FCC200 Report

Affine Cipher and S-DES Implementation

Connor Beardsmore - 15504319

Curtin University
Science and Engineering
Perth, Australia
April 2017

Affine Cipher

Compute Eligible Keys

There are two keys required, a and b. The first is required to be *coprime* with the length of the alphabet, in this scenario *26*. The second key representing the linear shift must be both positive and less than the length of the alphabet. To check if the a value is coprime, the following greatest common denominator check was utilized. If the greatest common denominator of a and 26 is 1, the value of a is *coprime* and the key is valid in combination with any valid b value.

```
//-------------------------------------------------------------------------------
// FUNCTION: gcd
// IMPORT: a (int), b (int)
// PURPOSE: Find greatest common denominator of 2 numbers

int gcdFunction( int a, int b )
{
    int quotient, residue, temp, gcd = 1;
    // SWAP ELEMENTS TO GET THE MAX
    if ( a < b )
    {
        temp = a;
        a = b;
        b = temp;
    }
    // CHECK IF EITHER NUMBER IS 0
    if ( a == 0 )   return b;
    if ( b == 0 )   return a;
    // SATISFY THE EQUATION: A = B * quotient + residue
    quotient = a / b;
    residue = a - ( b * quotient );
    // RECURSIVELY CALL GCD
    gcd = gcdFunction( b, residue );
    return gcd;
}

//-------------------------------------------------------------------------------
```

The results of calling this function on all a values from 1 to 25 are as follows:

- **gcdFunction(1, 26) = 1**
- gcdFunction(2, 26) = 2
- **gcdFunction(3, 26) = 1**
- gcdFunction(4, 26) = 2
- **gcdFunction(5, 26) = 1**
- gcdFunction(6, 26) = 2
- **gcdFunction(7, 26) = 1**
- gcdFunction(8, 26) = 2
- **gcdFunction(9, 26) = 1**
- gcdFunction(10, 26) = 2
- **gcdFunction(11, 26) = 1**
- gcdFunction(12, 26) = 2
- gcdFunction(13, 26) = 13
- gcdFunction(14, 26) = 2
- **gcdFunction(15, 26) = 1**
- gcdFunction(16, 26) = 2

- **gcdFunction(17, 26) = 1**
- gcdFunction(18, 26) = 2
- **gcdFunction(19, 26) = 1**
- gcdFunction(20, 26) = 2
- **gcdFunction(21, 26) = 1**
- gcdFunction(22, 26) = 2
- **gcdFunction(23, 26) = 1**
- gcdFunction(24, 26) = 2
- **gcdFunction(25, 26) = 1**

The full list of valid a values is: **1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25**

There are a total of 12 possible a values that are coprime with *26*. Each of these values can have a shift value (b) of 0 to 25. Thus, the total number of eligible keys is:

$$12 * 26 = 312$$

Of these, 26 keys are trivial Caesar ciphers and 286 are non-trivial.

Recovered Plaintext

```
1  Inthispaperweconsidertheproblemofrobustfacerecognitionusingcolor
2  informationinthiscontextsparserepresentationbasedalgorithmsarethe
3  stateoftheartsolutionsforgrayfacialimageSproposedmodelthecontrolpar
4  ameterizationTechniquetOgetherwiththeconstrainttranscriptionmethodi
5  susedbytransformingtheproposedproblemintoasequenceofoptimalparameter
6  selectionproblemsFinallyapracticalexampleonbeersalesisusedtoshowtheeffectiveness
7  ofproposedmodelandwepresenttheoptimAladvertisingstrategiescorrespondingtodifferent
8  competitionsituationS
9
```

Figure 1: Original Plaintext

```
[Connors-MacBook-Pro:affine connor$ make
 gcc -c affine.c -Wall -Wextra -std=c99
 gcc -c keyEligible.c -Wall -Wextra -std=c99
 gcc affine.o keyEligible.o -o affine -Wall -Wextra -std=c99
[Connors-MacBook-Pro:affine connor$ affine -e affine.txt output.txt 11 9
```

Figure 2: Encryption Process

```
1  Twkitzsjsborbfhwztqbokibsohuablhmohuvzkmjfbobfhxwtkthwvztwxfhaho
2  twmholjkthwtwkitzfhwkbckzsjozbobsobzbwkjkthwujzbqjaxhotkilzjobkib
3  zkjkbhmkibjokzhavkthwzmhoxojnmjftjatljxbZsohshzbqlhqbakibfhwkohasjo
4  jlbkbotyjkthwKbfiwtdvbkHxbkibortkikibfhwzkojtwkkojwzfotskthwlbkihqt
5  zvzbqunkojwzmholtwxkibsohshzbqsohuabltwkhjzbdvbwfbhmhsktljasjojlbkbo
6  zbabfkthwsohuablzMtwjaanjsojfktfjabcjlsabhwubbozjabztzvzbqkhzihrkibbmmbfktgbwbzz
7  hmsohshzbqlhqbajwqrbsobzbwkkibhsktlJajqgboktztwxzkojkbxtbzfhoobzshwqtwxkhqtmmbobwk
8  fhlsbktkthwztkvjkthwZ
9
```

