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 <= 1,000. We can consider this constant time of O(1,000 x 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 <= 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 of instead accessing table[i][j - 1] (left), table[i - 1][j] (top), and table[i - 1][j - 1] (diagonal) – like we do for the unbanded algorithm — we instead access table[i][j - 1] (left – this one stays the same), table[i - 1][j] (top, but functions as diagonal), and 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 <= 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.

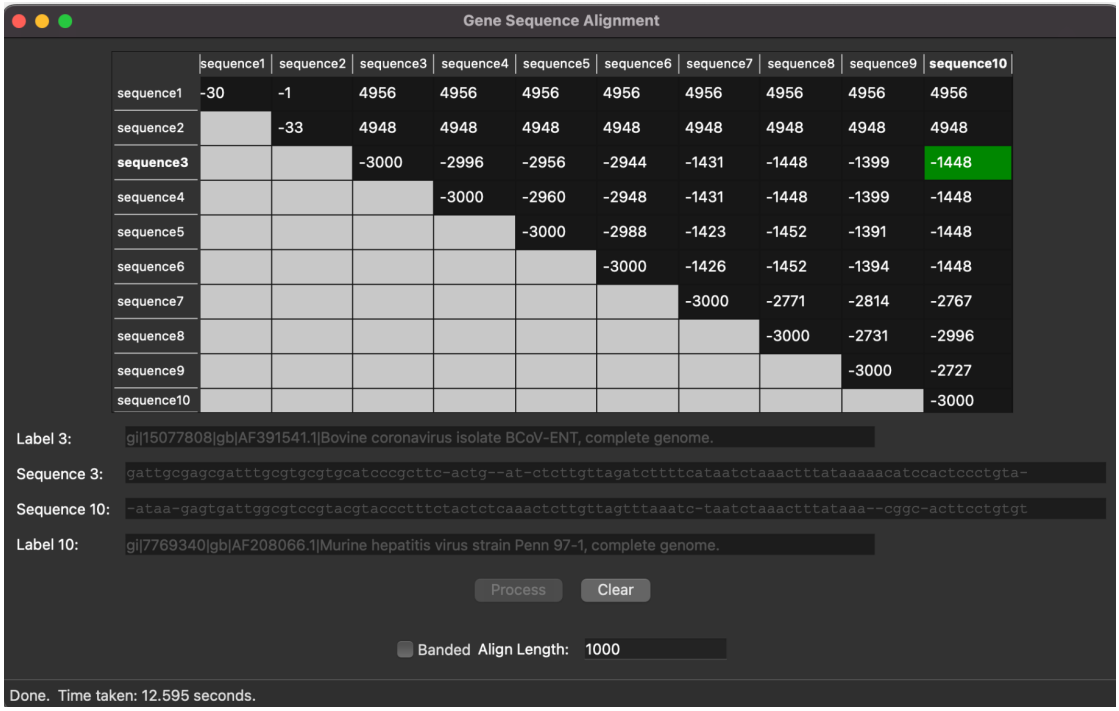
- Note that we also have to have a seq1 counter that allows us to access the sequence at the appropriate location, since our column is only k long and doesn't match the sequence length. This isn't too hard to keep track of, as anytime we move left or top (when it functions as a diagonal), we subtract 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.

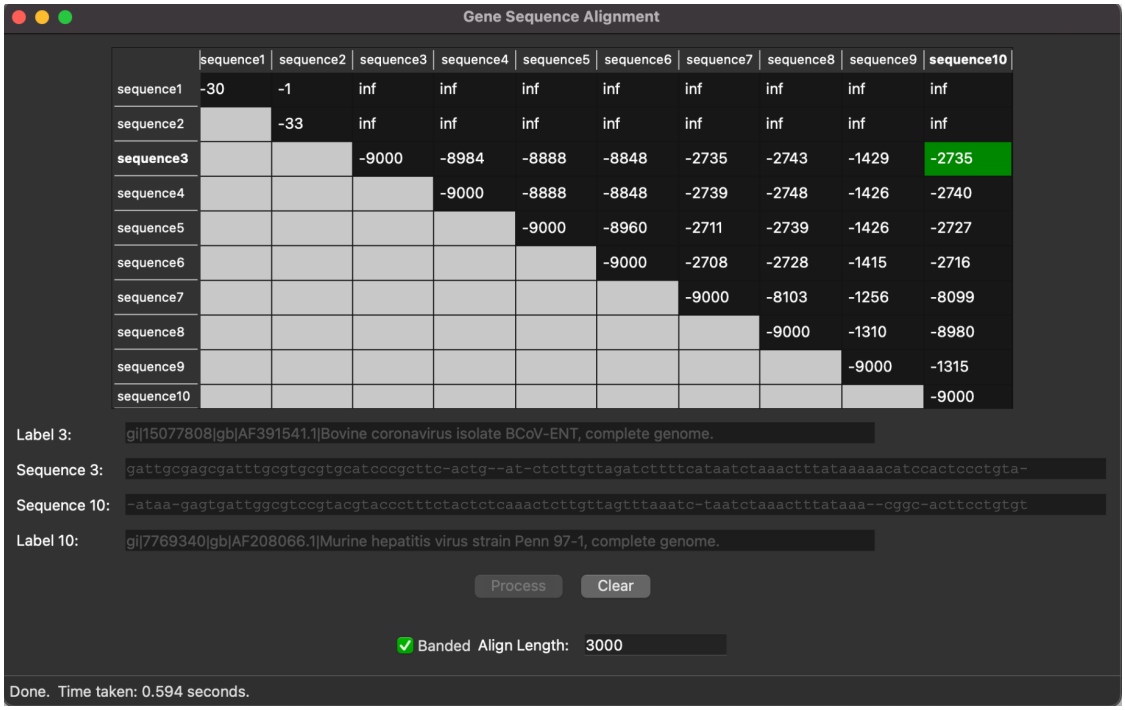
```
Initialize tables

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)
```

```
            else:
                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





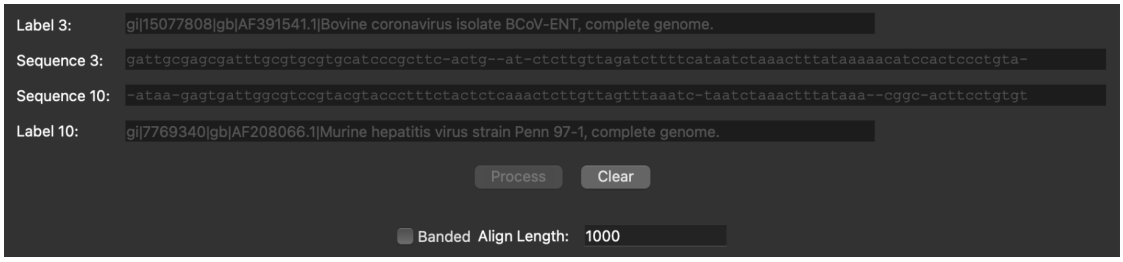
I got the same sequence for banded and unbanded:

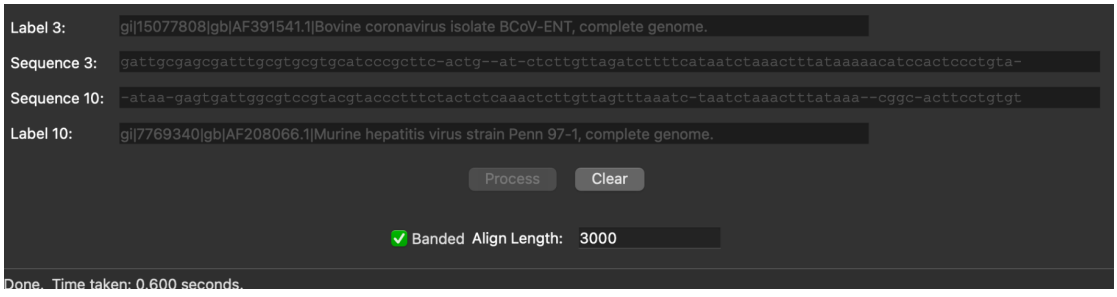
Unbanded:

```
gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaacttttataaaaacatccactccctgta-
-ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaacttttataaa--cggc-acttcctgtgt
```

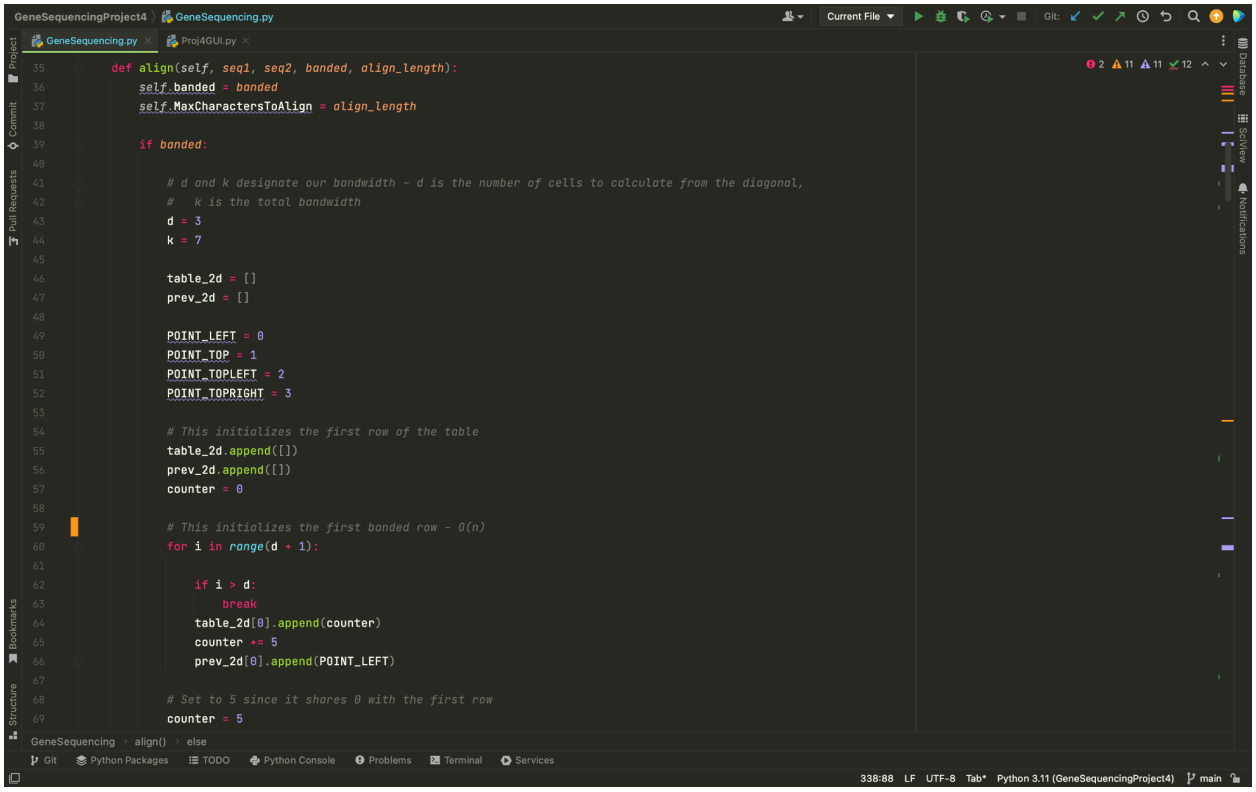
Banded:

