inf	inf	inf	inf	inf
sequence2		-33	inf	inf	inf	inf	inf	inf	inf	inf
sequence3			-9000	-8984	-8888	-8848	-2735	-2743	-1429	-2735
sequence4				-9000	-8888	-8848	-2739	-2748	-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

Label I:
Sequence I:
Sequence J:
Label J:

[✓ Banded] Align Length: 3000

Done. Time taken: 14.889 seconds.

4. The unrestricted alignment, with n=1000, of sequences #3 and #10:

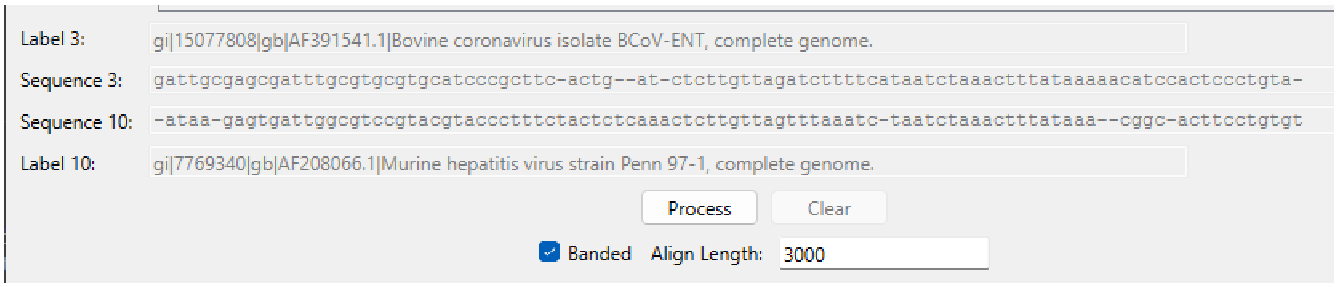


Label 3: gi|15077808|gb|AF391541.1|Bovine coronavirus isolate BCoV-ENT, complete genome.

