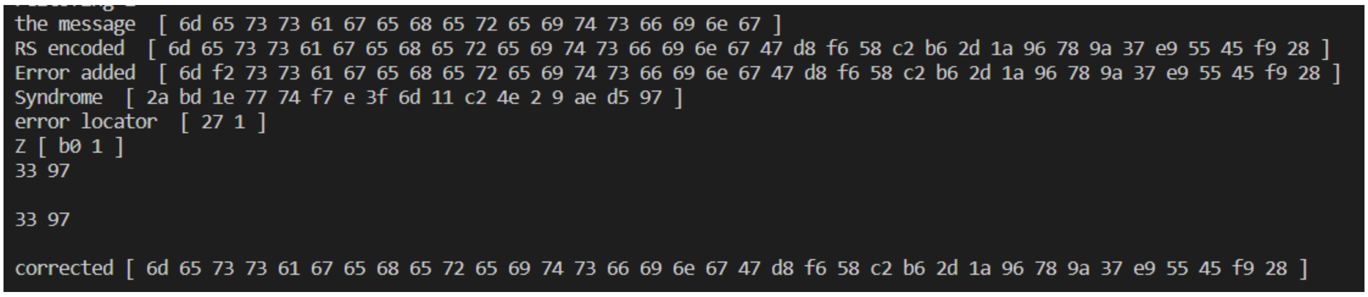
Assignment 7

Ethan Fidler 2/27/2023

Output

Correcting a message with 1 Error



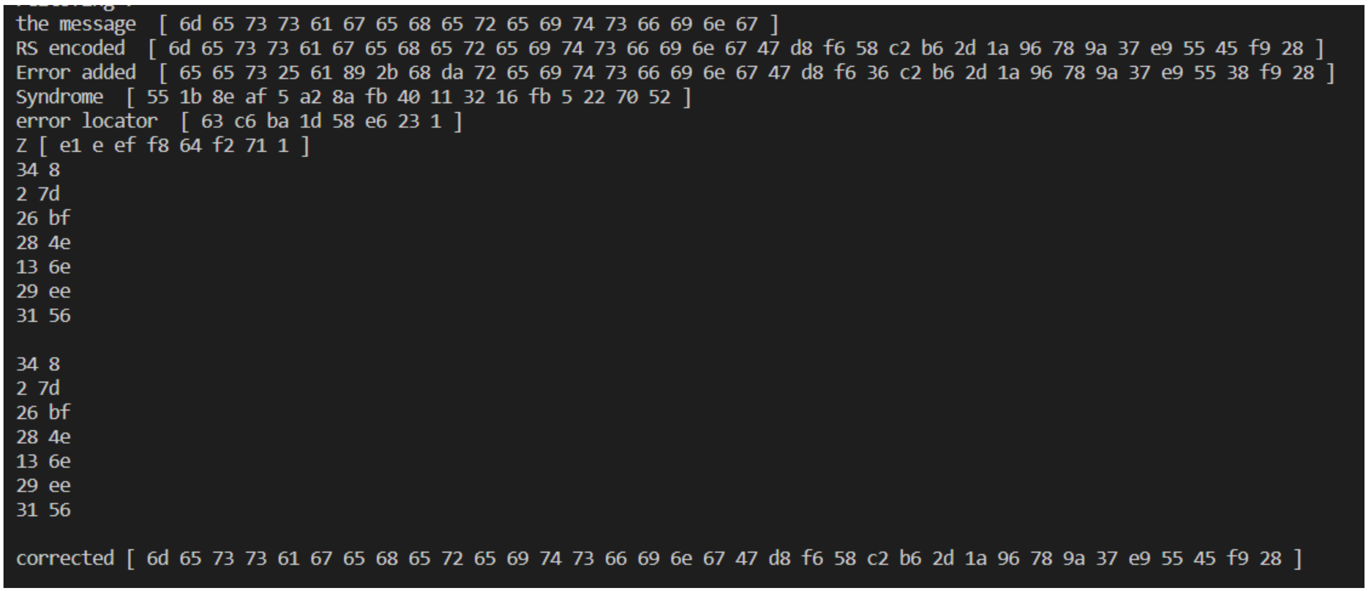
```
the message  [ 6d 65 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 ]
RS encoded  [ 6d 65 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 47 d8 f6 58 c2 b6 2d 1a 96 78 9a 37 e9 55 45 f9 28 ]
Error added  [ 6d f2 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 47 d8 f6 58 c2 b6 2d 1a 96 78 9a 37 e9 55 45 f9 28 ]
Syndrome  [ 2a bd 1e 77 74 f7 e 3f 6d 11 c2 4e 2 9 ae d5 97 ]
error locator  [ 27 1 ]
Z [ b0 1 ]
33 97

33 97

corrected [ 6d 65 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 47 d8 f6 58 c2 b6 2d 1a 96 78 9a 37 e9 55 45 f9 28 ]
```

