Gene Sequencing Report

Backlink: @P-CS 312

Thursday, November 9, 2017 5:11 PM

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

A - Unrestricted is at most O(nm) time and space.

I tried to be meticulous and reuse as much code as I could, so there isn't much different between my two algorithms, apart from their loops.

Time complexity is O(nm) as well, because I iterate over the entire cost matrix and compute each cell. There are nm cells, so it runs in O(nm) time.

Additionally, there is an O(n + m) initialization computation that could be removed if I were to optimize. It just fills in the first row and first column with values increasing by 5 each step.

```
initializeMatrices(matrix, prev, rows, cols);

if (!banded) unrestricted(matrix, prev, rows, cols, sequenceA, sequenceB);
```

Runs in O(n + m)

```
private void initializeMatrices(int[,] matrix, int[,] prev, int rows, int cols)
{
    int val = 0;
    for (int i = 0; i < cols; i++)
    {
        matrix[0, i] = val;
        prev[0, i] = LEFT;
        val += 5;
    }
    val = 5;
    for (int j = 1; j < rows; j++)
    {
        matrix[j, 0] = val;
        prev[j, 0] = UP;
        val += 5;
    }
}
```

Runs in O(nm)

```
private void unrestricted(int[,] matrix, int[,] prev, int rows, int cols, GeneSequence sequenceA, GeneSequence
sequenceB)
{
  for (int i = 1; i < rows; i++)   // O(n)
  {
    for (int j = 1; j < cols; j++)    // O(m)
    {
      matrix[i, j] = computeVal(matrix, prev, i, j, sequenceA, sequenceB);
    }
  }
}
```

```
private int computeVal(int[,] matrix, int[,] prev, int row, int col, GeneSequence sequenceA, GeneSequence sequenceB)
{
  char letterA = sequenceA.Sequence[row - 1];
  char letterB = sequenceB.Sequence[col - 1];
  int diagVal = letterA == letterB ? -3 : 1;
  int indelVal = 5;

  if (matrix[row - 1, col - 1] + diagVal <= matrix[row - 1, col] + indelVal && matrix[row - 1, col - 1] + diagVal <=
matrix[row, col - 1] + indelVal)
  {
    prev[row, col] = DIAG;
    return matrix[row - 1, col - 1] + diagVal;
  }
  else if (matrix[row - 1, col] + indelVal < matrix[row - 1, col - 1] + diagVal && matrix[row - 1, col] + indelVal <=
matrix[row, col - 1] + indelVal)
  {
    prev[row, col] = UP;
    return matrix[row - 1, col] + indelVal;
  }
  else
  {
    prev[row, col] = LEFT;
    return matrix[row, col - 1] + indelVal;
```

```
    }
}
```

Both algorithms uses two 2D arrays that have n rows and m columns at most. One is used to store the intermediate cost score, the other to keep track of the path the cost scores take. This gives O(nm) space complexity.

The int[,] prev array stores values indicating what cost was accepted. An indel from above, an indel from the left, or a match/mismatch value from the diagonal.

```
int rows = maxLengthVal < sequenceA.Sequence.Length + 1? maxLengthVal : sequenceA.Sequence.Length + 1;
int cols = maxLengthVal < sequenceB.Sequence.Length + 1? maxLengthVal : sequenceB.Sequence.Length + 1;

int[,] matrix = new int[rows,cols];
int[,] prev = new int[rows,cols];
```

B - Banded is at most O(n+m) time and O(nm) space

Time complexity is O(n+m) because the banded algorithm just follows the diagonal down the matrix. As long as |n - m| <= bandwidth, the two sequences can be aligned. There are n cells in the diagonal, and m additional cells included in the bandwidth. Thus, O(n+m).

