**Gene Sequencing (Dynamic Programming) Report**

1. Explain the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your pseudo-code.

a. Your analysis should show that your unrestricted algorithm is at most **O(nm)** time and space.

In the code below there are a lot of look-ups and assignments which have a simple cost of O(1). The complexity of the unrestricted algorithm lies on specific parts of the code. We have two matrices of size nxm. Where n is the length of string 1 and m is the length of string 2. This makes the time complexity of the algorithm to be O(nm). One matrix is used for the calculated cost and the other one is used for the back pointers. In order to calculate the cost and fill out the matrices we must visit every single cell in it. We do that by iterating through the rows, “for i in range(1, rows): # O(n),” and visiting every cell in each row, “for j in range(1, cols): # O(m).” The second biggest cost in the complexity comes when we use the back pointers to form the alignment sequences. We do not visit every single cell of this matrix we only visit the cells that are needed to form the string sequences. In this case the cost of doing this is of O(n + m). Using the max rule we can conclude that the total cost of the unrestricted algorithm is of O(nm) since O(n + m) is smaller than O(nm). That is, O(nm) = O(nm) + O(n + m). The space complexity of the algorithm is the cost of the matrix, which is O(nm). Nowhere else in the code below we have a cost of space bigger than the matrix.

"def unbandedAlignment(seq1, seq2):

        matrix = [[]] # nxm matrix of cost

        rows, cols = (len(seq1) + 1, len(seq2) + 1)

        pointers = [[]] # nxm matrix of pointers

        diagonal = 0

        left = 0

        top = 0

        options = [] #left, top, diagonal

# O(n \* m)

        for i in range(1, rows): # O(n)

            for j in range(1, cols): # O(m)

                if(seq1[i-1] == seq2[j-1]): # look ups are constant time

                    diagonal = matrix[i-1][j-1] – 3 # assignment is constant time

                else:

                    diagonal = matrix[i-1][j-1] + 1

                left = matrix[i][j-1] + 5

                top = matrix[i-1][j] + 5

                options.append(left) # appends are constant time

                options.append(top)

                options.append(diagonal)

                min\_v = min(options) # O(3) = O(1)

                matrix[i][j] = min\_v

                index = options.index(min\_v)#O(3) = O(1)

                if(index == 0): #left

                    pointers[i][j] = (i,j-1)

                if(index == 1): #top

                    pointers[i][j] = (i-1,j)

                if(index == 2): #diagonal

                    pointers[i][j] = (i-1,j-1)

                options = []

        score = matrix[rows-1][cols-1] #goal

        origin = (0,0)

        tuple = pointers[rows - 1][cols - 1]

        dist = (rows-1, cols-1)

        align1 = ""

        align2 = ""

        while(origin != dist): # O(n + m)

            i\_index = dist[0] - tuple[0]

            j\_index = dist[1] - tuple[1]

# diagonal

            if(i\_index == 1 and j\_index == 1): # O(1), basic assignments

                align1 = seq1[dist[0] - 1] + align1

                align2 = seq2[dist[1] - 1] + align2

            # top

            if(i\_index == 1 and j\_index == 0): # O(1), basic assignments

                align1 = seq1[dist[0] - 1] + align1

                align2 = "-" + align2

            # left

            if(i\_index == 0 and j\_index == 1): # O(1), basic assignments

                align1 = "-" + align1

                align2 = seq2[dist[1] - 1] + align2

            dist = tuple

            tuple = pointers[dist[0]][dist[1]]

        return score, align1, align2

b. Your analysis should show that your banded algorithm is at most **O(kn)** time and space.

Similarly to the previous algorithm, the banded algorithm contains a lot of look-ups and assignments which again are of complexity O(1). This algorithm also uses two matrices, but in this case the matrices are of n\*bandwith, where n is the length of the sequence 1 and bandwith is a constant value of 7. This makes the time complexity of the banded algorithm to be O(kn) or (7n). One matrix is used for the calculated cost and the other one is used for the back pointers. In order to calculate the cost and fill out the matrices we must visit every single cell in it. We do that by iterating through the rows, “for i in range(1, rows): # O(n),” and visiting every cell in each row, which will be at most 7 cells in each row. “for j in range(start, cols, 1).” Moreover, just like in the unrestricted algorithm, the second biggest cost in the complexity comes when we use the back pointers to form the alignment sequences. We do not visit every single cell of this back-pointers matrix, we only visit the cells that are needed to form the string sequences. In this case the cost of doing this is of O(n + 7). Using the max rule we can conclude that the total cost of the unrestricted algorithm is of O(7n) since O(n + 7) is smaller than O(n7). That is, O(7n) = O(7n) + O(n + 7). The space complexity of the algorithm is the cost of the matrix, O(7n) or O(kn). Nowhere else in the code below we have a cost of space bigger than the matrix.