Original Message: `6d 65 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 47 d8 f6 58 c2 b6 2d 1a 96 78 9a 37 e9 55 45 f9 28`

Message with Error: `6d f2 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 47 d8 f6 58 c2 b6 2d 1a 96 78 9a 37 e9 55 45 f9 28`

Corrected Message: `6d 65 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 47 d8 f6 58 c2 b6 2d 1a 96 78 9a 37 e9 55 45 f9 28`

Correcting a message with 8 Errors



```
the message  [ 6d 65 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 ]
RS encoded  [ 6d 65 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 47 d8 f6 58 c2 b6 2d 1a 96 78 9a 37 e9 55 45 f9 28 ]
Error added  [ 65 65 73 25 61 89 2b 68 da 72 65 69 74 73 66 69 6e 67 47 d8 f6 36 c2 b6 2d 1a 96 78 9a 37 e9 55 38 f9 28 ]
Syndrome  [ 55 1b 8e af 5 a2 8a fb 40 11 32 16 fb 5 22 70 52 ]
error locator  [ 63 c6 ba 1d 58 e6 23 1 ]
Z [ e1 e ef f8 64 f2 71 1 ]
34 8
2 7d
26 bf
28 4e
13 6e
29 ee
31 56

34 8
2 7d
26 bf
28 4e
13 6e
29 ee
31 56

corrected [ 6d 65 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 47 d8 f6 58 c2 b6 2d 1a 96 78 9a 37 e9 55 45 f9 28 ]
```

Original Message: `6d 65 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 47 d8 f6 58 c2 b6 2d 1a 96 78 9a 37 e9 55 45 f9 28`

Message with Error: `65 65 73 25 61 89 2b 68 da 72 65 69 74 73 66 69 6e 67 47 d8 f6 36 c2 b6 2d 1a 96 78 9a 37 e9 55 38 f9 28`

Corrected Message: 6d 65 73 73 61 67 65 68 65 72 65 69 74 73 66 69 6e 67 47 d8 f6 58 c2 b6 2d 1a 96 78 9a 37 e9 55 45 f9 28

Code

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

public class DE11{
    GF f = new GF(256, 2, 0x11d);
    static int Gdegree = 17;  // generator polynomial degree, indicating maximum 8 errors to correct
    int numberOfErrors = 0; // number of random errors
    HashMap<Integer, Integer> errors = new HashMap<Integer, Integer>(); // random errors
    Polynomial G = makeRSG(); // the generator
    Polynomial M = null; // message
    Polynomial C = null; // reed-solomon encoded
    Polynomial CplusE = null; // error added
    Polynomial Syndrome = null; // syndrome after error added
    Polynomial errorLocator = null;

class GF{  // finite field of 2^k elements so that addition and subtractions are XOR
    int fieldSize = 0;  // 2^k
    public int logBase = 0;  // a primative element
    int irreducible = 0;  // a irreducible polynomial of degree k
    public int[] alog = null;  // all powers of the logBase
    public int[] log = null;  // discrete log, log[0] is not defined

    public GF(int size, int base, int irr){  // constructor
        fieldSize = size; logBase = base; irreducible = irr;
        alog = new int[fieldSize]; log = new int[fieldSize];
        makeLog();
    }

    int modMultiply(int a, int b, int m){  // multiply based on XOR as addition
        int product = 0;
        for (; b > 0; b >>= 1){
            if ((b & 1) > 0) product ^= a;
            a <<= 1;
            if ((a & fieldSize) > 0) a ^= m;
            }
        return product;
    }

    void makeLog(){  // first make all powers and then discrete log
        alog[0] = 1;
        for (int i = 1; i < fieldSize; i++)
            alog[i] = modMultiply(logBase, alog[i - 1], irreducible);
        for (int i = 0; i < fieldSize - 1; i++) log[alog[i]] = i;
    }
    public int multiply(int a, int b){  // multiplication in GF
            return (a == 0 || b == 0) ? 0 : alog[(log[a] + log[b]) % (fieldSize -
1)];
    }

