# Penerapan *Reed-Solomon Code* dalam Pembuatan *Self-Correcting* Data

Farhan Nabil Suryono - 13521114[1]
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
*[1]13521114@std.stei.itb.ac.id*

***Abstract*—Data merupakan cara komputer untuk menyimpan informasi. Ketika menyimpan atau melakukan pengiriman data, data dapat mengalami data *corruption*. Data yang telah *corrupted* akan menghasilkan informasi yang salah ketika digunakan sehingga dapat menimbulkan kekacauan. *Error correction code* atau kode yang dapat memperbaiki dirinya sendiri muncul sebagai suatu solusi akan masalah ini. Makalah ini akan membahas melakukan implementasi salah satu *Error Correction Code* yaitu *Reed-Solomon Code* secara sederhana.**

***Keywords—Reed-Solomon Code, Error Correction Code,* Aritmatika Modular*, Welch-Berlekamp Algorithm*.**

## I. Pendahuluan

Data merupakan salah satu hal yang sangat penting di era modern. Berdasarkan definisi dari kamus *Cambridge*, data adalah informasi yang tersimpan dalam bentuk elektronik dan dapat disimpan dan digunakan oleh komputer. Selain itu, data juga seringkali dikirim dari komputer satu ke komputer lainnya.

Sebagai salah satu komponen penting di dunia informasi, kesalahan data merupakan sesuatu yang sangat tidak diinginkan. Namun, pada kenyataannya, data sangatlah rawan terkena *corruption* yang akan menyebabkan kesalahan informasi ketika digunakan. Hal ini sangatlah berbahaya karena kesalahan informasi dapat menyebabkan kekacauan meski dalam jumlah kecil sekalipun. Salah satu contoh data *corruption* yang paling umum adalah data rusak pada CD yang terjadi akibat tergores.

Untuk meminimalisir *data corruption*, salah satu usaha yang dilakukan para peneliti adalah membuat data yang dapat mengoreksi dirinya sendiri. Konsep ini diperkenalkan oleh Richard Hamming pada tahun 1950 dengan *Hamming Code* buatannya. Namun, *Hamming Code* memiliki batasan yaitu jumlah maksimal *error* yang sangat rendah. Melanjutkan inovasi yang telah dibuat Hamming, pada tahun 1960, Irving S. Reed dan Gustave Solomon membuat *self-correction code* yang jauh lebih baik dibanding *Hamming Code* yang kelak dinamakan *Reed-Solomon Code*. *Reed-Solomon Code* bekerja dengan memanfaatkan polinomial dalam suatu medan hingga yang disebut *Galois Field*.

Makalah ini berfokus dalam mengimplementasikan *Reed-Solomon Code* dalam suatu teks atau pesan. Penulis juga akan mensimulasikan *data corruption* pada data yang telah di-*encode*.

## II. Dasar Teori

### A. Aritmatika Modular

Ketika kita membagi dua bilangan bulat a dengan m, kita sebenarnya membuat suatu persamaan yang memiliki bentuk sebagai berikut

$$a = mq + r \text{ dengan } 0 \leq r < n$$

dimana q adalah hasil bagi dan r adalah sisa dari pembagian kedua bilangan bulat tersebut. Persamaan ini dapat ditulis dalam bentuk lain yaitu dalam bentuk modulo seperti berikut

$$a \bmod m = r$$

Aritmatika Modular adalah suatu sistem aritmatika untuk bilangan bulat yang didasari dengan konsep modulo seperti contoh di atas. Di contoh, m disebut sebagai modulus atau modulo dan hasil perhitungan dalam sistem ini akan selalu terletak dalam himpunan {0, 1, 2, ..., m – 1}. Sistem ini merupakan salah satu contoh *Finite Field* atau biasa disebut *Galois Field* karena memiliki jumlah elemen yang terbatas.

