17.1 Graph Colouring

Easter 2024

Contents

17.1 Graph Colouring

This project's code has been written in C#. The code can be found in the appendix at "Appendix A: Code" (p. 5). The output can be found in "Appendix B: Output" (p. 55).

Question 1

The output can be found in "Question 1 output" (p. 55).

The methods i-iv as discussed in the project instructions have been implemented, as well as the greedy algorithm. The following tables show the number of colours used by the greedy algorithm

for each randomly generated graph in the $\mathcal{G}(70, 0.5)$ and $\mathcal{G}_3(70, 0.5)$ cases respectively.

Graph tested	Method (i)	Method (ii)	Method (iii)	Method (iv)
1	16	16	15	17
2	17	17	17	18
3	16	16	18	17
4	17	16	17	17
5	15	18	17	16
6	16	19	16	17
7	17	17	15	16
8	17	17	18	16
9	17	17	17	18
10	17	16	16	16
average	16.5	16.9	16.6	16.8

Table 1: Number of colours used to colour ten randomly generated graphs in $\mathcal{G}(70, 0.5)$.

Graph tested	Method (i)	Method (ii)	Method (iii)	Method (iv)
1	4	6	4	3
2	3	3	6	4
3	3	3	4	5
4	4	5	5	3
5	4	3	4	3
6	3	4	3	5
7	4	3	3	3
8	4	4	7	4
9	5	4	4	4
10	3	3	4	4
average	3.7	3.8	4.4	3.8

Table 2: Number of colours used to colour ten randomly generated graphs in $\mathcal{G}_3(70, 0.75)$.

As can be seen in both cases, each of the four methods for labelling the vertices are about as equally effective as each other when using the greedy algorithm.

Graphs randomly selected from $\mathcal{G}_3(70, 0.75)$ notably require much fewer colours on average than those in $\mathcal{G}(70, 0.5)$ via the greedy algorithm. This is notable, as the expected number of edges of a graph in $\mathcal{G}(70, 0.5)$ is

$$\binom{70}{2} \cdot 0.5 = 1207.5,$$

which is lower than the expected number of edges of a graph in $\mathcal{G}_3(70, 0.75)$ which is

$$\left(\binom{70}{2} - \underbrace{\binom{24}{2} - 2 \cdot \binom{23}{2}}_{(*)} \right) \cdot 0.75 = 1224.75,$$

[$(*)$ accounts for the pairs of vertices i, j with $i - j \equiv 0 \mod 3$, which shouldn't be counted.]

despite the much higher average number of colours used in the case of $\mathcal{G}(70, 0.5)$.

This result does however make sense when we note that the rule that no edge ij can exist with $i - j \equiv 0 \mod 3$ gives rise to a lot of structure within a graph randomly chosen from $\mathcal{G}_3(70, 0.75)$, allowing a colouring with few colours to arise more easily, whereas a graph randomly chosen from $\mathcal{G}(70, 0.5)$ is expected to have no such structure, meaning little symmetry can be exploited by a colouring to allow for a small number of colours to exist.

Question 2

An ordering guaranteeing that the greedy algorithm uses no more than 3 colours for $\mathcal{G}_3(70, 0.75)$

An order guaranteeing such a result can be obtained as follows:

Suppose $v_1, \ldots, v_{70}$ is the original labelling of the graph's vertices after generating it from $\mathcal{G}_3(70, 0.75)$. In particular, this labelling obeys the rule that $v_i v_j$ is not an edge when $i - j \equiv 0 \mod 3$.

We construct a new labelling $v'_1, \ldots, v'_{70}$ as follows, where the square brackets denote ordered sets:

$$[v'_1, \ldots, v'_{70}] = [v_1, v_4, v_7, \ldots, v_{67}, v_{70} v_2, v_5, v_8, \ldots, v_{68}, v_3, v_6, v_9, \ldots, v_{69}].$$

That is, we group the original v_is by congruency class: require all v_i with $i \equiv 1 \mod 3$ to come first, followed by the v_i with $i \equiv 2 \mod 3$, then the v_i with $i \equiv 0 \mod 3$.

Such an ordering guarantees that the vertices that were originally v_i with $i \equiv 1 \mod 3$ will be labelled with the colour 1, due to no edges existing between these vertices. The $i \equiv 2 \mod 3$ will then be labelled with at most the colour 2 similarly (some may be coloured with 1) and the final congruence class labelled with at most the colour 3.

This idea can be easily extended to give an ordering that guarantees a graph selected from $\mathcal{G}_k(n, p)$ will be coloured in no more than k colours.

Why was $p = 0.75$ chosen here?

As per our discussion in question 1, we note that

$$\mathbb{E}(\#\text{edges of } G \text{ chosen from } \mathcal{G}(70, 0.5)) \approx \mathbb{E}(\#\text{edges of } G \text{ chosen from } \mathcal{G}_3(70, p))$$

when $p = 0.75$. Despite the similar number of edges, in question 1 we remarked that the greedy algorithm is able to find a much better colouring $\mathcal{G}_3(70, 0.75)$ due to the structure that arises in such graphs. In this question we proved that such a graph can indeed by coloured by greedy in at most 3 colours by exploiting this structure.

No such structure can be exploited for a graph G chosen in $\mathcal{G}(70, 0.5)$, meaning we are unable to do similar analysis to give an upper bound on $\chi(G)$, let alone provide an ordering such that the greedy algorithm constructs it. Indeed, it is hypothetically possible we pick $G = K_{70}$, the complete graph on 70 vertices, requiring 70 colours.

A graph G of order $3n$ such that $\chi(G) = 3$ but on which greedy might need $n + 2$ colours

Consider the graph G as defined. G has vertices labelled $\{v_{i,j} \mid 1 \leq i \leq 3, 1 \leq j \leq n\}$, in particular $3n$ vertices. Say $v_{i,j}$ is connected to $v_{i',j'}$ provided either:

- $i \neq i'$ and $j \neq j'$,
- $j = j' = n$.

The following demonstrates a 3-colouring of G:

Let $A_i = \{v_{i,j} \mid 1 \leq j \leq n\}$ for $i = 1, 2, 3$. Then colour all the vertices in the set A_i with the colour i, $i = 1, 2, 3$. This is a 3-colouring.

The greedy algorithm produces an $n + 2$-colouring if we order the vertices as $v'_1, \ldots, v'_{3n}$ as follows:

$$[v'_1, \ldots, v'_{3n}] = [v_{1,1}, v_{2,1}, v_{3,1}, v_{1,2}, v_{2,2}, v_{3,2}, \ldots v_{1,n}, v_{2,n}, v_{3,n},].$$

Under this ordering: $v_{1,1}, v_{2,1}, v_{3,1}$ are not connected to each other, so they all get coloured 1. $v_{1,2}, v_{2,2}, v_{3,2}$ are all not connected to each other, but are connected to two of $v_{1,1}, v_{2,1}, v_{3,1}$, thus

get coloured 2. $v_{1,3}, v_{2,3}, v_{3,3}$ are not connected to each other but are connected to two of those vertices we coloured 1 and two of those vertices we coloured 2. Thus they get coloured 3. In general $v_{1,j}, v_{2,j}, v_{3,j}$ get coloured j for $j < n$. Then $v_{1,n}, v_{2,n}, v_{3,n}$ form a triangle, with each connected to something of colour $1, 2, \ldots, n-1$, thus they get coloured $n, n+1, n+2$ respectively.

A similar construction can be used to find a graph G of order kn such that $\chi(G) = k$, but which greedy uses $n + k - 1$ colours.

Appendix A: Code

Matrices.cs

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

namespace Matrices
{
   public class Matrix
   {

      public static Random random = new Random();
      private int[] underlyingArray;

      private int rows;
      private int columns;

      public Matrix(int height, int width)
      {
         underlyingArray = new int[width * height];

         rows = height;
         columns = width;
      }

      public int this[int row, int column]
      {
         get
         {
            return underlyingArray[(column - 1) * rows + row - 1];
         }
         set
         {
            underlyingArray[(column - 1) * rows + row - 1] = value;
         }
      }

      public override string ToString()
      {
         string lineBreak = "|";
         string str = "[ ";
         for (int i = 1; i <= Rows; i++)
         {
            for (int j = 1; j <= Columns; j++)
            {
               str += this[i, j] + " ";

            }

            if (i != Rows)
            {
               str += lineBreak + " ";
            }

         }
         str += "]";
         return str;
```

```
}

public (int, int) size
{
   get
   {
      return (rows, columns);
   }
}

public int Rows
{
   get
   {
      return rows;
   }
}

public int Columns
{
   get
   {
      return columns;
   }
}

public static Matrix operator *(Matrix A, Matrix B) //multiply 2 matrices together
{

   if (A.columns != B.rows)
   {
      throw new Exception("Cannot multiply a " + A.size + " matrix with a " +
          B.size + " matrix.");
   }

   Matrix AB = new Matrix(A.rows, B.columns);

   for (int i = 1; i <= AB.Rows; i++)
   {
      for (int j = 1; j <= AB.Columns; j++)
      {
         AB[i, j] = 0;

         for (int k = 1; k <= A.columns; k++)
         {
            AB[i, j] += A[i, k] * B[k, j];
         }

      }
   }

   return AB;
}

public static Matrix operator *(Matrix A, Vector B) //multiply a matrix with a
    vector
{
   return A * B.AsMatrix();
}
```