Figure 3: Encrypted Ciphertext

```
[Connors-MacBook-Pro:affine connor$ affine -d output.txt original.txt 11 9
```

Figure 4: Decryption Process

```
1  Inthispaperweconsidertheproblemofrobustfacerecognitionusingcolor
2  informationinthiscontextsparserepresentationbasedalgorithmsarethe
3  stateoftheartsolutionsforgrayfacialimageSproposedmodelthecontrolpar
4  ameterizationTechniquetOgetherwiththeconstrainttranscriptionmethodi
5  susedbytransformingtheproposedproblemintoasequenceofoptimalparameter
6  selectionproblemsFinallyapracticalexampleonbeersalesisusedtoshowtheeffectiveness
7  ofproposedmodelandwepresenttheoptimAladvertisingstrategiescorrespondingtodifferent
8  competitionsituationS
9
```
d

Figure 5: Recovered Plaintext

Affine Mathematical Proof

The encryption and decryption functions for the affine cipher are as follows:

$$E(x) = (ax + b)\ mod\ m$$

$$D(x) = a^{-1}(x - b)\ mod\ m$$

The modular multiplicative inverse of a is defined as:

$$1 = aa^{-1}\ mod\ m$$

It can be shown that *D(x)* is the inverse of *E(x)* via the modular arithmetic laws.

$$\begin{aligned} D(E(x)) &= a^{-1}(E(x) - b)\ mod\ m \\ &= a^{-1}((ax + b\ mod\ m) - b)\ mod\ m \\ &= a^{-1}(ax + b - b)\ mod\ m \\ &= a^{-1}ax\ mod\ m \\ &= x\ mod\ m \end{aligned}$$

Letter Distribution

For the given test file shown in Figure 2, the following table and Figure 7 illustrate the letter distribution and relative frequencies.

- A: 35
- B: 7
- C: 17
- D: 14
- E: 65
- F: 14
- G: 10
- H: 16
- I: 41
- J: 0
- K: 0
- L: 18
- M: 16
- N: 35
- O: 50
- P: 23
- Q: 2
- R: 39
- S: 39
- T: 53
- U: 8
- V: 2
- W: 4
- X: 2
- Y: 3
- Z: 1

```
A: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
B: xxxxxxx
C: xxxxxxxxxxxxxxxx
D: xxxxxxxxxxxxxx
E: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
F: xxxxxxxxxxxxxx
G: xxxxxxxxxx
H: xxxxxxxxxxxxxxxx
I: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
J:
K:
L: xxxxxxxxxxxxxxxxx
M: xxxxxxxxxxxxxxx
N: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
O: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
P: xxxxxxxxxxxxxxxxxxxxx
Q: xx
R: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
S: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
T: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
U: xxxxxxxx
V: xx
W: xxxx
X: xx
Y: xxx
Z: x
```

Figure 6: Letter Distributions

S-DES

S-DES Mathematical Proof

hello

Pseudo Code Structure

The pseudo-code structure of the three key functions utilized in the S-DES implementation is illustrated below.

function KeyGeneration(int key)
    $key \leftarrow$ permute(key, P10)
    leftshift(key, 1)
    $subkeys[0] \leftarrow$ permute(key, P8)
    leftshift(key, 2)
    $subkeys[1] \leftarrow$ permute(key, P8)
    return subkeys
end function

function SwitchFunction(int input)
    $right \leftarrow bits \;\&\&\; ((1 << 4) - 1)$
    $left \leftarrow bits >>> 4$
    $output \leftarrow left \;||\; (right << 4)$
    return output
end function

function FeistalKeyRound(int message, int subkey)
    $halves \leftarrow$ split(message)
    $fMap \leftarrow$ fMapping(rightHalf, subkey)
    $leftHalf \leftarrow leftHalf \oplus fMap$
    $combined \leftarrow leftHalf + rightHalf$
    return combined
end function

Encrypted Test File

```
[Connors-MacBook-Pro:SDES connor$ javac *.java
[Connors-MacBook-Pro:SDES connor$ java SDES -e 130 des.txt output.txt
```

