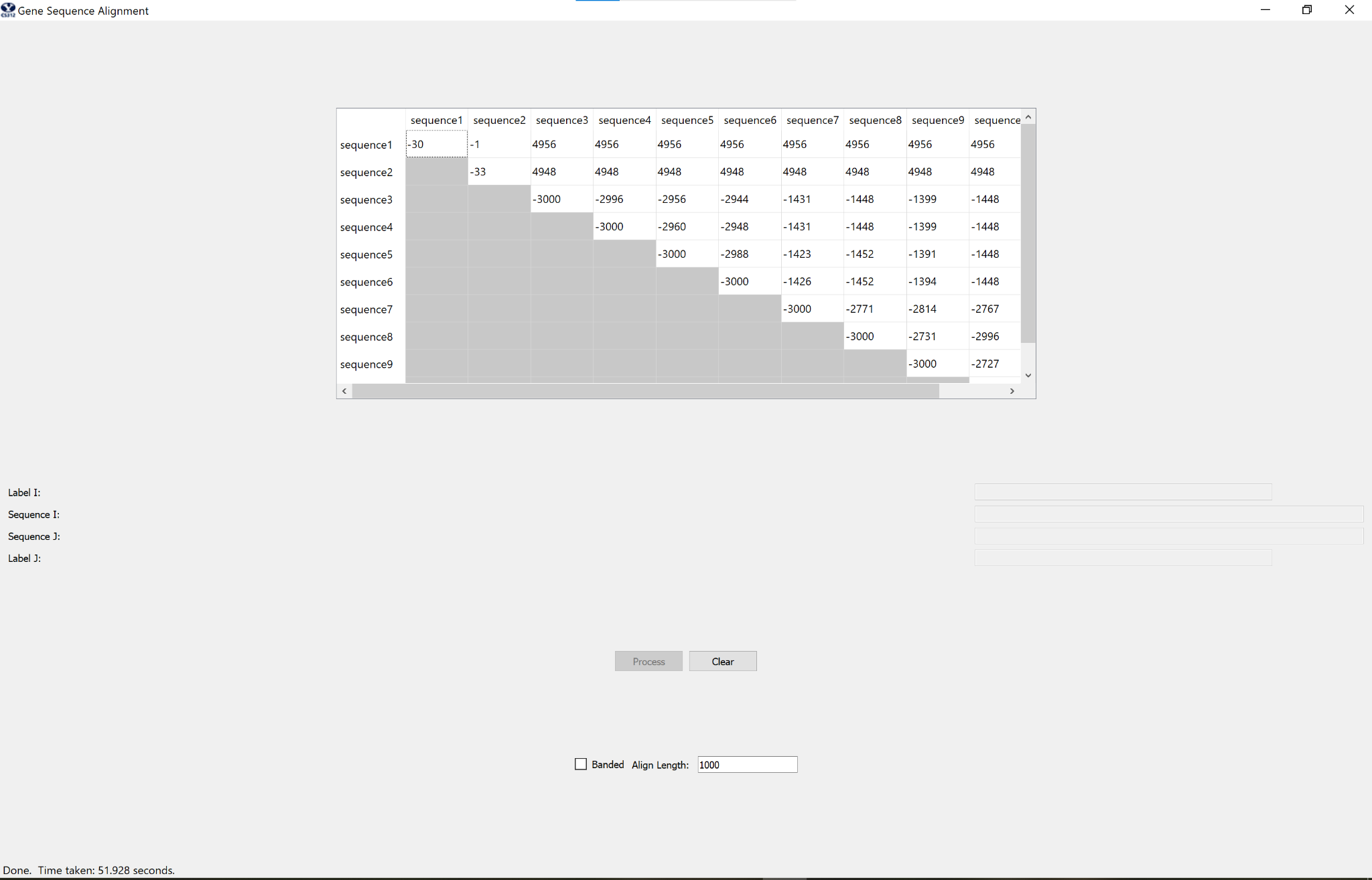