    int multiplicativeInverse(int a){
            return alog[fieldSize - 1 - log[a]];
    }
};

class Polynomial{
    int[] coeff = null;
    // coeff[0] is the constant term, coeff[coeff.length - 1] is the highest power
term

    public Polynomial(int length){ coeff = new int[length]; } // constructor

    public Polynomial(String data){ // turn string around as data polynomial
        coeff = new int[data.length()];
        for (int i = 0; i < coeff.length; i++)
            coeff[coeff.length - 1 - i] = data.charAt(i);
    }

    public int evaluate(int x){  // Horner's algorithm
        int sum = coeff[coeff.length - 1];
        for (int i = coeff.length - 2; i >= 0; i--)
            sum = f.multiply(sum, x) ^ coeff[i];
        return sum;
    }

    public void display(String title) { // display with highest power first
        if (coeff.length == 0){
            System.out.println(title + " [ ]");
            return;
        }
        System.out.print(title + " [ ");
        for (int i = coeff.length - 1; i > 0; i--)
            System.out.print(Integer.toHexString(coeff[i]) + " ");
        System.out.println(Integer.toHexString(coeff[0]) + " ]");
    }

    public Polynomial scale(int a){ // ap(x)
        Polynomial newp = new Polynomial(coeff.length);
        for (int i = 0; i < coeff.length; i++)
            newp.coeff[i] = f.multiply(coeff[i], a);
            return newp;
    }

    public Polynomial shift(int r){ // x^r p(x)
        Polynomial newp = new Polynomial(coeff.length + r);
        for (int i = 0; i < coeff.length; i++) newp.coeff[i + r] = coeff[i];
        for (int i = 0; i < r; i++) newp.coeff[i] = 0;
            return newp;
    }

    public Polynomial add(Polynomial p2){ // p(x) + p2(x)
        if (coeff.length >= p2.coeff.length){
            Polynomial newp = new Polynomial(coeff.length);
            for (int i = 0; i < p2.coeff.length; i++)
                newp.coeff[i] = coeff[i] ^ p2.coeff[i];
            for (int i = p2.coeff.length; i < coeff.length; i++)
                newp.coeff[i] = coeff[i];
                    return newp;
        }else{
            Polynomial newp = new Polynomial(p2.coeff.length);
            for (int i = 0; i < coeff.length; i++)
                newp.coeff[i] = coeff[i] ^ p2.coeff[i];
            for (int i = coeff.length; i < p2.coeff.length; i++)
                newp.coeff[i] = p2.coeff[i];
                    return newp;
        }
    }

    public Polynomial RSencode(){ // shift, mod G and add remainder
        Polynomial tmp = shift(Gdegree);
        int head = tmp.coeff.length - 1;
        for (int i = tmp.coeff.length - G.coeff.length; i >= 0; i--)
            tmp = tmp.add(G.scale(tmp.coeff[head--]).shift(i));
        Polynomial ret = shift(Gdegree);
        for (int i = 0; i < Gdegree; i++) ret.coeff[i] = tmp.coeff[i];
        return ret;
    }

    public Polynomial computeSyndrome(){
        Polynomial S = new Polynomial(Gdegree);
        for (int i = 0; i < Gdegree; i++)
            S.coeff[i] = evaluate(f.alog[i]);
        return S;
    }

    Polynomial addError(HashMap<Integer, Integer> errors){
        // used on result of RSencode to get errorAdded
        // used on errorAdded to get result of RSencode
        int numberOfCodewords = coeff.length;
        Polynomial errorAdded = new Polynomial(numberOfCodewords);
        for (int i = 0; i < numberOfCodewords; i++)
            errorAdded.coeff[i] = coeff[i];
        errors.forEach((k,v)->{ errorAdded.coeff[k] ^= v; });
        return errorAdded;
    }

    Polynomial berlekampMassey(){  // used on Syndrome and returns errorLocator
        Polynomial op = new Polynomial(1); op.coeff[0] = 1;
        Polynomial ep = new Polynomial(1); ep.coeff[0] = 1;
        Polynomial np = null;
        for (int i = 0; i < Gdegree; i++){
            op = op.shift(1);
```