```
public Matrix Mod(int number) //returns this matrix mod some number
{
   Matrix newMatrix = new Matrix(this.Rows, this.Columns);

   for (int i = 1; i <= this.Rows; i++)
   {
      for (int j = 1; j <= this.Columns; j++)
      {
         newMatrix[i, j] = this[i, j] % number;

         while (newMatrix[i, j] < 0) //the % in C# may not work properly if
             negative, so we need to increase until positive.
         {
            newMatrix[i, j] += number;
         }
      }
   }

   return newMatrix;
}

public Vector[] RowsAsVectors()
{
   List<Vector> rowVectors = new List<Vector>();
   for (int i = 1; i <= this.Rows; i++)
   {
      Vector vector = new Vector(this.Columns);
      for (int j = 1; j <= this.Columns; j++)
      {
         vector[j] = this[i, j];
      }
      rowVectors.Add(vector);
   }
   return rowVectors.ToArray();
}

public Matrix Transpose()
{
   Matrix transpose = new Matrix(this.Columns, this.Rows);

   for (int i = 1; i <= this.Rows; i++)
   {
      for (int j = 1; j <= this.Columns; j++)
      {
         transpose[j, i] = this[i, j];
      }
   }
   return transpose;
}

public static Matrix RandomMatrix(int inPrimeField, int width, int height)
{
   Matrix matrix = new Matrix(width, height);
   for (int i = 1; i <= width; i++)
   {
      for (int j = 1; j <= height; j++)
      {
         matrix[i, j] = random.Next(0, inPrimeField);
```

```
      }
   }
   return matrix;
}

/// <summary>
/// Makes a nice LaTeX string for this matrix
/// </summary>
/// <returns></returns>
public string ToLaTeX()
{

   string result = "";
   result += "\\begin{pmatrix}\n";
   for (int i = 1; i <= this.Rows; i++)
   {
      string line = "";
      for (int j = 1; j <= this.Columns; j++)
      {
         int value = this[i, j];

         line += value;
         if (j == this.Columns)
         {
            if (i != this.Rows)
            {
               line += " \\\\";
            }
         }
         else
         {
            line += " & ";
         }

      }
      result += line;
      result += "\n";
   }
   result += "\\end{pmatrix}";
   return result;
}

public Matrix transpose
{
   get
   {
      Matrix Transpose = new Matrix(Columns, Rows);

      for (int i = 1; i <= Transpose.Rows; i++)
      {
         for (int j = 1; j <= Transpose.Columns; j++)
         {
            Transpose[i, j] = this[j, i];
         }
      }

      return Transpose;
   }
}
```

```
    public bool AllEntriesZero()
    {
        foreach (int entry in underlyingArray)
        {
            if (entry != 0)
            {
                return false;
            }
        }
        return true;
    }
}

public class Vector
{
    Matrix underlyingMatrix;

    public Vector(int size)
    {
        underlyingMatrix = new Matrix(size, 1);
    }

    public Vector(Matrix matrix) //turn some matrix into a vector, if possible.
    {
        if (matrix.Rows == 1)
        {
            underlyingMatrix = matrix.Transpose();
        }
        else if (matrix.Columns == 1)
        {
            underlyingMatrix = matrix;
        }
        else
        {
            throw new Exception("Given matrix is not either a row or column vector so
                cannot convert into vector object.");
        }

    }

    public int this[int entry]
    {
        get
        {
            return underlyingMatrix[entry, 1];
        }
        set
        {
            underlyingMatrix[entry, 1] = value;
        }
    }

    public int Size
    {
        get
        {
            return underlyingMatrix.Rows;
        }
```

```
        }

        public Matrix AsMatrix()
        {
            return underlyingMatrix;
        }

        public Vector Mod(int number) //returns this vector mod some number
        {
            return new Vector(AsMatrix().Mod(number));
        }

        public override string ToString()
        {
            return underlyingMatrix.ToString();
        }


        public string ToLaTeX()
        {
            return underlyingMatrix.ToLaTeX();
        }

        /// <summary>
        /// Returns the last non-zero index of the vector. E.g. (1,1,0,1,0,0) returns 4 as
            the 4th index is non-zero, all others are zero.
        /// If it is the zero vector, returns 0
        /// </summary>
        /// <returns></returns>
        public int LastNonZeroIndex()
        {
            int lastIndex = 0;
            for (int i = 1; i <= this.Size; i++) {

                if (this[i] != 0)
                {
                    lastIndex = i;
                }

            }
            return lastIndex;
        }

    }
}
```

MatricesKernel.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ContinuedFractionProject;
using Matrices;
using Matrices.RREF;
```

```
namespace Matrices.Kernel
{
   public static class MatricesKernel
   {
      /*
      * This class provides extensions to the matrices that allow the calculation of the
          kernel, similarly to what was done in Q3 with the RREF.
      */

      public static Vector[] GetBasisOfKernel(this Matrix matrix, int prime)
      {

         if (matrix.AllEntriesZero()) //special case, this algorithm doesnt work.
         {

            List<Vector> vectors = new List<Vector>();

            for (int i = 1; i <= matrix.Columns; i++) {
               Vector v = new Vector(matrix.Columns);

               v[i] = 1;
               vectors.Add(v);
            }
            return vectors.ToArray();
         }
         //Console.WriteLine("Going to find the RREF");
         //Get the RREF and its ls
         Matrix RREF = matrix.GetReducedRowEchelonForm(prime);

         //Console.WriteLine("Found RREF");

         int m = RREF.Rows;
         int n = RREF.Columns;
         int r = RREF.LastNonZeroRow();
         //Console.WriteLine("n = " + n);
         //Console.WriteLine("m = " + m);
         //Console.WriteLine("r = " + r);

         int[] ls = RREF.GetRREFLValues(); //the values of l(1), ..., l(r). due to array
             indexing we store l(1) at 0, etc.

         //we need to construct a matrix B such that the x_{l(i)} are expressed in terms
             of the other x_j.
         //see write up in document - this is how we do it:

         Matrix XRelationMatrix = new Matrix(r, n - r); //relates the xs to each other.
         //Console.WriteLine(XRelationMatrix.size);

         for (int i = 1; i <= r; i++)
         {
            //Console.WriteLine("l(" + i + ") = " + ls[i - 1]);
            int newIndex = 1;

            for (int j = 1; j <= n; j++)
            {
               if (ls.Contains(j) == false) //this is not one of the x_{l(i)}
               {
                  //Console.WriteLine(j);
```

```
            XRelationMatrix[i, newIndex] = -RREF[i, j]; //negative due to
                rearrangement to other side of the equation: vector of x_l(i)s = B
                x_others
            newIndex++;
        }
    }
}

//Console.WriteLine(XRelationMatrix.ToString());

//we have the required matrix.

//now we can work out the basis.

Vector[] basis = new Vector[n - r];

for (int vectorComponentToSetToOne = 1; vectorComponentToSetToOne <= n - r;
    vectorComponentToSetToOne++)
{
    int xToSetToOne = 1;
    int lsFound = 0;
    if (ls.Contains(xToSetToOne)) //this is not one of the x_{l(i)}
    {
        lsFound++;
    }

    while (xToSetToOne - lsFound != vectorComponentToSetToOne)
    {
        xToSetToOne++;
        if (ls.Contains(xToSetToOne)) //this is not one of the x_{l(i)}
        {
            lsFound++;
        }
    }

    //construct the vector of the xjs

    Vector xjsVector = new Vector(n - r); //defaults to all 0 so set required
        component to 1
    xjsVector[vectorComponentToSetToOne] = 1;

    Vector valuesOfLs = new Vector(XRelationMatrix * xjsVector);

    valuesOfLs = valuesOfLs.Mod(prime); //mod it in case of any overflow out of
        the range 0, ... p-1

    //now construct the basis vector

    //set the non l x to 1
    Vector basisVector = new Vector(n);
    basisVector[xToSetToOne] = 1;

    //set the l xs to whatever value calculated.
    for (int i = 1; i <= ls.Length; i++)
    {
        int l = ls[i - 1]; //off by 1 due to array indexing
        basisVector[l] = valuesOfLs[i];
    }
```

```
            basisVector = basisVector.Mod(prime); //mod it in case of any overflow out of
                the range 0, ... p-1 (may happen due to negatives)

            basis[vectorComponentToSetToOne - 1] = basisVector; //done!
         }

         //just as a check, make sure to remove all zero vectors. shouldnt have any any
             way.

         List<Vector> basisFinal = new List<Vector>();

         foreach (Vector vector in basis)
         {
            if (vector.LastNonZeroIndex() != 0)
            {
               basisFinal.Add(vector);
            }
         }

         return basisFinal.ToArray();
      }
   }
}
```