```
gattgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaacttttataaaaacatccactccctgta-
-ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaacttttataaa--cggc-acttcctgtgt
```

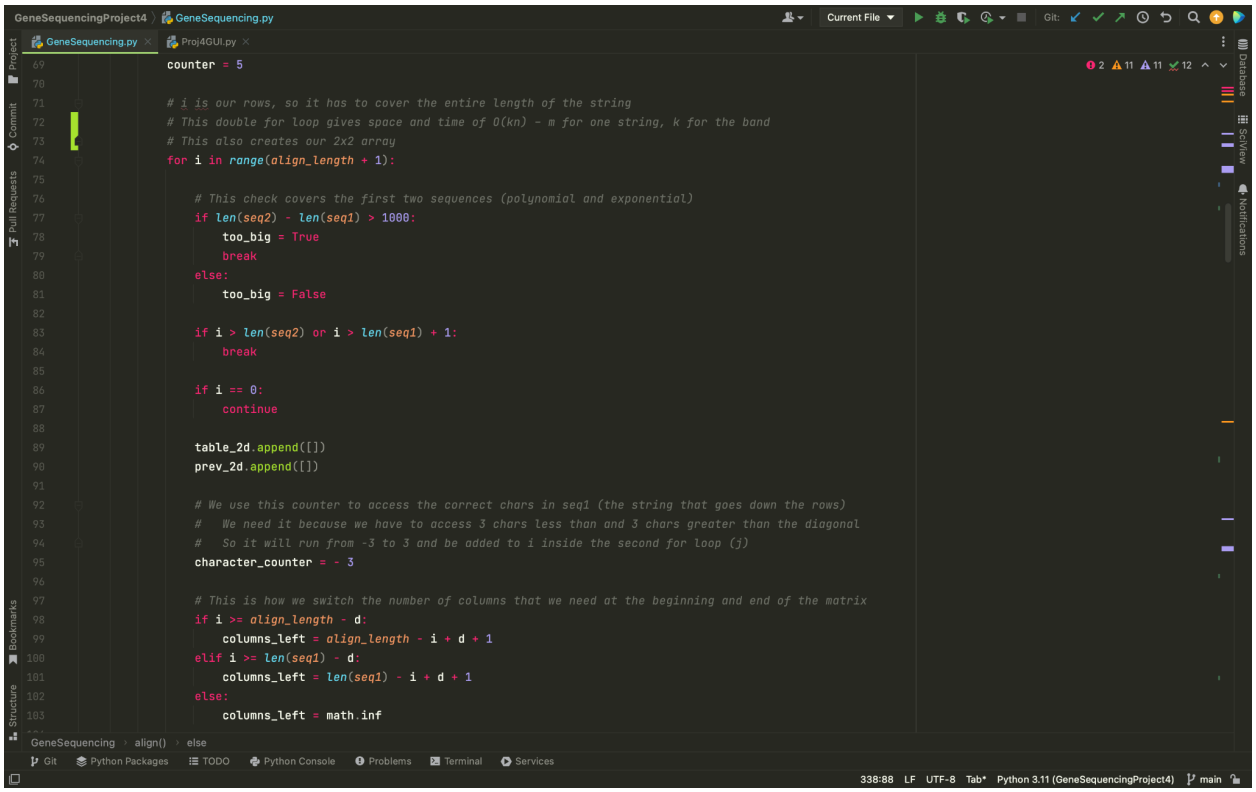




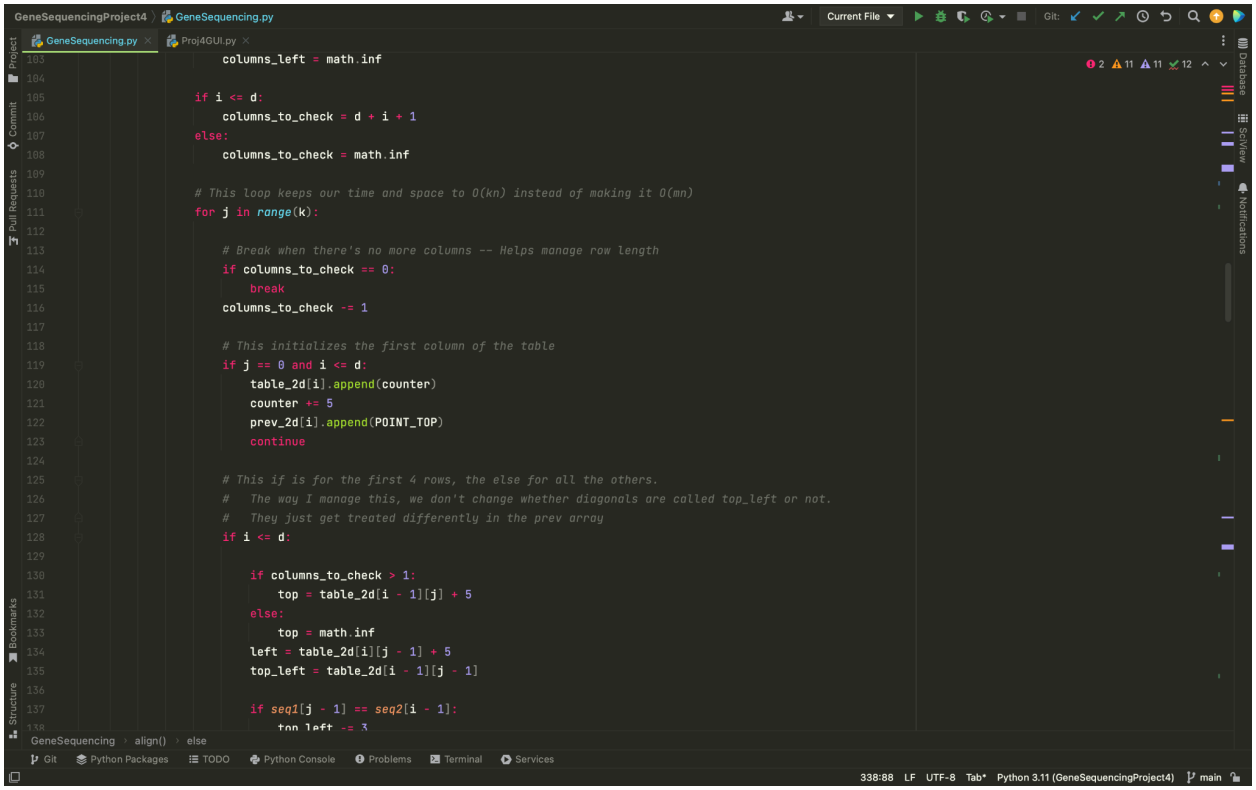
Appendix



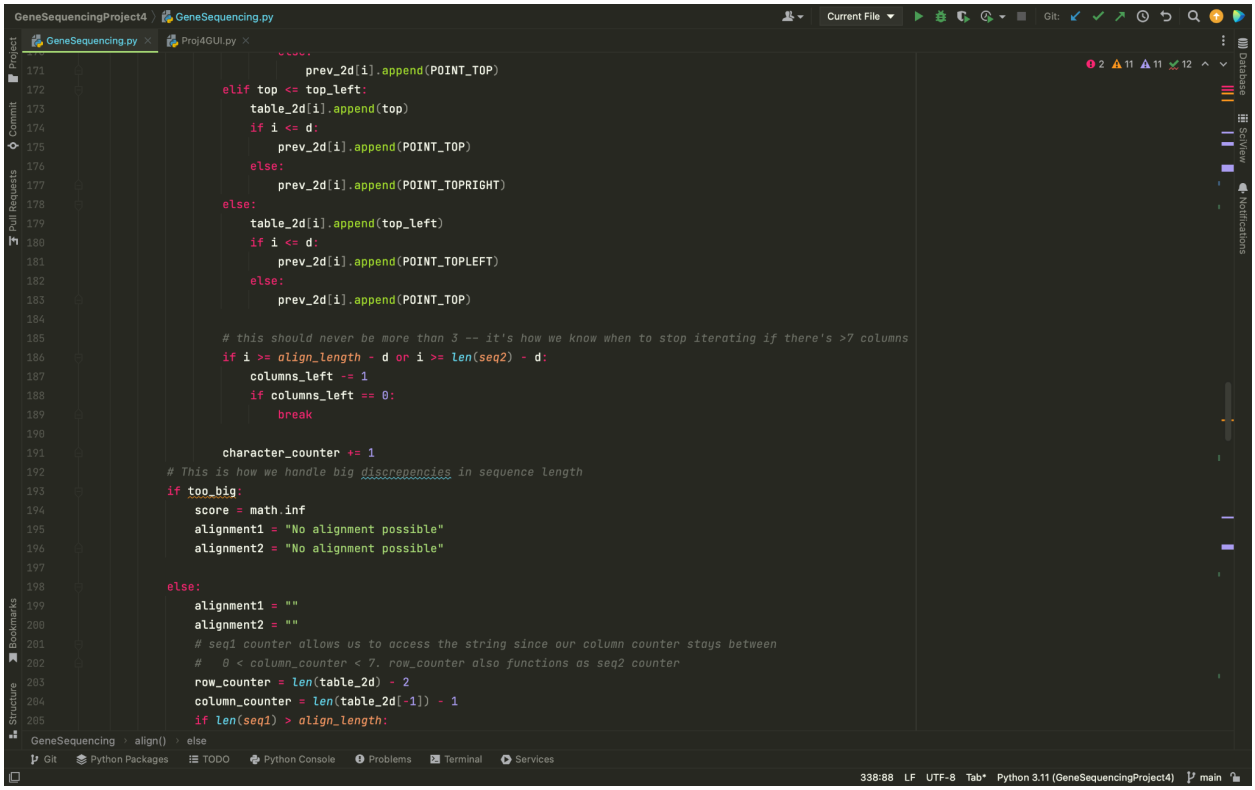
```
35      def align(self, seq1, seq2, banded, align_length):
36          self.banded = banded
37          self.MaxCharactersToAlign = align_length
38
39          if banded:
40
41              # d and k designate our bandwidth - d is the number of cells to calculate from the diagonal,
42              #   k is the total bandwidth
43              d = 3
44              k = 7
45
46              table_2d = []
47              prev_2d = []
48
49              POINT_LEFT = 0
50              POINT_TOP = 1
51              POINT_TOPLEFT = 2
52              POINT_TOPRIGHT = 3
53
54              # This initializes the first row of the table
55              table_2d.append([])
56              prev_2d.append([])
57              counter = 0
58