Sequence 3:
```
gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-
```
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: 1000

The banded alignment, with n=3000, of sequences #3 and #10:



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

Sequence 3:
```
gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-
```

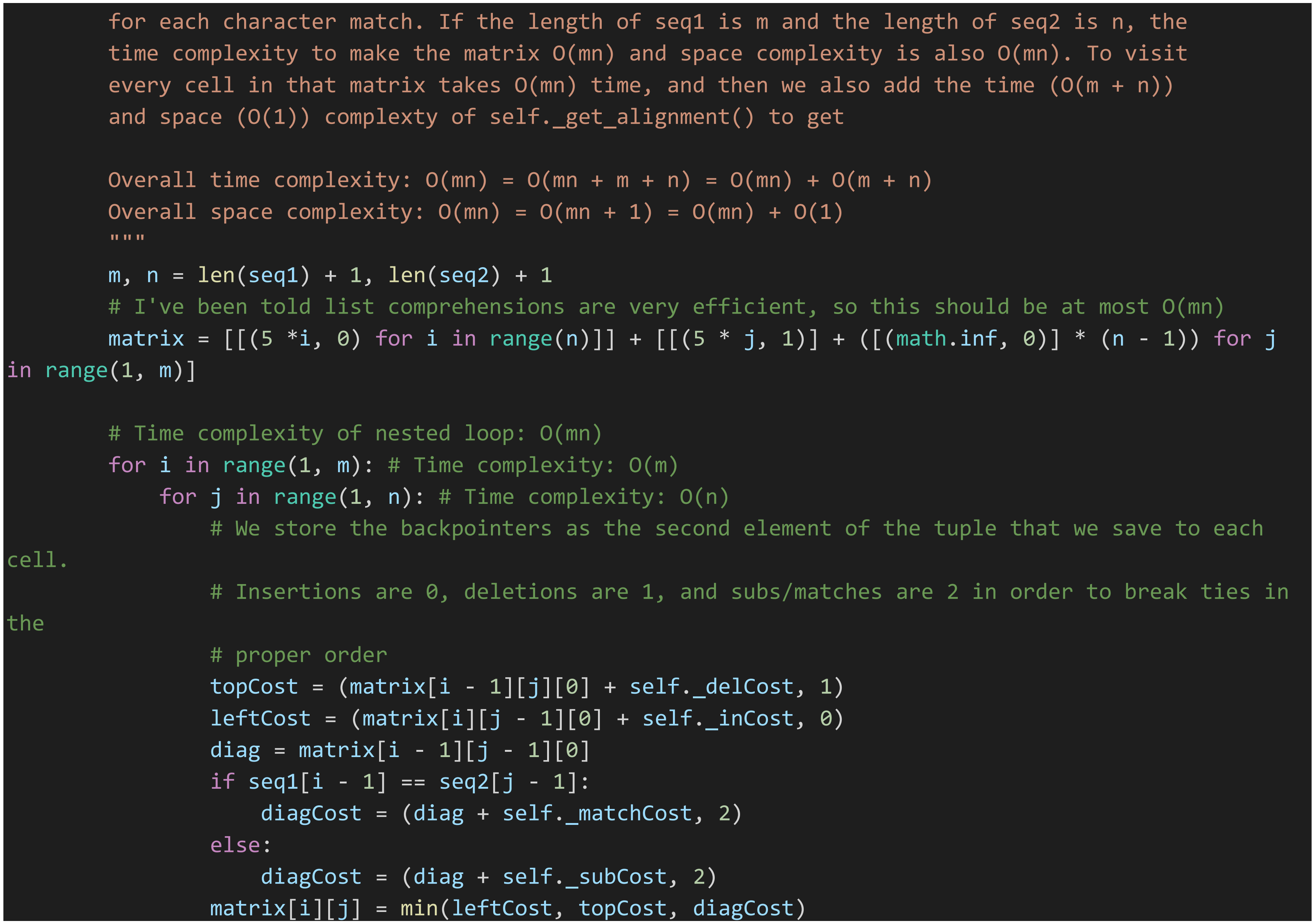
Sequence 10:
```
-ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgt
```