### B. Pembagi Bersama Terbesar

Bila terdapat dua bilangan bulat tidak nol a dan b, pembagi bersama terbesar atau PBB adalah suatu bilangan bulat terbesar d dimana d habis membagi a dan b. Dengan demikian, hal ini dapat ditulis dalam bentuk PBB(a, b) = d.

### C. Extended Euclidean Algorithm

Diberikan dua bilangan bulat a dan b, PBB(a, b) selalu dapat dinyatakan dalam suatu persamaan yang memiliki bentuk sebagai berikut

$$ma + nb = PBB(a, b)$$

*Extended Euclidean Algorithm* merupakan suatu algoritma yang dapat menghitung nilai PBB(a, b) sekaligus menghitung nilai m dan n pada persamaan diatas. Karena perhitungan dilakukan bersamaan, m dan n dapat didapat hampir tanpa beban tambahan bila dibandingkan dengan hanya menghitung PBB(a, b).

Algoritma ini dapat dikerjakan dengan rekursi dengan memanggil PBB(b *mod* a, a) dan menyimpan nilai m dan n. Misalkan nilai m dan n yang didapat dalam pemanggilan rekursif adalah $m_1$ dan $n_1$, m dan n bisa didapat dengan persamaan sebagai berikut:

$$m = n_1 - \lfloor b/a \rfloor * m_1$$
$$n = m_1$$

### D. Invers Modulo

Di dalam aritmatika riil, balikan dari suatu bilangan bulat tidak nol a adalah 1/a sedemikian sehingga a * 1/a = 1. Pada dasarnya konsep invers modulo cukup mirip dengan konsep ini yaitu untuk suatu bilangan bulat tidak nol a dalam aritmatika modulo dengan modulus m maka inversnya adalah suatu bilangan bulat x sedemikian sehingga ax = 1 (mod m). Invers modulo ada apabila a dan m memenuhi syarat yaitu a dan m relatif prima dan m lebih dari 1.

Invers modulo jauh lebih sulit untuk dihitung dibandingkan menghitung invers dalam aritmatika riil. Salah satu metode untuk menghitung invers modulo adalah dengan memanfaatkan *Extended Euclidean Algorithm.* Hal ini memungkinkan karena *Extended Euclidean Algorithm* akan menghasilkan persamaan

$$ax + my = 1$$

karena a dan m relatif prima. Bila persamaan di atas kita lakukan modulo m maka didapat persamaan berikut

$$ax \equiv 1 \ (mod\ m)$$

Dengan demikian, kita cukup mengambil nilai x dari hasil melakukan *Extended Euclidean Algorithm* untuk mendapatkan invers modulo.

### E. Polinomial

Berdasarkan definisi dari kamus *Cambridge*, polinomial adalah penjumlahan atau pengurangan dari angka dan suatu variabel atau lebih. Bentuk umum polinomial dengan satu variabel berderajat n adalah sebagai berikut

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

dimana $a_n$ hingga $a_1$ adalah koefisien, $a_0$ adalah konstanta, dan $x^n$ hingga x adalah variabel.

### F. Reed-Solomon Code

*Reed-Solomon Code* merupakan salah satu *error correcting codes* dengan konsep *block.* Kode Reed-Solomon berbasi *Galois Field* yang memiliki elemen terbatas dan setiap nilai yang didapat melalui operasi aritmatika pasti akan terdapat di *Field.*

*Reed-Solomon Code* didapat dengan membuat suatu polinomial berdasarkan pesan yang ingin di-*encode*. Sebagai contoh bila seseorang ingin menyampaikan pesan [2, 3, 1] dalam *Field* dengan modulus m maka polinomial yang akan terbentuk adalah

$$F(x) = 2x^2 + 3x + 1 \ (mod\ m)$$

Setelah mendapat polinomial, *codewords* akan dihasilkan berdasarkan hasil evaluasi polinomial di titik 0 hingga panjang pesan + t dimana t adalah jumlah simbol tambahan yang akan digunakan untuk pengoreksian kode.