Figure 7: S-DES Encryption Process

Figure 8: S-DES Cipher Text

Decrypted Test File

```
[Connors-MacBook-Pro:SDES connor$ java SDES -d 130 output.txt original.txt
```

Figure 9: S-DES Decryption Process

```
\subsection{AFS Algebras}
###&&&
The Iris dataset is used as an illustrative example for AFS algebras through
this paper. It has 150 samples which are evenly distributed in three
classes and 4 features of sepal length($f_1$), sepal
width($f_2$), petal length($f_3$), and petal width($f_4$). Let a
pattern $x=(x_{1},x_{2},x_{3},x_{4})$, where $x_{i}$ is the $i$th
feature value of $x$. The following three linguist fuzzy rules have been obtained for Class 1
to build the AFS fuzzy classifier in Section 4.

\bigskip
\textbf{Rule} $R_1$: If $x_{1}$ is \emph{short sepal}, and $x_{2}$
is \emph{wide sepal}, and $x_{4}$ is \emph{narrow petal}, then $x$
belongs to Class 1;
```

```
 \subsection{Shannon¡¯s Entropy}
 Let $X$ be a discrete random variable with a finite set containing $N$ symbols
 $x_{0}, x_{1}, \ldots, x_{N}$. If an output $x_{j}$ occurs with probability $p(x_{j})$, then
 the
 amount of information associated with the known occurrence of the output $x_{j}$ is defined as
 \begin{equation}
 I(x_{j}) = -log_{2} p(x_{j})
 \end{equation}
 Based on this, the concept of Shannon¡¯s entropy is defined as follows:
 )))))~~~~
```

Figure 10: S-DES Plain Text

Utilization of an all 1 Key

Performing encryption and decryption with S-DES utilizing a key of all 1's (11111111) does not significantly alter how the algorithm performs. However, during the two feistal key rounds, the subkeys will be equivalent. This leads to both encryption and decryption being the same process. Thus in this situation:

$$x = E(E(x))$$

Modify S-Boxes

The S-BOX values in the SDESConstants.java file were modified to ensure that the S-DES algorithm still performs accurately.

Additional Questions

Threats

hello

Source Coding

Source coding in information transmission aims to compress natural messages for highly efficient message transfer.

Error Coding

Error coding in information transmission attempts to enable a high information rate by the introduction of redundancy to data, as well as via error detection and correction mechanisms.

S-DES Coding

hello

S-DES Confusion and Diffusion

In S-DES, confusion is provided by the S-BOX substitutions performed within the feistal key round. Diffusion in contrast is provided by the permutations applied to the data included the expansion permutation utilized.

Affine Source Code

keyeligible.h

```
/*****************************************************************************
* FILE: keyEligible.h
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Header file for key eligiblility checks
*   LAST MOD: 24/03/17
*   REQUIRES: NONE
*****************************************************************************/

// PROTOTYPES
int keyEligible(int,int,int);
int gcdFunction(int,int);
int extendEuclid(int,int);

//----------------------------------------------------------------------------
```

keyeligible.c

```c
/*****************************************************************************
* FILE: keyEligible.c
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Check the eligibility of keys a and b
*   LAST MOD: 28/03/17
*   REQUIRES: keyEligible.h
*****************************************************************************/

#include "keyEligible.h"

//----------------------------------------------------------------------------
// FUNCTION: keyEligible
// IMPORT: a (int), b (int)
// EXPORT: eligible (int)
// PURPOSE: Check that the two given keys are eligible via coprime check

int keyEligible( int a, int b, int alphabet )
{
    int eligible = 1;

    // a must be positive and less than the alpabet (26)
    if ( ( a < 0 ) || ( b > ( alphabet - 1 ) ) )
        eligible = 0;
    // a must be coprime to the alphabet length (26)
    if ( gcdFunction( a, alphabet ) != 1 )
        eligible = 0;
    // b must be positive and less than the alphabet (26)
    if ( ( b < 0 ) || ( b > ( alphabet - 1 ) ) )
        eligible = 0;
    return eligible;
}

//----------------------------------------------------------------------------
// FUNCTION: gcd
// IMPORT: a (int), b (int)
// PURPOSE: Find greatest common denominator of 2 numbers

int gcdFunction( int a, int b )
{
    int quotient, residue, temp, gcd = 1;
    // SWAP ELEMENTS TO GET THE MAX
    if ( a < b )
    {
        temp = a;
        a = b;
        b = temp;
    }
    // CHECK IF EITHER NUMBER IS 0
    if ( a == 0 )   return b;
    if ( b == 0 )   return a;
    // SATISFY THE EQUATION: A = B * quotient + residue
    quotient = a / b;
    residue = a - ( b * quotient );
    // RECURSIVELY CALL GCD
    gcd = gcdFunction( b, residue );
    return gcd;
}

//----------------------------------------------------------------------------
// FUNCTION: extendEuclid
// IMPORT: a (int), n (int)
// PURPOSE: Extended Euclidean algorithm to find inverse modular
```

```
64
65 int extendEuclid( int a, int n )
66 {
67     int t = 0, newt = 1;
68     int r = n, newr = a;
69     int q = 0, temp = 0;
70
71     // IF GCD IS NOT 1 THEN NO COPRIME EXISTS
72     if ( gcdFunction( a, n ) != 1 )
73         return -1;
74
75     // PERFORM EXTENDED EUCLIDEAN
76     while ( newr != 0 )
77     {
78         q = r / newr;
79         temp = t;
80         t = newt;
81         newt = temp - ( q * newt );
82         temp = r;
83         r = newr;
84         newr = temp - ( q * newr );
85     }
86
87     // MAKE SURE T IS NOT NEGATIVE
88     if ( t < 0 )
89         t = t + n;
90
91     return t;
92 }
93
94 //-----------------------------------------------------------------------------
```