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

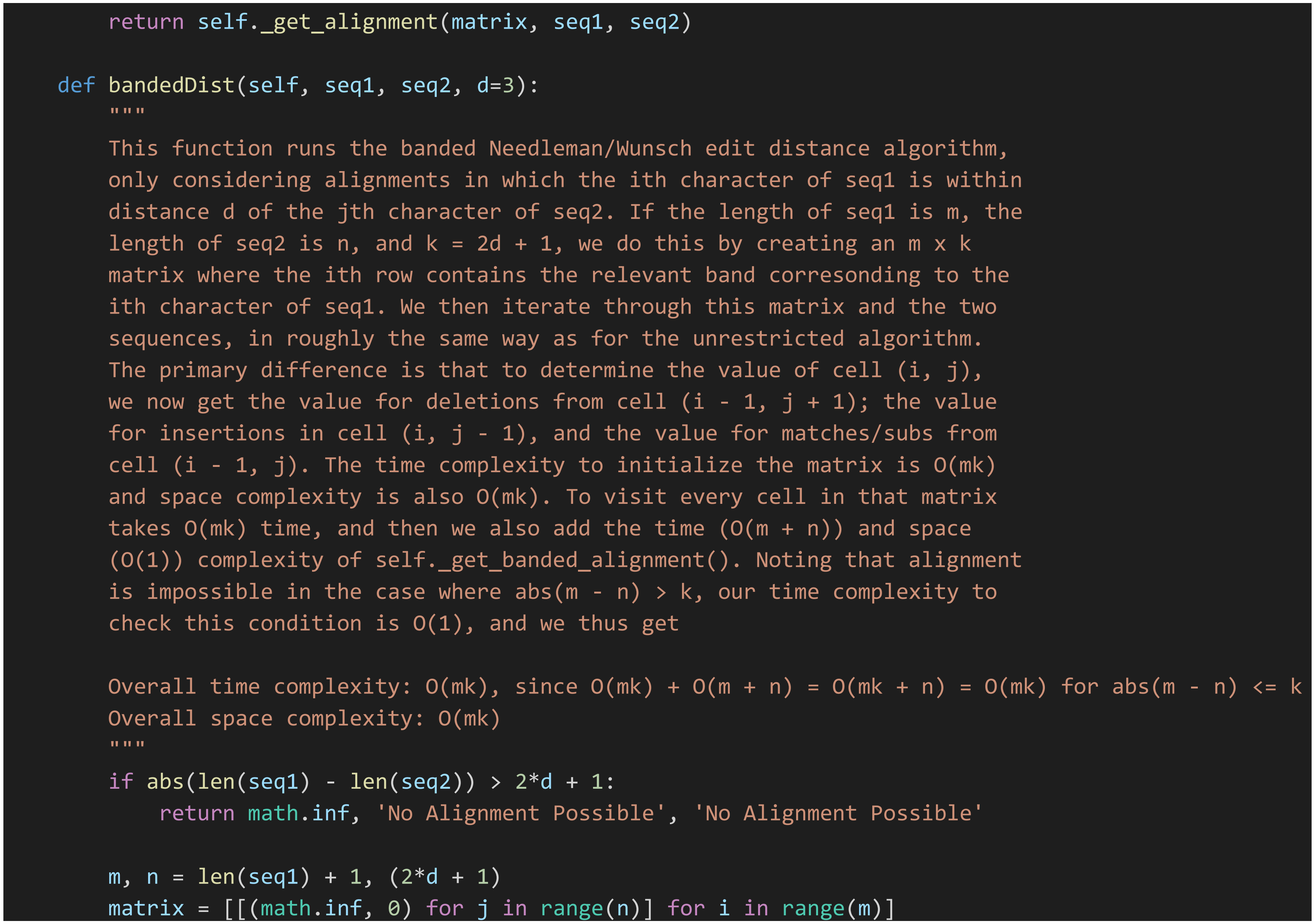
Banded ✓ Align Length: 3000

Appendix

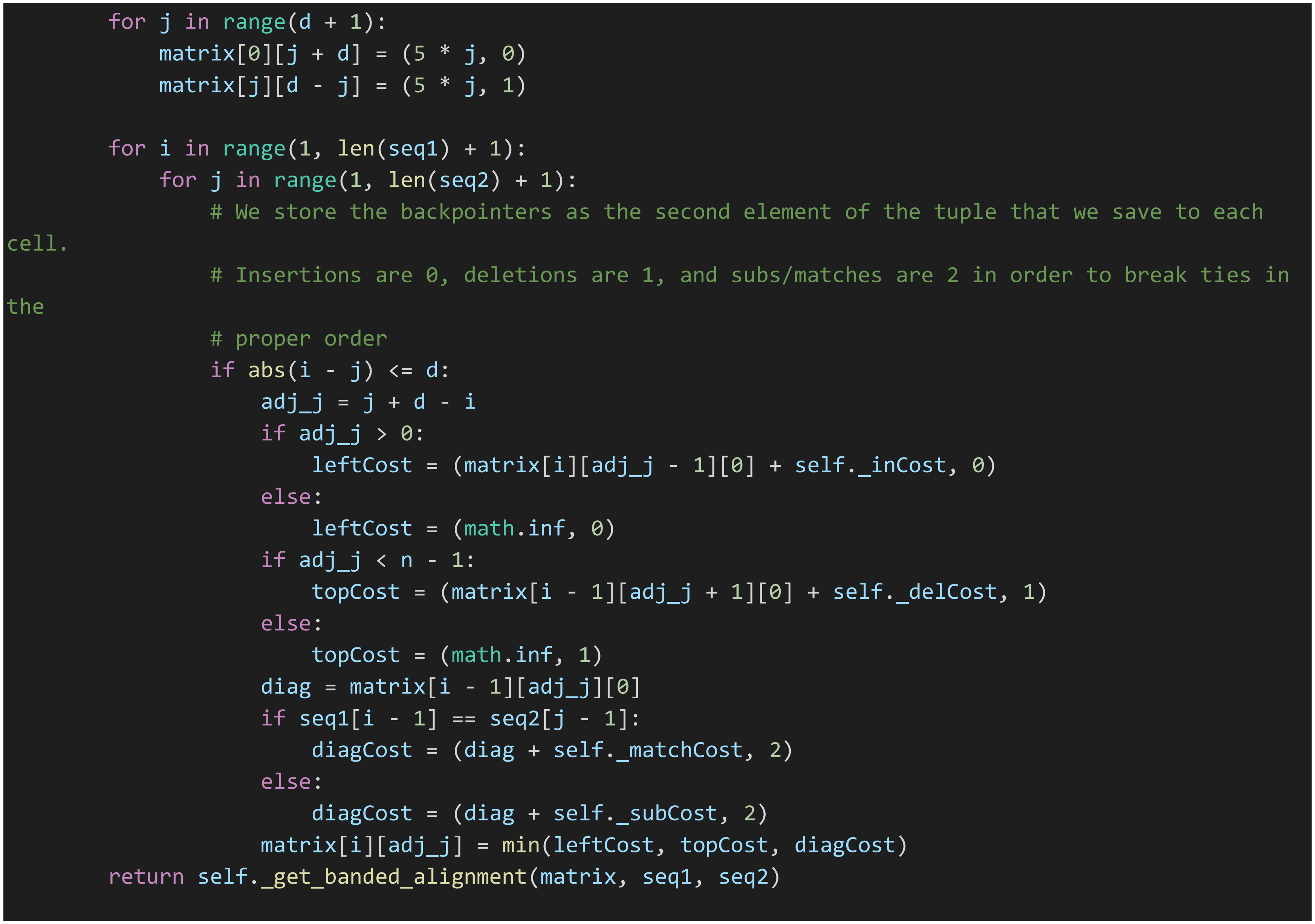
```python
    def __init__( self ):
        self._subCost = 1
        self._matchCost = -3
        self._delCost = 5
        self._inCost = 5
        pass

    def align( self, seq1, seq2, banded, align_length):
        self.banded = banded
        self.MaxCharactersToAlign = align_length

        # only align the sequences up to the specified length
        seq1 = seq1[:align_length]
        seq2 = seq2[:align_length]

        if self.banded:
            score, alignment1, alignment2 = self.bandedDist(seq1, seq2, d=3)
        else:
            score, alignment1, alignment2 = self.editDist(seq1, seq2)

        # We always return the first 100 characters of each alignment, for visualization
        return {'align_cost':score, 'seqi_first100':alignment1[:100],
'seqj_first100':alignment2[:100]}

    def editDist(self, seq1, seq2):
        """
        This function runs the normal, unrestricted Needleman/Wunsch edit distance algorithm,
        using penalty values of 1 for each substitution, 5 for each insertion/deletion, and -3
```



