

## Part 1 – Source code

Included as Appendix

## Part 2 – Space and time complexity discussion

$O(mn)$  unbanded algorithm

- The time complexity for this algorithm comes in the double for loop, one loop for each of the two strings. If we consider one string  $m$  and one string  $n$ , then the time and space complexity will be  $O(mn)$  because of the double for loop.
- Time
  - The first for loop that initializes the first row takes  $O(n)$  time, since it only loops through the string on the top. The work done here is only additions and inserts into the table, so this doesn't add any additional complexity.
  - The double for loop takes  $O(mn)$  time, where  $m$  is the first string and  $n$  is the second. The work done within the loops is minimum, since we only ever check 3 values (it's bounded, so  $O(1)$  constant time). Grabbing values, comparing them, and inserting into the table all take constant time. We do perform 3 additions or 2 additions and 1 subtraction every time, but for our scope, we consider this constant, as the additions and subtractions will never exceed 5.
  - Rebuilding the string takes  $O(n)$  time where  $n$  is the longer of the two strings. As we build the strings, we perform concatenations on each string for each back-pointer in the prev array. Accessing the prev array is constant time. This will never be worse than  $O(n)$  because the number of the prev array's pointers will never exceed the length of the longer string.
  - While theoretically the time is  $O(mn)$ , in reality, the time is bounded by our alignment length, since we only check characters up to that point. Since we're bounding our lengths at 1,000,  $O(mn)$  is only a true time complexity for strings of length  $\leq 1,000$ . We can consider this constant time of  $O(1,000 \times 1,000)$  or  $O(1,000,000)$  since we are bounding both  $m$  and  $n$  at 1,000.
- Space
  - Space will be  $O(mn)$ . All we have to keep track of in memory is the value array, the prev array, the two sequences, and our rebuild of the two sequences. The strings will never exceed linear space, so  $O(n)$  space for the longer string. In reality, it might be  $O(4n)$  because we have 4 strings. The real time complexity comes with the two tables. We never exceed  $O(mn)$  because we only add 1 value and 1 back-pointer for each of the strings' comparisons. This would make it  $O(2mn)$  because the prev array and the value table are both  $O(mn)$

```

Initialize value table and prev array

for i in range(align_length + 1):
    Initialize first row to values +5 each time

for i in range(align_length + 1):
    Add row to table

for j in range(align_length + 1):

    # comparing values within the table as they get populated
    left = table_2d[i][j - 1] + 5
    top = table_2d[i - 1][j] + 5
    top_left = table_2d[i - 1][j - 1]
    if seq1[j - 1] == seq2[i - 1]:
        top_left -= 3
    else:
        top_left += 1
    Choose minimum of left, top, top_left

```

Rebuild strings from bottom right corner using prev array  
Get score from last cell in the 2d array

O(kn)

- The time complexity for this algorithm also comes in the double for loop, one loop for one string, the other for the band. If we consider the band k and one string n, then the time and space complexity will be O(kn) because of the double for loop.
- Time
  - The work done here is similar to the unbanded algorithm (differences will be detailed below). All of the work comes in the double for loop, but since we limit the second loop to k values, the overall work done is limited to O(kn). Although the code for the banded algorithm is a lot more complicated, the work done per iteration is very similar to the unbanded version. The increased complexity in the code comes from an increased number of if-branches and checks, not from extra work. The work done doesn't increase at all, because we're still just checking cells, chars in the strings, and performing simple additions and subtractions in addition to table inserts. So the only real difference is that the second for loop is limited to k iterations instead of m for the second string.
- Space
  - Same as the above, we don't add anything beyond what we did with banded, we're just limiting our time and space to n x k iterations instead of n x m.
- For O(kn) table, discuss how you modified your dependency pointers to the adjacent cells with your smaller array
  - The big adjustments we have to make to the dependency pointers are 1) how we access the values in the array to compare, 2) how we add back pointers to the array, and 3) how we traverse the back pointers to rebuild the string.

