Atkinson, Jenna – Gene Sequencing

GitHub Link: https://github.com/jennanatkinson/cs312-proj4-gene-sequencing

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

(**Note**: These screenshots summarize the major time/space complexity elements of the align function. Inside the for loop on line 151, it is just comparing costs which is O(1) time. You can see the full code at the end of the report)

```
114     # UNBAN_Time: O(mn) UNBAN_Space: O(mn) [seq1=m, seq2=n]
115     # BAN_Time: O(kn) BAN_Space: O(kn) [k=2(MAXINDELS)+1, seq2=n]
116     def align(self, seq1, seq2, banded, align_length):
117       #Cut sequences and check for matching
118       if len(seq1) > align_length:
119         seq1 = seq1[:align_length]
120       if len(seq2) > align_length:
121         seq2 = seq2[:align_length]
122       if seq1 == seq2:
123         return {'align_cost':len(seq1)*MATCH, 'seqi_first100':seq1, 'seqj_first100':seq2}
124
125       self.banded = banded
126       self.numColumns = len(seq1)+1
127       self.numRows = len(seq2)+1
```

```
128
129       # CostDict => Key[tuple((rowIndex, columnIndex))] : [Cost(val:int, prevCellTuple, prevDirection)]
130       # UNBAN_Space: O(mn), BAN_Space: O(kn)
131       self.costDict = dict()
132
133       self.costDict[tuple((0,0))] = Cost(0, None, None)
134
135       # Fill first row, UNBAN_Time: O(n), BAN_Time: O(k+1)
136       numFirstRow = self.checkBanded(min(1+MAXINDELS,self.numColumns), self.numColumns)
137       for i in range(1, numFirstRow):
138         self.costDict[tuple((0,i))] = Cost(i*INDEL, tuple((0, i-1)), Direction.LEFT)
139
140       #Fill first col, UNBAN_Time: O(m), BAN_Time: O(k+1)
141       numFirstCol = self.checkBanded(min(1+MAXINDELS,self.numRows), self.numRows)
142       for i in range(1, numFirstCol):
143         self.costDict[tuple((i,0))] = Cost(i*INDEL, tuple((i-1, 0)), Direction.TOP)
144
145       #Run algorithm for all cells, UNBAN_Time: O((m-1)*(n-1)), BAN_Time: O(k*n)
146       for rowIndex in range(1, self.numRows):
147         colStart = self.checkBanded(max(rowIndex-MAXINDELS,1), 1)
148         colEnd = self.checkBanded(min(rowIndex+MAXINDELS+1,self.numColumns), self.numColumns)
```

```
149
150         # Calculate costs with priority: left > top > diagonal
151         for colIndex in range(colStart, colEnd):
152           minCost = Cost(math.inf, None, None)
```

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

m=sequence1, n=sequence2

For the unbanded algorithm, I initialize the first row and column in my dictionary, which together is O(m+n) time. Then I run a nested for loop for the rest of the matrix, building up my dictionary as I go with the smallest cost. To calculate the smallest cost, I lookup the previous cell in my dictionary, Time:O(1), then compare to the other possibilities, Time:O(1), then add to the dictionary, Time:O(1). Even though the inside of the loop is trivial, we repeat these calculations for every combination of every character in m and n, therefore the **Time Complexity in total is O(nm)**. The dictionary will hold O(nm) values to keep track of the matrix, so the **Space Complexity is also O(nm)**.

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

k=7, n=sequence2

The banded algorithm is similar to the unbanded, however the columns we look at are restricted to be at most k. I only initialize the first few values in the first row and column, for Time:O(k). Then, my nested for loop visits all the rows for Time:O(n) but only compares each row to k columns, moving to the next row once the index is out of range of the band. The time complexity for the double for loop dominates the function, so the total **Time Complexity is O(kn).** The dictionary will also only store the elements in this band, so the **Space Complexity is also O(kn).**