affine.h

```
/*****************************************************************************
* FILE: affine.h
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Header file for affine cipher
*   LAST MOD: 11/03/17
*   REQUIRES: stdio.h, ctype.h, stdlib.h, string.h, keyEligible.h
*****************************************************************************/

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include "keyEligible.h"

// PROTOTYPES
char encrypt(char,int,int);
char decrypt(char,int,int);

// FUNCTION POINTER
typedef char (*FuncPtr)(char,int,int);

// CONSTANTS
#define ARGS 6
#define ALPHABET 26

//---------------------------------------------------------------------------
```

affine.c

```
/*****************************************************************************
* FILE: affine.c
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Run Affine cipher given text and key, either encrypt or decrypt
*   LAST MOD: 28/03/17
*   REQUIRES: affine.h
*****************************************************************************/

#include "affine.h"

//----------------------------------------------------------------------------
// FUNCTION: main

int main( int argc, char* argv[] )
{
    if ( argc != ARGS )
    {
        printf("\nUSAGE: <FLAG> <INPUT FILE> <OUTPUT FILE> <KEY A> <KEY B>\n");
        printf("FLAGS ARE: -e for encryption, -d for decryption\n\n");
        return 1;
    }

    // CONVERT ARGC NAMES
    char* flag = argv[1];
    char* inFile = argv[2];
    char* outFile = argv[3];
    int a = atoi( argv[4] );
    int b = atoi( argv[5] );

    // CHECK THAT THE KEYS ARE ELIGIBLE
    int validity = keyEligible( a, b, ALPHABET );
    if ( validity != 1 )
    {
        printf("\nKEYS %d AND %d ARE NOT VALID.\n", a, b );
        return 2;
    }

    // OPEN INPUT AND OUTPUT FILES
    FILE* inF = fopen( inFile, "r" );
    FILE* outF = fopen( outFile, "w" );

    // CHECK OPEN FOR ERRORS
    if ( ( inF == NULL ) || ( outF == NULL ) )
    {
        perror("\nERROR OPENING INPUT OR OUTPUT FILE\n");
        return 3;
    }

    // FUNCTION POINTER FOR encrypt() OR decrypt()
    FuncPtr fp;

    // PERFORM ENCRYPTION IF -e FLAG PROVIDED AND VICE VERSA
    if ( !strncmp( flag, "-e", 2 ) )
        fp = &encrypt;
    else if ( !strncmp( flag, "-d", 2 ) )
        fp = &decrypt;
    else
    {
        printf("\nFLAG IS INCORRECT, MUST BE -e OR -d\n");
        return 4;
    }