MatricesRREF.cs

```
using System;
using ContinuedFractionProject;
using Matrices;
using PrimeFieldInverses;

namespace Matrices.RREF
{
   public static class MatricesRREF
   /*
   * This class provides extensions to the matrices that allow the calculation of the
       Reduced Row Echelon Form.
   */
   {

      public static int LastNonZeroRow(this Matrix matrix)
      {
         //returns the number of the last row which is non zero.

         int currentRowTesting = matrix.Rows; //start at the bottom

         bool rowIsAllZeroes = true;

         do
         {
            for (int j = 1; j <= matrix.Columns; j++)
            {
               if (matrix[currentRowTesting, j] != 0) //this row is not completely made
                   of 0s
               {
                  rowIsAllZeroes = false;
```

```
            }
        }
        if (rowIsAllZeroes) //this row we have just tested is all 0s. check the next
            highest one.
        {
            currentRowTesting--;

            if (currentRowTesting == 0) //we have gone before the first row - meaning
                this matrix is the zero matrix, a special case. say it is in RREF.
            {
                return matrix.Rows;
            }
        }
    } while (rowIsAllZeroes);

    //the row we have just tested has some non zero element so we can work out what
        r is now

    return currentRowTesting;
}
public static bool IsInReducedRowEchelonForm(this Matrix matrixToTest, int prime)
{

    /*
    * CONDITION 1: "For some r..."
    */

    int r = matrixToTest.LastNonZeroRow();
    /*
    * CONDITION 2: "For each i..."
    */

    int[] ls = new int[r]; //the values of l(1), ..., l(r). due to array indexing
        we store l(1) at 0, etc.

    for (int i = 1; i <= r; i++)
    {
        //calculate l(i)

        int j = 1;

        while (matrixToTest[i, j] == 0)
        {
            j++;
            if (j > matrixToTest.Columns) //the whole row was zeroes (not at the
                bottom but somewhere in the middle)
            {
                return false;
            }
        }

        if (matrixToTest[i, j] != 1) //the row does not start with a 1 after the 0s
            so this is not in the form.
        {
            return false;
        }
        else //weve found l(i)
        {
            ls[i - 1] = j;
```

```
      }
   }

   //we have all the values of l.

   /*
   * CONDITION 3: "l(1) < l(2) < ..."
   */

   //check to make sure the ls obey this.

   int last = ls[0];

   for (int i = 1; i < r; i++)
   {
      if (ls[i] <= last) //fails this condition
      {
         return false;
      }
      last = ls[i];
   }

   /*
   * CONDITION 4: "For each k..."
   */
   for (int k = 2; k <= r; k++)
   {
      int j = ls[k - 1];

      for (int i = 1; i < k; i++)
      {
         if (matrixToTest[i, j] != 0) //fails the test.
         {
            return false;
         }
      }
   }

   //meets all 4 conditions so this is in RREF.

   return true;
}

public static int[] GetRREFLValues(this Matrix matrixInRREF)
{
   //returns the values of l(1), ... for some matrix in RREf.
   int[] ls = new int[matrixInRREF.LastNonZeroRow()]; //the values of l(1), ...,
       l(r). due to array indexing we store l(1) at 0, etc.

   for (int i = 1; i <= matrixInRREF.LastNonZeroRow(); i++)
   {
      //calculate l(i)

      int j = 1;

      while (matrixInRREF[i, j] == 0)
      {
         j++;
      }
```

```
      if (matrixInRREF[i, j] != 1) //the row does not start with a 1 after the 0s
          so this is not in the form.
      {
         throw new Exception("This matrix is not in RREF.");
      }
      else //weve found 1(i)
      {
         ls[i - 1] = j;
      }
   }
   return ls;
}

//Below we code the operations T, D, S that we will need for GE.
public static Matrix TransposeRows(this Matrix matrix, int i, int j, int prime)
{
   Matrix newMatrix = matrix.Mod(prime);

   //new matrix is currently a copy of the original matrix, modded. perform the
       transposition

   for (int column = 1; column <= matrix.Columns; column++)
   {
      newMatrix[i, column] = matrix[j, column];
      newMatrix[j, column] = matrix[i, column];
   }

   return newMatrix;
}

public static Matrix Divide(this Matrix matrix, int i, int a, int prime)
{
   //to divide, we need to get the inverses of this prime


   if (a == 0)
   {
      throw new Exception("Cannot divide by zero.");
   }
   Matrix newMatrix = matrix.Mod(prime);

   //new matrix is currently a copy of the original matrix, modded. perform the
       division

   for (int column = 1; column <= matrix.Columns; column++)
   {
      newMatrix[i, column] *= Inverses.InverseOf(a, prime); //multiplying by the
          inverse same as dividing.
   }

   newMatrix = newMatrix.Mod(prime); //mod the matrix after applying the operation
       in case anything has gone out of the range.
   return newMatrix;
}

public static Matrix Subtract(this Matrix matrix, int i, int a, int j, int prime)
{
```

```
   if (i == j) //we require i =/= j.
   {
      throw new Exception("Cannot allow the subtraction of a row from itself.");
   }
   Matrix newMatrix = matrix.Mod(prime);

   //new matrix is currently a copy of the original matrix, modded. perform the
       transposition

   for (int column = 1; column <= matrix.Columns; column++)
   {
      newMatrix[i, column] -= a * newMatrix[j, column];
   }

   newMatrix = newMatrix.Mod(prime); //mod the matrix after applying the operation
       in case anything has gone out of the range.
   return newMatrix;
}


public static Matrix GetReducedRowEchelonForm(this Matrix matrix, int prime) //get
    RREF for this matrix.
{

   if (matrix.AllEntriesZero()) //technically not in RREF, but we cannot put it
       into such. just return it, if we need to deal with zero matrices we do it
       elsewhere
   {
      return matrix;
   }
   if (IsInReducedRowEchelonForm(matrix, prime)) //we are done
   {
      return matrix;
   }

   Matrix newMatrix = matrix.Mod(prime);

   int currentColumn = 1; //the current column that we shall intend to modify into
       the form it should be in for RREF.
   int currentRowToLead = 1;


   while ((currentColumn > newMatrix.Columns || currentRowToLead > newMatrix.Rows)
       == false && IsInReducedRowEchelonForm(newMatrix, prime) == false) //when
       this condition is true, we are at the end of the process since we have gone
       beyond the edge of the matrix. or if in RREF before, just quit since solved.
   {
      //Find some row in this column that we are sorting out that has a non-zero
          number, if any exists.

      int rowWithNonZeroInCurrentColumn = 0;

      for (int i = currentRowToLead; i <= newMatrix.Rows; i++)
      {
         if (newMatrix[i, currentColumn] != 0)
         {
            rowWithNonZeroInCurrentColumn = i;
            break;
         }
```

```
            }

            if (rowWithNonZeroInCurrentColumn == 0) //this column is full of nothing but
                zeroes, so it is fine to skip over trying to fix it. there is no new top
                row though
            {
               currentColumn++;
            }
            else
            {

               //move this row that we have just found to the top of the matrix
                   (excluding previously sorted out rows)
               newMatrix = newMatrix.TransposeRows(rowWithNonZeroInCurrentColumn,
                   currentRowToLead, prime);
               //we want the leading coefficient to be 1. we can do that by dividing by
                   whatever the current leading coeff is.
               newMatrix = newMatrix.Divide(currentRowToLead, newMatrix[currentRowToLead,
                   currentColumn], prime);
               //we want every other row to have a 0 in this column, so we can do this by
                   applying subtraction.
               //we do not need to worry about messing up any of the previous columns
                   above since by previous steps in this algorithm we have everything
                   before the leading 1 in this column a 0.

               for (int i = 1; i <= matrix.Rows; i++)
               {
                  if (i != currentRowToLead) //cant substract from this same row.
                  {
                     newMatrix = newMatrix.Subtract(i, newMatrix[i, currentColumn],
                         currentRowToLead, prime); //subtracting this amount from the row
                         will create a 0 since there is a leading 1
                  }
               }


               //we have sorted out this row and column. move onto the next ones
               currentColumn++;
               currentRowToLead++;
            }


         }




         if (IsInReducedRowEchelonForm(newMatrix, prime) == false) //something has gone
             wrong, this should not occur.
         {
            throw new Exception("Error in algorithm - does not satisfy RREF conditions");
         }

         return newMatrix;
      }
   }
}
```

ArrayExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ContinuedFractionProject
{
   public static class ArrayExtensions
   {
      public static string ToFormattedString<T>(this T[] thisArray)
      {
         if (thisArray.Length == 0)
         {
            return "[]";
         }
         string result = "[";

         foreach (T item in thisArray)
         {
            if (item is null)
            {
               result += "null";
            }
            else
            {
               result += item.ToString();
            }

            result += ", ";
         }

         result = result.Substring(0, result.Length - 2);
         result += "]";
         return result;
      }

      /// <summary>
      /// Remove all copies of a given item.
      /// </summary>
      /// <typeparam name="T"></typeparam>
      /// <param name="array"></param>
      /// <param name="itemToRemove"></param>
      /// <returns></returns>
      public static T[] RemoveAll<T>(this T[] array, T itemToRemove)
      {
         List<T> newArr = new List<T>();

         foreach (T item in array)
         {
            if (item.Equals(itemToRemove) == false)
            {
               newArr.Add(item);
            }
         }
         return newArr.ToArray();
```

```
        }

        /// <summary>
        /// Gets all elements that appear in the array more than once. Each
            more-than-once-appearing element will appear in the new array one time.
        /// </summary>
        /// <param name="array"></param>
        /// <example>{1, 2, 2, 3} returns {2}</example>
        /// <returns></returns>
        public static T[] AllEntriesThatAppearMoreThanOnce<T>(this T[] array)
        {
           List<T> duplicated = new List<T>();


           for (int i = 0; i < array.Length; i++)
           {
              T item = array[i];
              if (duplicated.Contains(item) == false) //we dont already know this one is
                  duplicated, i.e. its the first one of its type
              {
                 for (int j = i+1; j < array.Length; j++)
                 {
                    T item2 = array[j];
                    if (item.Equals(item2))
                    {
                       duplicated.Add(item);
                       break;

                    }
                 }
              }
           }
           return duplicated.ToArray();
        }

