Part 1:

Before either banded/unrestricted are called, there is a cost of slicing each string to make it the size of the align length, but since that is a constant, it can be dropped from the overall time and space complexity.

For the unrestricted algorithm, the space complexity is simply O(nm) where n is the length of one string and m is the length of the other string. Since each cell of the table stores a tuple with an integer value and a short string dictating which direction that score came from, it does not affect the asymptotic space complexity of the algorithm.

There are two for loops in the unrestricted algorithm, the outer one being O(n) where n is the length of the string or the align length, and the inner loop being O(m) where m is the length of the other string or align length, whichever is shorter.

The backtrace time complexity for unrestricted is O(n), since it simply constructs the alignments from the directions stored in the table. This would make the overall time complexity O(nm + n), but asymptotically just O(nm)

The above adds another O(n + m) to the space complexity, but again does not contribute asymptotically.

For the banded algorithm, the space complexity is O(kn), where k is the bandwidth (in this case 7), and n is the length of the shorter string. The backtrace adds another O(n + m) to the space complexity, but again does not contribute asymptotically.

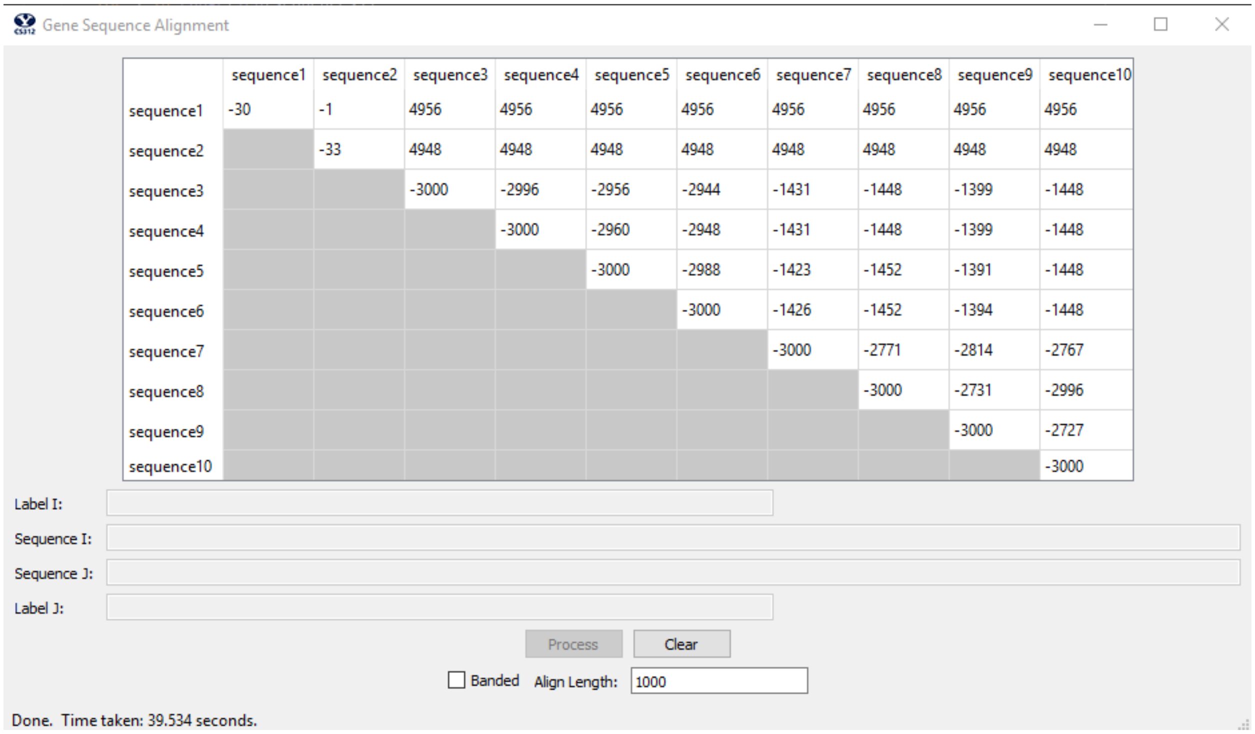
The first for loop in the banded algorithm is O(n) time complexity, where n is the minimum of the shorter string’s length or the align length. The inner for loop contributes an effectively constant time, in this case O(7) which is asymptotically O(1), since the bandwidth is 7 and the inner loop only goes as long as the bandwidth.