```

```
64      // PERFROM APPROPRIATE FUNCTION
65      while ( ( feof( inF ) == 0 ) && ( ferror( inF ) == 0) && (ferror( inF ) == 0) )
66      {
67          // GET THE NEXT CHARACTER FROM FILE
68          char next = fgetc( inF );
69          if ( feof( inF ) == 0 )
70              // WRITE THE CONVERTED CHARACTER TO FILE
71              fputc( ( *fp )( next, a, b ), outF );
72      }
73
74      // CLOSE FILES
75      fclose( inF );
76      fclose( outF );
77
78      return 0;
79  }
80
81  //-------------------------------------------------------------------------------
82  // FUNCTION: encrypt
83  // IMPORT: plain (char), a (int), b (int)
84  // PURPOSE: Convert a plaintext char into the encryped character
85
86  char encrypt( char plain, int a, int b)
87  {
88      char output = plain;
89      // ENCRYPT BASED ON plain * a + b MODULO 26
90      // IGNORE NON-CHARACTERS
91      if ( isupper(plain) )
92          output = ( ( ( plain - 'A' ) * a + b ) % ALPHABET ) + 'A';
93      else if ( islower(plain) )
94          output = ( ( ( plain - 'a' ) * a + b ) % ALPHABET ) + 'a';
95      return output;
96  }
97
98  //-------------------------------------------------------------------------------
99  // FUNCTION: decrypt
100 // IMPORT: plain (char*), a (int), b (int)
101 // PURPOSE: Convert a ciphertect char into the decrypted character
102
103 char decrypt( char cipher, int a, int b )
104 {
105     // FIND THE MODULO INVERSE USING EUCLIDEAN
106     int inverse = extendEuclid( a, ALPHABET );
107     char output = cipher;
108     // DECRYPT BASED ON inverse * cipher - b MODULO 26
109     // IGNORE NON-CHARACTERS
110     if ( isupper(cipher) )
111         output = ( ( inverse * ( cipher - 'A' - b + ALPHABET ) ) % ALPHABET ) + 'A';
112     else if ( islower(cipher) )
113         output = ( ( inverse * ( cipher - 'a' - b + ALPHABET ) ) % ALPHABET ) + 'a';
114     return output;
115 }
116
117 //-------------------------------------------------------------------------------
```

S-DES Source Code

SDESConstants.java

```
/*****************************************************************************
* FILE: SDESConstants
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Structures to represent the constants in the SDES algorithm
*   LAST MOD: 21/03/17
*   REQUIRES: NONE
*****************************************************************************/

public class SDESConstants
{
    // P10 PERMUTATION FOR THE 10-BIT KEY
    public static final int[] P10 = { 2, 4, 1, 6, 3, 9, 0, 8, 7, 5 };

//---------------------------------------------------------------------------

    // P8 PERMUTATION FOR THE 10-BIT KEY
    public static final int[] P8 = { 5, 2, 6, 3, 7, 4, 9, 8 };

//---------------------------------------------------------------------------

    // INITIAL PERMUTATION FOR THE 8-BIT PLAINTEXT
    public static final int[] IP = { 1, 5, 2, 0, 3, 7, 4, 6 };

//---------------------------------------------------------------------------

    // INVERSE PERMUTATION FOR THE 8-BIT PLAINTEXT
    public static final int[] IPI = { 3, 0, 2, 4, 6, 1, 7, 5 };

//---------------------------------------------------------------------------

    // EXPANSION PERMUTATION FOR 4-BITS IN Fk
    public static final int[] EP = { 3, 0, 1, 2, 1, 2, 3, 0 };

//---------------------------------------------------------------------------

    // P4 PERMUTATION AFTER THE S-BOX SELECTION
    public static final int[] P4 = { 1, 3, 2, 0 };

//---------------------------------------------------------------------------

    // SBOX ONE
    public static final int[][] S0 = {  { 1, 0, 3, 2 },
                                        { 3, 2, 1, 0 },
                                        { 0, 2, 1, 3 },
                                        { 3, 1, 3, 2 }  };

//---------------------------------------------------------------------------

    // SBOX TWO
    public static final int[][] S1 = {  { 0, 1, 2, 3 },
                                        { 2, 0, 1, 3 },
                                        { 3, 0, 1, 0},
                                        { 2, 1, 0, 3 }  };

//---------------------------------------------------------------------------
}
```