It is also O(n+m) because I use the same initialization function to populate the first row and first column with indel values.

```
private void bandedAlg(int[,] matrix, int[,] prev, int rows, int cols, GeneSequence sequenceA, GeneSequence
sequenceB)
{
   if (Math.Abs(rows - cols) > 3)
   {
      matrix[rows - 1, cols - 1] = int.MaxValue;
      return;
   }

   int i, j;
   i = j = 0;
   int maxD = rows > cols ? rows : cols;
   for (i = 1; i < maxD; i++)
   {
      for (j = 0; j < 4; j++)
      {
         if (i + j < cols && i < rows) matrix[i, i + j] = computeVal(matrix, prev, i, i + j, sequenceA, sequenceB);
         if (i + j < rows && i < cols) matrix[i + j, i] = computeVal(matrix, prev, i + j, i, sequenceA, sequenceB);
      }
   }
}

private int computeVal(int[,] matrix, int[,] prev, int row, int col, GeneSequence sequenceA, GeneSequence sequenceB)
{
   char letterA = sequenceA.Sequence[row - 1];
   char letterB = sequenceB.Sequence[col - 1];
   int diagVal = letterA == letterB ? -3 : 1;
   int indelVal = 5;

   if (matrix[row - 1, col - 1] + diagVal <= matrix[row - 1, col] + indelVal && matrix[row - 1, col - 1] + diagVal <=
matrix[row, col - 1] + indelVal)
   {
      prev[row, col] = DIAG;
      return matrix[row - 1, col - 1] + diagVal;
   }
   else if (matrix[row - 1, col] + indelVal < matrix[row - 1, col - 1] + diagVal && matrix[row - 1, col] + indelVal <=
matrix[row, col - 1] + indelVal)
   {
      prev[row, col] = UP;
      return matrix[row - 1, col] + indelVal;
   }
   else
   {
      prev[row, col] = LEFT;
      return matrix[row, col - 1] + indelVal;
   }
}
```

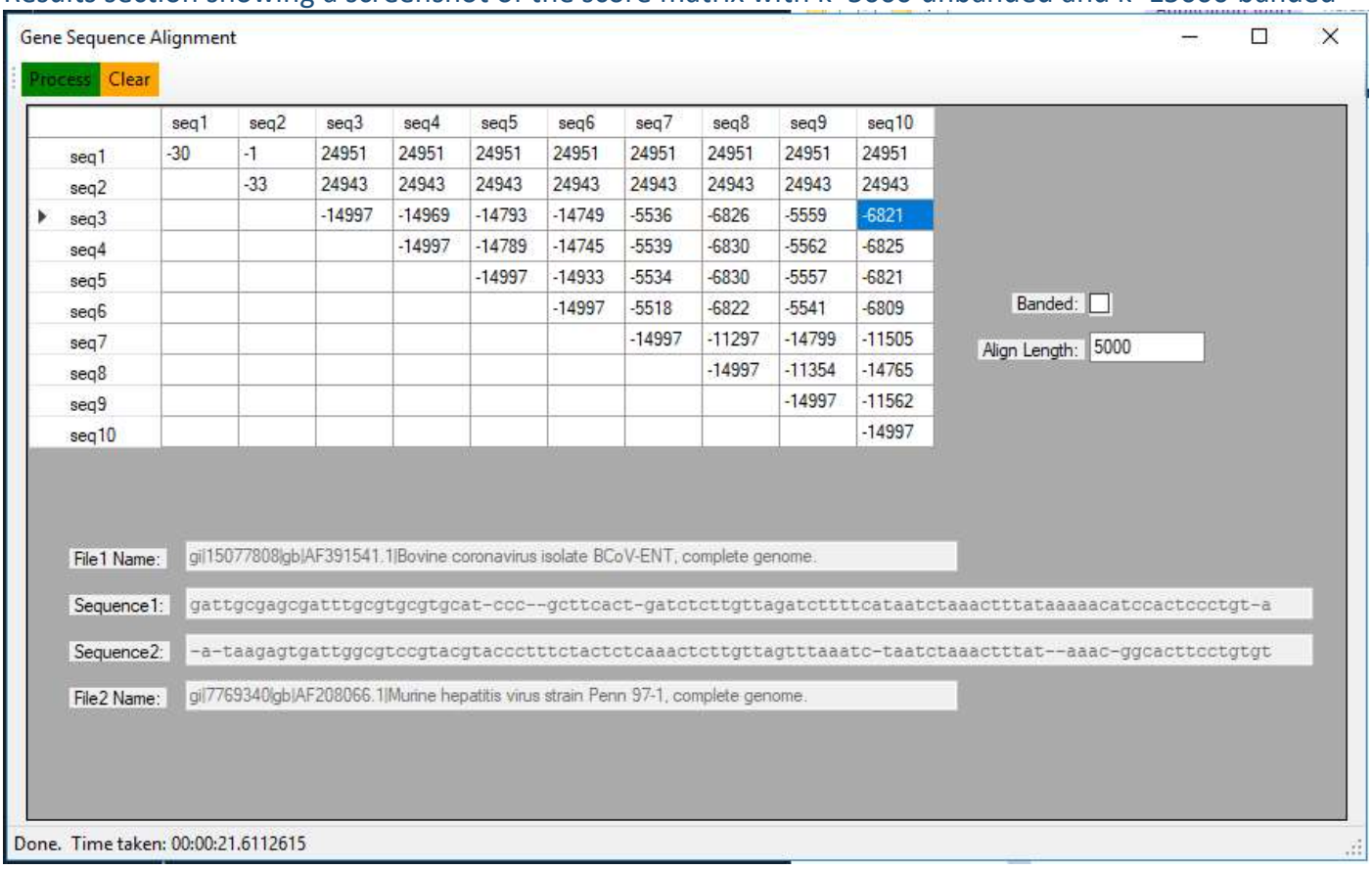
Both algorithms uses two 2D arrays that have n rows and m columns at most. One is used to store the intermediate cost score, the other to keep track of the path the cost scores take. This gives O(nm) space complexity.
 	