        /// <summary>
        /// Returns the indices of the array matching a given item.
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="arr"></param>
        /// <param name="item"></param>
        /// <returns></returns>
        public static int[] IndicesOf<T>(this T[] arr, T item)
        {
           List<int> indices = new List<int>();
           for (int i = 0; i < arr.Length;i++)
           {
              if (arr[i].Equals(item))
              {
                 indices.Add(i);
              }
           }
           return indices.ToArray();
        }
     }
  }
```

ContinuedFractions.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Text;
using System.Threading.Tasks;

namespace ContinuedFractionProject
{
   public static class ContinuedFractions
   {

      public static int PeriodOfContinuedFractionExpansionOfSquareRoot(this ulong
          integerUnderRoot)
      {
         List<(ulong, ulong)> foundRSValues = new List<(ulong, ulong)>();

         List<ulong> AList = new List<ulong>();

         bool found = false;

         //start with (0 + sqrt(N))/1
         ulong rCurrent = 0;
         ulong sCurrent = 1;

         int i = 0;
         while (true) //this will eventually terminate by itself
         {


            //get the new values

            double convergentCurrent = ((double)rCurrent + Math.Sqrt(integerUnderRoot)) /
                (double)sCurrent;
            ulong floorCurrent = (ulong)Math.Floor(convergentCurrent); //a_n


            if (i > 0)
            {
               //add current r and s
               foundRSValues.Add((rCurrent, sCurrent));

               //add a
               AList.Add(floorCurrent);
            }


            if (convergentCurrent == floorCurrent) //xn = an so terminate, its a square
                number
            {
               return 0;
            }

            ulong rNext = floorCurrent * sCurrent - rCurrent; //as-r
            ulong sNext = (integerUnderRoot - (rNext) * (rNext)) / sCurrent;
                //(N-(r-as)^2) / s
```

```
      if (foundRSValues.Contains((rNext, sNext))) //were looping
      {
         return i;
      }

      rCurrent = rNext;
      sCurrent = sNext;
      i++;

   }
}

/// <summary>
/// Returns the CFE of a given square root of an integer.
/// </summary>
/// <param name="integerUnderRoot">N in the program</param>
/// <param name="RSValues">The values of r and s as defined in question 2 for each
    x_n.</param>
/// <returns></returns>
public static ulong[] ContinuedFractionExpansionOfSquareRoot(this ulong
    integerUnderRoot, out (ulong, ulong)[] RSValues, int upToConvergent)
{
   List<(ulong, ulong)> RSValuesList = new List<(ulong, ulong)>();

   List<ulong> AList = new List<ulong>();

   //start with (0 + sqrt(N))/1
   ulong rCurrent = 0;
   ulong sCurrent = 1;
   for (int i = 0; i <= upToConvergent; i++)
   {


      //get the new values

      double convergentCurrent = ((double)rCurrent + Math.Sqrt(integerUnderRoot)) /
          (double)sCurrent;
      ulong floorCurrent = (ulong)Math.Floor(convergentCurrent); //a_n


      //add current r and s
      RSValuesList.Add((rCurrent, sCurrent));

      //add a
      AList.Add(floorCurrent);


      if (convergentCurrent == floorCurrent) //xn = an so terminate
      {
         break;
      }

      ulong rNext = floorCurrent * sCurrent - rCurrent; //as-r
      ulong sNext = (integerUnderRoot - (rNext) * (rNext)) / sCurrent;
          //(N-(r-as)^2) / s

      rCurrent = rNext;
      sCurrent = sNext;
```

```
   }

   RSValues = RSValuesList.ToArray();

   return AList.ToArray();

}

/// <summary>
/// Returns a list of partial quotients as tuples, with the first entry
    corresponding to p, the second corresponding to q. Likely will cause overflow
    if represents the partial quotients for sqrt(N) with large N, and given array
    is long.
/// </summary>
/// <param name="arr"></param>
/// <returns></returns>
public static (ulong, ulong)[] PartialQuotients(this ulong[] arr)
{
   return PartialQuotientsModulo(arr, 0);
}

/// <summary>
/// Returns a list of partial quotients as tuples, with the first entry
    corresponding to p, the second corresponding to q, modulo a given number.
/// </summary>
/// <param name="arr"></param>
/// <param name="mod">What number to work out the partial quotients mod. put mod =
    0 for no modding. Likely to cause overflows in this case</param>
/// <returns></returns>
public static (ulong, ulong)[] PartialQuotientsModulo(this ulong[] arr, ulong mod)
{
   List<(ulong, ulong)> partialQuotientsList = new List<(ulong, ulong)>();

   //p, q at index i-2, start at i=0
   ulong P_iMinusTwo = 0;
   ulong Q_iMinusTwo = 1;

   //p, q at index i-1, start at i=0
   ulong P_iMinusOne = 1;
   ulong Q_iMinusOne = 0;

   int i = 0;

   while (i < arr.Length)
   {
      //recursion relations
      ulong P_i, Q_i;

      if (mod == 0)
      {
         P_i = (ulong)arr[i] * P_iMinusOne + P_iMinusTwo;
         Q_i = (ulong)arr[i] * Q_iMinusOne + Q_iMinusTwo;
      }
      else
      {
         P_i = ModularArithmetic.Mod(ModularArithmetic.ModMult(arr[i], P_iMinusOne,
             mod) + P_iMinusTwo, mod);
```

```
            Q_i = ModularArithmetic.Mod(ModularArithmetic.ModMult(arr[i], Q_iMinusOne,
                mod) + Q_iMinusTwo, mod);
        }


        partialQuotientsList.Add((P_i, Q_i));

        //go on to next i
        P_iMinusTwo = P_iMinusOne;
        Q_iMinusTwo = Q_iMinusOne;

        P_iMinusOne = P_i;
        Q_iMinusOne = Q_i;

        i++;
      }

      return partialQuotientsList.ToArray();
    }

    /// <summary>
    /// Returns P_n mod N, P_n^2 mod N for a given N and n in a given range
    /// </summary>
    /// <returns>An array of tuples, with the nth entry in the array giving (P_n mod
        N, P_n^2 mod N)</returns>
    public static (ulong, ulong)[] ConvergentNumeratorsModulo(ulong N, int
        upToConvergent)
    {
      (ulong, ulong)[] unused = new (ulong, ulong)[0];
      ulong[] aValues = ContinuedFractionExpansionOfSquareRoot(N, out unused,
          upToConvergent);

      ulong[] PValues = aValues.PartialQuotientsModulo(N).Select(x =>
          x.Item1).ToArray();

      List<(ulong, ulong)> convergentModValues = new List<(ulong, ulong)>();

      foreach (ulong PValue in PValues)
      {
        ulong mod = ModularArithmetic.Mod(PValue, N);
        ulong modSquared = ModularArithmetic.ModMult(mod, mod, N);

        convergentModValues.Add((mod, modSquared));
      }

      return convergentModValues.ToArray();
    }

  }
}
```

GenericExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;

namespace ContinuedFractionProject
{
   public static class GenericExtensions
   {
      private static Random random = new Random();

      /// <summary>
      /// Returns a random integer with a given number of digits
      /// </summary>
      /// <param name="numberOfDigits"></param>
      /// <returns></returns>
      public static long RandomIntegerWithGivenDigits(int numberOfDigits)
      {
         if (numberOfDigits < 1)
         {
            throw new Exception("Need to be given a positive integer for given digits");
         }


         string result = "";

         for (int i = 1; i <= numberOfDigits; i++)
         {
            if (i == 1)
            {
               result += random.Next(1, 10).ToString(); //10 since exclusive upper bound
            }
            else
            {
               result += random.Next(0, 10).ToString(); //10 since exclusive upper bound
            }

         }

         return long.Parse(result);
      }

      public static long[] DistinctRandomIntegersWithGivenDigits(int numberOfDigits, int
          sampleSizeMax)
      {

         long min = (int)Math.Pow(10, numberOfDigits - 1);
         long max = (int)Math.Pow(10, numberOfDigits) - 1;

         long totalNumbersInRange = max - min + 1;

         long actualSampleSizeMax = sampleSizeMax;
         if (sampleSizeMax > totalNumbersInRange)
         {
            actualSampleSizeMax = totalNumbersInRange;
         }

         List<long> result = new List<long>();

         for (int i = 0; i < actualSampleSizeMax; i++)
         {
```

```
            long numGenerated = 0;

            do
            {
               numGenerated = RandomIntegerWithGivenDigits(numberOfDigits);
            } while (result.Contains(numGenerated));