```
                int d = coeff[i];
                for (int j = 1; j < ep.coeff.length; j++)
                    d ^= f.multiply(ep.coeff[j], coeff[i - j]);
                if (d != 0){
                    if (op.coeff.length > ep.coeff.length){
                        np = op.scale(d);
                        op = ep.scale(f.multiplicativeInverse(d));
                        ep = np;
                    }
                    ep = ep.add(op.scale(d));
                }
            }
            return ep;
        }

        HashSet<Integer> findZeros(){  // returns zeros of an polynomial as a set
            HashSet<Integer> zeros = new HashSet<Integer>();
            for (int i = 0; i < f.fieldSize; i++)
                if (evaluate(i) == 0) zeros.add(i);
            return zeros;
        }

        Polynomial computeZ(Polynomial locator){  // part of Forney, used on syndrome
            Polynomial Z = new Polynomial(locator.coeff.length);
            for (int i = 0; i < locator.coeff.length; i++){
                Z.coeff[i] = locator.coeff[i];
                for (int j = 0; j < i; j++)
                    Z.coeff[i] ^= f.multiply(locator.coeff[j] , coeff[i - 1 - j]);
    }
            return Z;
        }

        HashMap<Integer, Integer> forney(HashSet<Integer> betaInverses){ // used on Z
            HashMap<Integer, Integer> errors = new HashMap<Integer, Integer>();
            HashSet<Integer> betas = new HashSet<Integer>();
            for (int p: betaInverses) betas.add(f.multiplicativeInverse(p));
            for (int p: betaInverses){ // for each beta_i^-1 as p
                int d = p; // d is for denominator in Forney
                int q = f.multiplicativeInverse(p);
                for (int b: betas) if (q != b) // b is beta_k with k != i
                d = f.multiply(d, (1 ^ f.multiply(b, p)));
                    // d * (1 + beta_k * beta_i^-1)
                    // use f.multiply for * and ^ for +
                int e = f.multiply(evaluate(p), f.multiplicativeInverse(d));
                    // Z(beta_i^-1) / d
                    // use evaluate() for Z()
                    // use f.multiply(a, f.multiplicativeInverse(b)) for a/b
                errors.put(f.log[q], e);
            }
            return errors;
        }


    };
```

```
public DE11(String[] args){
    M = new Polynomial(args[0]);
    numberOfErrors = Integer.parseInt(args[1]);
}

 Polynomial makeRSG(){
    Polynomial G = new Polynomial(2);
    G.coeff[0] = G.coeff[1] = 1;
    for (int i = 1; i < Gdegree; i++) G = G.shift(1).add(G.scale(f.alog[i]));
    return G;
 }

 void encode(){
    M.display("the message ");
    C = M.RSencode();
    C.display("RS encoded ");
 }

 void randomErrors(){  // used on result of RSencode
    int numberOfPositions = C.coeff.length;
    Random random = new Random();
    while (errors.size() < numberOfErrors){ // random error positions
        int position = random.nextInt(numberOfPositions);
        if (!errors.containsKey(position))
            errors.put(position, 1 + random.nextInt(f.fieldSize - 1));
    }
 }

 void checkErrors(HashMap<Integer, Integer> e){  // print for errors and
recoveredErrors
    e.forEach((k, v) -> System.out.println(k + " " + Integer.toHexString(v)));
    System.out.println();
 }

 void errorCorrectionExperiment(){
    randomErrors();
    CplusE = C.addError(errors);
    CplusE.display("Error added ");
    Syndrome = CplusE.computeSyndrome();
    Syndrome.display("Syndrome ");
    errorLocator = Syndrome.berlekampMassey();
    errorLocator.display("error locator ");
    HashSet<Integer> zeros = errorLocator.findZeros();
    Polynomial Z = Syndrome.computeZ(errorLocator);
    Z.display("Z");
    HashSet<Integer> betaInverses = errorLocator.findZeros();
    HashMap<Integer, Integer> recoveredErrors = Z.forney(betaInverses);
    checkErrors(errors);
    checkErrors(recoveredErrors);  // comparing recovered errors to real errors
    Polynomial Crecovered = CplusE.addError(recoveredErrors); // correcting using
recoevered errors
    Crecovered.display("corrected");
 }
```

```
public static void main(String[] args){
   if (args.length < 2){
     System.err.println("Usage: java DE11 message numberOfErrors");
     return;
   }
   DE11 de11 = new DE11(args);
   de11.encode();
   de11.errorCorrectionExperiment();
}
}
```