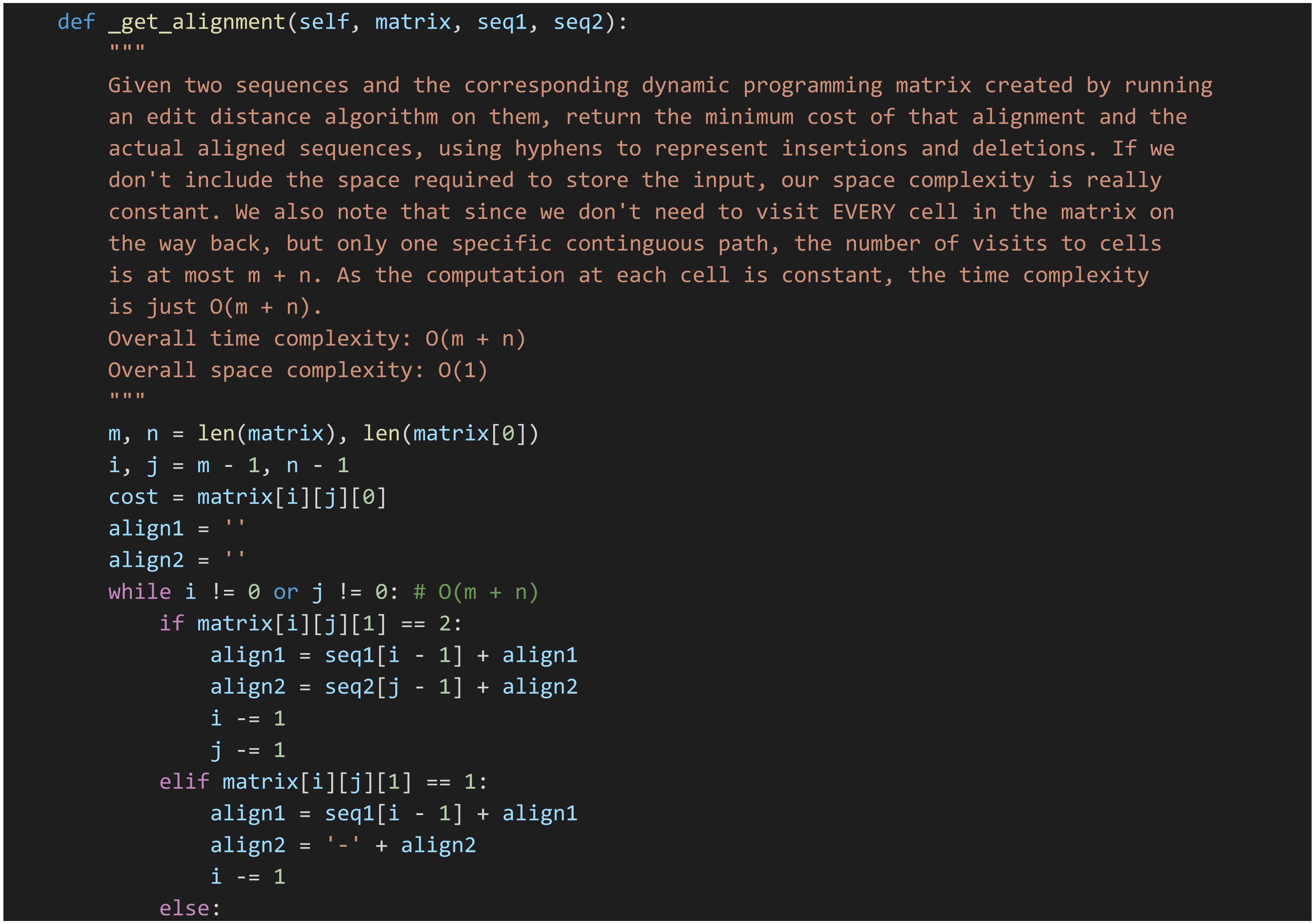
```
        for each character match. If the length of seq1 is m and the length of seq2 is n, the
        time complexity to make the matrix O(mn) and space complexity is also O(mn). To visit
        every cell in that matrix takes O(mn) time, and then we also add the time (O(m + n))
        and space (O(1)) complexty of self._get_alignment() to get

        Overall time complexity: O(mn) = O(mn + m + n) = O(mn) + O(m + n)
        Overall space complexity: O(mn) = O(mn + 1) = O(mn) + O(1)
        """
        m, n = len(seq1) + 1, len(seq2) + 1
        # I've been told list comprehensions are very efficient, so this should be at most O(mn)
        matrix = [[(5 *i, 0) for i in range(n)]] + [[(5 * j, 1)] + ([(math.inf, 0)] * (n - 1)) for j
in range(1, m)]

        # Time complexity of nested loop: O(mn)
        for i in range(1, m): # Time complexity: O(m)
            for j in range(1, n): # Time complexity: O(n)
                # We store the backpointers as the second element of the tuple that we save to each
cell.
                # Insertions are 0, deletions are 1, and subs/matches are 2 in order to break ties in
the
                # proper order
                topCost = (matrix[i - 1][j][0] + self._delCost, 1)
                leftCost = (matrix[i][j - 1][0] + self._inCost, 0)
                diag = matrix[i - 1][j - 1][0]
                if seq1[i - 1] == seq2[j - 1]:
                    diagCost = (diag + self._matchCost, 2)
                else:
                    diagCost = (diag + self._subCost, 2)
                matrix[i][j] = min(leftCost, topCost, diagCost)
```