The backtrace for unbanded then is also O(n), as it traces the path to the start from the end and constructs the alignments while doing so. It is O(n) because it is the length of the string (or the align length).

Part 2:

My algorithm uses the Needleman-Wunsch cost function, and goes through the shorter string character by character, comparing each one to every character of the longer (or horizontally-oriented) string. It formed a DAG because no higher row was dependent on a lower row, and no value was dependent on the value to its right. I stored the value and direction the value came from in tuples in each cell of the table, so that the backtrace could follow the directions in reverse to construct the alignment and arrive at the beginning of the string.

Part 3:



	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:

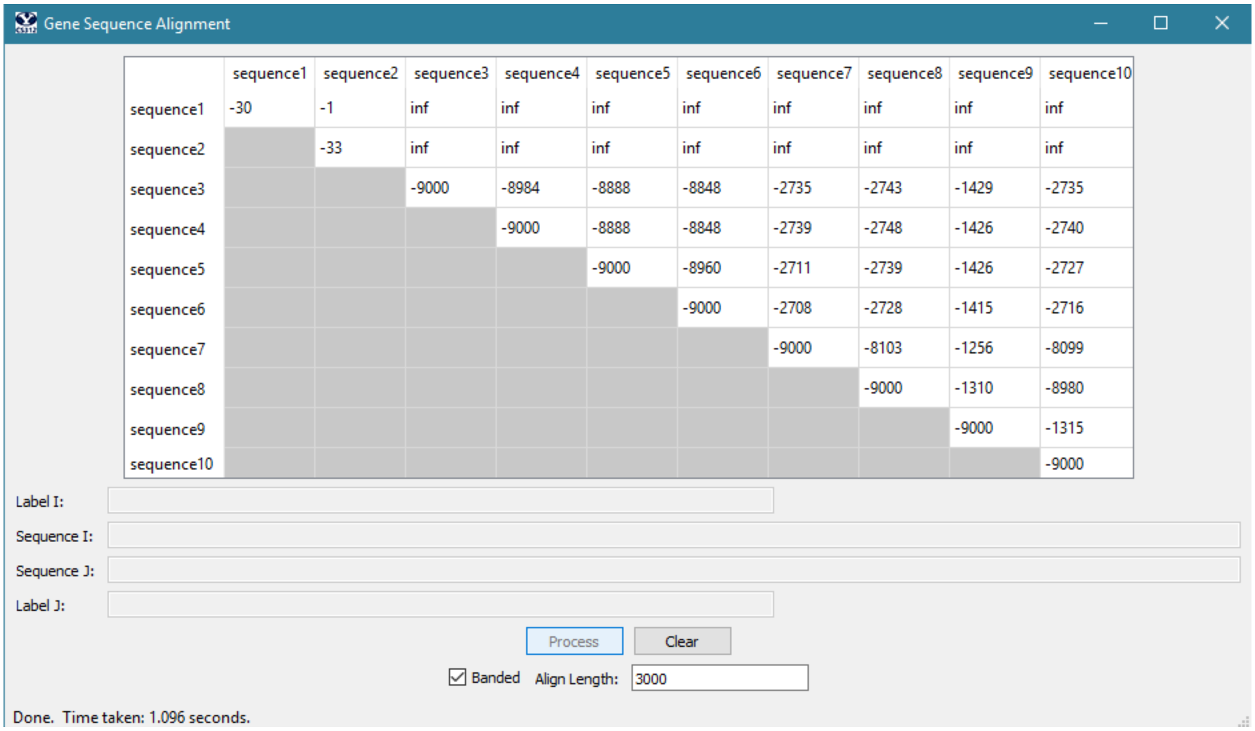
Sequence I:

Sequence J:

Label J:

☐ Banded Align Length: 1000

Done. Time taken: 39.534 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

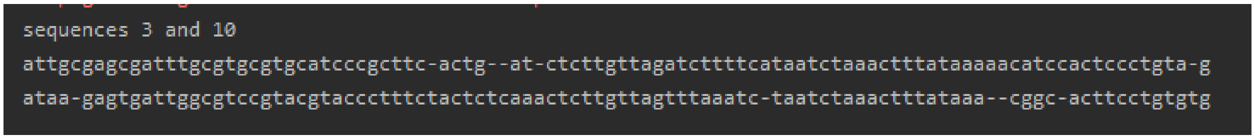
Done. Time taken: 1.096 seconds.