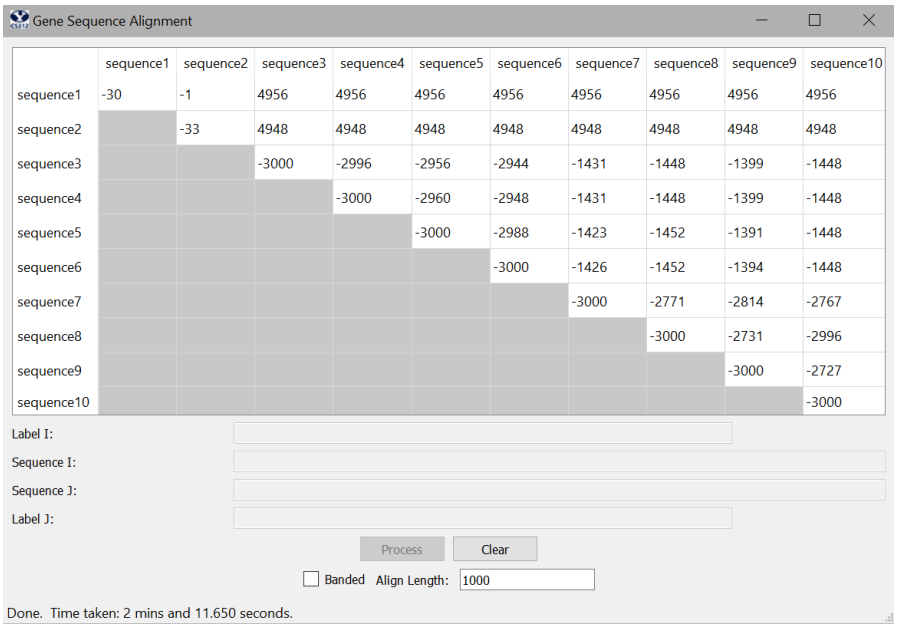
4. **Write a paragraph that explains how your alignment extraction algorithm works, including the backtrace**

```
83    # Traverse through the matrix and build up the alignment strings
84    # Time: O(x), Space: O(1) [x=max(len(seq1), len(seq2))]
85    def getAlignmentStrings(self, seq1, seq2, cell:tuple):
86      alignment1, alignment2 = "", ""
87      while not (cell[0] == 0 and cell[1] == 0):
88        cost = self.costDict.get(cell)
89        if cost.direction == Direction.LEFT:
90          alignment1 += seq1[cell[1]-1]
91          alignment2 += "-"
92        elif cost.direction == Direction.TOP:
93          alignment1 += "-"
94          alignment2 += seq2[cell[0]-1]
95        elif cost.direction == Direction.DIAGONAL:
96          alignment1 += seq1[cell[1]-1]
97          alignment2 += seq2[cell[0]-1]
98        cell = cost.prev
99
100     #Reverse strings and cut to 100, Time: O(x)
101     return alignment1[::-1][:100], alignment2[::-1][:100]
```

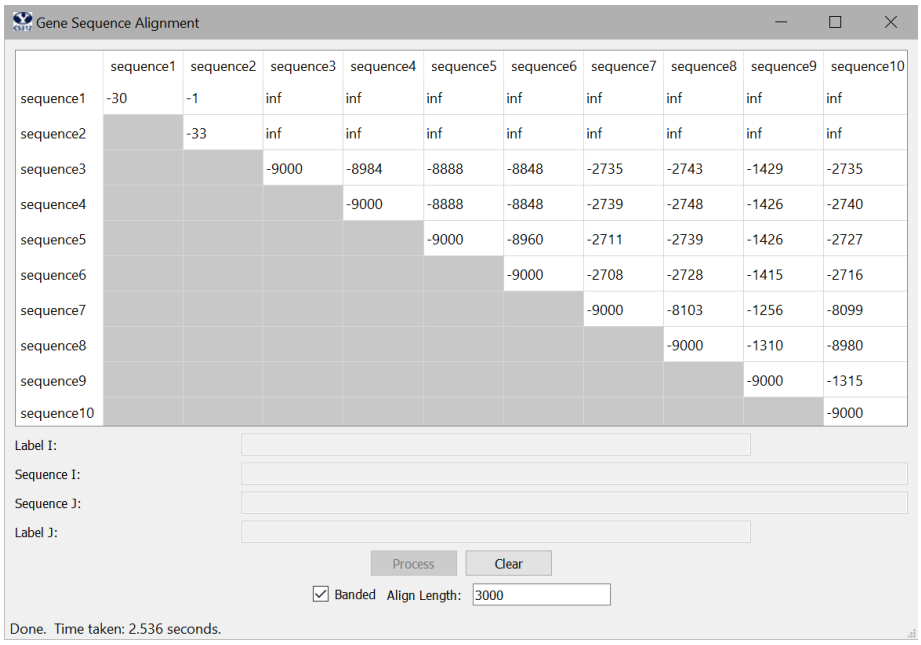
My backtrace method is passed the two sequences and then the final cell. *(Note: if there is no solution, the program will not even call this function.)* I loop until we arrive back at the origin, following the path of each cell's previous coordinates back up the chain. When calculating the cost in my algorithm, I store the direction to make this step easy. I then add strings to both alignments depending on if the direction indicated an INDEL or a MATCH/SUB. This builds the strings backwards, so I reverse them and then only return a slice of the first 100 characters. *Each alignment string will be the same length as max length of*

seq1 and seq2, so we will call that x. Traversing back up to the origin through the matrix will take a Time:O(x), and reversing the string will be the same, for a total **Time Complexity of O(x).**

