Fundamental Concepts of Cryptography
Assignment 2 Report
Semester 1, 2020

Mahmudul Hossain (19303235)

1

In the process of implementing RSA, in order to determine whether two numbers are co-prime, The Extended Euclidean Algorithm is used which calculated the greatest common divisor between two imported numbers alongwith calculating the multiplicative inverse of a mod n where numbers a and n are imported. Moreover, CalcGCD determines the greatest common divisor between two numbers using the Extended Euclidean Algorithm. The source code with detailed explanation is provided below :

Source code :

```
//Extended Euclidean algorithm
//Returns the greatest common divisor (d) between a and n stored in index 0
//Returns the Multiplicative Modular Inverse of a (x) with modulo n in index 1
//Generate x and y such that a*x + n*y + d = gcd(a, n)
//Code adapted from : https://www.sanfoundry.com/java-program-extended-euclid-algorithm/
public static long[] gcdExtended(long a, long n)
{
  //0th index stores the gcd value
  //1st index stores the value of x
  long[] array = new long[2];

  //Initial values of
  long x = 0, y = 1;
  long finalX = 1, finalY = 0;

  //The quotient and remainder to store in each iteration
  //Acts as an intermediate variable to swap between x, finalX and
  //y, finalY.
  long quotient, remainder, temp;

  //If finalX is negative, use initial
  //to make finalX positive
  long initial = n;

  //If n == 0, gcd has been obtained
  while(n != 0)
  {

    //Steps described indepth in the book :
    //Cryptography and Network Security : Principles and Practice (6th edition)
```

```
        //Page 98 and
        //Page 99 Table 4.4


        //Determines the greatest common divisor
        quotient = a / n;
        remainder = a % n;
        a = n;
        n = remainder;

        //Updates value of finalX and finalY in each iteration
        temp = x;
        x = finalX - (quotient * x);
        finalX = temp;

        temp = y;
        y = finalY - (quotient * y);
        finalY = temp;
    }

    //Here finalX and finalY have opposite signs to one another
    //finalX stores the actual multiplicative inverse so we
    //dont care about finalY
    //If finalX is positive then, finalX is the multiplicate inverse
    //else if it is negative, finalX must be turned into positive by
    //adding
    //finalX is such that (finalX * a) mod n == 1
    if(finalX < 0)
    {
        finalX = finalX + initial;
    }
    array[0] = a; //Greatest common divisor value
    array[1] = finalX; //Multiplicative inverse value

    return array;

}
```

Mathematical proof :

gcd(a, b) = gcd(b, a mod b)   in case a > b

Let d = gcd(a, b)

Producing d in a way such that, d | a and d | b

On the other hand, for any positive value of b we can expressed a as with the definition of GCD as,

a = k*b + r ≡ r (mod b)
a mod b = r   where k and r are integers,

Hence,   a mod b = a – k*b

But because d | b it also divides k*b
Thus, d | (a mod b) meaning d is a common divisor of b and (a mod b).

Also, if d is a common divisor of b and (a mod b), then d | k*b and

thus, d | [k*b + (a mod b)], which is equivalent to d | a.

As a result, the set of common divisors between a and b is equal to the set of common divisors of b and (a mod b), including the greatest common divisor.

Finally,

gcd(a, b) = gcd(b, a mod b) in case a > b   (Proven)

The proof has been adapted from the book, Cryptography and Network Security : Principles and Practice (6th edition) Page 96 (4.6)