Ada banyak metode untuk melakukan *decode* terhadap *Reed-Solomon Code*. Salah satunya adalah *Welch-Berlekamp Algorithm* yang juga akan dijelaskan dan digunakan di makalah ini.

### G. Sistem Persamaan Linear

Sistem Persamaan Linear adalah suatu sistem dimana terdapat m buah persamaan dan n variabel $x_1, x_2, \ldots, x_n$.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \vdots \quad\ \vdots \quad\quad\quad\ \vdots \quad\ \vdots & \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned}$$

*Gambar 2.1 Bentuk Umum Sistem Persamaan Linear*

*Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2020-2021/Algeo-03-Sistem-Persamaan-Linier.pdf*

Untuk mempermudah ketika melakukan komputasi, SPL sering dinyatakan dalam suatu bentuk yang dinamakan matriks *augmented*. Matriks *augmented* berisi persamaan-persamaan yang terdapat di SPL dan solusinya.

Contoh:

$$\begin{aligned} x_1 + 3x_2 - 6x_3 &= 9 \\ 2x_1 - 6x_2 + 4x_3 &= 7 \\ 5x_1 + 2x_2 - 5x_3 &= -2 \end{aligned} \quad \Rightarrow \quad \begin{bmatrix} 1 & 3 & -6 & 9 \\ 2 & -6 & 4 & 7 \\ 5 & 2 & -5 & -2 \end{bmatrix}$$

*Gambar 2.2 Contoh Matriks Augmented*

*Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2020-2021/Algeo-03-Sistem-Persamaan-Linier.pdf*

### H. Eliminasi Gauss Jordan

Eliminasi Gauss-Jordan adalah salah satu metode untuk menyelesaikan Sistem Persamaan Linear.

Langkah pertama dalam eliminasi Gauss-Jordan adalah membuat matriks *augmented* dari Sistem Persamaan Linear.

Langkah kedua adalah melakukan Operasi Baris Elementer (OBE) terhadap matriks *augmented* sehingga dihasilkan matriks eselon baris tereduksi. Terdapat 3 operasi pada OBE yaitu

1. Mengalikan sebuah baris dengan konstanta tidak 0
2. Menukar 2 buah baris
3. Menjumlahkan sebuah baris dengan kelipatan dari baris lainnya

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & \ldots & a_{2n} & b_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} & b_m \end{bmatrix} \sim \text{OBE} \sim \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 & * \\ 0 & 1 & 0 & \ldots & 0 & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \vdots & 1 & * \end{bmatrix}$$

*Gambar 2.3 Gambaran Hasil OBE pada Matriks Gauss Jordan*

*Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2020-2021/Algeo-05-Sistem-Persamaan-Linier-2.pdf*

Dari matriks hasil OBE, diperoleh beberapa persamaan. Solusi SPL akan didapat setelah menyelesaikan beberapa persamaan tersebut.

### *I. Welch-Berlekamp Algorithm*

*Welch-Berlekamp Algorithm* adalah salah satu metode untuk melakukan *decoding Reed-Solomon Code*.

Misalkan b adalah *string* kode yang akan di-*encode*, fungsi F(x) adalah fungsi polinom ketika melakukan *encode*, dan e adalah jumlah error, persamaan untuk algoritma ini adalah:

$$b_i E(a_i) = F(a_i)E(a_i)$$

Dimana $E(a_i)$ akan bernilai 0 bila $b_i \neq F(a_i)$. Persamaan ini tidak dapat dilakukan secara langsung. Namun, dengan memisalkan suatu polinom Q(x) berderajat q yang merupakan hasil kali F(x) dan E(x) dan memberi batasan koefisien untuk derajat tertinggi pada E(x) atau $ce_e$ bernilai 1, kita akan mendapat persamaan berikut

$$\begin{gathered} b_i E(a_i) = Q(a_i) \\ b_i E(a_i) - Q(a_i) = 0 \\ b_i(ce_0 + ce_1 a_i + ce_2 a_i^2 + \cdots + ce_e a_i^e) - \\ \left(cq_0 + cq_1 a_i + cq_2 a_i^2 + \cdots + cq_q\right) = 0 \end{gathered}$$

dengan $ce_i$ dan $cq_i$ adalah koefisien dari polinom E(x) dan Q(x) pada suku berpangkat i.

