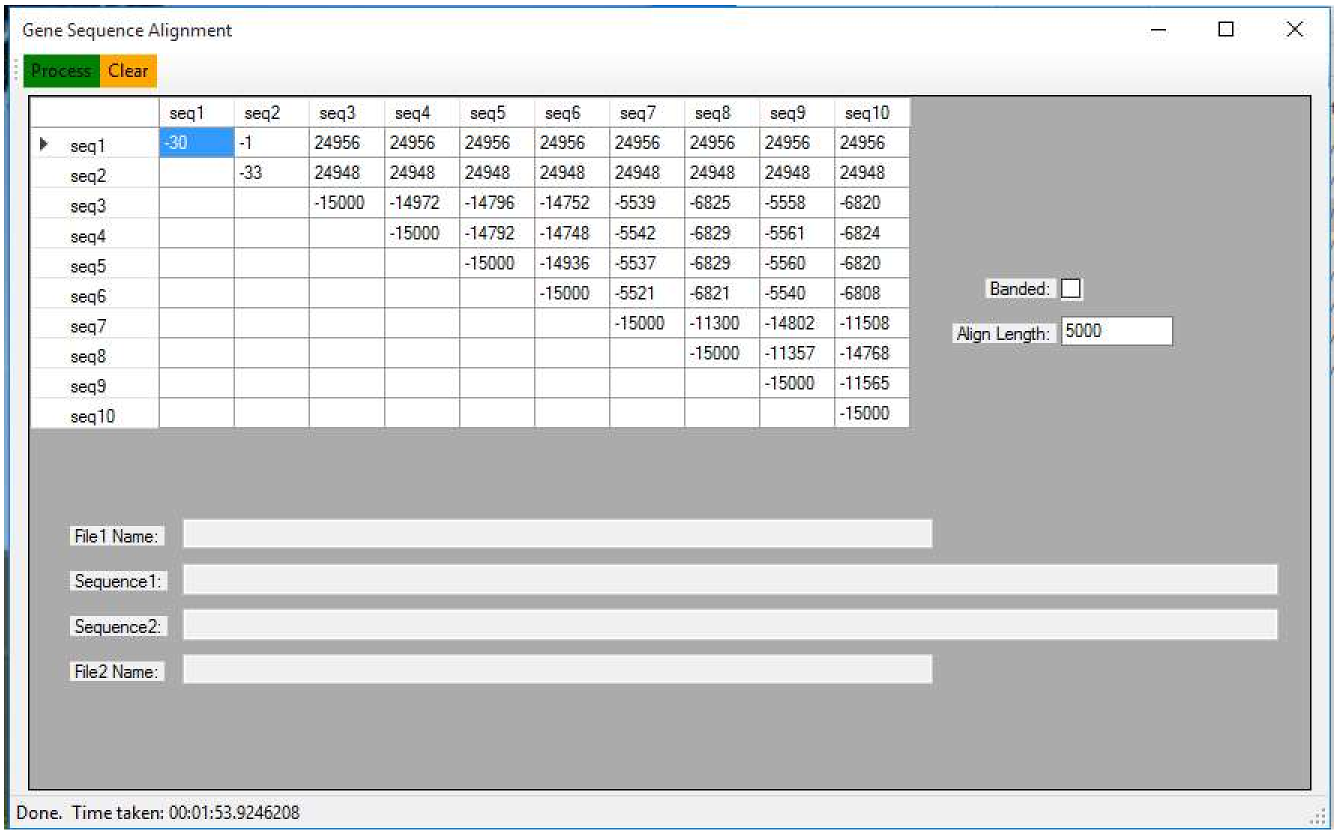
Taylor Cowley
CS 312 Lab 4: Gene Sequencing
November 03, 2016