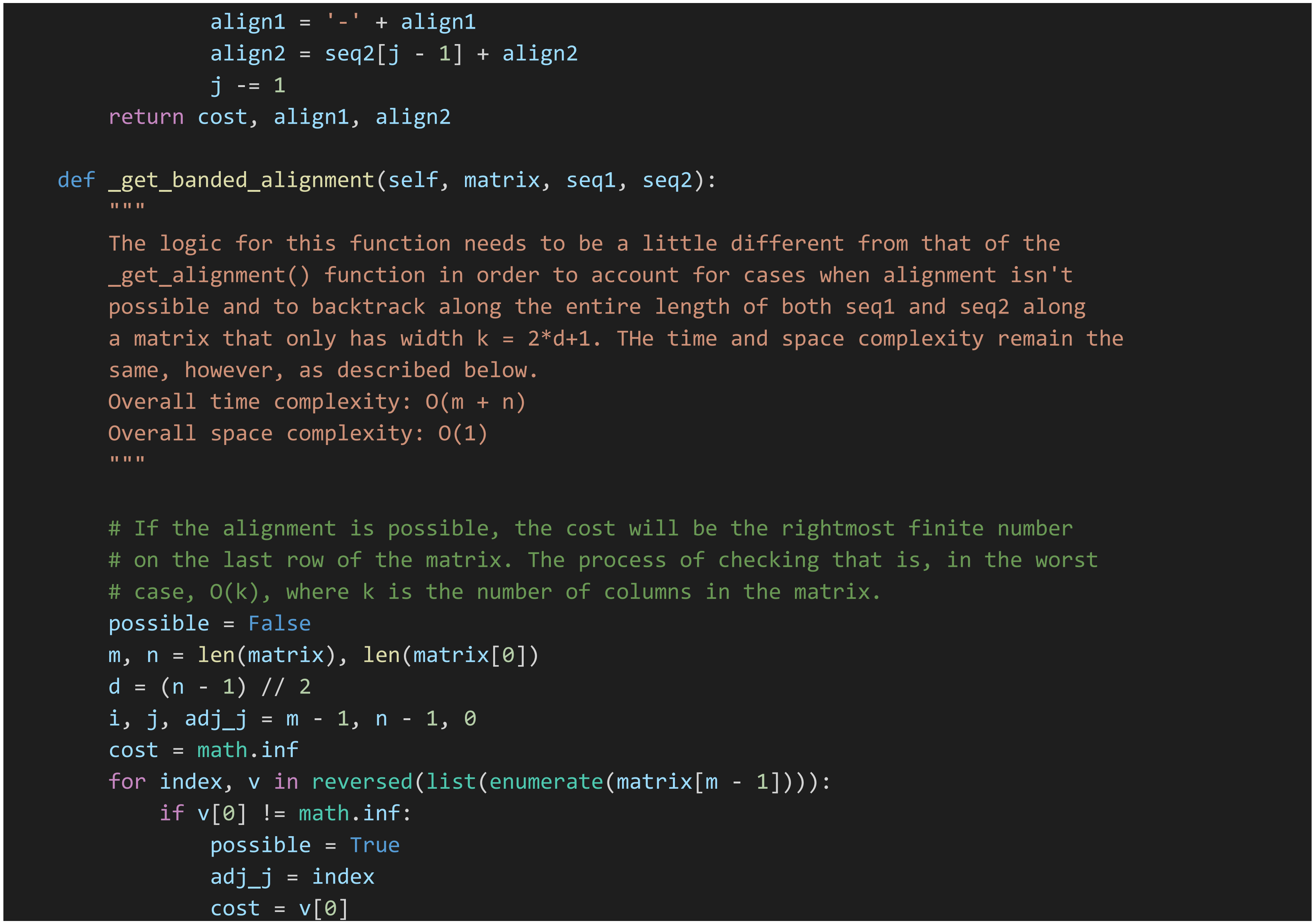
```python
        return self._get_alignment(matrix, seq1, seq2)

    def bandedDist(self, seq1, seq2, d=3):
        """
        This function runs the banded Needleman/Wunsch edit distance algorithm,
        only considering alignments in which the ith character of seq1 is within
        distance d of the jth character of seq2. If the length of seq1 is m, the
        length of seq2 is n, and k = 2d + 1, we do this by creating an m x k
        matrix where the ith row contains the relevant band corresonding to the
        ith character of seq1. We then iterate through this matrix and the two
        sequences, in roughly the same way as for the unrestricted algorithm.
        The primary difference is that to determine the value of cell (i, j),
        we now get the value for deletions from cell (i - 1, j + 1); the value
        for insertions in cell (i, j - 1), and the value for matches/subs from
        cell (i - 1, j). The time complexity to initialize the matrix is O(mk)
        and space complexity is also O(mk). To visit every cell in that matrix
        takes O(mk) time, and then we also add the time (O(m + n)) and space
        (O(1)) complexity of self._get_banded_alignment(). Noting that alignment
        is impossible in the case where abs(m - n) > k, our time complexity to
        check this condition is O(1), and we thus get

        Overall time complexity: O(mk), since O(mk) + O(m + n) = O(mk + n) = O(mk) for abs(m - n) <= k
        Overall space complexity: O(mk)
        """
        if abs(len(seq1) - len(seq2)) > 2*d + 1:
            return math.inf, 'No Alignment Possible', 'No Alignment Possible'

        m, n = len(seq1) + 1, (2*d + 1)
        matrix = [[(math.inf, 0) for j in range(n)] for i in range(m)]
```