The int[,] prev array stores values indicating what cost was accepted. An indel from above, an indel from the left, or a match/mismatch value from the diagonal.
	int rows = maxLengthVal < sequenceA.Sequence.Length + 1? maxLengthVal : sequenceA.Sequence.Length + 1;
	int cols = maxLengthVal < sequenceB.Sequence.Length + 1? maxLengthVal : sequenceB.Sequence.Length + 1;
	
	int[,] matrix = new int[rows,cols];
	int[,] prev = new int[rows,cols];

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

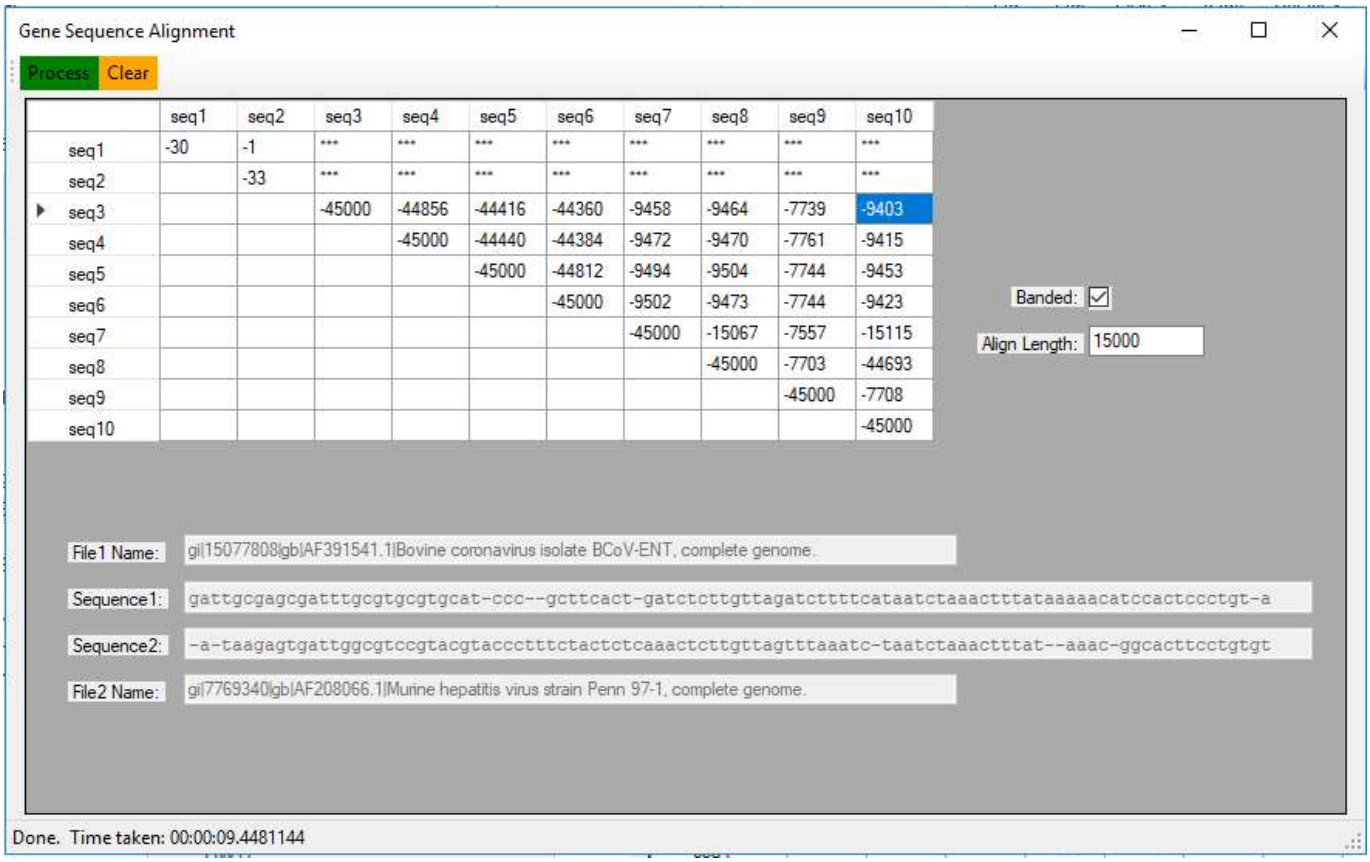
My extraction algorithm takes in the int[,] prev array shown above in the space complexity of both algorithms, the number of rows and columns, the score, and the two sequences as strings. This cells in this matrix point to where the corresponding cells in the cost matrix were derived from. A UP (1:int) indicates that it was an indel from above, a DIAG (2:int) indicates that it was from the diagonal, and a LEFT (3:int) indicates that it was an indel from the left. If the score is int.MaxValue, I set both alignment strings to "No Alignment Possible" and return. Otherwise I start at the bottom right cell in the matrix, using the rows and columns values passed as parameters, and check the value. If it stored a DIAG value, I grab the char from each sequence at the corresponding index and insert it to the beginning of the corresponding alignment string, and then decrement both row and col, moving up to the diagonal. Else, if it stores an UP value, I grab the char from the sequence from the left of the matrix, insert a '-' in the alignment for the top sequence, and decrement the row. If it stores a LEFT value, I do the inverse (save the top, insert a '-', decrement the column).

This effectively moves me up the lowest cost path by indicating which cell the current cell was computed from.