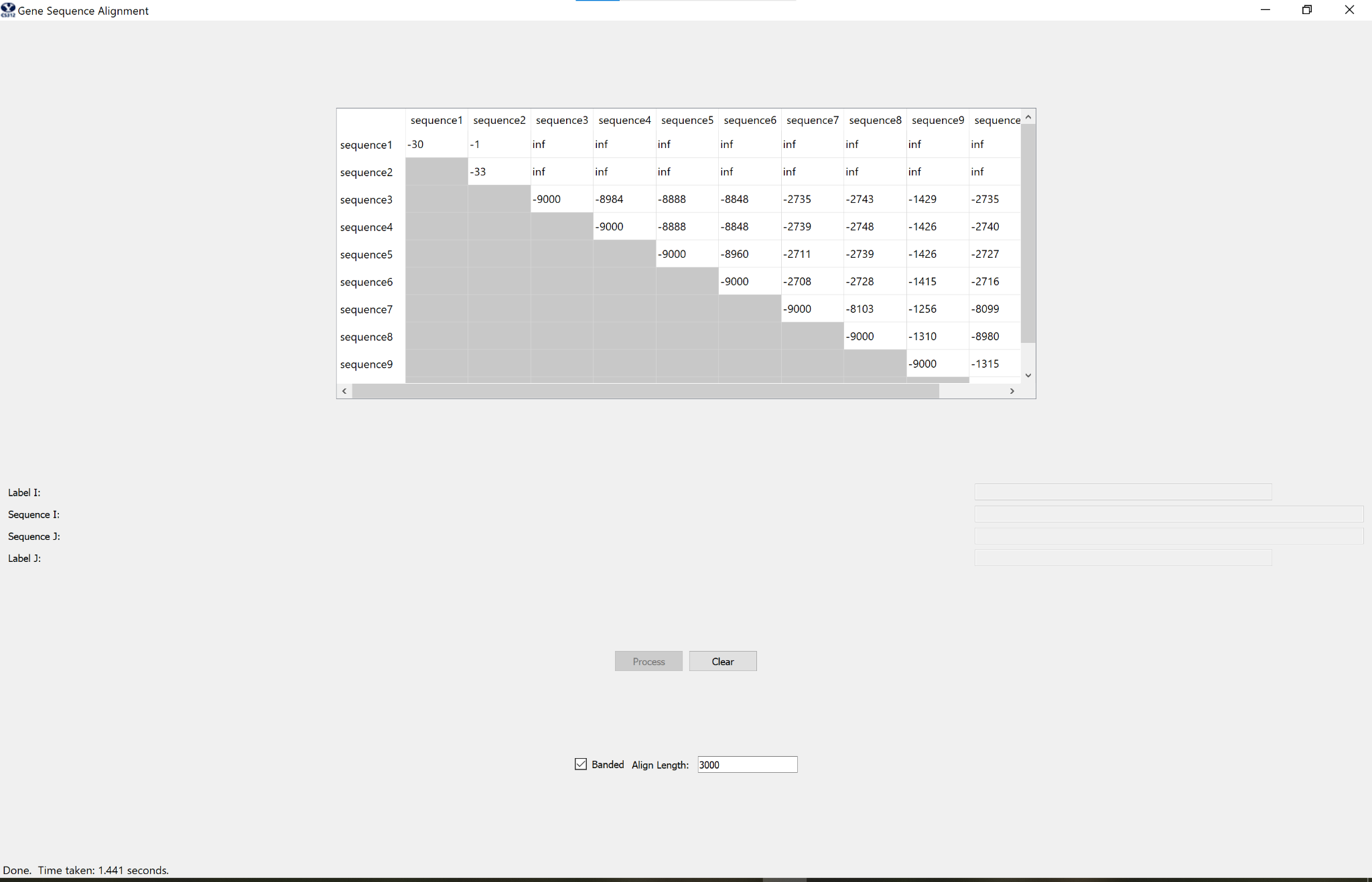
1. **See appendix**