1. Explain the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your code.
    a. [10 points] Your analysis should show that your unrestricted algorithm is at most *O*(*nm*) time and space.
    When doing the table calculation, we march through from 0 to A's length (n); each time going calculations from 0 to B's length (m). Because of this double for-loop action, the unrestricted algorithm is O(nm) in time. Because we store every result in a table of dimension n*m, the space complexity is also O(n*m).
    b. [10 points] Your analysis should show that your banded algorithm is at most *O*(*n*+*m*) time and *O*(*nm*) space.
    When doing the table calculation for the banded algorithm, we still go through 0 to A's length, but doing only 7 calculations with B each time. But because the difference in length from A and B can only be (at the maximum) 4, we get O(n+m) time complexity. We still, however, initialize a table that is n*m, hence taking up O(n*m) space complexity.
2. [10 points] Write a paragraph that explains how your alignment extraction algorithm works, including the backtrace.
    a. **We do the standard Needleman-Wunsch algorithm, with a cost of 5 for any "gap," or insert/delete in the sequence, a cost of 1 for any substitutions, and a "cost" (really, a benefit) of -3 for a matching pattern. To compute the total alignment cost, dynamic programming is used to construct a table showing all the costs of aligning the two strings in every possible way. Any given character of the combined sequence is produced by either taking the character from both sequences- substituting one for the other or finding a match, or by taking a character from one sequence and forcing a gap in the other. This corresponds to coming from the upper-left diagonal or from the top or left side, respectively. Every node in the table stores the minimum cost of arriving at this alignment, as well as a pointer to its parent node- the one from which it achieves the current minimum cost. At the end, we take the last node in the bottom right corner, and follow its lineage- either up, left, or diagonally, until we reach the ultimate parent node at the top left corner. This corresponds with traversing our aligned strings from the end to the beginning.**
3. [20 points] Include a "results" section showing both a screen-shot of your 10x10 score matrix for the unrestricted algorithm with align length *k* = 5000 and a screen-shot of your 10x10 score matrix for the banded algorithm with align length *k* = 15000.
    Results:

a. [screenshot of unrestricted k=5000]



Gene Sequence Alignment

	seq1	seq2	seq3	seq4	seq5	seq6	seq7	seq8	seq9	seq10
seq1	-30	-1	24956	24956	24956	24956	24956	24956	24956	24956
seq2		-33	24948	24948	24948	24948	24948	24948	24948	24948
seq3			-15000	-14972	-14796	-14752	-5539	-6825	-5558	-6820
seq4				-15000	-14792	-14748	-5542	-6829	-5561	-6824
seq5					-15000	-14936	-5537	-6829	-5560	-6820
seq6						-15000	-5521	-6821	-5540	-6808
seq7							-15000	-11300	-14802	-11508
seq8								-15000	-11357	-14768
seq9									-15000	-11565
seq10										-15000

Banded: ☐

Align Length: 5000

File1 Name:

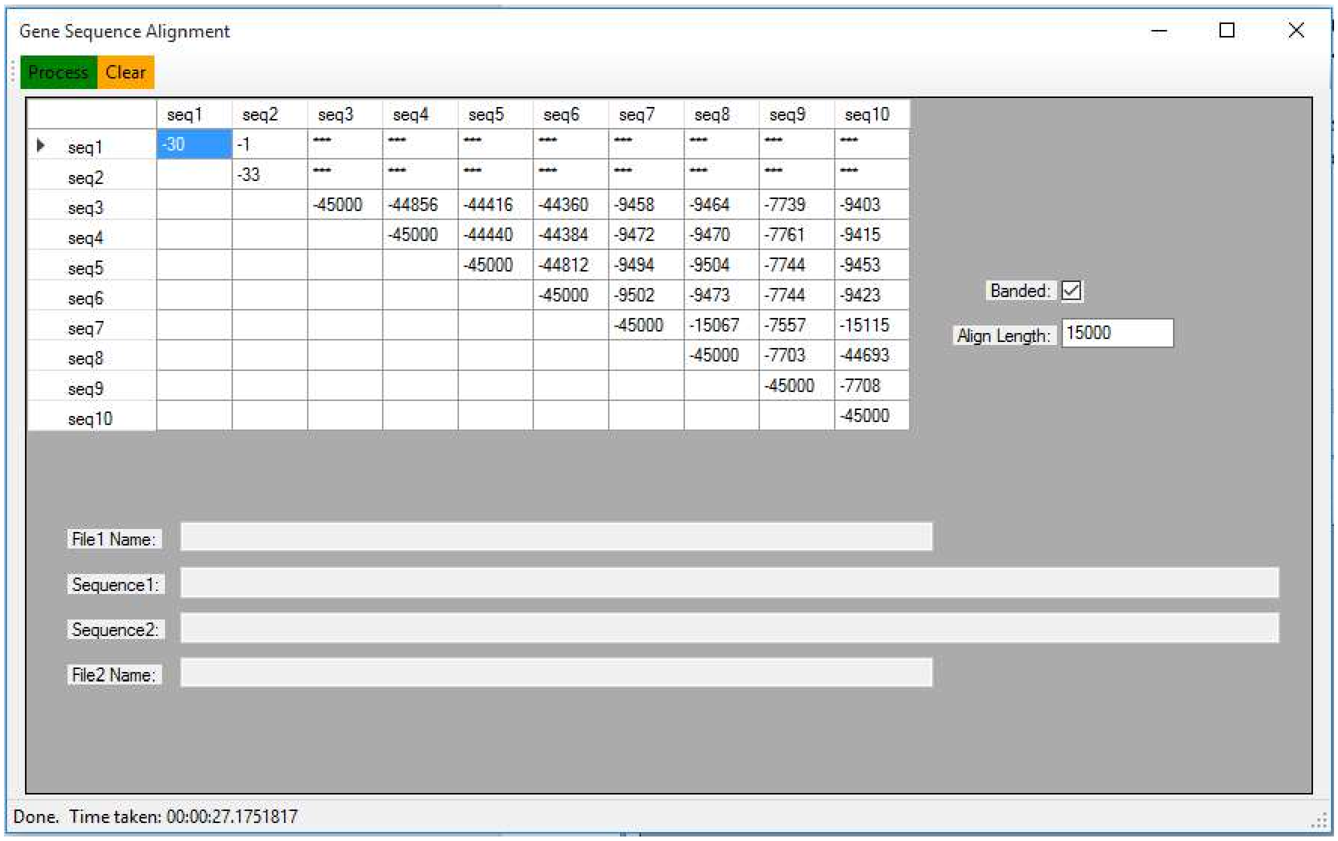
Sequence1:

Sequence2:

File2 Name:

Done. Time taken: 00:01:53.9246208

b. [screenshot of banded k=15000]



Gene Sequence Alignment

	seq1	seq2	seq3	seq4	seq5	seq6	seq7	seq8	seq9	seq10
seq1	-30	-1	***	***	***	***	***	***	***	***
seq2		-33	***	***	***	***	***	***	***	***
seq3			-45000	-44856	-44416	-44360	-9458	-9464	-7739	-9403
seq4				-45000	-44440	-44384	-9472	-9470	-7761	-9415
seq5					-45000	-44812	-9494	-9504	-7744	-9453
seq6						-45000	-9502	-9473	-7744	-9423
seq7							-45000	-15067	-7557	-15115
seq8								-45000	-7703	-44693
seq9									-45000	-7708
seq10										-45000

Banded: ✓

Align Length: 15000

File1 Name:

Sequence1:

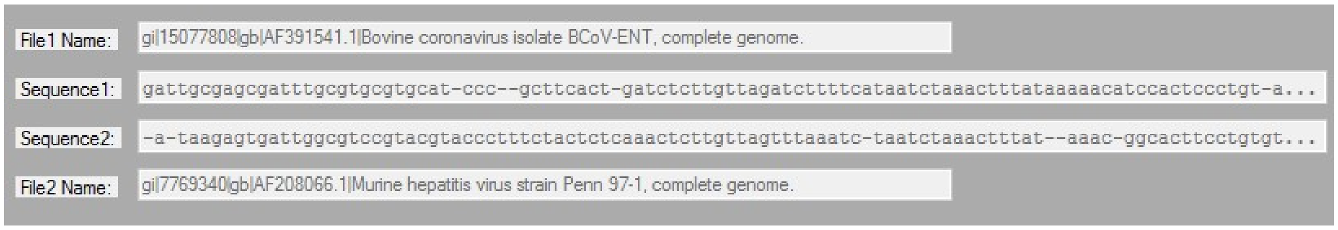
Sequence2:

File2 Name:

Done. Time taken: 00:00:27.1751817

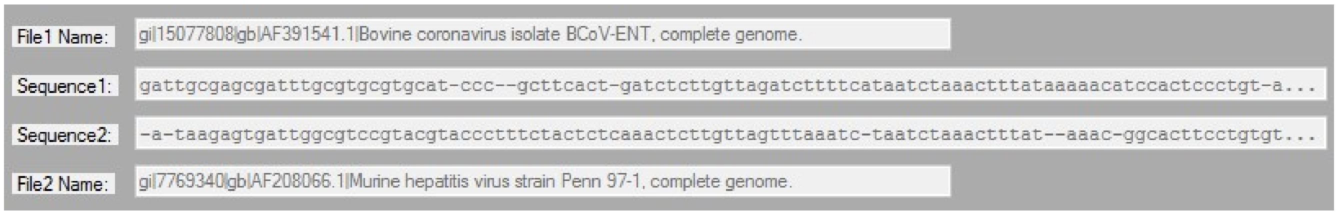
4. [10 points] Include in the “results” section the extracted alignment for the first 100 characters of sequences #3 and #10 (counting from 1), computed using the unrestricted algorithm with *k* = 5000. Display the sequences in a side-by-side fashion in such a way that matches, substitutions, and insertions/deletions 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 *k* = 15000.