5. **Include a "results" section showing both a screen-shot 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.**



	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



	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

6. **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 in a side-by-side fashion 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.**

Unbanded (Row=Seq3, Column=Seq10, n=1000)

```
gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-
-ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgt
```

Banded (Row=Seq3 and Column=Seq10, n=3000)

```
gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-
-ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgt
```

7. Attach your commented source code for both your unrestricted and banded algorithms.

```
#!/usr/bin/python3

# from os import SCHED_OTHER
from enum import Enum, auto
from re import X
from which_pyqt import PYQT_VER
if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF
elif PYQT_VER == 'PYQT4':
    from PyQt4.QtCore import QLineF, QPointF
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

import math

# Used to compute the bandwidth for banded version
MAXINDELS = 3
SEQRETURNLEN = 100

# Used to implement Needleman-Wunsch scoring
MATCH = -3
INDEL = 5
SUB = 1

# Indicates the direction the previous cell in the matrix
class Direction(Enum):
    LEFT = auto()
    TOP = auto()
    DIAGONAL = auto()

# Contains the costVal, previous cell which helped calculate that, and the direction the previous cell came from
# Space: O(1)
class Cost:
    def __init__(self, cost:int, prev:tuple, dir:Direction):
        self.costVal = cost
        self.prev = prev
        self.direction = dir

    def __eq__(self, other):
        return (self.costVal, self.prev, self.direction) == (other.costVal, other.prev, self.direction)

class GeneSequencing:
```

```
43.  def __init__( self ):
44.    pass
45.
46.  # Print the cost dictionary
47.  def printDict(self, seq1, seq2):
48.    table_data = [[]]
49.    # Put the first row aka listing the seq1
50.    for i in range(-1, self.numColumns):
51.      if i == -1:
52.        table_data[0].append(" ")
53.      elif i == 0:
54.        table_data[0].append("_")
55.      else:
56.        table_data[0].append(seq1[i-1])
57.
58.    for i in range(0, self.numRows):
59.      table_data.append([])
60.      last = len(table_data)-1
61.      for j in range(-1, self.numColumns):
62.        if j == -1 and i == 0:
63.          table_data[last].append("_")
64.        elif j == -1:
65.          table_data[last].append(seq2[i-1])
66.        else:
67.          string = "" #f"({i},{j}):"
68.          item = self.costDict.get(tuple((i, j)))
69.          if type(item) == Cost:
70.            string += f"{item.costVal}"
71.          else:
72.            string += " "
73.          table_data[last].append(string)
74.
75.    formatString = "{: >5} "
76.    for row in table_data:
77.      for item in row:
78.        print(formatString.format(item), end="")
79.      print()
80.    print()
81.
82.  # Traverse through the matrix and build up the alignment strings
83.  # Time: O(x), Space: O(1) [x=max(len(seq1), len(seq2))]
84.  def getAlignmentStrings(self, seq1, seq2, cell:tuple):
85.    alignment1, alignment2 = "", ""
86.    while not (cell[0] == 0 and cell[1] == 0):
87.      cost = self.costDict.get(cell)
```

```
88.       if cost.direction == Direction.LEFT:
89.         alignment1 += seq1[cell[1]-1]
90.         alignment2 += "-"
91.       elif cost.direction == Direction.TOP:
92.         alignment1 += "-"
93.         alignment2 += seq2[cell[0]-1]
94.       elif cost.direction == Direction.DIAGONAL:
95.         alignment1 += seq1[cell[1]-1]
96.         alignment2 += seq2[cell[0]-1]
97.       cell = cost.prev
98.
99.     #Reverse strings and cut to 100, Time: O(x)
100.       return alignment1[::-1][:100], alignment2[::-1][:100]
101.
102.     # Check the banded setting and return the value, Time: O(1)
103.     def checkBanded(self, trueVal, falseVal):
104.       if self.banded:
105.         return trueVal
106.       else:
107.         return falseVal
108.
109.     # Given two sequences, find the optimal alignment of Insert/Deletion,
    Substitution or Match
110.     #    @required _seq1_(top of matrix) and _seq2_(side of matrix) are two
    sequences to be aligned