59              # This initializes the first banded row - O(n)
60              for i in range(d + 1):
61
62                  if i > d:
63                      break
64                  table_2d[0].append(counter)
65                  counter += 5
66                  prev_2d[0].append(POINT_LEFT)
67
68              # Set to 5 since it shares 0 with the first row
69              counter = 5
```



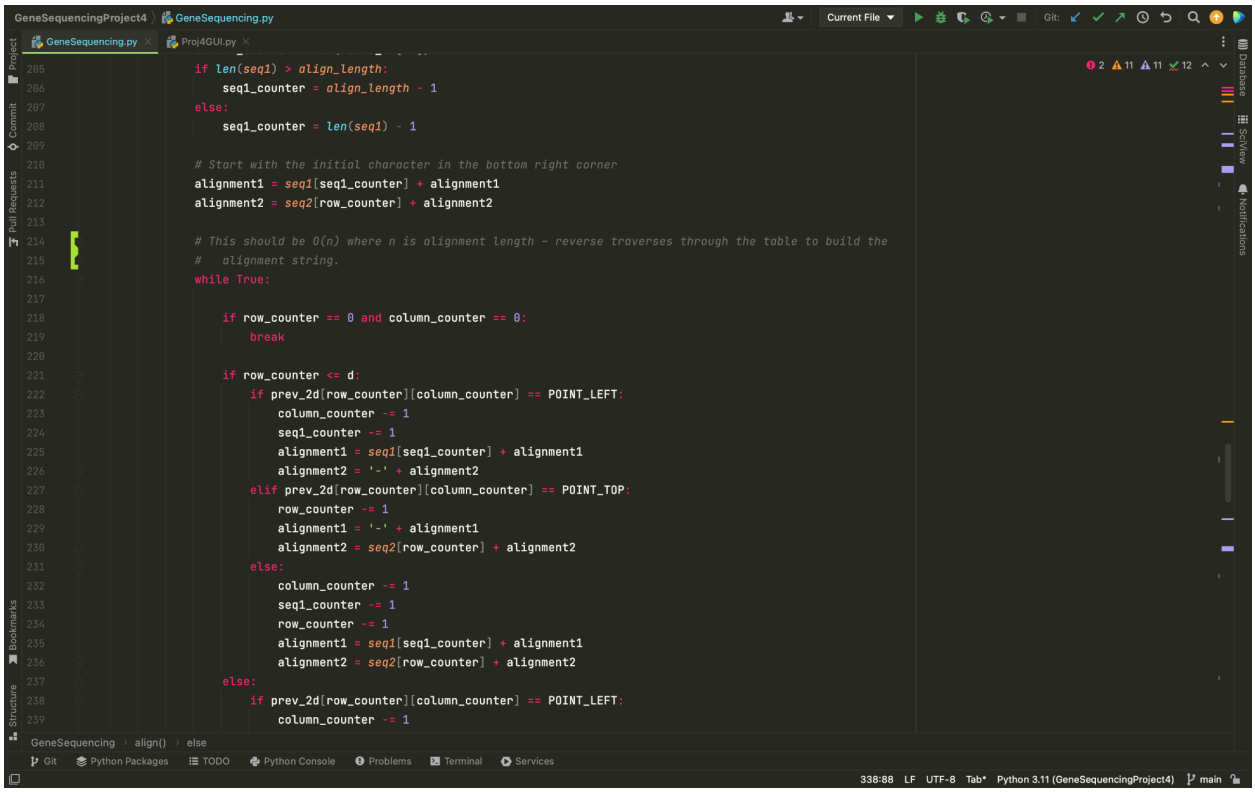
```
69              counter = 5
70
71              # i is our rows, so it has to cover the entire length of the string
72              # This double for loop gives space and time of O(kn) - m for one string, k for the band
73              # This also creates our 2x2 array
74              for i in range(align_length + 1):
75
76                  # This check covers the first two sequences (polynomial and exponential)
77                  if len(seq2) - len(seq1) > 1000:
78                      too_big = True
79                      break
80                  else:
81                      too_big = False
82
83                  if i > len(seq2) or i > len(seq1) + 1:
84                      break
85
86                  if i == 0:
87                      continue
88
89                  table_2d.append([])
90                  prev_2d.append([])
91
92                  # We use this counter to access the correct chars in seq1 (the string that goes down the rows)
93                  #   We need it because we have to access 3 chars less than and 3 chars greater than the diagonal
94                  #   So it will run from -3 to 3 and be added to i inside the second for loop (j)
95                  character_counter = - 3
96
97                  # This is how we switch the number of columns that we need at the beginning and end of the matrix
98                  if i >= align_length - d:
99                      columns_left = align_length - i + d + 1
100                 elif i >= len(seq1) - d:
101                     columns_left = len(seq1) - i + d + 1
102                 else:
103                     columns_left = math.inf
```



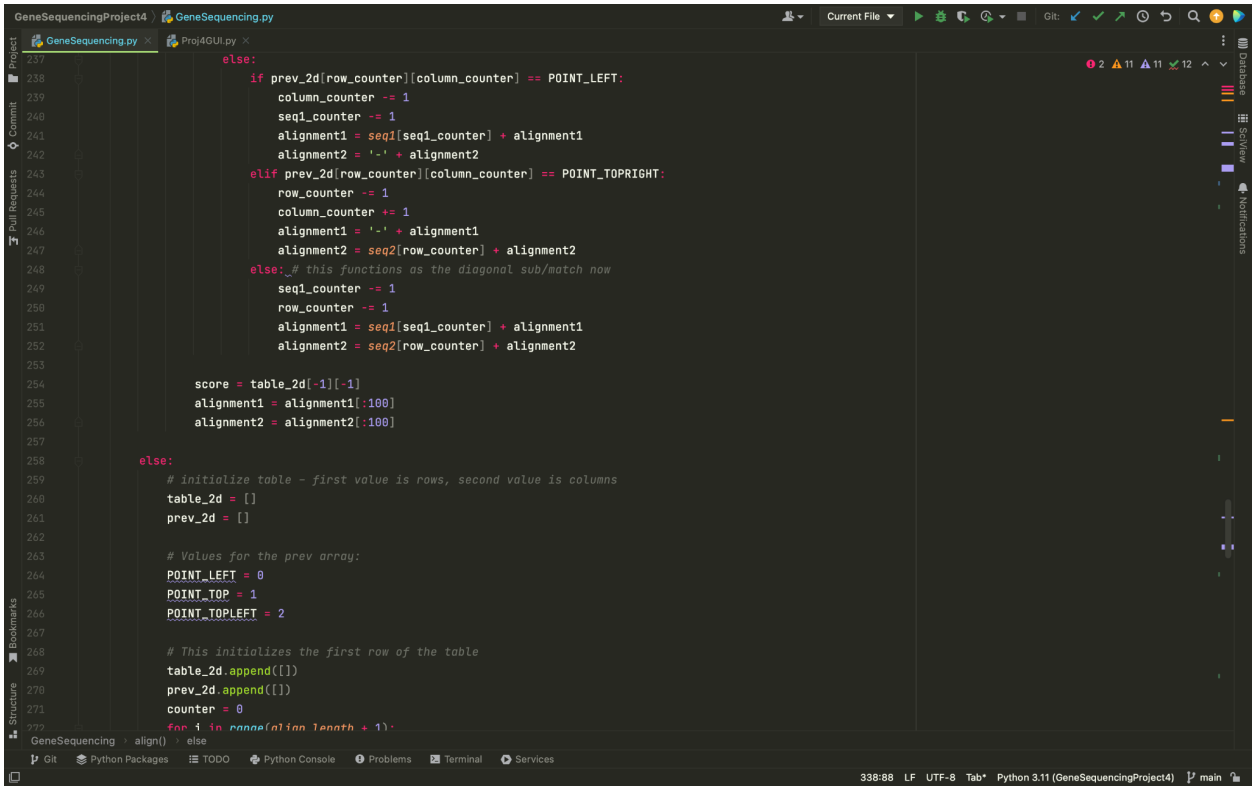
```
            columns_left = math.inf