SDESBits.java

```
/*****************************************************************************
* FILE: SDESBits.java
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: BitSet alternative using a int
*   LAST MOD: 24/03/17
*   REQUIRES: NONE
*****************************************************************************/

public class SDESBits
{
    //CONSTANTS
    public static final int MIN_SIZE = 4;
    public static final int MAX_SIZE = 10;

    //CLASSFIELDS
    private int bits;
    private int size;
    private int half;
    // private only applies to different classes, so we can
    // import an SDESBits and retreive bits without a getter

//-----------------------------------------------------------------------------
    //ALTERNATE CONSTRUCTOR

    public SDESBits( int inBits, int inSize )
    {
        // Check inBits and inSize validity
        if ( inBits < 0 )
            throw new IllegalArgumentException( "INVALID SDESBits VALUE" );
        if ( ( inSize < MIN_SIZE ) || ( inSize > MAX_SIZE ) )
            throw new IllegalArgumentException( "INVALID SDESBits SIZE" );
        if ( inSize % 2 != 0 )
            throw new IllegalArgumentException( "INVALID SDESBits SIZE" );

        bits = inBits;
        size = inSize;
        half = inSize >>> 1;
    }

//-----------------------------------------------------------------------------
    //COPY CONSTRUCTOR

    public SDESBits( SDESBits inBits )
    {
        bits = inBits.bits;
        size = inBits.size;
        half = inBits.half;
    }

//-----------------------------------------------------------------------------
    //FUNCTION: switchHalves()
    //PURPOSE: Switch the left half of bits with the right half

    public void switchHalves()
    {
        // Get the right half of the bits
        int oRight = bits & ( ( 1 << half ) - 1 );
        // Shift the left half of the bits down
        bits >>>= half;
        // Combine left half with right half shifted up
        bits |= ( oRight << half );
    }
```

```
64
65 //-------------------------------------------------------------------------
66     //FUNCTION: permute()
67     //IMPORT: permTable (int[])
68     //EXPORT: permuted (SDESBits)
69     //PURPOSE: Create a permutation of this objects bits in a new SDESBits
70
71     public SDESBits permute( int[] permTable )
72     {
73         // Create temporary space the size of the permutation
74         SDESBits permuted = new SDESBits( 0, permTable.length );
75         // Iterate across the permutation, getting and setting bits
76         for ( int ii = 0; ii < permTable.length; ii++ )
77             permuted.setBit( getBit( permTable[ii] ), ii );
78         return permuted;
79     }
80
81 //-------------------------------------------------------------------------
82     //FUNCTION: leftShift()
83     //IMPORT: shifts (int)
84     //PURPOSE: Perfrom a circular left shift on the bits of each half
85
86     public void leftShift( int shifts )
87     {
88         //Check shift validity
89         if ( shifts < 1 )
90             throw new IllegalArgumentException( "ILLEGAL SHIFT VALUE" );
91
92         // Temp variable for repeated 1's for a half
93         int ones = ( 1 << half ) - 1;
94         // Avoid shifting more than required
95         if ( half > shifts )
96             shifts %= half;
97
98         // Get the left half and right half
99         int left = bits >>> half;
100        int right = bits & ones;
101
102        // Loop for each shift individually
103        for ( int ii = 0; ii < shifts; ii++ )
104        {
105            // Get the leftmost bit of the left sub-half
106            int leftBit = ( left & ones );
107            leftBit >>>= MIN_SIZE;
108            // Get the rightmost bit of the right sub-half
109            int rightBit = ( right & ones );
110            rightBit >>>= MIN_SIZE;
111
112            // Perform the actual shifting of the bits
113            left = ( left << 1 ) & ones;
114            right = ( right << 1 ) & ones;
115
116            // If the first bits of the halves were one, set final bit
117            if ( leftBit == 1 )      left++;
118            if ( rightBit == 1 )     right++;
119        }
120
121        // Recombine both halves back together
122        bits = ( left << half ) | right;
123    }
124
125 //-------------------------------------------------------------------------
126    //FUNCTION: split()
127    //EXPORT: halves (SDESBits[])