            result.Add(numGenerated);
         }

         return result.ToArray();
      }

      public static long GCD(long a, long b)
      {
         //Console.WriteLine("GCD(" + a + ", " + b + ")");
         //Console.ReadLine();
         if (b > a)
         {
            return GCD(a, b);
         }
         else
         {
            if (b == 0)
            {
               return a;
            }
            else
            {
               return GCD(b, a % b);
            }
         }
      }
   }
}
```

ModularArithmetic.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ContinuedFractionProject
{
   /// <summary>
   /// Performs modular arithmetic.
   /// </summary>
   public static class ModularArithmetic //Reused from my 15.1 Primality tests project
       with some other additions. May contain methods not relevant to this project.
   {

      #region Modulus methods
      /// <summary>
      /// Calculates a^b mod N
```

```
/// </summary>
public static int ModExp(int a, int b, int N)
{
   if (b < 0)
   {
      throw new Exception("Exponent must be positive.");
   }
   if (a == 0 && b == 0)
   {
      throw new Exception("0^0 not defined.");
   }


   /*
   * In order to make sure we dont get overflows, we multiply by a and do the mod
       at each step.
   * This ensures that we never have result be larger than a*(N-1) (result is
       never more than
   * N-1 after each for loop, so it could be on a*(N-1). In this project we dont
       work with N > 10^10
   * so we can work with a up to about 10^5 which is more than enough for our
       purpouses, we wont really
   * go above bases in double digits).
   */

   /*
   * We dont just do a*a* ... * a, b times, since for b large (which we will need
       since we exponent to the prime)
   * this takes ages. Instead heres a better way:
   * Write b in binary, and calculate a, then a^2, then a^4 by a^2 * a^2, up to
       the largest binary digit in b.
   * Multiply all together where we have it. E.g. if we want to find a^42, 42 is
       101010 in binary so we
   * do a^32 * a^8 * a^2.
   */

   int result = 1;

   int newA = a;
   int newB = b;
   newA = Mod(newA, N);

   while (newB > 0)
   {
      if (newB % 2 == 1)
      result = ModMult(result, newA, N);

      newB /= 2;
      newA = ModMult(newA, newA, N);


   }
   //Console.WriteLine(a + "^" + b + " = " + result + " mod " + N);
   return result;
}

/// <summary>
/// Calculates a^b mod N
/// </summary>
```

```
public static ulong ModExp(ulong a, ulong b, ulong N)
{
   if (b < 0)
   {
      throw new Exception("Exponent must be positive.");
   }
   if (a == 0 && b == 0)
   {
      throw new Exception("0^0 not defined.");
   }


   /*
   * In order to make sure we dont get overflows, we multiply by a and do the mod
       at each step.
   * This ensures that we never have result be larger than a*(N-1) (result is
       never more than
   * N-1 after each for loop, so it could be on a*(N-1). In this project we dont
       work with N > 10^10
   * so we can work with a up to about 10^5 which is more than enough for our
       purpouses, we wont really
   * go above bases in double digits).
   */

   /*
   * We dont just do a*a* ... * a, b times, since for b large (which we will need
       since we exponent to the prime)
   * this takes ages. Instead heres a better way:
   * Write b in binary, and calculate a, then a^2, then a^4 by a^2 * a^2, up to
       the largest binary digit in b.
   * Multiply all together where we have it. E.g. if we want to find a^42, 42 is
       101010 in binary so we
   * do a^32 * a^8 * a^2.
   */

   ulong result = 1;

   ulong newA = a;
   ulong newB = b;
   newA = Mod(newA, N);

   while (newB > 0)
   {
      if (newB % 2 == 1)
      result = ModMult(result, newA, N);

      newB /= 2;
      newA = ModMult(newA, newA, N);


   }
   //Console.WriteLine(a + "^" + b + " = " + result + " mod " + N);
   return result;
}

/// <summary>
/// Works out a*b mod N.
/// </summary>
public static int ModMult(int a, int b, int N)
```

```
{
   int result = 0;
   a = Mod(a, N); //initialise a to be in the interval


   while (b > 0)
   {
      if (b % 2 == 1)
      result = Mod(result + a, N);

      a = Mod(a * 2, N);
      b /= 2; //divide b by 2
   }

   return result;
}

/// <summary>
/// Works out a*b mod N.
/// </summary>
public static ulong ModMult(ulong a, ulong b, ulong N)
{
   ulong result = 0;
   a = Mod(a, N); //initialise a to be in the interval


   while (b > 0)
   {
      if (b % 2 == 1)
      result = Mod(result + a, N);

      a = Mod(a * 2, N);
      b /= 2; //divide b by 2
   }

   return result;
}

/// <summary>
/// Calculates a mod N. We dont just use a % N since C# is weird with negative
    numbers. Probably wont use
/// negative numbers ever in this project but just being careful.
/// </summary>

public static int Mod(this int a, int N)
{
   int result = a;
   while (result < 0)
   {
      result += N;
   }

   result = result % N;
   return result;
}

/// <summary>
/// Calculates a mod N. We dont just use a % N since C# is weird with negative
    numbers.
```

```
        /// </summary>

        public static ulong Mod(this ulong a, ulong N)
        {
            ulong result = a;
            while (result < 0)
            {
                result += N;
            }

            result = result % N;
            return result;
        }

        public static long Mod(this long a, ulong N)
        {
            long result = a;
            while (result < 0)
            {
                result += (long)N;
            }

            result = result % (long)N;
            return result;
        }

        #endregion

        /// <summary>
        /// Calculates a mod N. We dont just use a % N since C# is weird with negative
            numbers. Make it within -N/2 and N/2.
        /// </summary>
        public static long ModWithinHalf(this ulong a, long N)
        {
            long result = (long)a.Mod((ulong)N);

            if (result >= N/2)
            {
                result -= N;
            }

            return result;
        }

    }
}
```

PellsEquation.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ContinuedFractionProject
```

```
{
   internal class PellsEquation
   {
      /// <summary>
      /// Does this tuple x, y, N satisfy x^2 - Ny^2 = 1
      /// </summary>
      public static bool SatisfiesPellsEquation(ulong x, ulong y, ulong N)
      {
         return (SatisfiesPMPellsEquation(x, y, N) == (true, 1));
      }

      /// <summary>
      /// Does this tuple x, y, N satisfy x^2 - Ny^2 = -1
      /// </summary>
      public static bool SatisfiesNegativePellsEquation(ulong x, ulong y, ulong N)
      {
         return (SatisfiesPMPellsEquation(x, y, N) == (true, -1));
      }

      /// <summary>
      /// Does this tuple x, y, N satisfy x^2 - Ny^2 = +-1? Returns which one it
          satisfies.
      /// </summary>
      public static (bool, int) SatisfiesPMPellsEquation(ulong x, ulong y, ulong N)
      {

         int checkPrimesUpTo = 151; //a magic number, as explained in the writeup
             checking up to here proves the result.

         int[] primesToTest = PrimesAndFactorisation.PrimesLessThan(checkPrimesUpTo + 1);

         int pmOne = 0; //whether its plus or minus one. needs to be the same for all of
             3, ..., 151 (mod 2, 1 = -1)

         foreach (int p in primesToTest)
         {
            ulong xSquaredModP = ModularArithmetic.ModMult(x, x, Convert.ToUInt64(p));
                //x^2 mod p

            ulong ySquaredModP = ModularArithmetic.ModMult(y, y, Convert.ToUInt64(p));
                //y^2 mod p

            ulong NySquaredModP = ModularArithmetic.ModMult(ySquaredModP, N,
                Convert.ToUInt64(p)); //Ny^2 mod p

            ulong negativeNySquaredModP = Convert.ToUInt64(p) - NySquaredModP; //-Ny^2
                mod p, but get it as a positive number as ulong only deals with that

            ulong result = ModularArithmetic.Mod(xSquaredModP + negativeNySquaredModP,
                Convert.ToUInt64(p)); //x^2 - Ny^2 mod p, within the range 0, ..., p-1


            if (!(result == 1 || result == Convert.ToUInt64(p) - 1)) //its congruent to
                neither 1 or -1 mod p
            {
               return (false, 0);
            }
            else
            {
```

```
        if (p > 2)
        {
           if (p == 3) //first prime we test where 1 != -1, so assign pmOne here
           {
              if (result == 1)
              {
                 pmOne = 1;
              }
              else //result is p-1
              {
                 pmOne = -1;
              }
           }
           else
           {
              int thispmOne;
              if (result == 1)
              {
                 thispmOne = 1;
              }
              else //result is p-1
              {
                 thispmOne = -1;
              }
              if (pmOne != thispmOne) //is 1 on some primes > 2, -1 on the others,
                   so is not a solution
              {
                 return (false, 0);
              }
           }
        }

     }

   }
   //passed all the tests so its true, return true and whichever it is



   return (true, pmOne);
}

/// <summary>
/// Find a solution to x^2 - Ny^2 = +-1 for a given N using the continued fraction
    of sqrt(N).
/// </summary>
/// <returns>A tuple, x is the first entry and y the second entry. (0, 0) if none
    found</returns>
public static (ulong, ulong) FindPMPellSolutionUsingContinuedFractions(ulong N,
    int checkUpToConvergent, int plusOrMinusOne)
{
   if (plusOrMinusOne != 1 && plusOrMinusOne != -1)
   {
      throw new Exception("Invalid input, need to provide plus or minus one.");
   }
   //Theorem says that the first solution will be either x_{k-1}, y_{k-1} or
       x_{2k-1}, y_{2k-1} for k the period of the CFE.
   //project instructions does not necessarily need that so we take a somewhat
       more naive approach and just check all of them
```

```
            (ulong, ulong)[] unused = new (ulong, ulong)[0];
            ulong[] aValues = ContinuedFractions.ContinuedFractionExpansionOfSquareRoot(N,
                out unused, checkUpToConvergent);

            //Console.WriteLine(aValues.ToFormattedString());

            (ulong, ulong)[] PQValues = ContinuedFractions.PartialQuotients(aValues);

            foreach ((ulong, ulong) tuple in PQValues)
            {
                ulong x = tuple.Item1;
                ulong y = tuple.Item2;

                (bool, int) satisfies = SatisfiesPMPellsEquation(x, y, Convert.ToUInt64(N));



                if (satisfies.Item1 && satisfies.Item2 == plusOrMinusOne) //satisfies
                    relevant pells
                {
                    return (x, y);
                }
            }

            return (0, 0); //none found
        }

        /// <summary>
        /// Find a solution to x^2 - Ny^2 = 1 for a given N using the continued fraction
            of sqrt(N).
        /// </summary>
        /// <returns>A tuple, x is the first entry and y the second entry. (0, 0) if none
            found</returns>
        public static (ulong, ulong) FindPellSolutionUsingContinuedFractions(ulong N, int
            checkUpToConvergent)
        {
            return FindPMPellSolutionUsingContinuedFractions(N, checkUpToConvergent, 1);
        }

        /// <summary>
        /// Find a solution to x^2 - Ny^2 = -1 for a given N using the continued fraction
            of sqrt(N).
        /// </summary>
        /// <returns>A tuple, x is the first entry and y the second entry. (0, 0) if none
            found</returns>
        public static (ulong, ulong) FindNegativePellSolutionUsingContinuedFractions(ulong
            N, int checkUpToConvergent)
        {
            return FindPMPellSolutionUsingContinuedFractions(N, checkUpToConvergent, -1);
        }
    }
}
```

PrimeFieldInverses.cs

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

namespace PrimeFieldInverses
{
   public static class Inverses
   {



      public static int InverseOf(int integer, int prime)
      {
         if (integer <= 0 || integer >= prime)
         {
            throw new Exception("Integer given is not in the range 1, ..., p-1");
         }
         else
         {
            for (int b = 1; b <= prime - 1; b++)
            {
               int ab = (integer * b) % prime; //the value of ab mod p
               //steps += 2;

               if (ab == 1) //b is as inverse.
               {
                  return b;
               }
            }
            throw new Exception("Could not find an inverse");
         }
      }
   }
}
```