Unrestricted algorithm #3 and #10, first 100 characters of aligned sequences



```
File1 Name:  gi|15077808|gb|AF391541.1|Bovine coronavirus isolate BCoV-ENT, complete genome.
Sequence1:   gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a...
Sequence2:   -a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt...
File2 Name:  gi|7769340|gb|AF208066.1|Murine hepatitis virus strain Penn 97-1, complete genome.
```

Banded algorithm #3 and #10, first 100 characters of aligned sequences



```
File1 Name:  gi|15077808|gb|AF391541.1|Bovine coronavirus isolate BCoV-ENT, complete genome.
Sequence1:   gattgcgagcgatttgcgtgcgtgcat-ccc--gcttcact-gatctcttgttagatcttttcataatctaaactttataaaaacatccactccctgt-a...
Sequence2:   -a-taagagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttat--aaac-ggcacttcctgtgt...
File2 Name:  gi|7769340|gb|AF208066.1|Murine hepatitis virus strain Penn 97-1, complete genome.
```

5. [30 points] Attach your commented source code for both your unrestricted and banded algorithms.

See next several pages

```
using System;
using System.Collections.Generic;
using System.Text;

namespace GeneticsLab
{
    class PairWiseAlign
    {
        int MaxCharactersToAlign;

        public PairWiseAlign()
        {
            // Default is to align only 5000 characters in each sequence.
            this.MaxCharactersToAlign = 5000;
        }

        public PairWiseAlign(int len)
        {
            // Alternatively, we can use an different length; typically used with the
              banded option checked.
            this.MaxCharactersToAlign = len;
        }

        /// <summary>
        /// this is the function you implement.
        /// </summary>
        /// <param name="sequenceA">the first sequence</param>
        /// <param name="sequenceB">the second sequence, may have length not equal to
          the length of the first seq.</param>
        /// <param name="banded">true if alignment should be band limited.</param>
        /// <returns>the alignment score and the alignment (in a Result object) for
          sequenceA and sequenceB.  The calling function places the result in the
          dispay appropriately.
        ///
        public ResultTable.Result Align_And_Extract(GeneSequence sequenceA,
          GeneSequence sequenceB, bool banded)
        {
            //
              -----------------------------------------------------------------------
              ---------------------
            // Setup section. Also returns without calculation for banded analysis of
              sequences which will be impossible
            // O(1)

            int maxlength = MaxCharactersToAlign;

            const int indel = 5;
            const int sub = 1;
            const int match = -3;

            //Console.WriteLine("Sequence a: " + sequenceA.Sequence);
            //Console.WriteLine("Sequence b: " + sequenceB.Sequence);
            int lengthA, lengthB;
            if (sequenceA.Sequence.Length > maxlength) {
                lengthA = maxlength;
            } else {
                lengthA = sequenceA.Sequence.Length;
            }

            if (sequenceB.Sequence.Length > maxlength)
            {
                lengthB = maxlength;
            }
            else
            {
                lengthB = sequenceB.Sequence.Length;
            }


            ResultTable.Result result = new ResultTable.Result();
            int score;                                                  // place 
              your computed alignment score here
            string[] alignment = new string[2];                         // place 
              your two computed alignments here
            // these "alignments" are just the two strings, with "-" added where an 
              insertion/deletion has occured. simple enough.

            // ********* these are placeholder assignments that you'll replace with 
              your code  *******
            score = 0;
            alignment[0] = "";
            alignment[1] = "";
            //                                                                      
              ***********************************************************************
              ***************

            // We will not be able to get a banded result if the lengths differ by 
              more than 3
            if (banded && Math.Abs(lengthA - lengthB) > 3) {
                score = int.MaxValue;
                result.Update(score, "No Alignment Possible", "No Alignment 
                  Possible");
                return result;
            }




            // Sequence length because it has to fit the string AND a space for the 
              "empty string" at the beginning
            node[,] calcTable = new node[lengthA+1 , lengthB+1];
            // Go through the entire table to calculate the things.
            int a, b = 0;

            // do top left corner first
            calcTable[0, 0] = new node(-1, -1, 0);


            // END setup section
            //
              ------------------------------------------------------------------------
              -----------------------------------

            if (banded)
            {
                //--------------------------------------------------------------------
                  -----------------------------------
                // Banded table scores calculation.
                // It goes through the length of A, each time doing 7 calculations for
                   B.
                // O(7n) - where n is the length of A
                // Also note that the length difference between A and B is MAX 4, so O
                  (n)

                // Do entire top row
                for (a = 1; a < 4; a++)
                {
                    calcTable[a, 0] = new node(a - 1, 0, a * indel);
                }

                // And entire left row