111.     #    @required _banded_ is a boolean for computing banded alignment or
    full alignment
112.     #    @required align_length_ = how many base pairs to use in computing
    the alignment
113.     # UNBAN_Time: O(mn) UNBAN_Space: O(mn) [seq1=m, seq2=n]
114.     # BAN_Time: O(kn) BAN_Space: O(kn) [k=2(MAXINDELS)+1, seq2=n]
115.     def align(self, seq1, seq2, banded, align_length):
116.       #Cut sequences and check for matching
117.       if len(seq1) > align_length:
118.         seq1 = seq1[:align_length]
119.       if len(seq2) > align_length:
120.         seq2 = seq2[:align_length]
121.       if seq1 == seq2:
122.         return {'align_cost':len(seq1)*MATCH, 'seqi_first100':seq1,
    'seqj_first100':seq2}
123.
124.       self.banded = banded
125.       self.numColumns = len(seq1)+1
126.       self.numRows = len(seq2)+1
127.
```

```
128.      # CostDict => Key[tuple((rowIndex, columnIndex))] : [Cost(val:int,
   prevCellTuple, prevDirection)]
129.      # UNBAN_Space: O(mn), BAN_Space: O(kn)
130.      self.costDict = dict()
131.
132.      self.costDict[tuple((0,0))] = Cost(0, None, None)
133.
134.      # Fill first row, UNBAN_Time: O(n), BAN_Time: O(k+1)
135.      numFirstRow = self.checkBanded(min(1+MAXINDELS,self.numColumns),
   self.numColumns)
136.      for i in range(1, numFirstRow):
137.        self.costDict[tuple((0,i))] = Cost(i*INDEL, tuple((0, i-1)),
   Direction.LEFT)
138.
139.      #Fill first col, UNBAN_Time: O(m), BAN_Time: O(k+1)
140.      numFirstCol = self.checkBanded(min(1+MAXINDELS,self.numRows),
   self.numRows)
141.      for i in range(1, numFirstCol):
142.        self.costDict[tuple((i,0))] = Cost(i*INDEL, tuple((i-1, 0)),
   Direction.TOP)
143.
144.      #Run algorithm for all cells, UNBAN_Time: O((m-1)*(n-1)), BAN_Time:
   O(k*n)
145.      for rowIndex in range(1, self.numRows):
146.        colStart = self.checkBanded(max(rowIndex-MAXINDELS,1), 1)
147.        colEnd = self.checkBanded(min(rowIndex+MAXINDELS+1,self.numColumns),
   self.numColumns)
148.
149.        # Calculate costs with priority: left > top > diagonal
150.        for colIndex in range(colStart, colEnd):
151.          minCost = Cost(math.inf, None, None)
152.
153.          # Calculate diagonal cost if match/sub
154.          diagonalPrevCell = tuple((rowIndex-1, colIndex-1))
155.          diagonalPrevCost = self.costDict.get(diagonalPrevCell)
156.          if diagonalPrevCost is not None:
157.            diagonalCost = Cost(diagonalPrevCost.costVal, diagonalPrevCell,
   Direction.DIAGONAL)
158.            if (seq1[colIndex-1] == seq2[rowIndex-1]):
159.              diagonalCost.costVal += MATCH
160.            else:
161.              diagonalCost.costVal += SUB
162.            if diagonalCost.costVal < minCost.costVal:
163.              minCost = diagonalCost
164.
```

```
165.          # Calculate top cost if indel
166.          topPrevCell = tuple((rowIndex-1, colIndex))
167.          topPrevCost = self.costDict.get(topPrevCell)
168.          if topPrevCost is not None:
169.            topCost = Cost(topPrevCost.costVal+INDEL, topPrevCell,
    Direction.TOP)
170.            if topCost.costVal <= minCost.costVal:
171.              minCost = topCost
172.
173.          # Calculate left cost if indel
174.          leftPrevCell = tuple((rowIndex, colIndex-1))
175.          leftPrevCost = self.costDict.get(leftPrevCell)
176.          if leftPrevCost is not None:
177.            leftCost = Cost(leftPrevCost.costVal+INDEL, leftPrevCell,
    Direction.LEFT)
178.            if leftCost.costVal <= minCost.costVal:
179.              minCost = leftCost
180.
181.          self.costDict[tuple((rowIndex,colIndex))] = minCost
182.          # self.printDict(seq1, seq2)
183.
184.      cell = tuple((len(seq2), len(seq1)))
185.      cost = self.costDict.get(cell)
186.      score = math.inf
187.      if cost is not None: # This would happen if it is outside of the band
188.        score = cost.costVal
189.      if score != math.inf:
190.        alignment1, alignment2 = self.getAlignmentStrings(seq1, seq2, cell)
191.      else:
192.        alignment1 = alignment2 = "No Alignment Possible"
193.      # print(alignment1)
194.      # print(alignment2)
195.
196.      return {'align_cost':score, 'seqi_first100':alignment1,
    'seqj_first100':alignment2}
197.
```