        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
                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]

                # The character_counter is how we know where in seq1 to access, since we only iterate through
                #   k times
                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
            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:
```

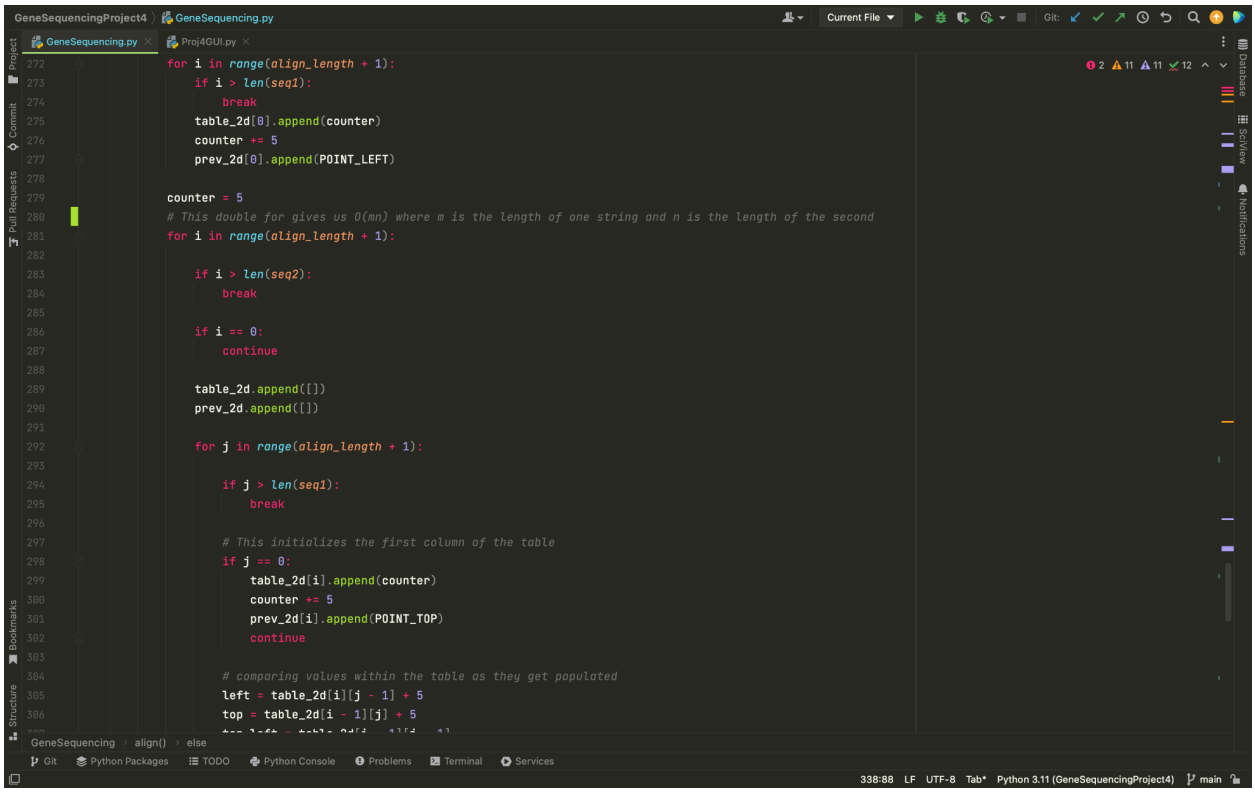


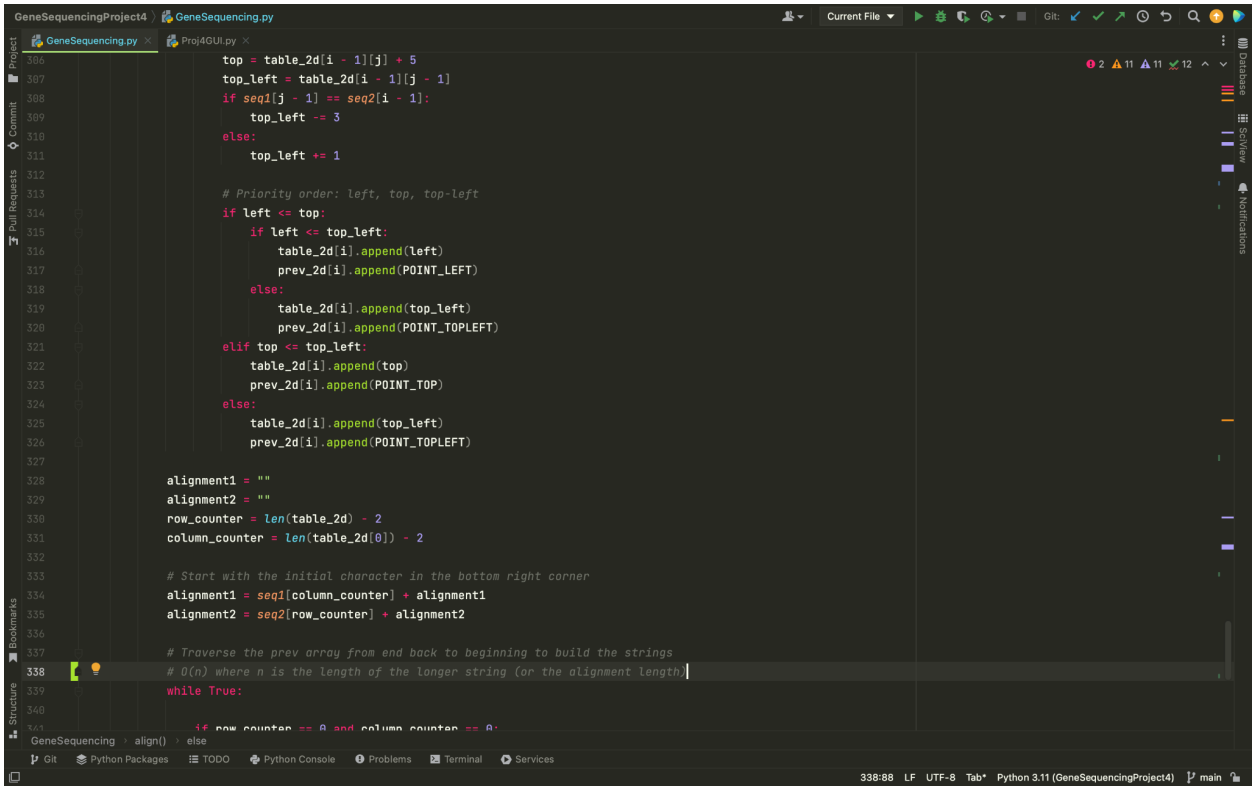
```
                        prev_2d[i].append(POINT_TOP)
                elif top <= top_left:
                    table_2d[i].append(top)
                    if i <= d:
                        prev_2d[i].append(POINT_TOP)
                    else:
                        prev_2d[i].append(POINT_TOPRIGHT)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