128    //PURPOSE: Split the bits into two sub-halves and return as objects
```

```
129
130     public SDESBits[] split()
131     {
132         // New container for the halves
133         SDESBits[] halves = new SDESBits[2];
134         // Get the left half and create object
135         int leftInt = bits >>> half;
136         halves[0] = new SDESBits( leftInt, half );
137         // Get the right half and create object
138         int rightInt = ( bits & ( ( 1 << half ) - 1 ) );
139         halves[1] = new SDESBits( rightInt, half );
140
141         return halves;
142     }
143
144 //-------------------------------------------------------------------------
145     //FUNCTION: xor()
146     //IMPORT: inBits (SDESBits)
147     //PURPOSE: XOR bits with the bits value of inBits
148
149     public void xor( SDESBits inBits )
150     {
151         // Ensure the same size
152         if ( size != inBits.size )
153             throw new IllegalArgumentException( "CANNOT XOR DIFFERENT SIZES" );
154
155         // Call simple exclusive-or on both bits
156         bits ^= inBits.bits;
157     }
158
159 //-------------------------------------------------------------------------
160     //FUNCTION: setBit()
161     //IMPORT: val (boolean), index (int)
162     //PURPOSE: Set the value at the specifed index with the specified value
163
164     public void setBit( boolean val, int index )
165     {
166         // Validity
167         if ( ( index < 0 ) || ( index >= size) )
168             throw new IllegalArgumentException("SETBIT IMPORTS INVALID");
169
170         // Reset the given bit
171         bits &= ~(1 << ( size - index - 1 ) );
172         // Reset the bits greater than the size we want
173         bits &= (1 << size)-1;
174         // Set the required bit
175         bits |= ((val) ? 1 : 0 ) << ( size - index - 1 );
176     }
177
178 //-------------------------------------------------------------------------
179     //FUNCTION: getBit()
180     //IMPORT: index (int)
181     //EXPORT: value (boolean)
182     //PURPOSE: Get the value of the bit at the specified index
183
184     public boolean getBit( int index )
185     {
186         if ( ( index < 0 ) || ( index >= size) )
187             throw new IllegalArgumentException("SETBIT IMPORTS INVALID");
188
189         // Bits are reverse ordered
190         return (bits & 1 << ( size - index - 1 ) ) != 0;
191     }
192
193 //-------------------------------------------------------------------------
```

```
194
195     public int getBits() { return bits; }
196
197 //-------------------------------------------------------------------------
198     //FUNCTION: append()
199     //IMPORT: newBits (SDESBits)
200     //PURPOSE: Append new set of bits to the original set
201
202     public void append( SDESBits newBits )
203     {
204         // Increment size
205         size += newBits.size;
206         // Shift original across and add new bits
207         bits = ( bits << newBits.size ) | newBits.bits;
208         // Update half value
209         half = size >>> 1;
210     }
211
212 //-------------------------------------------------------------------------
213     //FUNCTION: sbox()
214     //EXPORT: result (int)
215     //PURPOSE: Find the sbox values for the bits in this object
216
217     public int sbox()
218     {
219         // Split into halves
220         SDESBits halves[] = this.split();
221
222         // Get row and column of the first four bits
223         int colS0 = ( halves[0].bits & 6 ) >>> 1;
224         int rowS0 = ( ( halves[0].bits & 8 ) >>> 2 ) | ( halves[0].bits & 1 );
225         // Get row and column of the second four bits
226         int colS1 = ( halves[1].bits & 6 ) >>> 1;
227         int rowS1 = ( ( halves[1].bits & 8 ) >>> 2 ) | ( halves[1].bits & 1 );
228
229         // Get the appropriate sbox value
230         int s0Val = SDESConstants.S0[rowS0][colS0];
231         int s1Val = SDESConstants.S1[rowS1][colS1];
232
233         // Combine the result
234         int result = ( s0Val << 2 ) | s1Val;
235
236         return result;
237     }
238
239 //-------------------------------------------------------------------------
240     //FUNCTION: toString()
241     //EXPORT: state (String)
242     //PURPOSE: Export bits in a readable binary format
243
244     public String toString()
245     {
246         return Integer.toBinaryString( bits );
247     }
248
249 //-------------------------------------------------------------------------
250 }
```