Karena kita membatasi nilai $ce_e$ dengan 1, maka persamaan diatas berubah menjadi

$$\begin{gathered} b_i(ce_0 + ce_1 a_i + ce_2 a_i^2 + \cdots + ce_{e-1} a_i^{e-1}) - \\ \left(cq_0 + cq_1 a_i + cq_2 a_i^2 + \cdots + cq_q\right) = -b_i a_i^e \end{gathered}$$

Bila kita membuat persamaan tersebut menjadi suatu persamaan linear, kita akan mendapat E(x) dan Q(x). Lalu, kita mendapatkan F(x) dengan cara membagi Q(x) dengan E(x) dengan catatan pembagian polinom tersebut tidak bersisa. Bila bersisa, maka jumlah error telah melebihi batas maksimal dan tidak bisa di-*recover*.

## III. Implementasi Reed-Solomon Code Pada Suatu Pesan

Implementasi program untuk melakukan *encode* dan *decode* Reed-Solomon Code pada makalah ini dibuat dengan bahasa Python versi 3.9.8 dan menggunakan format RS(255, 223). Mekanisme program *encode* pada makalah ini adalah menerima sebuah input pengguna berupa pesan yang akan di-*encode* lalu hasil *encode* nya akan tersimpan dalam sebuah *file* txt. Lalu, untuk program *decode*, mekanisme programnya adalah menerima *file* txt hasil *encode* pesan lalu melakukan *decoding* menggunakan *Welch-Berlekamp Algorithm* lalu meng-output pesan aslinya.

Salah satu struktur data yang digunakan disini adalah struktur data polinomial yang diimplementasikan menggunakan *list of coefficients*, dimulai dari koefisien suku dengan derajat 0 hingga derajat terbesar dikurangi 1. Sebagai contoh. untuk polinomial

$$F(x) = 2x^3 + 3x + 1$$

akan tersimpan dalam bentuk [1, 3, 0, 2]. Suku dengan nilai koefisien 0 tetap ditulis untuk memudahkan pemakaian.

Dalam kode Python, dapat dibuat suatu *class* polynomial. Polinomial diinisasi dengan masukan *list of coefficients* dan memiliki representasi dalam bentuk polinomial dengan variabel x.

```python
class Polynomial:
    def __init__(self, listCoef):
        for i in range(len(listCoef) - 1, -1, -1):
            if listCoef[i] == 0:
                listCoef = listCoef[:i]
            else:
                break

        self.coeffs = listCoef


    def __repr__(self):
        chunks = []
        power = -1

        for coef in self.coeffs:
            power += 1
            if coef == 0:
                continue
            chunk = f" + {coef}"
            if power != 0:
                chunk += "x"
            if power > 1:
                chunk += f"^{power}"
            chunks.append(chunk)

        chunks[0] = chunks[0].lstrip(" + ")
        return ''.join(chunks)
```

*Gambar 3.1 Implementasi Inisiasi dan Representasi pada Class Polynomial*

*Sumber: Dokumen Pribadi*

Selanjutnya, terdapat 2 *method* yaitu *evaluate* dan *isZero* yang masing-masing digunakan untuk mengevaluasi nilai F(x) dalam *Finite Field* berbasis modulus m dan mengecek apakah polinomial F(x) bernilai 0.

```python
def __call__(self, x, m):
    return self.evaluate(x, m)


def evaluate(self, x, m):
    result = 0

    for power, coef in enumerate(self.coeffs):
        result = (result + coef * (x ** power)) % m

    return result


def isZero(self):
    return self.coeffs == []
```