Part 4:

Unrestricted:

attgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-g

ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgtg

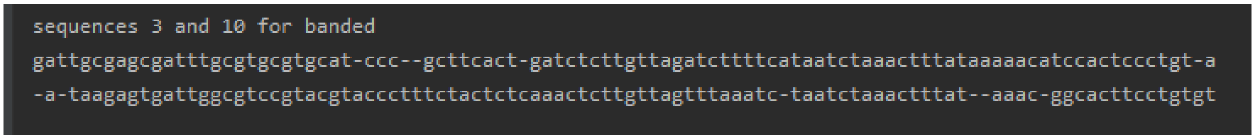


```
sequences 3 and 10
attgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-g
ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgtg
```

Banded:

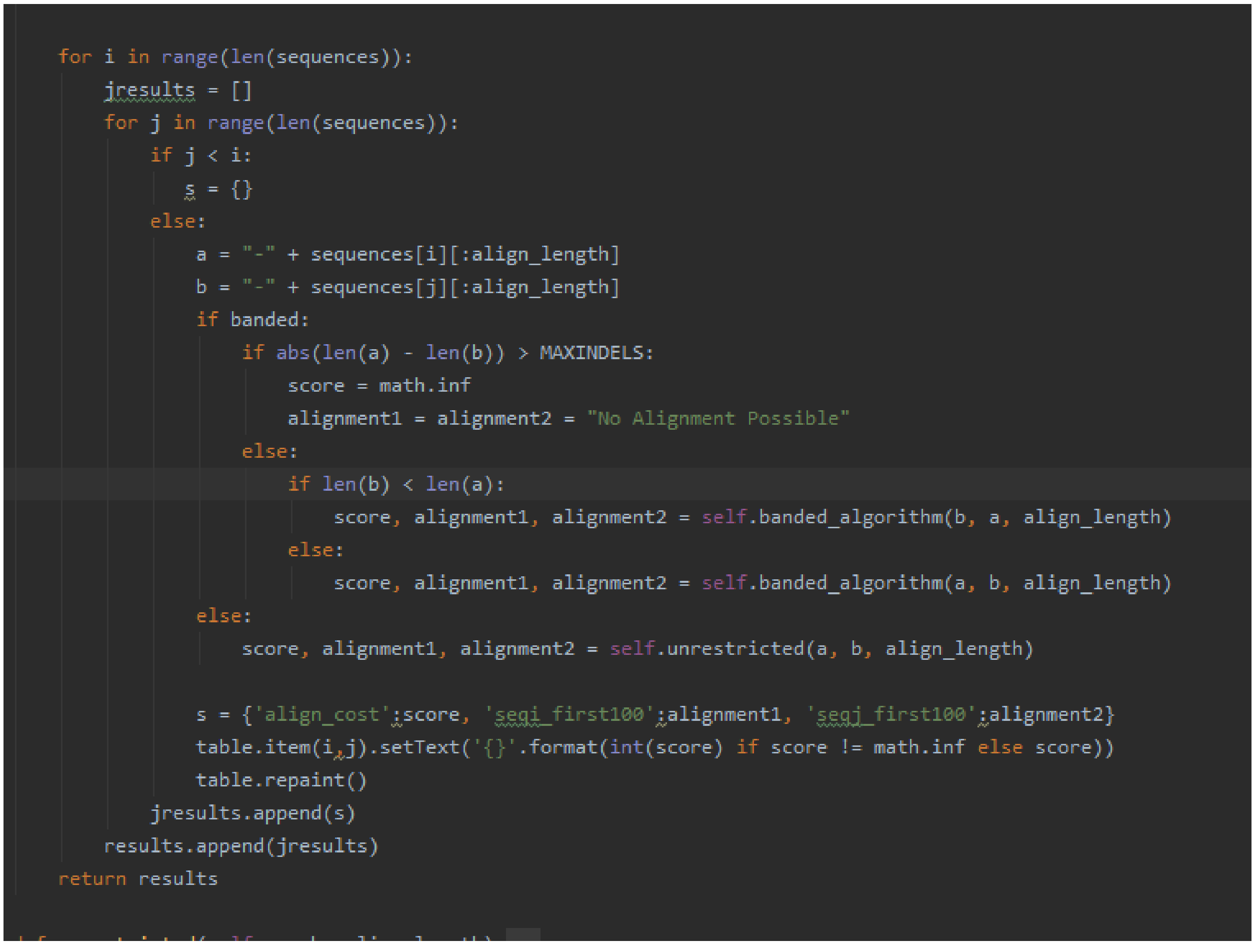
gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a