PrimesAndFactorisation.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ContinuedFractionProject
{
   public static class PrimesAndFactorisation
   {
      /// <summary>
      /// Returns whether the given integer is prime.
      /// </summary>
      /// <param name="n"></param>
      /// <returns></returns>
      public static bool IsPrime(this long n)
      {
         if (n <= 0)
         {
```

```
      throw new Exception("Need to give a positive integer as input for IsPrime");
   }
   if (n == 1)
   {
      return false;
   }
   for (long i = 2; i <= Math.Sqrt(n); i++) //trial division will be good enough
       for us in this project
   {
      if (n % i == 0) //composite
      {
         return false;
      }
   }
   return true;
}

/// <summary>
/// Returns whether the given integer is prime.
/// </summary>
/// <param name="n"></param>
/// <returns></returns>
public static bool IsPrime(this int n)
{
   return ((long)n).IsPrime();
}

public static bool IsPrime(this ulong n)
{
   return ((long)n).IsPrime();
}

/// <summary>
/// Returns the primes less than a given integer
/// </summary>
/// <param name="n"></param>
/// <returns></returns>
public static int[] PrimesLessThan(int n)
{
   if (n <= 1)
   {
      return new int[0];
   }
   List<int> list = new List<int>();
   for (int i = 2; i < n; i++)
   {
      if (i.IsPrime())
      {
         list.Add(i);
      }
   }
   return list.ToArray();
}

/// <summary>
/// Can n be written as a product of some given primes, possibly with -1?
/// </summary>
/// <param name="n"></param>
/// <param name="list"></param>
```

```
/// <returns></returns>
public static bool IsProductOfBNumbers(this long n, int[] list, bool
    checkIfListIsValid, out (int, int)[] primeFactorisation)
{
   primeFactorisation = new (int, int)[0];

   if (n == 0) //by convention
   {
      return true;
   }

   List<(int, int)> primeFactorisationList = new List<(int, int)>();
   if (n < 0 && list.Contains(-1)) //possible if we can write the positive -n as
       all the things other than -1.
   {
      bool isPositiveBNumberProduct = IsProductOfBNumbers(-n, list.RemoveAll(-1),
          checkIfListIsValid, out primeFactorisation);
      primeFactorisation.Append((-1, 1)); //factor of -1
      return isPositiveBNumberProduct;
   }
   else if (n > 0 && list.Contains(-1)) //no need for -1
   {
      bool isPositiveBNumberProduct = IsProductOfBNumbers(n, list.RemoveAll(-1),
          checkIfListIsValid, out primeFactorisation);
      return isPositiveBNumberProduct;
   }
   else if (n < 0) //n is negative, with no -1 in it, so we cant do it.
   {
      return false;
   }

   if (checkIfListIsValid)
   {
      foreach (int number in list)
      {
         if (number != -1 && number.IsPrime() == false)
         {
            throw new Exception("Given list contains the composite number " +
                number);
         }
      }
   }

   //valid list or didnt check

   //now check if n is a product of the B numbers
   //we know all the things in the list are positive primes at this point due to
       earlier handled bits

   long currentN = n;

   foreach (int number in list)
   {
      int timesDividedBy = 0;
      while (currentN % number == 0)
      {
         timesDividedBy++;
         currentN /= number;
      }
```

```
            if (timesDividedBy > 0) //has this as a factor at least once
            {
               primeFactorisationList.Add((number, timesDividedBy));
            }

         }

         if (currentN == 1) //we did it
         {
            primeFactorisation = primeFactorisationList.ToArray();
            return true;
         }
         else //there must be some other stuff not in B.
         {
            return false;
         }

      }

      public static bool IsProductOfBNumbers(this long n, int[] list, bool
          checkIfListIsValid)
      {
         (int, int)[] unusedList = new (int, int)[0];


         return IsProductOfBNumbers(n, list, checkIfListIsValid, out unusedList);
      }

      /// <summary>
      /// Returns whether the given integer is a square number.
      /// </summary>
      /// <param name="n"></param>
      /// <returns></returns>
      public static bool IsSquare(this int n)
      {
         return (Math.Sqrt(n) == (int)Math.Sqrt(n));
      }

      /// <summary>
      /// Returns whether the given integer is a square number.
      /// </summary>
      /// <param name="n"></param>
      /// <returns></returns>
      public static bool IsSquare(this ulong n)
      {
         return (Math.Sqrt(n) == (ulong)Math.Sqrt(n));
      }
   }
}
```

Program.cs

```
using ContinuedFractionProject;
using System.IO;
using Matrices;
```

```
using Matrices.Kernel;
using Matrices.RREF;
using System;
using System.Linq;

//Where will we output the results of the program?
string logFilePath = "";
string outputFolder = @"C:\Users\robin\Documents\University\Computational
    Projects\II\15.10 The Continued Fraction Method For Factorization\Report\Program
    Output\";



Question1();
Question2();
Question3();
Question5();
Question6();
Question7();

void Question1()
{

   logFilePath = outputFolder + "Question1.txt";
   string tableFilePath = outputFolder + "Question1Table.txt";
   ClearLogFile();
   ClearFile(tableFilePath);

   Log("Running question 1");
   Log("");


   int[] primesBelow50 = PrimesAndFactorisation.PrimesLessThan(50);

   long[] numbersToTestSpecifically = new long[] { 1, 318, 28577, 65536 };

   Log("Testing if numbers are B-numbers with B = " + primesBelow50.ToFormattedString());
   Log("n\tB-number\tFactorisation if a B-number");
   foreach (long number in numbersToTestSpecifically)
   {
      (int, int)[] factorisation;

      bool BNumber = number.IsProductOfBNumbers(primesBelow50, false, out factorisation);
      Log(number + "\t" + BNumber + "\t" + factorisation.ToFormattedString());
   }

   int numberOfSamples = 100000;

   Log("");
   Log("Generating max(" + numberOfSamples + ", 10^d - 10^{d-1} + 1) samples of d digit
       numbers and finding the probability they can be written as a product of B = " +
       primesBelow50.ToFormattedString());
   Log("");

   string tableHeading = "d\tsamples\tB_numbers_found\tprobability";
   Log(tableHeading);
   LogGivenFile(tableHeading, tableFilePath);
```

```
   for (int d = 1; d <= 14; d++)
   {
      long[] sample = GenericExtensions.DistinctRandomIntegersWithGivenDigits(d,
          numberOfSamples);
      int actualNumberOfSamples = sample.Length;
      int BNumberCount = 0;

      foreach (long number in sample)
      {
         if (number.IsProductOfBNumbers(primesBelow50, false))
         {
            BNumberCount++;
         }
      }
      double probability = (double)BNumberCount / (double)actualNumberOfSamples;


      string line = d + "\t" + actualNumberOfSamples + "\t" + BNumberCount + "\t" +
          probability.ToString("0.00000000");
      Log(line);
      LogGivenFile(line, tableFilePath);
   }


}

void Question2()
{
   logFilePath = outputFolder + "Question2.txt";
   string tableFilePath;
   string heading;
   ClearLogFile();

   int length = 50;

   ulong[] NsToFindCFE = new ulong[] { 7, 13, 19 };
   foreach (ulong N in NsToFindCFE)
   {
      tableFilePath = outputFolder + "Question2Table_" + N + ".txt";
      ClearFile(tableFilePath);
      Log("Calculating the CFE of sqrt(" + N + ")");
      (ulong, ulong)[] RSValues;
      ulong[] AValues = N.ContinuedFractionExpansionOfSquareRoot(out RSValues, length);

      Log("A values: " + AValues.ToFormattedString());

      (ulong, ulong)[] PQValues = AValues.PartialQuotients();

      heading = "n\tr_n\ts_n\ta_n\tp_n\tq_n";
      Log(heading);
      LogGivenFile(heading, tableFilePath);
      for (int i = 0; i < length; i++)
      {
         string tableLine = (i+1) + "\t" + RSValues[i].Item1 + "\t" + RSValues[i].Item2
             + "\t" + AValues[i] + "\t" + PQValues[i].Item1 + "\t" + PQValues[i].Item2;
         Log(tableLine);
         LogGivenFile(tableLine, tableFilePath);
      }
   }
```

```
//investigate how large r and s can become

tableFilePath = outputFolder + "Question2Table_investigation.txt";
ClearFile(tableFilePath);

int investigateUpTo = 1000;

int investigateUpToLength = 1000;

//how far should we check maxmaxSOverRootN up to?
ulong investigateUpToBestMaxMaxSOverRootN = 1000000;


Log("Investigating how large r and s become in the first " + investigateUpToLength +
    " convergents for sqrt(N) up to " + investigateUpTo);


heading = "N\tmax_r_found\tmax_s_found\tmax_s_found_over_root_N";

Log(heading);
LogGivenFile(heading, tableFilePath);

List<ulong> MaxSIsOne = new List<ulong>();

double maxmaxSOverRootN = 0;

for (ulong N = 2; N <= Convert.ToUInt64(investigateUpTo); N++)
{
   if (N.IsSquare() == false)
   {
      (ulong, ulong)[] RSValues;
      N.ContinuedFractionExpansionOfSquareRoot(out RSValues, investigateUpToLength);
      //remark: the values for a will have overflows here, but we dont care about
          them, r and s are sufficiently small (as defined by recursion relations)
          that we dont care

      ulong maxR = RSValues.Select(i => i.Item1).Max();
      ulong maxS = RSValues.Select(i => i.Item2).Max();

      double maxSOverRootN = (double)maxS / Math.Sqrt(N);

      string line = N + "\t" + maxR + "\t" + maxS + "\t" + maxSOverRootN;

      if (maxS == 1)
      {
         MaxSIsOne.Add(N);
      }
      if (maxSOverRootN > maxmaxSOverRootN)
      {
         maxmaxSOverRootN = maxSOverRootN;
      }
      Log(line);
      LogGivenFile(line, tableFilePath);
   }