*Gambar 3.2 Implementasi Methods pada Class Polynomial*

*Sumber: Dokumen Pribadi*

### A. Encoding Reed-Solomon Code

Proses *Encoding* suatu *Reed-Solomon Code* sangatlah simpel. *Encoder* menerima suatu pesan. Setelah itu, pesan yang berupa *string* diubah menjadi *array of integers* yang bernilai kode ascii dari setiap karakter pada pesan. Lalu, berdasarkan *array of integers* yang telah terbentuk, dibuat suatu polinom F(x) dengan koefisien-koefisien bernilai kode ascii pesan. Sebagai contoh, pesan "Hai" akan menghasilkan polinomial berikut

$$F(x) = 72x^2 + 97x + 105 \ (mod\ 257)$$

257 digunakan sebagai basis *Galois Field* disini karena 257 adalah bilangan prima terkecil yang lebih besar daripada nilai maksimum kode ascii yaitu 255.

Setelah polinom berhasil dibuat, data *Encoded* dibuat berdasarkan hasil evaluasi nilai F(0) hingga F(n – 1) dimana n adalah panjang *Reed-Solomon Code*.

```
def encoder(message):
    global modulus, n

    intMessage = [ord(i) for i in reversed(message)]
    polynom = polynomial.Polynomial(intMessage)

    encodedMessage = []
    for i in range(n):
        encodedMessage += [polynom(i, modulus)]

    return encodedMessage
```

*Gambar 3.3 Implementasi Encoder Reed-Solomon Code*

*Sumber: Dokumen Pribadi*

### B. Decoding Reed-Solomon Code

Ada banyak metode untuk melakukan *decoding Reed-Solomon Code*. Di makalah ini, penulis menggunakan *Welch-Berlekamp Algorithm* untuk melakukan *decode*.

Langkah pertama adalah membuat matriks *augmented* yang berisi persamaan berikut

$$b_i(ce_0 + ce_1 a_i + ce_2 a_i^2 + \cdots + ce_{e-1} a_i^{e-1}) - \left(cq_0 + cq_1 a_i + cq_2 a_i^2 + \cdots + cq_q\right) = -b_i a_i^e$$

```
def createDecodingEqSystem(encoded, e):
    global n, k

    q = n - e

    eqSystem = []
    for i in range(n):
        eqSystem += [[]]

        for j in range(e):
            eqSystem[i] += [encoded[i] * (i ** j) % modulus]

        for j in range(q):
            eqSystem[i] += [-1 * (i ** j) % modulus]

        eqSystem[i] += [(-1) * encoded[i] * (i ** e) % modulus]

    return eqSystem
```

*Gambar 3.4 Implementasi Generator Matriks Augmented*

*Sumber: Dokumen Pribadi*

Langkah kedua adalah mencari solusi dari sistem persamaan linear tersebut. Di program ini, solusi SPL didapat menggunakan eliminasi Gauss-Jordan dengan modifikasi menyesuaikan dengan *Galois Field*.

```
def modularArithmeticOBE(matrix, irow, k, modulus):
    matrix[irow] = [i * inverseModulus(k, modulus) % modulus for i in matrix[irow]]

    return matrix


def swapRow(matrix, r1, r2):
    temp = matrix[r1]
    matrix[r1] = matrix[r2]
    matrix[r2] = temp

    return matrix


def searchNonZeroInColumn(matrix, col, rowmin):
    for i in range(rowmin, len(matrix)):
        if matrix[i][col] != 0:
            return i

    return -1
```

*Gambar 3.5 Implementasi Fungsi Pembantu dan OBE yang Dimodifikasi untuk Eliminasi Gauss Jordan pada Aritmatika Modular*