                for (b = 1; b < 4; b++)
                {
                    calcTable[0, b] = new node(0, b - 1, b * indel);
                }

                // and the rest
                for (a = 1; a < lengthA + 1; a++)
                {
                    for (b = a - 3; b < a + 4; b++)
                    { // Can only calculate in the band
                        if (b < 1 || b > lengthB) {
                            // Can't be having those System.IndexOutOfRangeExceptions,
                      can we?
                            continue;
                        }

                        int topCost, leftCost, diagCost;

                        // Calculate cost of coming from top
                        if (calcTable[a, b - 1] == null)
                        {
```

```
                    topCost = int.MaxValue;         // null is very bad.
                }
                else
                {
                    topCost = calcTable[a, b - 1].score + indel;        //     ⤶
              coming from top is an insert/delete
                }



                // Calculate cost of coming from left
                if (calcTable[a - 1, b] == null)
                {
                    leftCost = int.MaxValue;        // again, null is very bad
                }
                else
                {
                    leftCost = calcTable[a - 1, b].score + indel;       //     ⤶
              coming from left is also an insert/delete
                }

                // Calculate cost of coming from the diagonal
                // We don't worry about nulls here, because they are          ⤶
              impossible
                if (sequenceA.Sequence[a - 1] == sequenceB.Sequence[b - 1])
                {       // If the two strings match at this character
                    diagCost = calcTable[a - 1, b - 1].score + match;   //     ⤶
              coming from diagonal on a match!
                }
                else
                {
                    diagCost = calcTable[a - 1, b - 1].score + sub;     //     ⤶
              coming from diagonal on a substitution
                }


                // Now to make our table entry
                if (diagCost <= leftCost && diagCost <= topCost)
                {          // Diagonal is cheapest
                    calcTable[a, b] = new node(a - 1, b - 1, diagCost);
                }
                else if (leftCost <= diagCost && leftCost <= topCost)
                {   // Left is cheapest
                    calcTable[a, b] = new node(a - 1, b, leftCost);
                }
                else
                {                                                       // Top is⤶
               cheapest
                    calcTable[a, b] = new node(a, b - 1, topCost);
                }
            }
        }
```

```
                // END Banded table scores calculation
                //------------------------------------------------------------------↩
                  ------------
            } else { // ends our if section for banded; starts our section to        ↩
              calculate unbanded


                //                                                                  ↩
                  ------------------------------------------------------------------↩
                  --------------
                // Start unbanded table scores calculation
                // Goes through the length of A, each time going through the length of↩
                   B.
                // Therefore O(n*m), where n is the length of A, and m is the length ↩
                  of B.


                // Do the entire top row first
                for (a = 1; a < lengthA + 1; a++)
                {
                    calcTable[a, 0] = new node(a - 1, 0, a * indel);
                }
                // and the entire left row
                for (b = 1; b < lengthB + 1; b++)
                {      // Skipping the first one which was already done
                    calcTable[0, b] = new node(0, b - 1, b * indel);
                }

                // And the rest
                for (a = 1; a < lengthA + 1; a++)
                {
                    for (b = 1; b < lengthB + 1; b++)
                    {

                        // Calculate cost of coming from top
                        int topCost = calcTable[a, b - 1].score + indel;        //    ↩
                      coming from top is an insert/delete

                        // Calculate cost of coming from left
                        int leftCost = calcTable[a - 1, b].score + indel;       //    ↩
                      coming from left is also an insert/delete

                        // Calculate cost of coming from the diagonal
                        int diagCost;
                        if (sequenceA.Sequence[a - 1] == sequenceB.Sequence[b - 1])
                        {       // If the two strings match at this character
                            diagCost = calcTable[a - 1, b - 1].score + match;   //    ↩
                      coming from diagonal on a match!
                        }
                        else
                        {
                            diagCost = calcTable[a - 1, b - 1].score + sub;     //    ↩
                      coming from diagonal on a substitution
                        }


                        // Now to make our table entry
                        if (diagCost <= leftCost && diagCost <= topCost)
                        {          // Diagonal is cheapest
                            calcTable[a, b] = new node(a - 1, b - 1, diagCost);
                        }
                        else if (leftCost <= diagCost && leftCost <= topCost)
                        {   // Left is cheapest
                            calcTable[a, b] = new node(a - 1, b, leftCost);
                        }