}
```

```
   Log("max s is one at " + MaxSIsOne.ToArray().ToFormattedString());


   for (ulong N = 2; N <= investigateUpToBestMaxMaxSOverRootN; N++)
   {
      if (N.IsSquare() == false)
      {
         (ulong, ulong)[] RSValues;
         N.ContinuedFractionExpansionOfSquareRoot(out RSValues, investigateUpToLength);
         //remark: the values for a will have overflows here, but we dont care about
             them, r and s are sufficiently small (as defined by recursion relations)
             that we dont care

         ulong maxS = RSValues.Select(i => i.Item2).Max();

         double maxSOverRootN = (double)maxS / Math.Sqrt(N);


         if (maxSOverRootN > maxmaxSOverRootN)
         {
            maxmaxSOverRootN = maxSOverRootN;
         }
      }

      if (N % 100000 == 0)
      {
         Log("maximum of max s over root N found up to " + N + " is " +
             maxmaxSOverRootN);
      }
   }


}

void Question3()
{
   logFilePath = outputFolder + "Question3.txt";
   string tableFilePath = outputFolder + "Question3Table.txt";
   ClearLogFile();
   ClearFile(tableFilePath);


   Log("Testing select (x, y, N) pairs to see if they solve Pell's equation.");


   (ulong, ulong, ulong)[] xyNPairs = new (ulong, ulong, ulong)[] { (32080051, 3115890,
       106), (8890182, 851525, 109), (253293,35989,113) };


   foreach ((ulong, ulong, ulong) pair in xyNPairs)
   {

      (bool, int) satisfies = PellsEquation.SatisfiesPMPellsEquation(pair.Item1,
          pair.Item2, pair.Item3);

      if (satisfies.Item1)
      {
```

```
            Console.WriteLine("(x,y,N) = (" + pair.Item1 + ", " + pair.Item2 + ", " +
                pair.Item3 + ") satisfies x^2 - Ny^2 = " + satisfies.Item2);
        }
        else
        {
            Console.WriteLine("(x,y,N) = (" + pair.Item1 + ", " + pair.Item2 + ", " +
                pair.Item3 + ") does not satisfy x^2 - Ny^2 = +-1");
        }


    }


    Log("Solutions to x^2 - Ny^2 = 1 and x'^2 - Ny'^2 = -1 for required N:");
    string heading = "N\tx\ty\tx'\ty'";

    Log(heading);
    LogGivenFile(heading, tableFilePath);

    for (ulong N = 1; N <= 100; N++)
    {
        Question3PellTest(N, tableFilePath);
    }
    for (ulong N = 500; N <= 550; N++)
    {
        Question3PellTest(N, tableFilePath);
    }
}

void Question3PellTest(ulong N, string tableFilePath)
{
    //Theorem says that the first solution will be either x_{k-1}, y_{k-1} or x_{2k-1},
        y_{2k-1} for k the period of the CFE.
    //project instructions does not necessarily need that so we take a somewhat more
        naive approach and just check all partial convergents
    //up to some number. Let's check it up to 2k-1 then.

    int period = ContinuedFractions.PeriodOfContinuedFractionExpansionOfSquareRoot(N);

    (ulong, ulong) pellSol = PellsEquation.FindPellSolutionUsingContinuedFractions(N, 2 *
        period);
    (ulong, ulong) negativePellSol =
        PellsEquation.FindNegativePellSolutionUsingContinuedFractions(N, period - 1);


    string line = N.ToString();


    if (pellSol == (0, 0)) //we failed to find one!
    {
        //throw new Exception("Did not find a solution for " + N + ".");
        line += "\t\t";

    }
    else
    {
        line += "\t" + pellSol.Item1 + "\t" + pellSol.Item2;
    }
    if (negativePellSol == (0, 0)) //we failed to find one!
```

```
   {
      //throw new Exception("Did not find a solution for " + N + ".");
      line += "\t\t";

   }
   else
   {
      line += "\t" + negativePellSol.Item1 + "\t" + negativePellSol.Item2;
   }

   Log(line);
   LogGivenFile(line, tableFilePath);
}

void Question5()
{
   logFilePath = outputFolder + "Question5.txt";
   string tableFilePath = outputFolder + "Question5Table.txt";
   ClearLogFile();
   ClearFile(tableFilePath);

   ulong[] Nvalues = new ulong[] { 2012449237, 2575992413, 3548710699 };
   int upTo = 50;




   List<(ulong, ulong)[]> results = new List<(ulong, ulong)[]>();

   string heading = "n";

   foreach (ulong N in Nvalues)
   {
      results.Add(ContinuedFractions.ConvergentNumeratorsModulo(N, upTo+1)); //find the
          results

      heading += "\tP_n_mod_" + N + "\t" + "P_n^2_mod_" + N;
   }

   Log(heading);
   LogGivenFile(heading, tableFilePath);


   for (int i = 0; i <= upTo; i++)
   {
      string line = i + "\t";
      for (int j = 0; j < Nvalues.Length; j++)
      {
         line += "\t" + results[j][i].Item1 + "\t" + results[j][i].Item2;
      }
      Log(line);
      LogGivenFile(line, tableFilePath);
   }

   int upToFindMatches = 500;
   Log("Matches:");
   //See if we have matches.
   foreach (ulong N in Nvalues)
   {
```

```
      Log("\tFor N = " + N + ":");
      (ulong, ulong)[] values = ContinuedFractions.ConvergentNumeratorsModulo(N,
          upToFindMatches + 1);

      ulong[] squares = values.Select(x => x.Item2).ToArray();

      ulong[] duplicatedSquares = squares.AllEntriesThatAppearMoreThanOnce();

      foreach (ulong duplicatedSquare in duplicatedSquares)
      {
         Log("\t\tThe square " + duplicatedSquare + " is duplicated at:");
         int[] indices = squares.IndicesOf(duplicatedSquare);

         Log("\t\tindex\tP_n mod N\tP_n^2 mod N");

         foreach (int index in indices)
         {
            Log("\t\t" + index + "\t" + values[index].Item1 + "\t" + values[index].Item2
                );
         }
      }

   }

}

//Gaussian elimination of a matrix
void Question6()
{
   logFilePath = outputFolder + "Question6.txt";
   //string tableFilePath = outputFolder + "Question5Table.txt";
   ClearLogFile();
   //ClearFile(tableFilePath);
   (int, int)[] sizesToGenerate = new (int, int)[] { (2, 2), (2, 2), (2, 3), (3, 2), (3,
       3), (3, 3), (3, 4), (4, 3), (2, 4), (4,2) };

   foreach ((int, int) sizeToGenerate in sizesToGenerate)
   {

      Matrix m;

      do
      {
         m = Matrix.RandomMatrix(2, sizeToGenerate.Item1, sizeToGenerate.Item2);
      } while (m.AllEntriesZero());

      Matrix rref = m.GetReducedRowEchelonForm(2);


      Matrices.Vector[] basis = rref.GetBasisOfKernel(2);

      string basisToString;

      if (basis.Length == 0)
      {
         basisToString = "\\emptyset";
      }
      else
      {
```

```
            basisToString = "\\left\\{";

               foreach (Matrices.Vector v in basis)
               {
                  basisToString += "" + v.ToLaTeX() + ", ";
               }
               basisToString = basisToString.Substring(0, basisToString.Length - 2);

               basisToString += "\\right\\}";
         }




         Log("$" + m.ToLaTeX() + "$ & $" + rref.ToLaTeX() + "$ & $" + basisToString +
             "$\\\\");
      }



   }

   void Question7()
   {
      logFilePath = outputFolder + "Question7.txt";
      //
      ClearLogFile();


      //Part one of question 7, test some N



      ulong[] numbersToTest = new ulong[] { 2012449237, 2575992413, 3548710699, 377691131,
          175224311, 48958009, 483205427 };

      Log("Testing the following N: " + numbersToTest.ToFormattedString());

      foreach (ulong number in numbersToTest)
      {

         int convergentsRequired;
         long foundFactor = Question7WithNumber(number, out convergentsRequired, true, 0,
             50);

         if (foundFactor == 0)
         {
            Log(number + " is prime");

            Log("");
         }
         else
         {
            Log("Factored " + number + " as " + foundFactor + " x " + number /
                (ulong)foundFactor);
```

```
      Log("");
   }

   Log("Was able to figure out the factorisation of " + number + " in " +
       convergentsRequired + " convergents.");


}

Log("");



//part 2 of question 7, Investigate the number of convergents typically required for
    factorization.

string tableFilePath = outputFolder + "Question7ConvergentInvestigationTable.txt";

ClearFile(tableFilePath);

int giveUpAfter = 1500;

for (ulong i = 1; i <= 100; i++) //was: 1000000
{
   int numberOfConvergentsRequired;



   Question7WithNumber(i, out numberOfConvergentsRequired, false, giveUpAfter, 50);

   if (numberOfConvergentsRequired == -1) //failed
   {
      //Log(i + "\t>" + giveUpAfter);
      Console.WriteLine(i + "\t>" + giveUpAfter);
      LogGivenFile(i + "\t>" + giveUpAfter, tableFilePath);
   }
   else
   {
      //Log(i + "\t" + numberOfConvergentsRequired);
      Console.WriteLine(i + "\t" + numberOfConvergentsRequired);
      LogGivenFile(i + "\t" + numberOfConvergentsRequired, tableFilePath);
   }
}

Log("[Rest of testing omitted in output.]");

Log("");

Log("Third part of question 7, investigate different choices of B");

giveUpAfter = 10000;

tableFilePath = outputFolder + "Question7BChoices.txt";

ClearFile(tableFilePath);