    def bandedAlignment(seq1, seq2):

        bandWith = 7

        if((len(seq2) - len(seq1)) > 3):

            return ∞

        else :

            rows = len(seq1) + 1 # n + 1

            cols = bandWith # 7

            table = [[]] # nx7 matrix

            pointers = [[]] # nx7 matrix of back-pointers

            start = 4

            table\_min\_val = ∞

            t\_min\_val\_location = (0,0)

            diagonal = ∞

            left = ∞

            top = ∞

            options = [] # left, top, diagonal

            for i in range(1, rows): # O(7n) = O(n - 1) \* O(7)

                if(start > 0):

                    start -= 1

                for j in range(start, cols, 1): # O(7) = O(1)

                    index\_of\_seq2 = (i + j) - 4

                    if(index\_of\_seq2 >= len(seq2)):

                        break

                    if(seq1[i-1] == seq2[index\_of\_seq2]):

                        diagonal = table[i-1][j] - 3

                    else:

                        diagonal = table[i-1][j] + 1

                    if(j != cols - 1): # right limit

                        top = table[i-1][j+1] + 5

                    if(j != 0): # left limit

                        left = table[i][j-1] + 5

                    options.append(left)

                    options.append(top)

                    options.append(diagonal)

                    min\_val = min(options) # O(3) = O(1)

                    table[i][j] = min\_val

                    # tracking back pointer

                    index = options.index(min\_val) # O(3) = O(1)

                    if(index == 0): #left

                        pointers[i][j] = (i,j-1)

                    if(index == 1): #top

                        pointers[i][j] = (i-1,j+1)

                    if(index == 2): #diagonal

                        pointers[i][j] = (i-1,j)

                    options = []

                    diagonal = ∞

                    left = ∞

                    top = ∞

            j\_ind = (len(seq2) + 4) – rows # O(1)

            table\_min\_val = table[rows-1][j\_ind] # O(1)

            i = rows - 1

            j = j\_ind

            score = table\_min\_val

            origin = (0,0)

            tuple = pointers[i][j]

            dist = (i, j)

            align1 = ""

            align2 = ""

            while(origin != dist): # O(n + 7)

                i\_index = dist[0] - tuple[0]

                j\_index = dist[1] - tuple[1]

                if(i\_index == 1 and j\_index == -1): # O(1), basic assignments

                    ind\_char\_seq1 = dist[0] - 1

                    align1 = seq1[ind\_char\_seq1] + align1

                    align2 = "-" + align2

                if(i\_index == 1 and j\_index == 0): # O(1), basic assignments

                    ind\_char\_seq1 = dist[0] - 1

                    ind\_char\_seq2 = (dist[0] + dist[1]) - 4

                    align1 = seq1[ind\_char\_seq1] + align1

                    align2 = seq2[ind\_char\_seq2] + align2

                if(i\_index == 0 and j\_index == 1): # O(1), basic assignments

                    align1 = "-" + align1

                    ind\_char\_seq2 = (dist[0] + dist[1]) - 4

                    align2 = seq2[ind\_char\_seq2] + align2

                dist = tuple

                tuple = pointers[dist[0]][dist[1]]

        return score, align1, align2

2. Write a paragraph that explains how your alignment extraction algorithm works, including the backtrace through previous pointers.

In this algorithm we use matrices to store and calculate the cost of using insertion, deletion, or substitution to align two strings. Dynamic programming makes the work optimal. We are able to calculate the cost in a cell based on the information of previous cells we have already calculated. We divide this big problem into ordered subproblems with a relation that helps calculate the next subproblems. We first build the matrices and fill in the base cases, and then we fill the cells until the desired cell is reached.

For the unrestricted algorithm as well for the restricted algorithm we fill the base case cells for both the cots matrix and the back-pointers matrix. Once we have the base cases in our matrices we can iterate through and calculate the other cells based on these base case cells. Using specific cost for deletion, substitution, and insertion we determine which cell we choose we our algorithm picks the smallest and most optimal value. Once we have calculated this value, we save it in the respective cell. Then we save the location, i and j indices, of the cell we used to calculate our current value in our back-point matrices.