To provide a brief generalisation, I have implemented RSA algorithm using Java.
All the mathematical functions that were used to generate public and private keys are stored inside the Number class. My RSA basically imported a filename containing plain text, opened it and encrypted each line of the file converting each character of the line to ASCII into a large number that is stored in encrypted.txt, where each number is delimited by a space. Using the numbers obtained from encryption, decryption is performed sequentially obtaining characters equivalent to the plain text and saving it into decrypted.txt.
For detailed explanations of each function used in my code, I will now provide my code with detailed comments below :

```
 public class RSA
{

   //Minimum range for prime number to be generated
   public static final long MINPRIME = 10000;
   //Maximum range for prime number to be generated
   public static final long MAXPRIME = 100000;


   //Prime numbers p and q
   private long p;
   private long q;

   //n = p * q
   //Used as the modulus for both encryption
   //and decryption
   private long n;

   //Euler Totient value
   // phi = (p-1) * (q-1)
   private long phi;

   //Public key for encryption
   private long publicE;

   //Private key for decryption
   private long privateD;


   public RSA(String filename)
   {
      //Generate random prime numbers for p and q
      //within the specified range
      p = Number.generateRandomPrime(MINPRIME, MAXPRIME);
      q = Number.generateRandomPrime(MINPRIME, MAXPRIME);

      //If by any chance p and q are same, this will generate
      //weak keys, which should be avoided for good practice
```

```
    while(p == q)
    {
        p = Number.generateRandomPrime(MINPRIME, MAXPRIME);
        q = Number.generateRandomPrime(MINPRIME, MAXPRIME);
    }

    //Calculate n to be used in encryption and decryption
    n = p * q;

    //Calculate Euler Totient value to be used to generate keys
    phi = (p - 1) * (q - 1);


    //Generate public key for encryption
    //in range 1 < publicE < phi where
    //publicE and phi are co-prime to each other
    publicE = this.getPublicKey();

    //Generate private key, finding the modular multiplicative
    //inverse of publicE mod phi
    //it satifies the condition :
    //(privateD * publicE) mod phi = 1
    privateD = Number.gcdExtended(publicE, phi)[1];

    //Perfrom encryption and decryption on file
    this.fileOp(filename);



}

//Perform rsa encryption to the imported line
//by encrypting each character M in the expression generating C :
//C = (M^publicKey) mod n   where M < n
//Returns encrypted line containing numeric value
public String encrypt(String line)
{

    String encrypted = new String();
    for(int ii = 0; ii < line.length(); ii++)
    {
        //Convert each character to ASCII equivalent
        long plain = (long)line.charAt(ii);

        //Calculates (plain ^ publicE) mod n
        //BigInteger datatype used here to prevent overflow of numbers
        //since the returned expression might be a very large value
        //where primitive data type might fail to store
        BigInteger value = Number.modularExponent(plain, publicE, n);

        //Store each calculated large value as string separated by <space>
        //making it easier for decryption process
        encrypted += value.toString() + " ";
    }

    return encrypted;

}
```

```java
//Perform rsa decryption to the imported line containing numeric values
//by decrypting each value C in the expression generating M :
//M = (C ^ privateKey) mod n where M < n
//Return decrypted line of characters
public String decrypt(String line)
{
   //Store each encrypted values in a string array
   String[] array = line.split(" ");
   String decrypt = new String();
   for(int ii = 0; ii < array.length; ii++)
   {
      //Obtain the encrypted value C for each character
      long value = Long.parseLong(array[ii]);

      //Calculates (value ^ privateKey) mod n
      //BigInteger datatype used here to prevent overflow of numbers
      //since the returned expression might be a very large value
      //where primitive data type might fail to store
      BigInteger d = Number.modularExponent(value, privateD, n);

      //The decrypted value d will be same as the ASCII value of plain character
      long c = d.longValue();

      //Map the ASCII number to appropriate character
      decrypt += (char)c;

   }

   return decrypt;

}

//Generate public key such that :
//1 < public key < phi     AND
//public key is co-prime to phi
public long getPublicKey()
{

   Random rand = new Random();
   boolean valid = false;

   //Public key to be determined
   long e = 0;