//part 3 of question 7, investigate different choices of B

string header = "";
```

```
   header += "primeUpTo\t";


   foreach (ulong number in numbersToTest)
   {
      header += number.ToString();
      header += "\t";
   }

   Log(header);
   LogGivenFile(header, tableFilePath);


   for (int numberOfPrimes = 10; numberOfPrimes <= 200; numberOfPrimes += 10)
   {

      string line = "";

      line += numberOfPrimes + "\t";
      foreach (ulong number in numbersToTest)
      {
         int convergentsRequired;
         Question7WithNumber(number, out convergentsRequired, false, giveUpAfter,
             numberOfPrimes);
         line += convergentsRequired + "\t";
      }
      Log(line);
      LogGivenFile(line, tableFilePath);
   }

}


long Question7WithNumber(ulong N, out int convergentsRequired, bool printOutDetails, int
    giveAfterNumberOfConvergents, int primeUpTo)
{
   convergentsRequired = 0;
   if (printOutDetails)
   {
      Log("Running Q7 with N=" + N);
   }


   if (N.IsSquare()) //special case
   {
      return (long)Math.Sqrt(N);
   }
   else if (N.IsPrime()) //special case
   {
      return 0;
   }

   int[] B = PrimesAndFactorisation.PrimesLessThan(primeUpTo);



   B = B.Append(-1).ToArray();

   if (printOutDetails)
```

```
{
   Log("B = " + B.ToFormattedString());

}


//B is the set as described in the project

int upToConvergent = 50;



bool foundFactorisation = false;

do
{
   if (upToConvergent > giveAfterNumberOfConvergents && giveAfterNumberOfConvergents
       > 0)
   {
      convergentsRequired = -1;
      return 0;
   }
   if (printOutDetails)
   {
      Log("Going to check up to n=" + upToConvergent + " for the convergents. Will
          increase if nothing is found.");
   }


   //First, we find some B numbers



   (ulong, ulong)[] RSValues;

   //get the p_n mod N and p_n^2 mod N
   (ulong, ulong)[] convergents = ContinuedFractions.ConvergentNumeratorsModulo(N,
       upToConvergent);

   //The way we do it, p_n^2 mod N is within 0 and N-1. Want it within -N/2, N/2.
   (ulong, long)[] adjustedConvergents = convergents.Select(x => (x.Item1,
       x.Item2.ModWithinHalf((long)N))).ToArray();



   //filter out which ones are B numbers in their p_n^2 entry. also keep track of n

   List<(ulong, long, int)> adjustedConvergentsBNumbers = new List<(ulong, long,
       int)>();

   int n = 0;

   if (printOutDetails)
   {
      Log("We found the following B numbers in the continued fraction of sqrt(N)");

      Log("n\tp_n mod N\t<p_n^2>'_N");
   }
```

```
foreach ((ulong, long) pnpnsquaredpair in adjustedConvergents)
{

   if (pnpnsquaredpair.Item2.IsProductOfBNumbers(B, false)) //is it a B number?
   {

      adjustedConvergentsBNumbers.Add((pnpnsquaredpair.Item1,
          pnpnsquaredpair.Item2, n));

      if (printOutDetails)
      {
         Log(n + "\t" + pnpnsquaredpair.Item1 + "\t" + pnpnsquaredpair.Item2);
      }




   }
   n++;
}

if (adjustedConvergentsBNumbers.Count > 0)
{
   //we now have the B numbers. make the matrix for which we will use as outlined
       in Q7 in the project.


   int numberOfBNumbers = adjustedConvergentsBNumbers.Count;

   int sizeOfB = B.Length;

   Matrix primeParityMatrix = new Matrix(sizeOfB, numberOfBNumbers);

   int column = 1;
   foreach ((ulong, long, int) pairWithbNumber in
       adjustedConvergentsBNumbers.ToArray())
   {
      long bNumber = pairWithbNumber.Item2;

      int row = 1;
      foreach (int primeOrMinusOne in B)
      {
         int parity = 0;
         //what is the parity of primeOrMinusOne in the factorisation of the b
             number?

         if (primeOrMinusOne == -1) //do this separately
         {
            if (bNumber < 0)
            {
               parity = 1;
            }
         }
         else
         {
            long current = bNumber;
            while (current % primeOrMinusOne == 0)
```

```
        {
           parity++;
           current /= primeOrMinusOne;
        }
     }

     parity = parity % 2;

     primeParityMatrix[row, column] = parity;


     row++;
  }

  column++;
}


//We have constructed the prime parity matrix.

if (printOutDetails)
{
  Log("Prime parity matrix:");

  Log(primeParityMatrix.ToLaTeX());
}



//find its kernel. sort it by minimal number of convergents required.

Vector[] kernel = primeParityMatrix.GetBasisOfKernel(2);


kernel = kernel.OrderBy(x => x.LastNonZeroIndex()).ToArray();

if (printOutDetails)
{
  Log("Found " + kernel.Length + " vectors in the kernel. Sorting them through
      and checking them by latest index:");

  foreach (Vector vector in kernel)
  {
     Log("\t" + vector.ToString());
  }
}




if (kernel.Length > 0) //we found a valid vector.
{

  if (printOutDetails)
  {
     Log("Found non-trivial vectors in the prime parity matrix kernel which we
         will now iterate through.");
  }
```

```
//loop through vectors in the kernel until we find a valid factorisation.

foreach (Vector kernelVector in kernel)
{

   if (printOutDetails)
   {
      Log("Testing if " + kernelVector.ToString() + " gives a valid
          factorisation.");
   }



   //we try to find x, y such that x^2 = y^2 like this

   long x = 1; //x

   int[] elementsOfBFactorCountInY = new int[B.Length];


   for (int i = 1; i <= kernelVector.Size; i++)
   {
      if (kernelVector[i] == 1)
      {
         x *= (long)adjustedConvergentsBNumbers[i - 1].Item1;
         x = x.Mod(N);

         long pnSquaredMod = adjustedConvergentsBNumbers[i - 1].Item2;

         for (int j = 0; j < B.Length; j++)
         {
            int elementOfB = B[j];

            if (elementOfB != -1) //ignore -1, its irrelevant when taing a
                square root as we know we have an even no of them
            {
               long current = pnSquaredMod;

               while (current % elementOfB == 0)
               {
                  current /= elementOfB;
                  elementsOfBFactorCountInY[j]++;
               }
            }
         }

         //productOfPnSquared *= adjustedConvergentsBNumbers[i - 1].Item2;
         // productOfPnSquared = productOfPnSquared.Mod(N);
      }
   }

   long y = 1;

   for (int j = 0; j < B.Length; j++)
   {
      int numberOfTimes = elementsOfBFactorCountInY[j]; //we know this must
          be even by the squareness required

      int number = B[j];
```

```
   for (int i = 0; i < numberOfTimes / 2; i++)
   {
      y *= (long)number;
      y = y.Mod(N);
   }
}

//we have y^2, we want to find y

if (printOutDetails)
{
   Log("Found the pair x=" + x + ", y=" + y);
}



//Hopefully now, x =/= +- y mod N

if (x.Mod(N) != y.Mod(N) && x.Mod(N) != y.Mod(N))
{
   //hopefully now, (N, x-y) and (N, x+y) are non trivial factors

   long nonTrivialFactorCandidate1 = GenericExtensions.GCD((long)N, (x -
       y).Mod(N));

   long nonTrivialFactorCandidate2 = GenericExtensions.GCD((long)N, (x -
       y).Mod(N));

   //bool works = false;



   if (nonTrivialFactorCandidate1 != 1 && nonTrivialFactorCandidate1 !=
       (long)N)
   {
      if ((long)N % nonTrivialFactorCandidate1 == 0)
      {
         //works = true;

         if (printOutDetails)
         {
            Log("Found the factor (N,x+y)=" + nonTrivialFactorCandidate1);
         }

         //worked! how many convergents did we need?

         convergentsRequired =
             adjustedConvergentsBNumbers[kernelVector.LastNonZeroIndex()-1].Item3;

         return nonTrivialFactorCandidate1;
      }
   }
   else if (nonTrivialFactorCandidate2 != 1 && nonTrivialFactorCandidate2
       != (long)N)
   {
      if ((long)N % nonTrivialFactorCandidate2 == 0)
      {
         //works = true;
```

```
                    if (printOutDetails)
                    {
                       Log("Found the factor (N, x-y)=" + nonTrivialFactorCandidate2);
                    }

                    //worked! how many convergents did we need?

                    convergentsRequired =
                        adjustedConvergentsBNumbers[kernelVector.LastNonZeroIndex()].Item3;

                    return nonTrivialFactorCandidate2;
                 }
              }
              else
              {

                 if (printOutDetails)
                 {
                    Log("No non-trivial factors found. (N, x+y)=" +
                        nonTrivialFactorCandidate1 + ", (N, x-y)=" +
                        nonTrivialFactorCandidate2);
                 }

              }
           }

           else
           {
              if (printOutDetails)
              {
                 Log("Has x = +- y");
              }

           }
        }


     }

     else
     {
        //did not find something

        if (printOutDetails)
        {
           Log("Couldn't find a valid vector. Searching further.");
        }

     }
  }



  upToConvergent += 50;
} while (true);
```

```
}


void Log(string str)
{
   Console.WriteLine(str);
   LogGivenFile(str, logFilePath);

}
void LogGivenFile(string str, string path)
{


   if (!File.Exists(path))
   {
      File.Create(path).Close();
   }
   using (StreamWriter sw = File.AppendText(path))
   {
      sw.WriteLine(str);
   }

}

void ClearLogFile()
{
   ClearFile(logFilePath);
}

void ClearFile(string path)
{
   File.Delete(path);
   File.Create(path).Close();
}
```

Appendix B: Output

Question 1 output