We can see that this makes the alignment of big strings in a reasonable time. For the unrestricted algorithm we can calculate the alignment of a thousand characters in less than 60 seconds (see image below). And with the banded algorithm we can calculate the alignment of three thousand characters in less 30 seconds (see image below). Again, this optimization for both algorithms comes from using dynamic programming. In the case of the restricted algorithm, we can calculate the alignment of big strings in such a short time because we have matrices with a small constant number of columns, 7. This reduces the time cost greatly because we do not need have to iterate through nxm where n is the length of the sequence 1 and m is the length of sequence 2, instead we use equations to do look-ups and find the characters needed from both strings. This improvement in the speed is gained though a loss in accuracy. With unrestricted algorithm we are guaranteed to find the most optimal solution, however this is not the case for the restricted one because we have a limit. However, in most cases the optimal solution will follow within our restriction.

Finally, the way we build our alignment is through the back-pointer matrix. This matrix contains the information of the path from our goal to the origin of our matrices, (0, 0) for the unrestricted and (0, 3) for the banded algorithm. We start with the cell that contains our goal and we retrieve the information in that cell which is the indices of the cell we used to calculate the value of that cell in our cost matrix. We continue doing that until we reach our origin cell. While we visit these cell in our back-pointer matrix we determine what character to add to the alignments depending on the cell we used to get the cost (diagonal, top, or left).

3. Include a “results” section showing both a screenshot of your 10x10 score matrix for the unrestricted algorithm with align length n = 1000 and a screen-shot of your 10x10 score matrix for the banded algorithm with align length n = 3000.

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

4. Include in the “results” section the extracted alignment for the first 100 characters of sequences #3 and #10 (counting from 1), computed using the unrestricted algorithm with n = 1000. Display the sequences one above the other in such a way that matches, substitutions, and insertions/deletions are clearly discernible as shown above in the To Do section. Also include the extracted alignment for the same pair of sequences when computed using the banded algorithm and n = 3000.

**100 characters of sequences #3 and #10 computed using the unrestricted algorithm with n = 1000.**

Alignment cost: -1448

seq1, seq2

gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-

-ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgt

**100 characters of sequences #3 and #10 computed using the banded algorithm with n = 3000.**

Alignment cost: -2735

seq1, seq2

gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-

-ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgt

5. Include your commented source code for both your unrestricted and banded algorithms as an appendix.

def align(self, seq1, seq2, banded, align\_length):

        self.banded = banded

        self.MaxCharactersToAlign = align\_length

        seq1 = seq1[:align\_length]

        seq2 = seq2[:align\_length]

        if(banded):

            result = self.bandedAlignment(seq1, seq2)

        else:

            result = self.unbandedAlignment(seq1, seq2)

        score = result[0]

        alignment1 = result[1]

        alignment2 = result[2]

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

    def unbandedAlignment(self, seq1, seq2):

        matrix = [[]]

        rows, cols = (len(seq1) + 1, len(seq2) + 1)

        matrix = [[0 for i in range(cols)] for j in range(rows)]

        pointers = [[]]

        pointers = [[(0, 0) for i in range(cols)] for j in range(rows)]

        for i in range(1):

            for j in range(1, cols):

                pointers[i][j] = (i, j-1)

        for j in range(1):

            for i in range(1, rows):

                pointers[i][j] = (i-1, j)

        for i in range(1):

            for j in range(cols):

                matrix[i][j] = j \* c\_insert

        for j in range(1): #O(length j)

            for i in range(rows):

                matrix[i][j] = i \* c\_delete

        diagonal = 0

        left = 0

        top = 0

        options = [] #left, top, diagonal

        for i in range(1, rows):

            for j in range(1, cols):

                if(seq1[i-1] == seq2[j-1]):

                    diagonal = matrix[i-1][j-1] - 3

                else:

                    diagonal = matrix[i-1][j-1] + 1

                left = matrix[i][j-1] + 5

                top = matrix[i-1][j] + 5

                options.append(left)

                options.append(top)

                options.append(diagonal)

                min\_v = min(options)

                matrix[i][j] = min\_v

                index = options.index(min\_v

                if(index == 0): #left

                    pointers[i][j] = (i,j-1)

                if(index == 1): #top

                    pointers[i][j] = (i-1,j)

                if(index == 2): #diagonal

                    pointers[i][j] = (i-1,j-1)

                options = []

        score = matrix[rows-1][cols-1]

        origin = (0,0)

        tuple = pointers[rows - 1][cols - 1]

        dist = (rows-1, cols-1) #goal

        align1 = ""

        align2 = ""

        while(origin != dist):

            i\_index = dist[0] - tuple[0]

            j\_index = dist[1] - tuple[1]