Results section showing a screenshot of the score matrix with k=5000 unbanded and k=15000 banded



	seq1	seq2	seq3	seq4	seq5	seq6	seq7	seq8	seq9	seq10
seq1	-30	-1	24951	24951	24951	24951	24951	24951	24951	24951
seq2		-33	24943	24943	24943	24943	24943	24943	24943	24943
seq3			-14997	-14969	-14793	-14749	-5536	-6826	-5559	-6821
seq4				-14997	-14789	-14745	-5539	-6830	-5562	-6825
seq5					-14997	-14933	-5534	-6830	-5557	-6821
seq6						-14997	-5518	-6822	-5541	-6809
seq7							-14997	-11297	-14799	-11505
seq8								-14997	-11354	-14765
seq9									-14997	-11562
seq10										-14997



Source:

```
const int UP = 1;
const int DIAG = 2;
const int LEFT = 3;
/// <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)
{
   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
   alignment[0] = alignment[1] = "";

   int maxLengthVal = banded ? 15001 : MaxCharactersToAlign;
   // If the sequences are longer than the desired alignment length, align only the desired amount.
   int rows = maxLengthVal < sequenceA.Sequence.Length + 1? maxLengthVal : sequenceA.Sequence.Length + 1;
   int cols = maxLengthVal < sequenceB.Sequence.Length + 1? maxLengthVal : sequenceB.Sequence.Length + 1;
   // Create the cost matrix and the matrix used to track the path.
   int[,] matrix = new int[rows,cols];
   int[,] prev = new int[rows,cols];
   initializeMatrices(matrix, prev, rows, cols);

   // If it's not banded, do the unrestriced algorithm. Otherwise do banded.
   if (!banded) unrestricted(matrix, prev, rows, cols, sequenceA, sequenceB);
   else bandedAlg(matrix, prev, rows, cols, sequenceA, sequenceB);

   // The score is stored in the last cell.
   score = matrix[rows - 1, cols - 1];
   // Find the alignment strings by using the path stored in prev
   findAlignments(alignment, prev, rows, cols, score, sequenceA.Sequence, sequenceB.Sequence);
```

```
    // If the strings are too long to display, just display 100 characters.
    if (alignment[0].Length > 100) alignment[0] = alignment[0].Substring(0, 100);
    if (alignment[1].Length > 100) alignment[1] = alignment[1].Substring(0, 100);

    result.Update(score,alignment[0],alignment[1]);              // bundling your results into the right object type
    return(result);
}

private void findAlignments(string[] alignment, int[,] prev, int rows, int cols, int score, string sequenceA, string sequenceB)
{
    // There was no alignment made. Return.
    if (score == int.MaxValue)
    {
        alignment[0] = alignment[1] = "No Alignment Possible";
        return;
    }

    int row, col;
    row = rows - 1;     // Index of last row
    col = cols - 1;     // Index of last column
    // Continue building the strings until we reach the first cell
    while (row > 0 && cols > 0)
    {
        // Get path value.
        switch (prev[row, col])
        {
            case DIAG:
                // Was either a match or mismatch
                alignment[0] = alignment[0].Insert(0, sequenceA.Substring(row-1, 1));
                alignment[1] = alignment[1].Insert(0, sequenceB.Substring(col-1, 1));
                row--;
                col--;
                break;
            case UP:
                // indel from above
                // keep left, insert - in top
                alignment[0] = alignment[0].Insert(0, sequenceA.Substring(row-1, 1));
                alignment[1] = alignment[1].Insert(0, "-");
                row--;
                break;
            case LEFT:
                // indel from the left
                // keep top, insert - in left
                alignment[0] = alignment[0].Insert(0, "-");
                alignment[1] = alignment[1].Insert(0, sequenceB.Substring(col-1, 1));
                col--;
                break;
        }
    }
}