- Also note that we have a lot of  $i \leq d$  checks, because for the first  $d + 1$  rows (rows 0-3), we treat them similar to the unbanded algorithm (in terms of checking, back pointers, etc.) because they shift like normal
- Accessing the values in the array:
  - To access the values in the banded array, we check the value to the left, the value directly above, and the value to the top-right. This is because the cells effectively “shift” over, even though their indexing remains the same. That means that the cell directly above our current cell acts as our “diagonal” sub/match cell, and our top-right acts as an indel cell. So left cells are accessed the same way, but the top two cells are shifted to the right one. This is why I added the POINT\_TOPRIGHT variable in the banded algorithm, that gives us a fourth direction to point and deal with.
  - We deal with accessing this differently, because instead of accessing  $\text{table}[i][j - 1]$  (left),  $\text{table}[i - 1][j]$  (top), and  $\text{table}[i - 1][j - 1]$  (diagonal) — like we do for the unbanded algorithm — we instead access  $\text{table}[i][j - 1]$  (left — this one stays the same),  $\text{table}[i - 1][j]$  (top, but functions as diagonal), and  $\text{table}[i - 1][j + 1]$  (top right, but functions as top). This isn’t too hard to think about in terms of the table, but it does create a lot of opportunities for index-out-of-bound errors, which is part of why there are so many if-branches. This is also why we have variables like `columns_left` and `columns_to_check`, which help us not only know how far the algorithm should go (i.e., how many columns to create on each row), but also help ensure we’re not accessing out of bounds.
- Adding back pointers to the array:
  - I may have over-engineered this part, but in order to simplify the minimization checks, I continued to use `left`, `top`, and `top_left` for my value checks from the value table; I just had to be careful about `top_left` accessing the value directly above the current one, and `top` accessing the top-right value. This allows for the minimization check to function like the unbanded algorithm, with one caveat: there is an extra  $i \leq d$  check to know whether we should add `top_left` vs. `top` pointers, and `top` vs. `top_right` pointers into the `prev` array. The priority stays the same, so we traverse the banded table the same way, but our back pointers for `top` and `top-right` will be treated differently when we traverse the `prev` array.
- Traversing the `prev` array differently to rebuild the string:
  - If we’ve added everything correctly, we should be ready to traverse our `prev` array pretty easily. `Left`, `top`, and `top-left` are all treated the same as in the unbanded version. The difference is that we will have `top-right` pointers as well that act as indels. When we point `top-right`, we add a value to our column-counter because we have to move back to the right through the `k` columns. For similar strings that have a lot of sub/matches, there will be a lot more `point-top` pointers, which will act as diagonal pointers. For dissimilar strings, we will have a lot more `point-left` and

point-top-right pointers that will take us back and forth through the banded table.

- Note that we also have to have a seq1 counter that allows us to access the sequence at the appropriate location, since our column is only k long and doesn't match the sequence length. This isn't too hard to keep track of, as anytime we move left or top (when it functions as a diagonal), we subtract 1 from the sequence counter. The only reason this is a little hard to keep track of is that our column-counter and our seq1 counter don't necessarily get added/subtracted from at the same time. But following the pointers as sub/match vs. indels helps to make this more clear, as sub/matches will always subtract 1 from seq1 but do nothing to column-counter, left pointers will also subtract 1 from seq1, but do nothing to column-counter, and top-right pointers will add 1 to the column-counter, but do nothing to the sequence counter, as it will effectively be traversing the table straight up.

```
Initialize tables

POINT_LEFT = 0
POINT_TOP = 1
POINT_TOPLEFT = 2
POINT_TOPRIGHT = 3

# This initializes the first banded row - O(n)
for i in range(d + 1):

    if i > d:
        break
    table_2d[0].append(counter)
    counter += 5
    prev_2d[0].append(POINT_LEFT)

for i in range(align_length + 1):

    # columns_left deals with the very bottom of the table when we start cutting
    #    columns off
    # columns_to_check deals with the top of the table when we add < k column
    if i >= align_length - d:
        columns_left = align_length - i + d + 1
    elif i >= len(seq1) - d:
        columns_left = len(seq1) - i + d + 1
    else:
        columns_left = math.inf

    if i <= d:
        columns_to_check = d + i + 1
    else:
        columns_to_check = math.inf
```

```

# This loop keeps our time and space to O(kn) instead of making it O(mn)
for j in range(k):
    if i <= d:

        if columns_to_check > 1:
            top = table_2d[i - 1][j] + 5
        else:
            top = math.inf
        left = table_2d[i][j - 1] + 5
        top_left = table_2d[i - 1][j - 1]

        if seq1[j - 1] == seq2[i - 1]:
            top_left -= 3
        else:
            top_left += 1

    else:

        if j == 0:
            left = math.inf
        else:
            left = table_2d[i][j - 1] + 5
        if j == k - 1:
            top = math.inf
        else:
            top = table_2d[i - 1][j + 1] + 5
        top_left = table_2d[i - 1][j]

        if seq1[i + character_counter - 1] == seq2[i - 1]:
            top_left -= 3
        else:
            top_left += 1

# priority -- left, top, diagonal -- we need the i <= d checks to know
#   which way to point, since we have an extra direction to deal with
if left <= top:
    if left <= top_left:
        table_2d[i].append(left)
        prev_2d[i].append(POINT_LEFT)
    else:
        table_2d[i].append(top_left)
        if i <= d:
            prev_2d[i].append(POINT_TOPLEFT)
        else:
            prev_2d[i].append(POINT_TOP)
elif top <= top_left:
    table_2d[i].append(top)
    if i <= d:
        prev_2d[i].append(POINT_TOP)