                # this should never be more than 3 -- it's how we know when to stop iterating if there's >7 columns
                if i >= align_length - d or i >= len(seq2) - d:
                    columns_left -= 1
                    if columns_left == 0:
                        break

                character_counter += 1
            # This is how we handle big discrepencies in sequence length
            if too_big:
                score = math.inf
                alignment1 = "No alignment possible"
                alignment2 = "No alignment possible"

            else:
                alignment1 = ""
                alignment2 = ""
                # seq1 counter allows us to access the string since our column counter stays between
                #   0 < column_counter < 7. row_counter also functions as seq2 counter
                row_counter = len(table_2d) - 2
                column_counter = len(table_2d[-1]) - 1
                if len(seq1) > align_length:
```



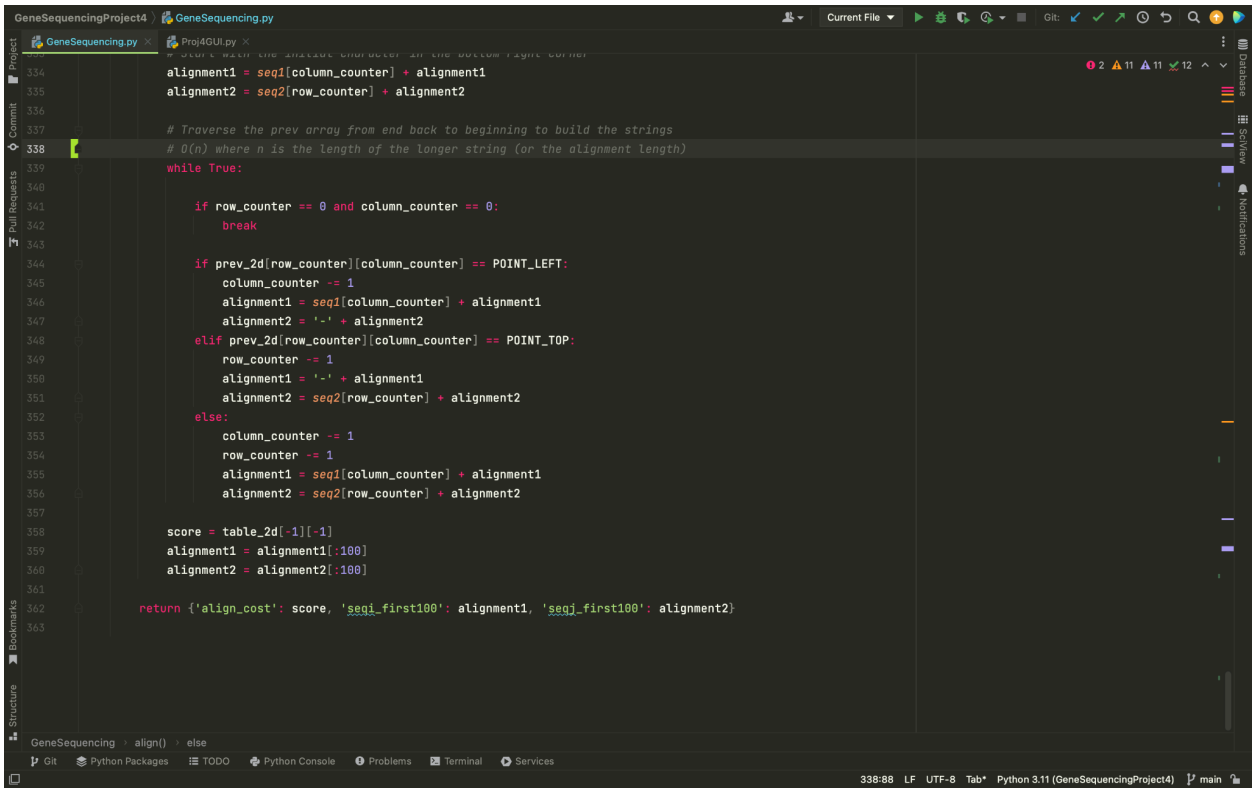
```
                if len(seq1) > align_length:
                    seq1_counter = align_length - 1
                else:
                    seq1_counter = len(seq1) - 1