*Sumber: Dokumen Pribadi*

```
def gaussJordanModular(matrix, modulus):
    j = 0

    for i in range(len(matrix)):
        while j < len(matrix[0]) - 1 and matrix[i][j] == 0:
            temp = searchNonZeroInColumn(matrix, j, i + 1)

            if temp != -1:
                matrix = swapRow(matrix, i, temp)
            else:
                j += 1

        if j == len(matrix[0]) - 1:
            break

        if matrix[i][j] != 0 and matrix[i][j] != 1:
            matrix = modularArithmeticOBE(matrix, i, matrix[i][j], modulus)

        for row in range(len(matrix)):
            if row != i and matrix[row][j] != 0:
                multiplier = matrix[row][j]

                for col in range(len(matrix[0])):
                    matrix[row][col] = (matrix[row][col] - multiplier * matrix[i][col]) % modulus

    return matrix
```

*Gambar 3.6 Implementasi Eliminasi Gauss-Jordan yang Dimodifikasi*

*Sumber: Dokumen Pribadi*

Dapat terlihat pada Gambar 3.5 bahwa salah satu OBE yang biasa dilakukan di Eliminasi Gauss-Jordan yaitu membagi satu baris agar nilai terkiri tak nolnya bernilai 1 menggunakan modulo Invers karena menggunakan aritmatika modular. Invers Modulo ini didapat dengan *Extended Euclidean Algorithm*.

```python
def extendedEuclideanAlgorithm(a, b):
    if a == 0:
        return b, 0, 1

    fpb, x, y = extendedEuclideanAlgorithm(b % a, a)

    m = y - (b // a) * x
    n = x

    return fpb, m, n


def inverseModulus(num, modulus):
    ret = extendedEuclideanAlgorithm(num, modulus)

    return ret[1]
```

*Gambar 3.7 Implementasi Extended Euclidean Algorithm dan Invers Modulo*

*Sumber: Dokumen Pribadi*

Langkah ketiga adalah membuat polinomial E(x) dan Q(x) dari solusi SPL, lalu kita lakukan pembagian polinomial Q(x) terhadap E(x). Bila sisa pembagiannya tidak ada, maka hasil bagi polinomial tersebut adalah F(x). Namun, bila ada sisa pembagian, maka ulangi dari langkah 1 dengan nilai e yang dikurangi sebanyak 1.

```python
def decoder(encoded):
    global n, k, maxError, modulus

    for e in range(maxError, -1, -1):
        eqSystem = createDecodingEqSystem(encoded, e)

        solution = getSolution(eqSystem, modulus)

        ePolynom = polynomial.Polynomial(solution[:e] + [1])
        qPolynom = polynomial.Polynomial(solution[e:])

        decoded, remainder = polynomial.dividePolynom(qPolynom, ePolynom, modulus)

        if remainder.isZero:
            return decoded, remainder

    return polynomial.dividePolynom(qPolynom, ePolynom, modulus)
```

*Gambar 3.8 Implementasi Fungsi Utama Decoder Reed-Solomon Code*

*Sumber: Dokumen Pribadi*

## IV. Simulasi Encode, Decode, dan Data Corruption

Untuk melakukan simulasi terjadinya Data Corruption, penulis mengambil *file* .txt hasil *encoding*. Lalu, beberapa nilai diubah oleh penulis. Agar lebih meyakinkan, nilai-nilai ini diubah secara *random* menggunakan library *random* bawaan Python.

```python
namafile = input("Masukkan Nama File yang Ingin Dicorrupt: ")
encoded = literal_eval(readEncoded(os.path.join(getCurrentDirectory(), namafile)))
nc = int(input("Masukkan Jumlah Data yang Ingin Dicorrupt: "))

for idx in random.sample(range(n), nc):
    encoded[idx] = (encoded[idx] + random.randint(0, n)) % modulus

writeEncoded(str(encoded), os.path.join(getCurrentDirectory(), namafile))
```

*Gambar 4.1 Implementasi Simulasi Data Corruption.*

*Sumber: Dokumen Pribadi*

Berikut adalah rangkaian langkah simulasi:

1. Memasukkan pesan yang akan di-*encode*.
2. Melakukan *corrupt* data
3. Mengecek perbedaan data awal dan akhir
4. Melakukan *decode* data

Penulis akan melakukan 2 kali simulasi yaitu bila jumlah data yang di-*corrupt* kurang dari sama dengan jumlah error maksimal (bernilai 16 di RS(255, 223)) dan bila jumlah data yang di-*corrupt* melebihi jumlah error maksimal.