```

```
        prev_2d[i].append(POINT_TOPRIGHT)
else:
    table_2d[i].append(top_left)
    if i <= d:
        prev_2d[i].append(POINT_TOPLEFT)
    else:
        prev_2d[i].append(POINT_TOP)
```

Rebuild the two alignment strings  
Get the score from the last cell in the table

## Part 3 & 4 – Table Screenshots and seq3 x seq10 alignment

**Gene Sequence Alignment**

	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	sequence10
sequence1	-30	-1	inf							
sequence2		-33	inf							
<b>sequence3</b>			<b>-9000</b>	<b>-8984</b>	<b>-8888</b>	<b>-8848</b>	<b>-2735</b>	<b>-2743</b>	<b>-1429</b>	<b>-2735</b>
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 3: gi|15077808|gb|AF391541.1|Bovine coronavirus isolate BCoV-ENT, complete genome.

Sequence 3: gattgcagcgatttgcgtgcgtgcacccgcttc-actg--at-ctcttgtagatctttcataatctaaactttataaaaacatccactccctgtat-

Sequence 10: -ataa-gagtgattggcgtccgtacgtacccttactctcaaactctttagttaaatc-taatctaaactttataaa-cggc-acttcctgtgt

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

Banded Align Length: 3000

Done. Time taken: 0.594 seconds.

I got the same sequence for banded and unbanded:

#### Unbanded:

```
gattgcagcgatttgcgtgcgtgcacccgcttc-actg--at-ctcttgtagatctttcataatctaaactttataaaaacatccactccctgtat-
-ataa-gagtgattggcgtccgtacgtacccttactctcaaactctttagttaaatc-taatctaaactttataaa-cggc-acttcctgtgt
```

#### Banded:

```
gattgcagcgatttgcgtgcgtgcacccgcttc-actg--at-ctcttgtagatctttcataatctaaactttataaaaacatccactccctgtat-
-ataa-gagtgattggcgtccgtacgtacccttactctcaaactctttagttaaatc-taatctaaactttataaa-cggc-acttcctgtgt
```

**Gene Sequence Alignment**

	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	sequence10
sequence1	-30	-1	inf							
sequence2		-33	inf							
<b>sequence3</b>			<b>-9000</b>	<b>-8984</b>	<b>-8888</b>	<b>-8848</b>	<b>-2735</b>	<b>-2743</b>	<b>-1429</b>	<b>-2735</b>
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 3: gi|15077808|gb|AF391541.1|Bovine coronavirus isolate BCoV-ENT, complete genome.

Sequence 3: gattgcagcgatttgcgtgcgtgcacccgcttc-actg--at-ctcttgtagatctttcataatctaaactttataaaaacatccactccctgtat-

Sequence 10: -ataa-gagtgattggcgtccgtacgtacccttactctcaaactctttagttaaatc-taatctaaactttataaa-cggc-acttcctgtgt

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

Banded Align Length: 1000

**Gene Sequence Alignment**

	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	sequence10
sequence1	-30	-1	inf							
sequence2		-33	inf							
<b>sequence3</b>			<b>-9000</b>	<b>-8984</b>	<b>-8888</b>	<b>-8848</b>	<b>-2735</b>	<b>-2743</b>	<b>-1429</b>	<b>-2735</b>
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 3: gi|15077808|gb|AF391541.1|Bovine coronavirus isolate BCoV-ENT, complete genome.