   //Generate e until it is avalid public key
   while(!(valid))
   {
      //Generate random values of e in range : 1 < e < phi
      e = (rand.nextLong() + 2) % phi;

      //Ensure that e is odd, co-prime to phi and positive to ensure that e
      //avoids any overflow during random generation
      if((e % 2 != 0) && (Number.gcdExtended(e, phi)[0] == 1) && (e > 0))
      {
         //Valid public key
         valid = true;
      }
```

```java
    }

    return e;

}


//Open and read the input filename
//Encrypts each line read which is saved into encrypted.txt
//Decrypts each encrypted line which is saved into decrypted.txt
//Code adjusted from assignment 1
public void fileOp(String filename)
{
    Scanner sc;
    File fileToOpen;
    String line = new String();
    File encryptedOutput, decryptedOutput;
    PrintWriter pwEncrypted, pwDecrypted;
    int lineNum  = 1;

    try
    {
        encryptedOutput = new File("encrypted.txt");
        decryptedOutput = new File("decrypted.txt");

        pwEncrypted = new PrintWriter(encryptedOutput);
        pwDecrypted = new PrintWriter(decryptedOutput);

        //Open the plain text file for rsa operation
        fileToOpen = new File(filename);
        sc = new Scanner(fileToOpen);

        //Read every line of input file
        //Perform RSA encryption and decryption
        while(sc.hasNextLine())
        {
            //Plain text to encrypt
            line = sc.nextLine();

            //Ignore empty lines and print a new line
            if(line.isEmpty())
            {
                pwEncrypted.println(" ");
                pwDecrypted.println(" ");
            }
            else
            {
                //Obtain the encrypted text
                String text = this.encrypt(line);

                //Obtain the decrypted text
                String decipher = this.decrypt(text);

                //Compare the plain text and decrypted line to ensure successful
                //rsa encryption and decryption
                //If an error occurs, output the line number where the rsa failed
                if(!(line.compareTo(decipher) == 0))
                {
```

```
                System.out.println("ERROR at line number : " + lineNum);
             }

             //Print the encrypted and decrypted texts into appropriate files
             pwEncrypted.println(text);
             pwDecrypted.println(decipher);
          }
          lineNum++;

       }

       pwEncrypted.close();
       pwDecrypted.close();
       sc.close();
    }
    catch(IOException e)
    {
       System.err.println(e.getMessage());
    }

 }

}
```

```
public class Number
{
   //Implementation adapted from the book :
   //Cryptography and Network Security : Principles and Practice (6th edition)
   //Page 269 Figure : 9.8
   //Calculates   (base^exponent) mod mudulus
   //Return as BigInteger datatype since the calculated value might
   //be very large
   //modularExponent acts as a replacement of Math.pow() % modulus
   //where this function is able to store very large numbers during the intermediate stages
   //of calculation ensuring good accuracy of calculated value for the encryption and
   //decryption process of RSA, avoiding chances of number overflow
   public static BigInteger modularExponent(long base, long exponent, long modulus)
   {
      //Convert exponent to string of binary bitstring
      String expBits = convertToBinary(exponent);

      //BigInteger used to store very large intermediate values
      BigInteger c = BigInteger.valueOf(0), f = BigInteger.valueOf(1);

      for(int i = 0; i < expBits.length(); i++)
      {
         c = c.multiply(new BigInteger("2")); // c = c * 2

         //f = (f * f) % modulus
         f = f.multiply(f);
         f = f.mod(BigInteger.valueOf(modulus));

         if(expBits.charAt(i) == '1')
         {
```

```
            c = c.add(new BigInteger("1")); // c = c + 1

            //f = (f * base) % modulus
            f = f.multiply(BigInteger.valueOf(base));
            f = f.mod(BigInteger.valueOf(modulus));
        }

    }

    //f = (base ^ exponent) % modulus
    return f;
}



//Finds the greatest common divisor among num1 and num2
//Code adapted from the pseudocode found in lecture 6 slide 19
//Same program used in assignment_1
public static long gcd(long num1, long num2)
{
    long temp;

    while(num2 != 0)
    {
        temp = num2;
        num2 = num1 % num2;
        num1 = temp;
    }

    return num1;
}


//Extended Euclidean algorithm
//Returns the greatest common divisor (d) between a and n stored in index 0
//Returns the Multiplicative Modular Inverse of a (x) with modulo n in index 1