Berikut adalah hasil percobaan simulasi untuk jumlah error kurang dari sama dengan 16.

```
Masukkan Pesan: Halo, Dunia!
Masukkan Nama File: testacc
Berhasil menyimpan hasil encode di testacc.txt
```

*Gambar 4.2 Melakukan Encode Halo, Dunia!*

*Sumber: Dokumen Pribadi*

```
Masukkan Pilihan: 2
Masukkan Nama File yang Ingin Dicorrupt: testacc
Masukkan Jumlah Data yang Ingin Dicorrupt: 16
Data yang berubah:
1. Nilai ke-0 berubah dari 33 menjadi 243
2. Nilai ke-1 berubah dari 223 menjadi 58
3. Nilai ke-2 berubah dari 142 menjadi 83
4. Nilai ke-3 berubah dari 197 menjadi 191
5. Nilai ke-5 berubah dari 77 menjadi 143
6. Nilai ke-21 berubah dari 37 menjadi 219
7. Nilai ke-47 berubah dari 63 menjadi 124
8. Nilai ke-52 berubah dari 142 menjadi 222
9. Nilai ke-59 berubah dari 175 menjadi 196
10. Nilai ke-64 berubah dari 202 menjadi 241
11. Nilai ke-71 berubah dari 87 menjadi 88
12. Nilai ke-155 berubah dari 218 menjadi 128
13. Nilai ke-166 berubah dari 115 menjadi 78
14. Nilai ke-173 berubah dari 72 menjadi 196
15. Nilai ke-231 berubah dari 80 menjadi 203
16. Nilai ke-232 berubah dari 99 menjadi 133
Berhasil mengcorrupt data pada file testacc.txt
```

*Gambar 4.3 Melakukan Data Corruption sebanyak 16 Data*

*Sumber: Dokumen Pribadi*

```
Masukkan Pilihan: 3
Masukkan Nama File yang Ingin Didecode: testacc
Apakah mau melakukan perbaikan pada file (y/n)? y
Pesan hasil decode: Halo, Dunia!
File testacc.txt berhasil diperbaiki
```

*Gambar 4.4 Melakukan Decode Pada TestAcc*

*Sumber: Dokumen Pribadi*

Dapat terlihat pada gambar 4.4 bahwa pesan yang asli berhasil didapat meskipun jumlah error bernilai tepat pada batas maksimal RS(255, 223). Lalu, berikut adalah hasil simulasi ke 2 dengan pesan yang berbeda dan jumlah error 17.

```
Masukkan Pesan: Hello, World!
Masukkan Nama File: testreject
Berhasil menyimpan hasil encode di testreject.txt
```

*Gambar 4.5 Melakukan Encode Hello, World!*

*Sumber: Dokumen Pribadi*

```
Masukkan Pilihan: 2
Masukkan Nama File yang Ingin Dicorrupt: testreject
Masukkan Jumlah Data yang Ingin Dicorrupt: 17
Data yang berubah:
1. Nilai ke-30 berubah dari 76 menjadi 77
2. Nilai ke-64 berubah dari 48 menjadi 147
3. Nilai ke-78 berubah dari 249 menjadi 77
4. Nilai ke-98 berubah dari 193 menjadi 4
5. Nilai ke-103 berubah dari 236 menjadi 76
6. Nilai ke-112 berubah dari 109 menjadi 234
7. Nilai ke-115 berubah dari 35 menjadi 83
8. Nilai ke-118 berubah dari 143 menjadi 112
9. Nilai ke-144 berubah dari 251 menjadi 50
10. Nilai ke-172 berubah dari 36 menjadi 77
11. Nilai ke-183 berubah dari 170 menjadi 92
12. Nilai ke-191 berubah dari 150 menjadi 24
13. Nilai ke-202 berubah dari 8 menjadi 65
14. Nilai ke-208 berubah dari 1 menjadi 189
15. Nilai ke-237 berubah dari 73 menjadi 14
16. Nilai ke-247 berubah dari 43 menjadi 3
17. Nilai ke-253 berubah dari 215 menjadi 156
Berhasil mengcorrupt data pada file testreject.txt
```