Sequence 3: gattgcagcgatttgcgtgcgtgcacccgcttc-actg--at-ctcttgtagatctttcataatctaaactttataaaaacatccactccctgtat-

Sequence 10: -ataa-gagtgattggcgtccgtacgtacccttactctcaaactctttagttaaatc-taatctaaactttataaa-cggc-acttcctgtgt

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

Banded Align Length: 3000

Done. Time taken: 0.600 seconds.

# Appendix

The image shows two side-by-side screenshots of a code editor, likely VS Code, displaying the same Python file, `GeneSequencing.py`, at different stages of development. The top screenshot shows an initial implementation of a function `align`. The bottom screenshot shows a revised version with significant changes, particularly in the handling of sequence lengths and the creation of a 2x2 array.

**Top Screenshot (Initial Implementation):**

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

    if banded:
        # d and k designate our bandwidth - d is the number of cells to calculate from the diagonal,
        # k is the total bandwidth
        d = 3
        k = 7

        table_2d = []
        prev_2d = []

        POINT_LEFT = 0
        POINT_TOP = 1
        POINT_TOPOLEFT = 2
        POINT_TORIGHT = 3

        # This initializes the first row of the table
        table_2d.append([])
        prev_2d.append([])
        counter = 0

        # This initializes the first banded row - O(n)
        for i in range(d + 1):

            if i > d:
                break
            table_2d[0].append(counter)
            counter += 5
            prev_2d[0].append(POINT_LEFT)

        # Set to 5 since it shares 0 with the first row
        counter = 5
```

**Bottom Screenshot (Revised Implementation):**

```
counter = 5

# i is our rows, so it has to cover the entire length of the string
# This double for loop gives space and time of O(kn) - m for one string, k for the band
# This also creates our 2x2 array
for i in range(align_length + 1):

    # This check covers the first two sequences (polynomial and exponential)
    if len(seq2) - len(seq1) > 1000:
        too_big = True
        break
    else:
        too_big = False

    if i > len(seq2) or i > len(seq1) + 1:
        break

    if i == 0:
        continue

    table_2d.append([])
    prev_2d.append([])

    # We use this counter to access the correct chars in seq1 (the string that goes down the rows)
    # We need it because we have to access 3 chars less than and 3 chars greater than the diagonal
    # So it will run from -3 to 3 and be added to i inside the second for loop (j)
    character_counter = - 3

    # This is how we switch the number of columns that we need at the beginning and end of the matrix
    if i >= align_length - d:
        columns_left = align_length - i + d + 1
    elif i >= len(seq1) - d:
        columns_left = len(seq1) - i + d + 1
    else:
        columns_left = math.inf
```

```
GeneSequencingProject4 / GeneSequencing.py
  GeneSequencing.py
  Proj4GUI.py

  columns_left = math.inf

  if i <= d:
    columns_to_check = d + i + 1
  else:
    columns_to_check = math.inf

  # This loop keeps our time and space to O(kn) instead of making it O(mn)
  for j in range(k):

    # Break when there's no more columns -- Helps manage row length
    if columns_to_check == 0:
      break
    columns_to_check -= 1

    # This initializes the first column of the table
    if j == 0 and i <= d:
      table_2d[i].append(counter)
      counter += 5
      prev_2d[i].append(POINT_TOP)
      continue

    # This if is for the first 4 rows, the else for all the others.
    # The way I manage this, we don't change whether diagonals are called top_left or not.
    # They just get treated differently in the prev array
    if i <= d:

      if columns_to_check > 1:
        top = table_2d[i - 1][j] + 5
      else:
        top = math.inf
      left = table_2d[i][j - 1] + 5
      top_left = table_2d[i - 1][j - 1]

      if seq1[j - 1] == seq2[i - 1]:
        top_left -= 3
```

```
GeneSequencingProject4 > GeneSequencing.py
Project GeneSequencing.py Proj4GUI.py
137     if seq1[j - 1] == seq2[i - 1]:
138         top_left -= 3
139     else:
140         top_left += 1
141
142     else:
143
144         if j == 0:
145             left = math.inf
146         else:
147             left = table_2d[i][j - 1] + 5
148         if j == k - 1:
149             top = math.inf
150         else:
151             top = table_2d[i - 1][j + 1] + 5
152         top_left = table_2d[i - 1][j]
153
154
155         # The character_counter is how we know where in seq1 to access, since we only iterate through
156         # k times
157         if seq1[i + character_counter - 1] == seq2[i - 1]:
158             top_left -= 3
159         else:
160             top_left += 1
161
162         # priority -- left, top, diagonal -- we need the i <= d checks to know which way to point
163         if left <= top:
164             if left <= top_left:
165                 table_2d[i].append(left)
166                 prev_2d[i].append(POINT_LEFT)
167             else:
168                 table_2d[i].append(top_left)
169                 if i <= d:
170                     prev_2d[i].append(POINT_TOPLEFT)
171                 else:
172                     prev_2d[i].append(POINT_TOP)
173
174
175 GeneSequencing > align() > else > if < top <= top_left
```

GeneSequencingProject4 > GeneSequencing.py