//Generate x and y such that a*x + n*y + d = gcd(a, n)
//Code adapted from : https://www.sanfoundry.com/java-program-extended-euclid-algorithm/
public static long[] gcdExtended(long a, long n)
{
    //0th index stores the gcd value
    //1st index stores the value of x
    long[] array = new long[2];

    //Initial values of
    long x = 0, y = 1;
    long finalX = 1, finalY = 0;

    //The quotient and remainder to store in each iteration
    //Acts as an intermediate variable to swap between x, finalX and
    //y, finalY.
    long quotient, remainder, temp;

    //If finalX is negative, use initial
    //to make finalX positive
    long initial = n;

    //If n == 0, gcd has been obtained
```

```java
    while(n != 0)
    {

        //Steps described indepth in the book :
        //Cryptography and Network Security : Principles and Practice (6th edition)
        //Page 98 and
        //Page 99 Table 4.4


        //Determines the greatest common divisor
        quotient = a / n;
        remainder = a % n;
        a = n;
        n = remainder;

        //Updates value of finalX and finalY in each iteration
        temp = x;
        x = finalX - (quotient * x);
        finalX = temp;

        temp = y;
        y = finalY - (quotient * y);
        finalY = temp;
    }

    //Here finalX and finalY have opposite signs to one another
    //finalX stores the actual multiplicative inverse so we
    //dont care about finalY
    //If finalX is positive then, finalX is the multiplicate inverse
    //else if it is negative, finalX must be turned into positive by
    //adding
    //finalX is such that (finalX * a) mod n == 1
    if(finalX < 0)
    {
        finalX = finalX + initial;
    }
    array[0] = a; //Greatest common divisor value
    array[1] = finalX; //Multiplicative inverse value

    return array;

}

//Generate a random prime number within range using Lehmann Algorithm
public static long generateRandomPrime(long minPrime, long maxPrime)
{
    int min = (int)minPrime;
    int max = (int)maxPrime;
    Random rand = new Random();
    boolean surePrime = false;
    long prime = 0;

    //Loop until a prime number is found
    while(!(surePrime))
    {
        //Generate a random number within the specified range
        prime = (long)rand.nextInt((max - min) + 1) + min;
```

```java
            //Use Lehmann Algorithm to determine if the randomly generate
            //number is a prime
            if(checkifPrime(prime))
            {
               surePrime = true;
            }
         }

         return prime;
      }


      //Use Lehmann algorithm to check if the imported number num is a prime
      //Code adjusted from FCC prac 2
      //Step 1 : Generate a random number randomNum such that 0 < randomNum < num
      //Step 2 : Calculate r in the form : r = randomNum^((num - 1) / 2) mod num
      //Step 3 : Check to see if r satisfies certain conditions to determine if
      //         num is actually prime
      //Step 4 : Repeat Steps 1 to 3 100 times to gain sufficient confidence
      //         that num is a prime number
      public static boolean checkifPrime(long num)
      {
         boolean flag = false;

         int t = 1;

         //Even numbers are not prime
         if( num % 2 == 0)
         {
            flag = false;
         }
         else
         {
            //Run the iteration 100 times to make a sufficient conclusion
            while(t < 100)
            {
               Random rand = new Random();

               //Step 1
               long randomNum = (rand.nextLong() + 1) % (num - 1) ;

               //Step 2
               //modularExponent is used to prevent any number overflow
               //which might result a prime number to be falsefully justified to be
               //not prime
               //If Math.pow() was used,the primality test failed for numbers
               //eg : 37, 41 etc at random runs resulting them to be not prime but infact
               //they are prime numbers
               long r = (modularExponent(randomNum, (num - 1)/2, num)).longValue();

               //Step 3
               if((r != 1) && (r != num - 1))
               {
                  //Here num is 100% not prime
                  flag = false;
               }
               else
               {
```

```
                //num is a prime number with probability 1 - (1/(2^t))
                 flag = true;
              }

              t++;
           }

        }

        return flag;
     }

     //Convert exponent number to a binary bit string
     public static String convertToBinary(long exponent)
     {
        return new String(Long.toBinaryString(exponent));
     }

}
```