            if(i\_index == 1 and j\_index == 1): # diagonal

                align1 = seq1[dist[0] - 1] + align1

                align2 = seq2[dist[1] - 1] + align2

            if(i\_index == 1 and j\_index == 0): # top

                align1 = seq1[dist[0] - 1] + align1

                align2 = "-" + align2

            if(i\_index == 0 and j\_index == 1): # left

                align1 = "-" + align1

                align2 = seq2[dist[1] - 1] + align2

            dist = tuple

            tuple = pointers[dist[0]][dist[1]]

        align1 = align1[:100]

        align2 = align2[:100]

        return score, align1, align2

    def bandedAlignment(self, seq1, seq2):

        bandWith = (2 \* 3) + 1 # 2\*d+1

        if((len(seq2) - len(seq1)) > 3):

            return float('inf'), "No Alignment Possible", "No Alignment Possible"

        else :

            rows = len(seq1) + 1

            cols = bandWith

            table = [[float("inf") for i in range(cols)] for j in range(rows)]

            pointers = [[(float("inf"), float("inf")) for i in range(cols)] for j in range(rows)]

            for j in range(3, -1, -1): # O(4)  # /

                table[3-j][j] = (3-j)\*5

                if(j != 0):

                    pointers[j][3-j] = (j-1, (3-j)+1)

            for i in range(1): # O(1) # \_\_

                for j in range(4): # O(4)

                    table[i][j+3] = j \* 5

                    if(j != 0):

                        pointers[i][j+3] = (0, j+2)

            pointers[0][3] = (0,0)

            start = 4

            table\_min\_val = float("inf")

            t\_min\_val\_location = (0,0)

            diagonal = float("inf")

            left = float("inf")

            top = float("inf")

            options = [] # left, top, diagonal

            for i in range(1, rows): # O(n-1) = O(n)

                if(start > 0):

                    start -= 1

                for j in range(start, cols, 1): # O(7) = O(1)

                    index\_of\_seq2 = (i + j) - 4

                    if(index\_of\_seq2 >= len(seq2)):

                        break

                    if(seq1[i-1] == seq2[index\_of\_seq2]):

                        diagonal = table[i-1][j] - 3

                    else:

                        diagonal = table[i-1][j] + 1

                    if(j != cols - 1): # right limit

                        top = table[i-1][j+1] + 5

                    if(j != 0): # left limit

                        left = table[i][j-1] + 5

                    options.append(left)

                    options.append(top)

                    options.append(diagonal)

                    min\_val = min(options) # O(3) = O(1)

                    table[i][j] = min\_val

                    if(min\_val < table\_min\_val): # O(1)

                        table\_min\_val = min\_val # O(1)

                        t\_min\_val\_location = (i, j)

                    # tracking back pointer

                    index = options.index(min\_val) # O(3) = O(1)

                    if(index == 0): #left

                        pointers[i][j] = (i,j-1)

                    if(index == 1): #top

                        pointers[i][j] = (i-1,j+1)

                    if(index == 2): #diagonal

                        pointers[i][j] = (i-1,j)

                    options = []

                    diagonal = float("inf")

                    left = float("inf")

                    top = float("inf")

            j\_ind = (len(seq2) + 4) - rows

            table\_min\_val = table[rows-1][j\_ind]

            i = rows - 1

            j = j\_ind

            score = table\_min\_val

            origin = (0,0)

            tuple = pointers[i][j]

            dist = (i, j) #goal

            align1 = ""

            align2 = ""

            while(origin != dist):

                i\_index = dist[0] - tuple[0]

                j\_index = dist[1] - tuple[1]

                if(i\_index == 1 and j\_index == -1): # / banded = | unbanded

                    ind\_char\_seq1 = dist[0] - 1

                    align1 = seq1[ind\_char\_seq1] + align1

                    align2 = "-" + align2

                if(i\_index == 1 and j\_index == 0): # | banded = \ unbanded

                    ind\_char\_seq1 = dist[0] - 1

                    ind\_char\_seq2 = (dist[0] + dist[1]) - 4 # (i + j) - 4

                    align1 = seq1[ind\_char\_seq1] + align1

                    align2 = seq2[ind\_char\_seq2] + align2

                if(i\_index == 0 and j\_index == 1): # \_\_ banded and unbanded

                    align1 = "-" + align1

                    ind\_char\_seq2 = (dist[0] + dist[1]) - 4 # (i + j) - 4

                    align2 = seq2[ind\_char\_seq2] + align2

                dist = tuple

                tuple = pointers[dist[0]][dist[1]]

            align1 = align1[:100]

            align2 = align2[:100]

        return score, align1, align2