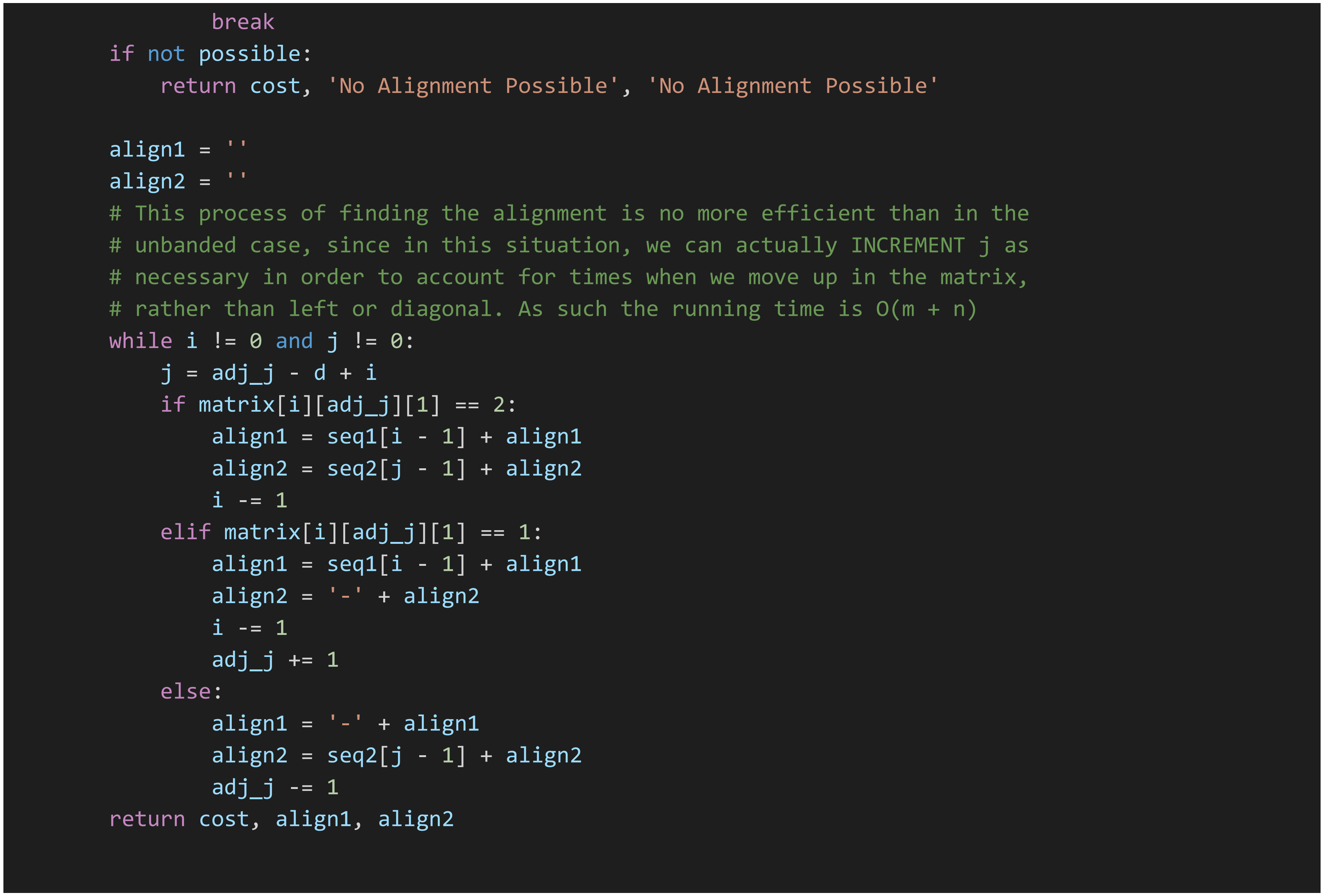
```python
        for j in range(d + 1):
            matrix[0][j + d] = (5 * j, 0)
            matrix[j][d - j] = (5 * j, 1)

        for i in range(1, len(seq1) + 1):
            for j in range(1, len(seq2) + 1):
                # We store the backpointers as the second element of the tuple that we save to each
cell.
                # Insertions are 0, deletions are 1, and subs/matches are 2 in order to break ties in
the
                # proper order
                if abs(i - j) <= d:
                    adj_j = j + d - i
                    if adj_j > 0:
                        leftCost = (matrix[i][adj_j - 1][0] + self._inCost, 0)
                    else:
                        leftCost = (math.inf, 0)
                    if adj_j < n - 1:
                        topCost = (matrix[i - 1][adj_j + 1][0] + self._delCost, 1)
                    else:
                        topCost = (math.inf, 1)
                    diag = matrix[i - 1][adj_j][0]
                    if seq1[i - 1] == seq2[j - 1]:
                        diagCost = (diag + self._matchCost, 2)
                    else:
                        diagCost = (diag + self._subCost, 2)
                    matrix[i][adj_j] = min(leftCost, topCost, diagCost)
        return self._get_banded_alignment(matrix, seq1, seq2)
```



```python
    def _get_alignment(self, matrix, seq1, seq2):
        """
        Given two sequences and the corresponding dynamic programming matrix created by running
        an edit distance algorithm on them, return the minimum cost of that alignment and the
        actual aligned sequences, using hyphens to represent insertions and deletions. If we
        don't include the space required to store the input, our space complexity is really
        constant. We also note that since we don't need to visit EVERY cell in the matrix on
        the way back, but only one specific continguous path, the number of visits to cells
        is at most m + n. As the computation at each cell is constant, the time complexity
        is just O(m + n).
        Overall time complexity: O(m + n)
        Overall space complexity: O(1)
        """
        m, n = len(matrix), len(matrix[0])
        i, j = m - 1, n - 1
        cost = matrix[i][j][0]
        align1 = ''
        align2 = ''
        while i != 0 or j != 0: # O(m + n)
            if matrix[i][j][1] == 2:
                align1 = seq1[i - 1] + align1
                align2 = seq2[j - 1] + align2
                i -= 1