```
171     prev_2d[i].append(POINT_TOP)
172     elif top <= top_left:
173         table_2d[i].append(top)
174         if i <= d:
175             prev_2d[i].append(POINT_TOP)
176         else:
177             prev_2d[i].append(POINT_TOPRIGHT)
178     else:
179         table_2d[i].append(top_left)
180         if i <= d:
181             prev_2d[i].append(POINT_TOPLEFT)
182         else:
183             prev_2d[i].append(POINT_TOP)
184
185     # this should never be more than 3 -- it's how we know when to stop iterating if there's >7 columns
186     if i >= align_length - d or i >= len(seq2) - d:
187         columns_left -= 1
188         if columns_left == 0:
189             break
190
191     character_counter += 1
192
193     # This is how we handle big discrepancies in sequence length
194     if too_big:
195         score = math.inf
196         alignment1 = "No alignment possible"
197         alignment2 = "No alignment possible"
198
199     else:
200         alignment1 = ""
201         alignment2 = ""
202
203         # seq1 counter allows us to access the string since our column counter stays between
204         # 0 < column_counter < 7. row_counter also functions as seq2 counter
205         row_counter = len(table_2d) - 2
206         column_counter = len(table_2d[-1]) - 1
207         if len(seq1) > align_length:
208
209             GeneSequencing > align() > else
210
211             Git Python Packages TODO Python Console Problems Terminal Services
212
213             338:88 LF UTF-8 Tab* Python 3.11 (GeneSequencingProject4) main
```

GeneSequencingProject4 > GeneSequencing.py

```
205     if len(seq1) > align_length:
206         seq1_counter = align_length - 1
207     else:
208         seq1_counter = len(seq1) - 1
209
210     # Start with the initial character in the bottom right corner
211     alignment1 = seq1[seq1_counter] + alignment1
212     alignment2 = seq2[row_counter] + alignment2
213
214     # This should be O(n) where n is alignment length - reverse traverses through the table to build the
215     # alignment string.
216     while True:
217
218         if row_counter == 0 and column_counter == 0:
219             break
220
221         if row_counter <= d:
222             if prev_2d[row_counter][column_counter] == POINT_LEFT:
223                 column_counter -= 1
224                 seq1_counter -= 1
225                 alignment1 = seq1[seq1_counter] + alignment1
226                 alignment2 = '-' + alignment2
227             elif prev_2d[row_counter][column_counter] == POINT_TOP:
228                 row_counter -= 1
229                 alignment1 = '-' + alignment1
230                 alignment2 = seq2[row_counter] + alignment2
231             else:
232                 column_counter -= 1
233                 seq1_counter -= 1
234                 row_counter -= 1
235                 alignment1 = seq1[seq1_counter] + alignment1
236                 alignment2 = seq2[row_counter] + alignment2
237
238         else:
239             if prev_2d[row_counter][column_counter] == POINT_LEFT:
240                 column_counter -= 1
241
242             Git Python Packages TODO Python Console Problems Terminal Services
243
244             338:88 LF UTF-8 Tab* Python 3.11 (GeneSequencingProject4) main
```

The screenshot shows a Python code editor with the following details:

- Project:** GeneSequencingProject4
- Current File:** GeneSequencing.py
- Code Content:** The code implements a dynamic programming algorithm for sequence alignment. It defines two sequences, seq1 and seq2, and initializes a 2D table for scoring. The code handles various cases for matching, mismatching, and aligning gaps. It also includes logic to initialize the table and set up the first row.