2.  **Discuss the time and space complexity of both your algorithms:**

* **Unrestricted:**
  + **Time complexity:**
    - The time complexity is O(nm). This is because we have to fill up a table that has n rows and m columns, where n is the number of characters in the first sequence, and m is the number of characters in the second sequence. Because we have to fill the table, we end up going over each and every cell, and filling them in. Filling in the cell takes O(1) time, so filling a table of size n\*m with an O(1) operation at each cell, takes O(nm) time. Since filling the table is the largest operation, everything else is dwarfed in comparison of complexity.
  + **Space complexity:**
    - The space complexity is O(nm). This is because we have a table that has n rows and m columns, where n is the number of characters in the first sequence, and m is the number of characters in the second sequence. Any other space we use (for variable storage and other utility functions) are not of great enough significance to overcome an array of size n\*m. Therefore, the largest space we take up is in the table of size n\*m, so the space complexity is O(nm).
* **Banded**
  + **Time complexity:**
    - The time complexity is O(kn). This is because we have to fill up a table that has n rows and k columns, where n is the number of characters in the first sequence, and k is 2x + 1, where x is the maximum number of indels you’d like to consider. Because we have to fill the table, we end up going over each and every cell, and filling them in. Filling in the cell takes O(1) time, so filling a table of size k\*n with an O(1) operation at each cell, takes O(kn) time. Since filling the table is the largest operation, everything else is dwarfed in comparison of complexity.
  + **Space complexity:**
    - The space complexity is O(kn). This is because we have a table that has n rows and k columns, where n is the number of characters in the first sequence, and k is 2x + 1, where x is the maximum number of indels you’d like to consider. Any other space we use (for variable storage and other utility functions) are not of great enough significance to overcome an array of size k\*n. Therefore, the largest space we take up is in the table of size k\*n, so the space complexity is O(kn)
  + **Discuss how you modified your dependency pointers to the adjacent cells with your smaller array:**
    - My pointers for both unrestricted and banded, were both integers 0, 1, and 2, which referred to a LEFT, DIAGONAL, and UP backpointer. Though I didn’t change the integer value of the pointer from unrestricted to banded, I DID change how it was handled.
      * In unrestricted, LEFT referred to the cell that was 1 column left of the current cell. This did not change in banded
      * In unrestricted, DIAGONAL referred to the cell 1 column left, and 1 row up from the current cell. However, in the banded version, I shifted all cells to the left 1 compared to the row above in the unrestricted array. This resulted in DIAGONAL referring to the cell directly 1 row above the current cell in the banded table.
      * In unrestricted, UP referred to the cell directly 1 row above the current cell. However, in the banded version, I shifted all cells to the left 1 compared to the row above in the unrestricted array. This resulted in UP referring to the cell 1 row above, and 1 to the right of the current cell in the banded table.

3. a. 10x10 score matrix for the unrestricted algorithm with align length *n* = 1000



b. 10x10 score matrix for the banded algorithm with align length *n* = 3000

4. a. **The alignment for the first 100 characters of sequences #3 and #10 (counting from 1), computed using the unrestricted algorithm with n = 1000**

* **attgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-g**
* **ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgtg**

b. **The alignment for the same pair of sequences when computed using the banded algorithm and n = 3000**

* **attgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-g**
* **ataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgtg**