private void bandedAlg(int[,] matrix, int[,] prev, int rows, int cols, GeneSequence sequenceA, GeneSequence sequenceB)
{
    // If the difference is greater than 3, they cannot be aligned.
    if (Math.Abs(rows - cols) > 3)
    {
        // Set the score cell to int.MaxValue to indicate that they weren't aligned.
        matrix[rows - 1, cols - 1] = int.MaxValue;
        return;
    }

    int i, j;
    i = j = 0;
    // Get the larger of the two out of the rows and columns. This is because we need to get to the bottom right.
    int maxD = rows > cols ? rows : cols;
    // This loop travels down the diagonal
    for (i = 1; i < maxD; i++)
    {
```

```
      // This loop uses an offset to compute the cell on the diagonal, 3 to the right, and 3 below.
      for (j = 0; j < 4; j++)
      {
        // Compute cell to the right
        if (i + j < cols && i < rows) matrix[i, i + j] = computeVal(matrix, prev, i, i + j, sequenceA, sequenceB);
        // Compute cell below.
        if (i + j < rows && i < cols) matrix[i + j, i] = computeVal(matrix, prev, i + j, i, sequenceA, sequenceB);
      }
    }
  }

  private int computeVal(int[,] matrix, int[,] prev, int row, int col, GeneSequence sequenceA, GeneSequence sequenceB)
  {
    // Get the two letters to compare.
    char letterA = sequenceA.Sequence[row - 1];
    char letterB = sequenceB.Sequence[col - 1];
    int diagVal = letterA == letterB ? -3 : 1;      // If they are the same, the diagonal score is -3, otherwise 1
    int indelVal = 5;

    // If the diagnoal score is the smallest of the three, store the DIAG value in the prev matrix and return the cost for the cell at matrix[row,col]
    if (matrix[row - 1, col - 1] + diagVal <= matrix[row - 1, col] + indelVal && matrix[row - 1, col - 1] + diagVal <= matrix[row, col - 1] + indelVal)
    {
      prev[row, col] = DIAG;
      return matrix[row - 1, col - 1] + diagVal;
    }
    // Else if the indel score from above is the smallest, store the UP value in the prev matrix and return the cost
    else if (matrix[row - 1, col] + indelVal < matrix[row - 1, col - 1] + diagVal && matrix[row - 1, col] + indelVal <= matrix[row, col - 1] + indelVal)
    {
      prev[row, col] = UP;
      return matrix[row - 1, col] + indelVal;
    }
    // Else the indel score from the left must be the smallest. Store it in prev and return the cost.
    else
    {
      prev[row, col] = LEFT;
      return matrix[row, col - 1] + indelVal;
    }
  }

  private void unrestricted(int[,] matrix, int[,] prev, int rows, int cols, GeneSequence sequenceA, GeneSequence sequenceB)
  {
    // For ever cell from top left to bottom right, compute the value. Compute by row.
    for (int i = 1; i < rows; i++)
    {
      for (int j = 1; j < cols; j++)
      {
        matrix[i, j] = computeVal(matrix, prev, i, j, sequenceA, sequenceB);
      }
    }
  }

  private void initializeMatrices(int[,] matrix, int[,] prev, int rows, int cols)
  {
    // Initialize the first row and first column with indel values.
    int val = 0;
    for (int i = 0; i < cols; i++)
    {
      matrix[0, i] = val;
      prev[0, i] = LEFT;
      val += 5;
    }
    val = 5;
    for (int j = 1; j < rows; j++)
    {
      matrix[j, 0] = val;
      prev[j, 0] = UP;
      val += 5;
    }
  }
```

```
// Helper function to pring matrices to the console. Helpful to debug the smaller problems before moving to bigger ones.
private void printMatrices(int[,] matrix, int[,] prev, int rows, int cols)
{
   for (int i = 0; i < rows; i++)
   {
      for (int j = 0; j < cols; j++)
      {
         Console.Write(matrix[i, j] + " ");
      }
      Console.Write("\n");
   }
   Console.Write("\n");

   for (int i = 0; i < rows; i++)
   {
      for (int j = 0; j < cols; j++)
      {
         Console.Write(prev[i, j] + " ");
      }
      Console.Write("\n");
   }
}
```