```
237
238     else:
239         if prev_2d[row_counter][column_counter] == POINT_LEFT:
240             column_counter -= 1
241             seq1_counter += 1
242             alignment1 = seq1[seq1_counter] + alignment1
243             alignment2 = '-' + alignment2
244         elif prev_2d[row_counter][column_counter] == POINT_TOPRIGHT:
245             row_counter -= 1
246             column_counter += 1
247             alignment1 = '-' + alignment1
248             alignment2 = seq2[row_counter] + alignment2
249         else: # this functions as the diagonal sub/match now
250             seq1_counter -= 1
251             row_counter -= 1
252             alignment1 = seq1[seq1_counter] + alignment1
253             alignment2 = seq2[row_counter] + alignment2
254
255             score = table_2d[-1][-1]
256             alignment1 = alignment1[:100]
257             alignment2 = alignment2[:100]
258
259     else:
260         # initialize table - first value is rows, second value is columns
261         table_2d = []
262         prev_2d = []
263
264         # Values for the prev array:
265         POINT_LEFT = 0
266         POINT_TOP = 1
267         POINT_TOPLEFT = 2
268
269         # This initializes the first row of the table
270         table_2d.append([])
271         prev_2d.append([])
272         counter = 0
273         for i in range(alignment_length + 1):
274             GeneSequencing > align() > else
```

GeneSequencingProject4 > GeneSequencing.py

```
306     top = table_2d[i - 1][j] + 5
307     top_left = table_2d[i - 1][j - 1]
308     if seq1[j - 1] == seq2[i - 1]:
309         top_left -= 3
310     else:
311         top_left += 1
312
313     # Priority order: left, top, top-left
314     if left <= top:
315         if left <= top_left:
316             table_2d[i].append(left)
317             prev_2d[i].append(POINT_LEFT)
318         else:
319             table_2d[i].append(top_left)
320             prev_2d[i].append(POINT_TOPLEFT)
321     elif top <= top_left:
322         table_2d[i].append(top)
323         prev_2d[i].append(POINT_TOP)
324     else:
325         table_2d[i].append(top_left)
326         prev_2d[i].append(POINT_TOPLEFT)
327
328     alignment1 = ""
329     alignment2 = ""
330     row_counter = len(table_2d) - 2
331     column_counter = len(table_2d[0]) - 2
332
333     # Start with the initial character in the bottom right corner
334     alignment1 = seq1[column_counter] + alignment1
335     alignment2 = seq2[row_counter] + alignment2
336
337     # Traverse the prev array from end back to beginning to build the strings
338     # O(n) where n is the length of the longer string (or the alignment length)
339     while True:
340
341         if row_counter == 0 and column_counter == 0:
342             break
343
344         if prev_2d[row_counter][column_counter] == POINT_LEFT:
345             column_counter -= 1
346             alignment1 = seq1[column_counter] + alignment1
347             alignment2 = '-' + alignment2
348         elif prev_2d[row_counter][column_counter] == POINT_TOP:
349             row_counter -= 1
350             alignment1 = '-' + alignment1
351             alignment2 = seq2[row_counter] + alignment2
352         else:
353             column_counter -= 1
354             row_counter -= 1
355             alignment1 = seq1[column_counter] + alignment1
356             alignment2 = seq2[row_counter] + alignment2
357
358         score = table_2d[-1][-1]
359         alignment1 = alignment1[:100]
360         alignment2 = alignment2[:100]
361
362     return {'align_cost': score, 'seq1_first100': alignment1, 'seq2_first100': alignment2}
```

GeneSequencingProject4 > GeneSequencing.py

```
334     # Start with the initial character in the bottom right corner
335     alignment1 = seq1[column_counter] + alignment1
336     alignment2 = seq2[row_counter] + alignment2
337
338     # Traverse the prev array from end back to beginning to build the strings
339     # O(n) where n is the length of the longer string (or the alignment length)
340     while True:
341
342         if row_counter == 0 and column_counter == 0:
343             break
344
345         if prev_2d[row_counter][column_counter] == POINT_LEFT:
346             column_counter -= 1
347             alignment1 = seq1[column_counter] + alignment1
348             alignment2 = '-' + alignment2
349         elif prev_2d[row_counter][column_counter] == POINT_TOP:
350             row_counter -= 1
351             alignment1 = '-' + alignment1
352             alignment2 = seq2[row_counter] + alignment2
353         else:
354             column_counter -= 1
355             row_counter -= 1
356             alignment1 = seq1[column_counter] + alignment1
357             alignment2 = seq2[row_counter] + alignment2
358
359         score = table_2d[-1][-1]
360         alignment1 = alignment1[:100]
361         alignment2 = alignment2[:100]
362
363     return {'align_cost': score, 'seq1_first100': alignment1, 'seq2_first100': alignment2}
```