Testing :

After a single run a comparison plain text, cipher text and decrypted text is shown below:

Plain text (testfile-DES.txt) :

```
For example as pointed out by researcher. For each set of fuzzy terms, $A \subseteq M$, $\prod_{m\in
A}m$ represents a conjunction of the fuzzy terms in $A$. For
instance,
 let $A=\{m_{1,2},m_{2,1},m_{4,2}\}\subseteq M$, a new
fuzzy concept ``$m_{1,2}$ and $m_{2,1}$ and $m_{4,2}$" with the linguist
interpretation ``\emph{short sepal and wide sepal and narrow petal}"
can be represented as $\prod_{m\in
 A}m=m_{1,2}m_{2,1}m_{4,2}$. Then the fuzzy rules can be represented as follows:

\bigskip

 \textbf{Rule} $R_1$ : If $x$ is
$m_{1,2}m_{2,1}m_{4,2}$, then $x$ belongs to Class 1;

\textbf{Rule} $R_2$ : If $x$ is $m_{2,1}m_{3,2}$, then $x$ belongs
to Class 1;

\textbf{Rule} $R_3$ : If $x$ is $m_{1,2}m_{4,2}$, then $x$ belongs
to Class 1.
\bigskip

\noindent Then, the antecedent of three fuzzy rules $R_1, R_2, R_3$ for Class
1 can be represented by ``or" as follows:
\bigskip

\textbf{Rule} $R$ : If $x$ is ``$m_{1,2}m_{2,1}m_{4,2}$ or
$m_{2,1}m_{3,2}$ or $m_{1,2}m_{4,2}$", then $x$ belongs to Class 1.

\bigskip

$\sum^{r}_{u=1}(\prod_{m\in A_u}m)$, which is a formal sum of the
fuzzy concepts $\prod_{m\in A_u}m$, $A_u \subseteq M$, is the disjunction of
the conjunctions represented by $\prod_{m\in A_u}m, u=1,\ldots,r$.
For example, let $A_1=\{m_{1,2},m_{2,1},m_{4,2}\},
A_2=\{m_{2,1},m_{3,2}\}, A_3=\{m_{1,2},m_{4,2}\} \subseteq M$, then
 a new
fuzzy set (i.e., fuzzy concept) as the disjunction of $\prod_{m\in A_1}m$,  $\prod_{m\in
A_2}m$,  $\prod_{m\in A_3}m$, i.e., ``$m_{1,2}m_{2,1}m_{4,2}$ or
$m_{2,1}m_{3,2}$ or $m_{1,2}m_{4,2}$'', can be represented as

\[\sum^{3}_{u=1}(\prod_{m\in A_u}m)=\prod_{m\in A_1}m + \prod_{m\in
                                                              40,0-1        Top
```