                # Start with the initial character in the bottom right corner
                alignment1 = seq1[seq1_counter] + alignment1
                alignment2 = seq2[row_counter] + alignment2

                # This should be O(n) where n is alignment length - reverse traverses through the table to build the
                #   alignment string.
                while True:

                    if row_counter == 0 and column_counter == 0:
                        break

                    if row_counter <= d:
                        if prev_2d[row_counter][column_counter] == POINT_LEFT:
                            column_counter -= 1
                            seq1_counter -= 1
                            alignment1 = seq1[seq1_counter] + alignment1
                            alignment2 = '-' + alignment2
                        elif prev_2d[row_counter][column_counter] == POINT_TOP:
                            row_counter -= 1
                            alignment1 = '-' + alignment1
                            alignment2 = seq2[row_counter] + alignment2
                        else:
                            column_counter -= 1
                            seq1_counter -= 1
                            row_counter -= 1
                            alignment1 = seq1[seq1_counter] + alignment1
                            alignment2 = seq2[row_counter] + alignment2
                    else:
                        if prev_2d[row_counter][column_counter] == POINT_LEFT:
                            column_counter -= 1
```


GeneSequencingProject4 〉 GeneSequencing.py
Current File
Git:
GeneSequencing.py
Proj4GUI.py
else:
if prev_2d[row_counter][column_counter] == POINT_LEFT:
column_counter -= 1
seq1_counter -= 1
alignment1 = seq1[seq1_counter] + alignment1
alignment2 = '-' + alignment2
elif prev_2d[row_counter][column_counter] == POINT_TOPRIGHT:
row_counter -= 1
column_counter += 1
alignment1 = '-' + alignment1
alignment2 = seq2[row_counter] + alignment2
else: # this functions as the diagonal sub/match now
seq1_counter -= 1
row_counter -= 1
alignment1 = seq1[seq1_counter] + alignment1
alignment2 = seq2[row_counter] + alignment2
score = table_2d[-1][-1]
alignment1 = alignment1[:100]
alignment2 = alignment2[:100]
else:
# initialize table - first value is rows, second value is columns
table_2d = []
prev_2d = []
# Values for the prev array:
POINT_LEFT = 0
POINT_TOP = 1
POINT_TOPLEFT = 2
# This initializes the first row of the table
table_2d.append([])
prev_2d.append([])
counter = 0
GeneSequencing 〉 align() 〉 else
Git
Python Packages
TODO
Python Console
Problems
Terminal
Services
338:88 LF UTF-8 Tab* Python 3.11 (GeneSequencingProject4) main


GeneSequencingProject4 〉 GeneSequencing.py
Current File
Git:
GeneSequencing.py
Proj4GUI.py
for i in range(align_length + 1):
if i > len(seq1):
break
table_2d[0].append(counter)
counter += 5
prev_2d[0].append(POINT_LEFT)
counter = 5
# This double for gives us O(mn) where m is the length of one string and n is the length of the second
for i in range(align_length + 1):
if i > len(seq2):
break
if i == 0:
continue
table_2d.append([])
prev_2d.append([])
for j in range(align_length + 1):
if j > len(seq1):
break
# This initializes the first column of the table
if j == 0:
table_2d[i].append(counter)
counter += 5
prev_2d[i].append(POINT_TOP)
continue
# comparing values within the table as they get populated
left = table_2d[i][j - 1] + 5
top = table_2d[i - 1][j] + 5
GeneSequencing 〉 align() 〉 else
Git
Python Packages
TODO
Python Console
Problems
Terminal
Services
338:88 LF UTF-8 Tab* Python 3.11 (GeneSequencingProject4) main



```
                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

                # Priority order: left, top, top-left
                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)
                        prev_2d[i].append(POINT_TOPLEFT)
                elif top <= top_left:
                    table_2d[i].append(top)
                    prev_2d[i].append(POINT_TOP)
                else:
                    table_2d[i].append(top_left)
                    prev_2d[i].append(POINT_TOPLEFT)

        alignment1 = ""
        alignment2 = ""
        row_counter = len(table_2d) - 2
        column_counter = len(table_2d[0]) - 2

        # Start with the initial character in the bottom right corner
        alignment1 = seq1[column_counter] + alignment1
        alignment2 = seq2[row_counter] + alignment2

        # Traverse the prev array from end back to beginning to build the strings
        # O(n) where n is the length of the longer string (or the alignment length)
        while True:

            if row_counter == 0 and column_counter == 0:
```



```
        alignment1 = seq1[column_counter] + alignment1
        alignment2 = seq2[row_counter] + alignment2

        # Traverse the prev array from end back to beginning to build the strings
        # O(n) where n is the length of the longer string (or the alignment length)
        while True:

            if row_counter == 0 and column_counter == 0:
                break

            if prev_2d[row_counter][column_counter] == POINT_LEFT:
                column_counter -= 1
                alignment1 = seq1[column_counter] + alignment1
                alignment2 = '-' + alignment2
            elif prev_2d[row_counter][column_counter] == POINT_TOP:
                row_counter -= 1
                alignment1 = '-' + alignment1
                alignment2 = seq2[row_counter] + alignment2
            else:
                column_counter -= 1
                row_counter -= 1
                alignment1 = seq1[column_counter] + alignment1
                alignment2 = seq2[row_counter] + alignment2

        score = table_2d[-1][-1]
        alignment1 = alignment1[:100]
        alignment2 = alignment2[:100]

    return {'align_cost': score, 'seqi_first100': alignment1, 'seqj_first100': alignment2}
```