-a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt

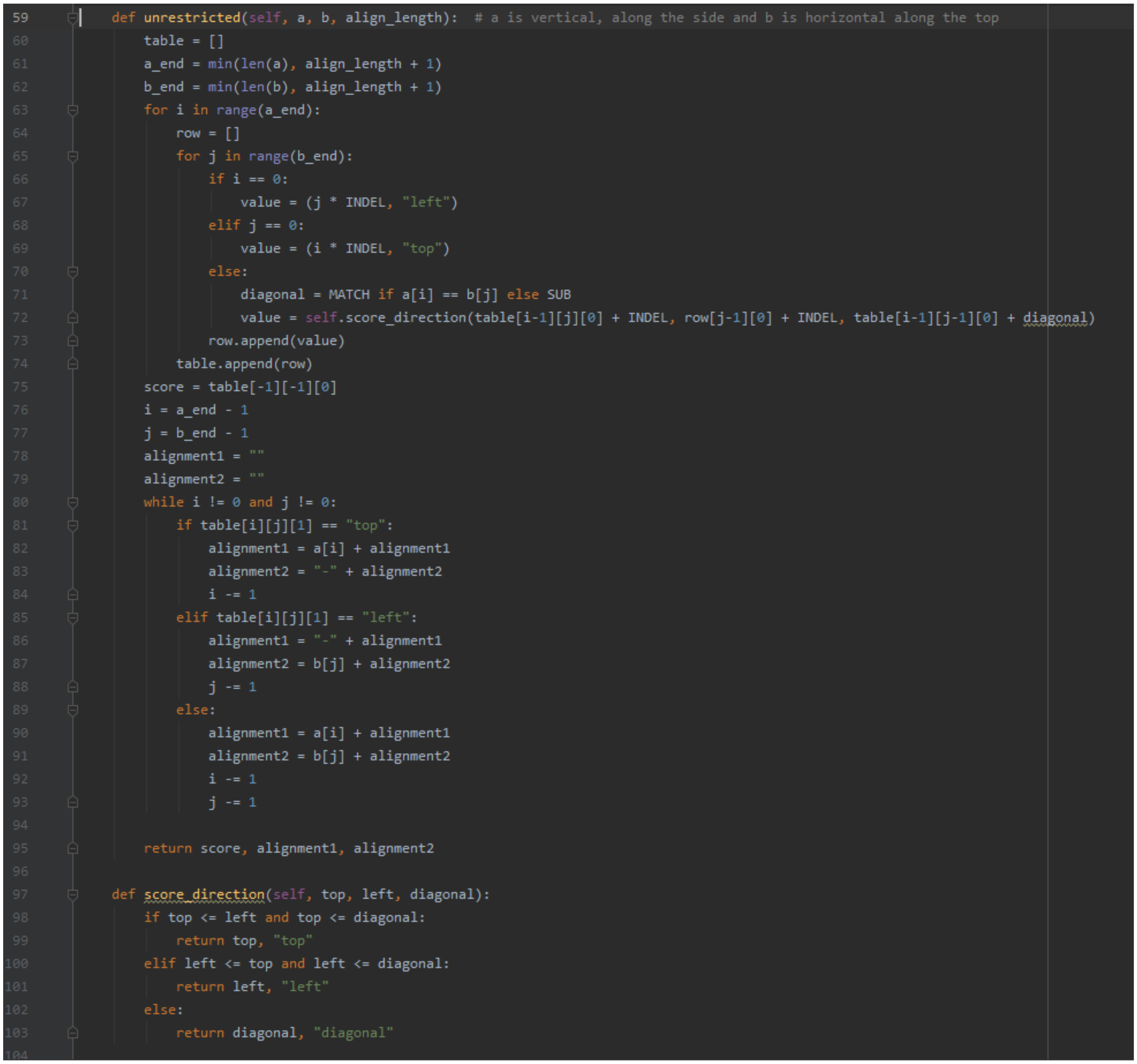


```
sequences 3 and 10 for banded
gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a
-a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt
```