SDES.java

```java
/*****************************************************************************
* FILE: SDES.java
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Performs SDES encryption or decryption on a given file
*   LAST MOD: 21/03/17
*   REQUIRES: NONE
*****************************************************************************/

import java.util.*;
import java.io.*;

public class SDES
{

    public static final int NUM_ARGS = 4;
    public static final int MAX_KEY = 1023;
    public static final int KEY_SIZE = 10;
    public static final int MESSAGE_SIZE = 8;

//---------------------------------------------------------------------------

    public static void main( String[] args )
    {
        // Check argument length and output usage
        if ( args.length != NUM_ARGS )
        {
            System.out.println("USAGE: SDES <mode> <key> <input file> <output file>");
            System.out.println("modes = -e encryption, -d decryption");
            System.out.println("keys = int between 0 and 255");
            System.exit(1);
        }

        // Rename variables for simplicity
        String mode = args[0];
        String key = args[1];
        String inFile = args[2];
        String outFile = args[3];
        SDESBits message, output;
        int intKey = Integer.parseInt( key );

        try
        {
            // Generate subkeys
            SDESBits subkeys[] = keyGeneration( intKey );

            // Open file streams
            FileInputStream fis = new FileInputStream( new File( inFile ) );
            FileOutputStream fos = new FileOutputStream( new File( outFile ) );

            // Read bytes until end of file
            int next = fis.read();
            while ( next != -1 )
            {
                message = new SDESBits( next, MESSAGE_SIZE );

                // Select function based on mode
                if ( mode.equals( "-e" ) )
                    output = encrypt( message, subkeys );
                else if ( mode.equals( "-d") )
                    output = decrypt( message, subkeys );
                else
                    throw new IllegalArgumentException("INVALID MODE");
```

```
64
65                  // Write converted output to file
66                  int outputInt = output.getBits();
67                  fos.write( outputInt );
68                  next = fis.read();
69              }
70          }
71          catch (Exception e)
72          {
73              System.out.println( e.getMessage() );
74          }
75
76      }
77
78  //-------------------------------------------------------------------------
79      //FUNCTION: encrypt()
80      //IMPORT: message (SDESBits), subkeys (SDESBits[])
81      //EXPORT: message (SDESBits)
82      //PURPOSE: Encrypt given message with given subkeys
83
84      public static SDESBits encrypt( SDESBits message, SDESBits[] subkeys )
85      {
86          // Initial Permutation
87          message = message.permute( SDESConstants.IP );
88          // First feistal key round with subkey 1
89          message = feistalRound( message, subkeys[0] );
90          // Switch left and right subhalves
91          switchFunction( message );
92          // Second feistal key round with subkey 2
93          message = feistalRound( message, subkeys[1] );
94          // Inverse of Initial Permutation
95          message = message.permute( SDESConstants.IPI );
96          return message;
97      }
98
99  //-------------------------------------------------------------------------
100     //FUNCTION: decrypt()
101     //IMPORT: message (SDESBits), subkeys (SDESBits[])
102     //EXPORT: message (SDESBits)
103     //PURPOSE: Decrypt given message with given subkeys
104
105     public static SDESBits decrypt( SDESBits message, SDESBits[] subkeys )
106     {
107         // Initial Permutation
108         message = message.permute( SDESConstants.IP );
109         // First feistal key round with subkey 2
110         message = feistalRound( message, subkeys[1] );
111         // Switch left and right subhalves
112         switchFunction( message );
113         // First feistal key round with subkey 2
114         message = feistalRound( message, subkeys[0] );
115         // Inverse of Initial Permutation
116         message = message.permute( SDESConstants.IPI );
117         return message;
118     }
119
120 //-------------------------------------------------------------------------
121     //FUNCTION: switchFunction()
122     //IMPORT: input (SDESBitSet)
123     //PURPOSE: Import 8-bit binary and swap the first and last 4 bits
124
125     public static void switchFunction( SDESBits input )
126     {
127         input.switchHalves();
128     }
```

```
129
130 //-------------------------------------------------------------------------
131     //FUNCTION: keyGeneration()
132     //IMPORT: keyDec (int)
133     //EXPORT: subkeys (SDESBits[])
134     //PURPOSE: Generate subkeys given the full key
135
136     public static SDESBits[] keyGeneration( int keyDec )
137     {
138         // Check key validity
139         if ( ( keyDec < 0 ) || ( keyDec > MAX_KEY ) )
140             throw new IllegalArgumentException("INVALID KEY");
141
142         // Convert int key into an SDESBits object and create subkey array
143         SDESBits key = new SDESBits( keyDec, KEY_SIZE );
144         SDESBits[] subkeys = new SDESBits[2];
145
146         // P10 permutation, left shift and P8 permutation to form subkey 1
147         key = key.permute( SDESConstants.P10 );
148         key.leftShift(1);
149         subkeys[0] = key.permute( SDESConstants.P8 );
150         // P8 permutation and double left shift to form subkey 2
151         key.leftShift(2);
152         subkeys[1] = key.permute( SDESConstants.P8 );
153
154         return subkeys;
155     }
156
157 //-------------------------------------------------------------------------
158     //FUNCTION: feistalRound()
159     //IMPORT: message (SDESBits), subkey (SDESBits)
160     //EXPORT: halves (SDESBits)
161     //PURPOSE: Perform feistal key round on message given a subkey
162
163     public static SDESBits feistalRound( SDESBits message, SDESBits subkey )
164     {
165         // Split message in half
166         SDESBits halves[] = message.split();
167         // Perform fMapping function
168         SDESBits fMap = fMapping( halves[1], subkey );
169         // XOR the halves and append
170         halves[0].xor( fMap );
171         halves[0].append(halves[1]);
172         return halves[0];
173     }
174
175 //-------------------------------------------------------------------------
176     //FUNCTION: fMapping()
177     //IMPORT: message (SDESBits), subkey (SDESBits)
178     //EXPORT: message (SDESBits)
179     //PURPOSE: Perform fMapping function on given message with subkey
180
181     public static SDESBits fMapping( SDESBits message, SDESBits subkey )
182     {
183         // Expansion permutation and XOR with subkey
184         message = message.permute( SDESConstants.EP );
185         message.xor( subkey );
186         // Calculate SBOX values and P4 permutation
187         message = new SDESBits( message.sbox(), MESSAGE_SIZE/2 );
188         message = message.permute( SDESConstants.P4 );
189         return message;
190     }
191
192 //-------------------------------------------------------------------------
193 }
```