Cipher text (encrypted.txt) :

```
1961372736 3849710606 595712271 2122420156 814994099 2580667700 2340976364 3580779285 238346605 224056731 814994099 2122420156 2340976364 970191216 2
122420156 238346605 3849710606 2003612082 2259367056 489648676 814994099 1856423009 2122420156 3849710606 559078655 489648676 2122420156 2961162766 3
917638184 2122420156 595712271 814994099 970191216 814994099 2340976364 595712271 394642512 2835134224 814994099 595712271 721408620 2122420156 19613
72736 3849710606 595712271 2122420156 814994099 2340976364 394642512 2835134224 2122420156 970191216 814994099 489648676 2122420156 3849710606 159379
2375 2122420156 1593792375 559078655 206256765 206256765 3917638184 2122420156 489648676 814994099 595712271 3580779285 970191216 3216615932 21224201
56 2491777443 1508810809 2122420156 3462913703 970191216 559078655 2961162766 970191216 814994099 489648676 814994099 1616589132 2122420156 591544631
 2491777443 3216615932 2122420156 2491777443 3462913703 238346605 595712271 3849710606 1856423009 3122132732 2585533069 3580779285 3462913703 2003612
082 2259367056
1508810809 222134277 3580779285 2491777443 2122420156 595712271 814994099 238346605 595712271 814994099 970191216 814994099 2259367056 489648676 9701
91216 2122420156 2340976364 2122420156 394642512 3849710606 2259367056 3430653908 559078655 2259367056 394642512 489648676 2003612082 3849710606 2259
367056 2122420156 3849710606 1593792375 2122420156 489648676 2835134224 814994099 2122420156 1593792375 559078655 206256765 206256765 3917638184 2122
420156 489648676 814994099 595712271 3580779285 970191216 2122420156 2003612082 2259367056 2122420156 2491777443 1508810809 2491777443 721408620 2122
420156 1961372736 3849710606 595712271
2003612082 2259367056 970191216 489648676 2340976364 2259367056 394642512 814994099 3216615932
2122420156 224056731 814994099 489648676 2122420156 2491777443 1508810809 1788298836 3462913703 2585533069 3580779285 3122132732 2585533069 126190793
4 3216615932 2387291850 222134277 3216615932 3580779285 3122132732 2585533069 2387291850 3216615932 1261907934 222134277 3216615932 3580779285 312213
2732 2585533069 1348017340 3216615932 2387291850 222134277 3462913703 222134277 3462913703 970191216 559078655 2961162766 970191216 814994099 4896486
76 814994099 1616589132 2122420156 591544631 2491777443 3216615932 2122420156 2340976364 2122420156 2259367056 814994099 3423596272
1593792375 559078655 206256765 206256765 3917638184 2122420156 394642512 3849710606 2259367056 394642512 814994099 238346605 489648676 2122420156 264
9562113 2649562113 2491777443 3580779285 3122132732 2585533069 1261907934 3216615932 2387291850 222134277 2491777443 2122420156 2340976364 2259367056
 1856423009 2122420156 2491777443 3580779285 3122132732 2585533069 2387291850 3216615932 1261907934 222134277 2491777443 2122420156 2340976364 225936
7056 1856423009 2122420156 2491777443 3580779285 3122132732 2585533069 1348017340 3216615932 2387291850 222134277 2491777443 1764679469 2122420156 34
23596272 2003612082 489648676 2835134224 2122420156 489648676 2835134224 814994099 2122420156 224056731 2003612082 2259367056 2251104968 559078655 20
03612082 970191216 489648676
2003612082 2259367056 489648676 814994099 595712271 238346605 595712271 814994099 489648676 2340976364 489648676 2003612082 3849710606 2259367056 212
2420156 2649562113 2649562113 3462913703 814994099 3580779285 238346605 2835134224 2585533069 970191216 2835134224 3849710606 595712271 489648676 212
2420156 970191216 814994099 238346605 2340976364 224056731 2122420156 2340976364 2259367056 1856423009 2122420156 3423596272 2003612082 1856423009 81
4994099 2122420156 970191216 814994099 238346605 2340976364 224056731 2122420156 2340976364 2259367056 1856423009 2122420156 2259367056 2340976364 59
5712271 595712271 3849710606 3423596272 2122420156 238346605 814994099 489648676 2340976364 224056731 222134277 1764679469
394642512 2340976364 2259367056 2122420156 2961162766 814994099 2122420156 595712271 814994099 238346605 595712271 814994099 970191216 814994099 2259
367056 489648676 814994099 1856423009 2122420156 2340976364 970191216 2122420156 2491777443 3462913703 238346605 595712271 3849710606 1856423009 3122
132732 2585533069 3580779285 3462913703 2003612082 2259367056
2122420156 1508810809 222134277 3580779285 1788298836 3580779285 3122132732 2585533069 1261907934 3216615932 2387291850 222134277 3580779285 31221327
32 2585533069 2387291850 3216615932 1261907934 222134277 3580779285 3122132732 2585533069 1348017340 3216615932 2387291850 222134277 2491777443 72140
8620 2122420156 3399076286 2835134224 814994099 2259367056 2122420156 489648676 2835134224 814994099 2122420156 1593792375 559078655 206256765 206256
765 3917638184 2122420156 595712271 559078655 224056731 814994099 970191216 2122420156 394642512 2340976364 2259367056 2122420156 2961162766 81499409
9 2122420156 595712271 814994099 238346605 595712271 814994099 970191216 814994099 2259367056 489648676 814994099 1856423009 2122420156 2340976364 97
0191216 2122420156 1593792375 3849710606 224056731 224056731 3849710606 3423596272 970191216 1320131822

3462913703 2961162766 2003612082 2251104968 970191216 178504902 2003612082 238346605

"encrypted.txt" 204L, 109516C                                                                          1,1           Top
```