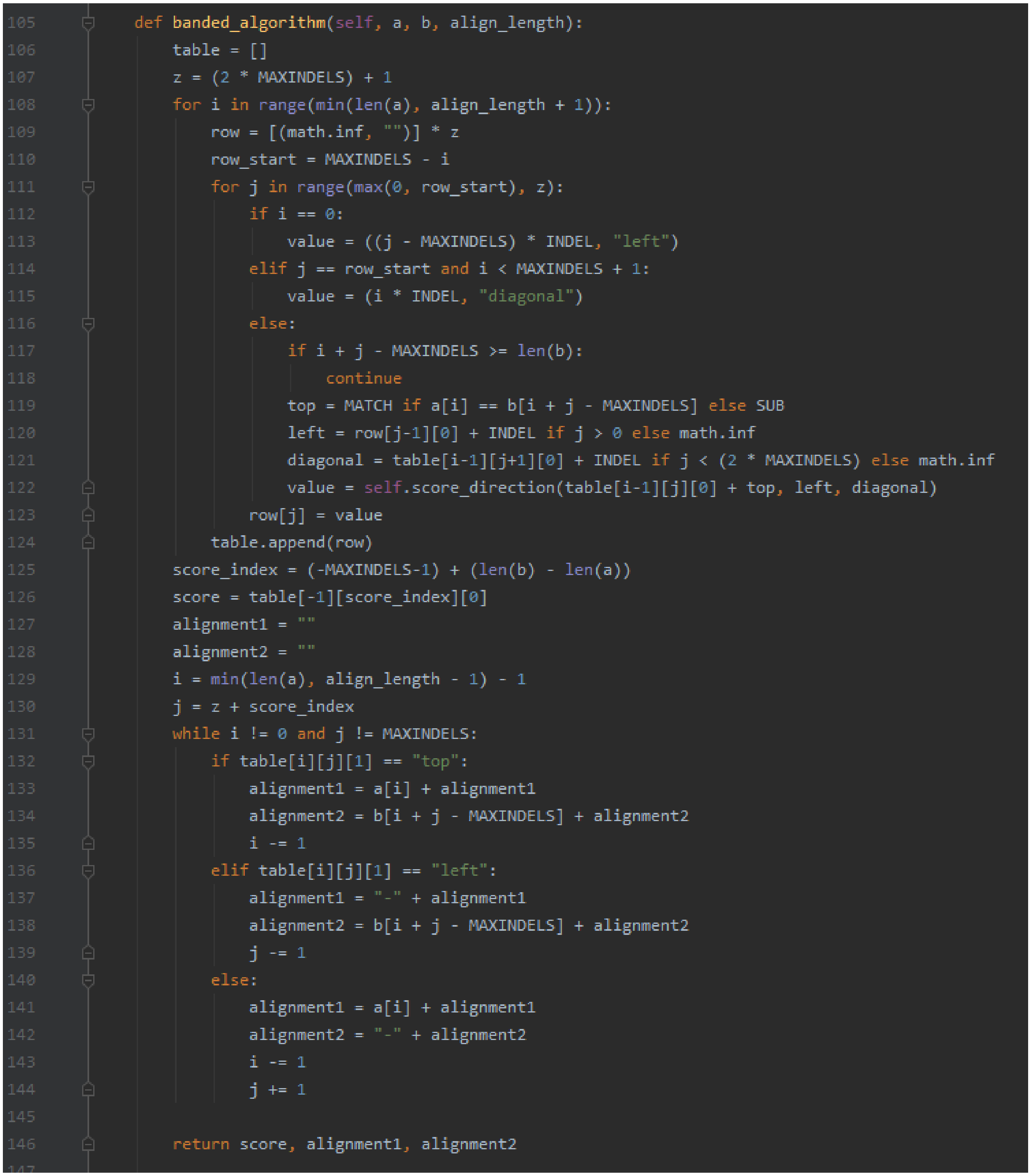
Part 5:



```
    for i in range(len(sequences)):
        jresults = []
        for j in range(len(sequences)):
            if j < i:
                s = {}
            else:
                a = "-" + sequences[i][:align_length]
                b = "-" + sequences[j][:align_length]
                if banded:
                    if abs(len(a) - len(b)) > MAXINDELS:
                        score = math.inf
                        alignment1 = alignment2 = "No Alignment Possible"
                    else:
                        if len(b) < len(a):
                            score, alignment1, alignment2 = self.banded_algorithm(b, a, align_length)
                        else:
                            score, alignment1, alignment2 = self.banded_algorithm(a, b, align_length)
                else:
                    score, alignment1, alignment2 = self.unrestricted(a, b, align_length)

                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
```



```python
    def unrestricted(self, a, b, align_length):  # a is vertical, along the side and b is horizontal along the top
        table = []
        a_end = min(len(a), align_length + 1)
        b_end = min(len(b), align_length + 1)
        for i in range(a_end):
            row = []
            for j in range(b_end):
                if i == 0:
                    value = (j * INDEL, "left")
                elif j == 0:
                    value = (i * INDEL, "top")
                else:
                    diagonal = MATCH if a[i] == b[j] else SUB
                    value = self.score_direction(table[i-1][j][0] + INDEL, row[j-1][0] + INDEL, table[i-1][j-1][0] + diagonal)
                row.append(value)
            table.append(row)
        score = table[-1][-1][0]
        i = a_end - 1
        j = b_end - 1
        alignment1 = ""
        alignment2 = ""
        while i != 0 and j != 0:
            if table[i][j][1] == "top":
                alignment1 = a[i] + alignment1
                alignment2 = "-" + alignment2
                i -= 1
            elif table[i][j][1] == "left":
                alignment1 = "-" + alignment1
                alignment2 = b[j] + alignment2
                j -= 1
            else:
                alignment1 = a[i] + alignment1
                alignment2 = b[j] + alignment2
                i -= 1
                j -= 1

        return score, alignment1, alignment2

    def score_direction(self, top, left, diagonal):
        if top <= left and top <= diagonal:
            return top, "top"
        elif left <= top and left <= diagonal:
            return left, "left"
        else:
            return diagonal, "diagonal"
```



```python
    def banded_algorithm(self, a, b, align_length):
        table = []
        z = (2 * MAXINDELS) + 1
        for i in range(min(len(a), align_length + 1)):
            row = [(math.inf, "")] * z
            row_start = MAXINDELS - i
            for j in range(max(0, row_start), z):
                if i == 0:
                    value = ((j - MAXINDELS) * INDEL, "left")
                elif j == row_start and i < MAXINDELS + 1:
                    value = (i * INDEL, "diagonal")
                else:
                    if i + j - MAXINDELS >= len(b):
                        continue
                    top = MATCH if a[i] == b[i + j - MAXINDELS] else SUB
                    left = row[j-1][0] + INDEL if j > 0 else math.inf
                    diagonal = table[i-1][j+1][0] + INDEL if j < (2 * MAXINDELS) else math.inf
                    value = self.score_direction(table[i-1][j][0] + top, left, diagonal)
                row[j] = value
            table.append(row)
        score_index = (-MAXINDELS-1) + (len(b) - len(a))
        score = table[-1][score_index][0]
        alignment1 = ""
        alignment2 = ""
        i = min(len(a), align_length - 1) - 1
        j = z + score_index
        while i != 0 and j != MAXINDELS:
            if table[i][j][1] == "top":
                alignment1 = a[i] + alignment1
                alignment2 = b[i + j - MAXINDELS] + alignment2
                i -= 1
            elif table[i][j][1] == "left":
                alignment1 = "-" + alignment1
                alignment2 = b[i + j - MAXINDELS] + alignment2
                j -= 1
            else:
                alignment1 = a[i] + alignment1
                alignment2 = "-" + alignment2
                i -= 1
                j += 1

        return score, alignment1, alignment2
```