**CODE (APPENDIX or whatever)  
#!/usr/bin/python3**

1. **import math**
2. **from which\_pyqt import PYQT\_VER**
3. **if PYQT\_VER == 'PYQT5':**
4. **from PyQt5.QtCore import QLineF, QPointF**
5. **elif PYQT\_VER == 'PYQT4':**
6. **from PyQt4.QtCore import QLineF, QPointF**
7. **elif PYQT\_VER == 'PYQT6':**
8. **from PyQt6.QtCore import QLineF, QPointF**
9. **else:**
10. **raise Exception('Unsupported Version of PyQt: {}'.format(PYQT\_VER))**
11. **import random**
12. **# Used to compute the bandwidth for banded version**
13. **MAXINDELS = 3**
14. **# Used to implement Needleman-Wunsch scoring**
15. **MATCH = -3**
16. **INDEL = 5**
17. **SUB = 1**
18. **LEFT = 0**
19. **DIAGONAL = 1**
20. **UP = 2**
21. **class GeneSequencing:**
22. **def \_\_init\_\_( self ):**
23. **pass**
24. **# This is the method called by the GUI.**
25. **# \_seq1\_ and \_seq2\_ are two sequences to be aligned,**
26. **# \_banded\_ is a boolean that tells you whether you should compute a banded alignment or full alignment,**
27. **# \_align\_length\_ tells you how many base pairs to use in computing the alignment**
28. **def align( self, seq1, seq2, banded, align\_length):**
29. **def BandedDistancesTableSetup():**
30. **newDistances = [[(0, -1) for \_ in range((2 \* MAXINDELS) + 1)] for \_ in range(self.MaxCharactersToAlign)]**
31. **# Set top row values**
32. **for num in range(4):**
33. **j = num + 3**
34. **newDistances[0][j] = (5 \* num, -1)**
35. **# Set diagonal top row values**
36. **for num in range(4):**
37. **j = 3 - num**
38. **newDistances[num][j] = (5 \* num, -1)**
39. **# Set top left empty triangle values**
40. **for i in range(MAXINDELS):**
41. **for j in range(MAXINDELS - i):**
42. **newDistances[i][j] = (-1, -1)**
43. **return newDistances**
44. **def UnrestrictedDistancesTableSetup():**
45. **newDistances = [[(0, -1) for \_ in range(self.MaxCharactersToAlign)] for \_ in range(self.MaxCharactersToAlign)]**
46. **for i in range(self.MaxCharactersToAlign):**
47. **newDistances[i][0] = (i \* 5, -1)**
48. **for j in range(self.MaxCharactersToAlign):**
49. **newDistances[0][j] = (j \* 5, -1)**
50. **return newDistances**
51. **def Banded(seq1, seq2):**
52. **if (len(seq1) < self.MaxCharactersToAlign or len(seq2) < self.MaxCharactersToAlign) and abs(len(seq1) - len(seq2)) > 3:**
53. **return False**
54. **for i in range(1, self.MaxCharactersToAlign):**
55. **isInGoofyRange = i < 4**
56. **if isInGoofyRange:**
57. **j = 4 - i**
58. **else:**
59. **j = 0**
60. **while j < ((2 \* MAXINDELS) + 1):**
61. **if isInGoofyRange:**
62. **seq2Index = j - (4 - i) # Isolates the incremental nature of j**
63. **else:**
64. **seq2Index = j + (i - 4) # Adds the number of rows since the goofy range**
65. **# Use null value if comparison string is too long**
66. **if (i - 1) < len(seq1):**
67. **iChar = seq1[i - 1]**
68. **else:**
69. **iChar = None**
70. **# Use null value if comparison string is too long**
71. **if (seq2Index) < len(seq2):**
72. **jChar = seq2[seq2Index]**
73. **else:**
74. **jChar = None**
75. **isMatch = iChar == jChar**
76. **if isMatch:**
77. **# If align\_length is larger than both words, then grabbing the diagonal does not increase score**
78. **if iChar is None and jChar is None:**
79. **diagonalValue = distances[i - 1][j][0]**
80. **else:**
81. **diagonalValue = -3 + distances[i - 1][j][0]**
82. **else:**
83. **diagonalValue = 1 + distances[i - 1][j][0]**
84. **if (j + 1) < (2 \* MAXINDELS + 1):**
85. **upValue = 5 + distances[i - 1][j + 1][0]**
86. **else:**
87. **upValue = math.inf**
88. **if (j - 1) >= 0:**
89. **leftValue = 5 + distances[i][j - 1][0]**
90. **else:**
91. **leftValue = math.inf**
92. **if leftValue <= upValue and leftValue <= diagonalValue:**
93. **distances[i][j] = (leftValue, LEFT)**
94. **elif upValue <= leftValue and upValue <= diagonalValue:**
95. **distances[i][j] = (upValue, UP)**
96. **else:**
97. **distances[i][j] = (diagonalValue, DIAGONAL)**
98. **j += 1**
99. **return True**
100. **def Unrestricted(seq1, seq2):**
101. **for i in range(1, self.MaxCharactersToAlign):**
102. **for j in range(1, self.MaxCharactersToAlign):**
103. **# Use null value if comparison string is too long**
104. **if (i - 1) < len(seq1):**
105. **iChar = seq1[i - 1]**
106. **else:**
107. **iChar = None**
108. **# Use null value if comparison string is too long**
109. **if (j - 1) < len(seq2):**