                        else
                        {                                                    // Top is ⤶
                       cheapest
                            calcTable[a, b] = new node(a, b - 1, topCost);
                        }
                    }
                }
                //END unbanded calculation
                //-----------------------------------------------------------------⤶
                  -------------------------
            }   // This ends the difference between banded and unbanded calculation


            //                                                                    ⤶
              ------------------------------------------------------------------⤶
              --------------------------------------------------
            // At this point our scores table should be complete. Now we just take the⤶
               final node and walk back to the beginning with it
            // The length of this string is the larger of the length of A and the     ⤶
              length of B
            // O(max(n,m)) where n is A's length and m is B's length
            a = lengthA;
            b = lengthB;

            StringBuilder strA = new StringBuilder(maxlength);
            StringBuilder strB = new StringBuilder(maxlength);



            // Go until we hit either the top or left row/column
            while (a != 0 && b != 0) {
                int parent_a = calcTable[a, b].parent_x;
                int parent_b = calcTable[a, b].parent_y;
                if(parent_a < a) {                  // To get here, we came from left
                    //alignment[0].Insert(0,sequenceA.Sequence[a-1].ToString         ⤶
                      ());                       // Which means we used a char of     ⤶
                      sequence a
                    strA.Insert(0, sequenceA.Sequence[a - 1].ToString());
                    if(parent_b < b) {                // Also came from top = came from⤶
```

```
                         diagonal
                          //alignment[1].Insert(0, sequenceB.Sequence[b-1].ToString
                        ());                 // Whicn means we ALSO used a char of
                        sequence b
                          strB.Insert(0, sequenceB.Sequence[b - 1].ToString());
                      } else {                                       // Only came from
                        left
                          //alignment[1].Insert(0,
                        "-");                                               // Used a
                        char of sequence a but not b
                          strB.Insert(0, "-");
                      }
                  } else {
                      //alignment[0].Insert(0, "-");                     // Did not come
                        from left nor diagonal; must have been from top
                      strA.Insert(0, "-");
                      //alignment[1].Insert(0, sequenceB.Sequence[b-1].ToString());
                      strB.Insert(0, sequenceB.Sequence[b - 1].ToString());
                  }
                  a = parent_a;
                  b = parent_b;
              }

              // Assume we hit the left column. This means a = 0 and b is getting
                smaller
              // This means we have used all of sequence a already.
              while (b != 0) {
                  int parent_a = 0;
                  int parent_b = calcTable[a, b].parent_y;

                  //alignment[0].Insert(0,"-");     // Already used all of a, so it has
                    gaps at the beginning
                  strA.Insert(0, "-");
                  //alignment[1].Insert(0, sequenceB.Sequence[b-1].ToString());
                  strB.Insert(0, sequenceB.Sequence[b - 1].ToString());

                  a = parent_a;
                  b = parent_b;
              }

              // Assume we hit the top row. This means b = 0 and a is getting smaller
              // This means we have used all of sequence b already.
              while (a != 0)
              {
                  int parent_a = calcTable[a, b].parent_x;
                  int parent_b = 0;

                  //alignment[0] = alignment[0].Insert(0, sequenceA.Sequence[a -
                    1].ToString());     // Already used all of a, so it has gaps at the
                    beginning
                  strA.Insert(0, sequenceA.Sequence[a - 1].ToString());
                  //alignment[1] = alignment[1].Insert(0, "-");
                  strB.Insert(0, "-");

                  a = parent_a;
                  b = parent_b;
              }

              // END final string calculation
              //-----------------------------------------------------------------------
                ---------------------------


              //-----------------------------------------------------------------------
                ---------------------------
              // From here on out, we are just wrapping up the calculations and
                returning the results
              // O(1)


              // If we reach here, we should have traced our strings back to the
                beginning. The alignment strings should be all good and we just need to
                get the score
              score = calcTable[lengthA, lengthB].score;
              result.Update(score, strA.ToString(), strB.ToString());//alignment
                [0],alignment[1]);                  // bundling your results into the
                right object type
              return(result);
          }
      }


      class node {
          public int parent_x;
          public int parent_y;
          public int score;

          public node(int x, int y, int s)
          {
              parent_x = x;
              parent_y = y;
              score = s;
          }
      }
  }
```