Decrypted text (decrypted.txt) :

```
For example as pointed out by researcher. For each set of fuzzy terms, $A \subseteq M$, $\prod_{m\in
A}m$ represents a conjunction of the fuzzy terms in $A$. For
instance,
 let $A=\{m_{1,2},m_{2,1},m_{4,2}\}\subseteq M$, a new
fuzzy concept ``$m_{1,2}$ and $m_{2,1}$ and $m_{4,2}$" with the linguist
interpretation ``\emph{short sepal and wide sepal and narrow petal}"
can be represented as $\prod_{m\in
 A}m=m_{1,2}m_{2,1}m_{4,2}$. Then the fuzzy rules can be represented as follows:

\bigskip

 \textbf{Rule} $R_1$ : If $x$ is
$m_{1,2}m_{2,1}m_{4,2}$, then $x$ belongs to Class 1;

\textbf{Rule} $R_2$ : If $x$ is $m_{2,1}m_{3,2}$, then $x$ belongs
to Class 1;

\textbf{Rule} $R_3$ : If $x$ is $m_{1,2}m_{4,2}$, then $x$ belongs
to Class 1.
\bigskip

\noindent Then, the antecedent of three fuzzy rules $R_1, R_2, R_3$ for Class
1 can be represented by ``or" as follows:
\bigskip

\textbf{Rule} $R$ : If $x$ is ``$m_{1,2}m_{2,1}m_{4,2}$ or
$m_{2,1}m_{3,2}$ or $m_{1,2}m_{4,2}$", then $x$ belongs to Class 1.

\bigskip

$\sum^{r}_{u=1}(\prod_{m\in A_u}m)$, which is a formal sum of the
fuzzy concepts $\prod_{m\in A_u}m$, $A_u \subseteq M$, is the disjunction of
the conjunctions represented by $\prod_{m\in A_u}m, u=1,\ldots,r$.
For example, let $A_1=\{m_{1,2},m_{2,1},m_{4,2}\},
A_2=\{m_{2,1},m_{3,2}\}, A_3=\{m_{1,2},m_{4,2}\} \subseteq M$, then
 a new
fuzzy set (i.e., fuzzy concept) as the disjunction of $\prod_{m\in A_1}m$,  $\prod_{m\in
A_2}m$,  $\prod_{m\in A_3}m$, i.e., ``$m_{1,2}m_{2,1}m_{4,2}$ or
$m_{2,1}m_{3,2}$ or $m_{1,2}m_{4,2}$'', can be represented as

\[\sum^{3}_{u=1}(\prod_{m\in A_u}m)=\prod_{m\in A_1}m + \prod_{m\in
"decrypted.txt" 204L, 10477C                                                                          30,1          Top
```

3

For the purpose of creating a signature, the process allows both encryption and decryption to be reversed when the opposite is applied vice versa. Meaning that a person encrypting a message with a private key can be deciphered by any person having the person's public key. The digital signature of sender is applied by passing the message m into a hash function H(), followed by signing the hashed value with private key. Generally, it can be assumed that the signature produces a unique value from H() and private key but at times collisions might occur where a different message m^1 generates the same hash value using H() where the signature for the original message m can be used to sign a different message resulting forgery which in this case, Bob found. Ideally, such occurrences are very rare, but it does happen. In order to minimise such occurrences, a hashing algorithm must be used which distributes generated hash values evenly minimising the likely hood of colliding with one another and H() should be designed in such a way that it is one way, meaning that given 'm^1' it is infeasible to generate m such that $H(m) = m^1$ (Second preimage resistant) . Also, the pre-decrypted message should not be completely available to the public to provide uncertainties that the collision message will be equal to the original message hash. Moreover, the message should have some error detection values eg : check-sum to ensure that if a collision is detected the message was not changed from the original state.

4

In a group of 24 randomly selected people, compute the probability that two of them share the same birthday :

With a group of 24 people, for two people to be chosen there are :

$(24 * 23) / 2 = 276$ combinations.

Assuming there are no leap years, the chance for a person to not share the same birthday with another person is :

$364 / 365$

Hence, the chance that no two person in a group share the same birthday is :
$(364/365)^{276}$

Therefore the compliment, that two person in a group share the same birthday is :

$1 - (364/365)^{276}$

which is 0.5310