110. **jChar = seq2[j - 1]**
111. **else:**
112. **jChar = None**
113. **isMatch = iChar == jChar**
114. **if isMatch:**
115. **# If align\_length is larger than both words, then grabbing the diagonal does not increase score**
116. **if iChar == None and jChar == None:**
117. **diagonalValue = distances[i - 1][j - 1][0]**
118. **else:**
119. **diagonalValue = -3 + distances[i - 1][j - 1][0]**
120. **elif iChar != None and jChar != None:**
121. **diagonalValue = 1 + distances[i - 1][j - 1][0]**
122. **else:**
123. **diagonalValue = math.inf**
124. **upValue = 5 + distances[i - 1][j][0]**
125. **leftValue = 5 + distances[i][j - 1][0]**
126. **if leftValue <= upValue and leftValue <= diagonalValue:**
127. **distances[i][j] = (leftValue, LEFT)**
128. **elif upValue <= leftValue and upValue <= diagonalValue:**
129. **distances[i][j] = (upValue, UP)**
130. **else:**
131. **distances[i][j] = (diagonalValue, DIAGONAL)**
132. **def DetermineUnrestrictedAlignmentStrings(seq1, seq2):**
133. **newSeq1 = ""**
134. **newSeq2 = ""**
135. **if len(seq1) < self.MaxCharactersToAlign:**
136. **i = len(seq1)**
137. **else:**
138. **i = self.MaxCharactersToAlign - 1**
139. **if len(seq2) < self.MaxCharactersToAlign:**
140. **j = len(seq2)**
141. **else:**
142. **j = self.MaxCharactersToAlign - 1**
143. **currentValue = distances[i][j][1]**
145. **while currentValue != -1:**
147. **if currentValue == DIAGONAL:**
148. **newSeq1 = seq1[i - 1] + newSeq1**
149. **newSeq2 = seq2[j - 1] + newSeq2**
150. **i -= 1**
151. **j -= 1**
152. **elif currentValue == UP:**
153. **newSeq1 = seq1[i - 1] + newSeq1**
154. **newSeq2 = "-" + newSeq2**
155. **i -= 1**
156. **else:**
157. **newSeq1 = "-" + newSeq1**
158. **newSeq2 = seq2[j - 1] + newSeq2**
159. **j -= 1**
160. **currentValue = distances[i][j][1]**
161. **return newSeq1, newSeq2**
162. **def DetermineBandedAlignmentStrings(seq1, seq2):**
163. **newSeq1 = ""**
164. **newSeq2 = ""**
165. **if len(seq1) < self.MaxCharactersToAlign:**
166. **i = len(seq1)**
167. **else:**
168. **i = self.MaxCharactersToAlign - 1**
169. **if len(seq2) < self.MaxCharactersToAlign:**
170. **j = MAXINDELS + (len(seq2) - len(seq1))**
171. **seq2Index = len(seq2) - 1**
172. **else:**
173. **j = MAXINDELS**
174. **seq2Index = self.MaxCharactersToAlign - 2**
175. **currentValue = distances[i][j][1]**
176. **while currentValue != -1:**
177. **if currentValue == DIAGONAL:**
178. **newSeq1 = seq1[i - 1] + newSeq1**
179. **newSeq2 = seq2[seq2Index] + newSeq2**
180. **i -= 1**
181. **seq2Index -= 1**
182. **elif currentValue == UP:**
183. **newSeq1 = seq1[i - 1] + newSeq1**
184. **newSeq2 = "-" + newSeq2**
185. **i -= 1**
186. **j += 1**
187. **else:**
188. **newSeq1 = "-" + newSeq1**
189. **newSeq2 = seq2[seq2Index] + newSeq2**
190. **j -= 1**
191. **seq2Index -= 1**
192. **currentValue = distances[i][j][1]**
193. **return newSeq1, newSeq2**
194. **self.banded = banded**
195. **self.MaxCharactersToAlign = align\_length + 1**
196. **if banded:**
197. **distances = BandedDistancesTableSetup()**
198. **isAlignmentPossible = Banded(seq1, seq2)**
199. **if isAlignmentPossible:**
200. **if len(seq1) < self.MaxCharactersToAlign:**
201. **rowNumber = len(seq1) + 1**
202. **else:**
203. **rowNumber = self.MaxCharactersToAlign - 1**
204. **if len(seq2) < self.MaxCharactersToAlign:**
205. **colNumber = MAXINDELS + (len(seq2) - len(seq1))**
206. **else:**
207. **colNumber = MAXINDELS**
208. **score = distances[rowNumber][colNumber][0]**
209. **alignment1, alignment2 = DetermineBandedAlignmentStrings(seq1, seq2)**
210. **else:**
211. **score = math.inf**
212. **alignment1 = alignment2 = "No Alignment Possible"**
213. **else:**
214. **distances = UnrestrictedDistancesTableSetup()**
215. **Unrestricted(seq1, seq2)**
216. **if len(seq1) < self.MaxCharactersToAlign:**
217. **rowNumber = len(seq1)**
218. **else:**
219. **rowNumber = self.MaxCharactersToAlign - 1**
220. **if len(seq2) < self.MaxCharactersToAlign:**
221. **colNumber = len(seq2)**
222. **else:**
223. **colNumber = self.MaxCharactersToAlign - 1**
224. **score = distances[rowNumber][colNumber][0]**
225. **alignment1, alignment2 = DetermineUnrestrictedAlignmentStrings(seq1, seq2)**
226. **alignment1 = alignment1[:100]**
227. **alignment2 = alignment2[:100]**
228. **return {'align\_cost':score, 'seqi\_first100':alignment1, 'seqj\_first100':alignment2}**