                j -= 1
            elif matrix[i][j][1] == 1:
                align1 = seq1[i - 1] + align1
                align2 = '-' + align2
                i -= 1
            else:
```



```python
                align1 = '-' + align1
                align2 = seq2[j - 1] + align2
                j -= 1
        return cost, align1, align2

    def _get_banded_alignment(self, matrix, seq1, seq2):
        """
        The logic for this function needs to be a little different from that of the
        _get_alignment() function in order to account for cases when alignment isn't
        possible and to backtrack along the entire length of both seq1 and seq2 along
        a matrix that only has width k = 2*d+1. THe time and space complexity remain the
        same, however, as described below.
        Overall time complexity: O(m + n)
        Overall space complexity: O(1)
        """

        # If the alignment is possible, the cost will be the rightmost finite number
        # on the last row of the matrix. The process of checking that is, in the worst
        # case, O(k), where k is the number of columns in the matrix.
        possible = False
        m, n = len(matrix), len(matrix[0])
        d = (n - 1) // 2
        i, j, adj_j = m - 1, n - 1, 0
        cost = math.inf
        for index, v in reversed(list(enumerate(matrix[m - 1]))):
            if v[0] != math.inf:
                possible = True
                adj_j = index
                cost = v[0]
```



```python
                break
        if not possible:
            return cost, 'No Alignment Possible', 'No Alignment Possible'

        align1 = ''
        align2 = ''
        # This process of finding the alignment is no more efficient than in the
        # unbanded case, since in this situation, we can actually INCREMENT j as
        # necessary in order to account for times when we move up in the matrix,
        # rather than left or diagonal. As such the running time is O(m + n)
        while i != 0 and j != 0:
            j = adj_j - d + i
            if matrix[i][adj_j][1] == 2:
                align1 = seq1[i - 1] + align1
                align2 = seq2[j - 1] + align2
                i -= 1
            elif matrix[i][adj_j][1] == 1:
                align1 = seq1[i - 1] + align1
                align2 = '-' + align2
                i -= 1
                adj_j += 1
            else:
                align1 = '-' + align1
                align2 = seq2[j - 1] + align2
                adj_j -= 1
        return cost, align1, align2
```