*Gambar 4.6 Melakukan Data Corruption sebanyak 17 Data*

*Sumber: Dokumen Pribadi*

```
Masukkan Pilihan: 3
Masukkan Nama File yang Ingin Didecode: testreject
Jumlah kesalahan melebihi maxError yang telah ditentukan
```

*Gambar 4.7 Gagal Melakukan Decode pada testreject.txt*

*Sumber: Dokumen Pribadi*

Dapat terlihat pada gambar 4.6 data yang terubah ada sebanyak 17 yang melebihi jumlah batas maksimal yaitu 16. Oleh karena itu, ketika melakukan *decode* yaitu pada gambar 4.7, *decoder* gagal mendapatkan pesan aslinya.

## V. KESIMPULAN DAN SARAN

Data sangatlah rawan terhadap *corruption*. Kesalahan pada data dapat menyebabkan misinformasi yang dapat berujung ke hal buruk. Oleh karena itu, *error correction code* diciptakan untuk mencegah hal buruk akibat *data corruption*. *Reed-Solomon Code* adalah salah satu dari *error correction code*. *Reed-Solomon Code* cukup hemat memori dengan kemampuannya yang dapat memperbaiki *error* yang lokasinya tidak diketahui. Memori yang dipakai *Reed-Solomon Code* sebesar ukuran data awal ditambah dengan 2 kali batas maksimal *error* yang kita atur di awal.

Implementasi pada makalah ini masih bisa di-*improve*. Salah satunya adalah pemilihan algoritma *decoding*. Algoritma yang digunakan di makalah ini yaitu *Welch-Berlekamp Algorithm* adalah algoritma yang cukup lambat karena kompleksitasnya adalah $O(n^3)$. Beberapa peneliti telah berhasil membuat algoritma *decoding* untuk *Reed-Solomon Code* yang lebih cepat, salah satunya adalah implementasi *decoder* dengan memanfaatkan *Fast Fourier Transform* (FFT).

## VI. LAMPIRAN

Berikut adalah lampiran berupa *link source code* GitHub yang berisi implementasi program *Reed-Solomon Code* yang dibuat di makalah ini: https://github.com/Altair1618/Implementasi-Reed-Solomon-Code-Sederhana

## VII. UCAPAN TERIMA KASIH

Penulis disini ingin menyampaikan ucapan terima kasih kepada semua pihak yang telah membantu penulis dalam penyusunan makalah ini, khususnya kepada:

1. Tuhan yang Maha Esa karena atas nikmat dan rahmat-Nya, penulis dapat menyelesaikan makalah ini dengan lancar.
2. Keluarga, terutama kedua orang tua yang senantiasa memberikan dukungan kepada penulis.
3. Dr. Fariska Zakhralativa Ruskanda, S.T., M.T., selaku dosen mata kuliah IF2120 Matematika Diskrit yang telah mengajar dan memberi banyak ilmu untuk penyelesaian makalah ini.
4. Teman-teman yang telah banyak membantu penulis menjalani perkuliahan.

## REFERENSI

[1] https://www.youtube.com/watch?v=X8jsijhllIA diakses pada 8 Desember 2022

[2] https://www.youtube.com/watch?v=1pQJkt7-R4Q diakses pada 8 Desember 2022

[3] https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html diakses pada 8 Desember 2022

[4] https://jeremykun.com/2015/09/07/welch-berlekamp/ diakses pada 9 Desember 2022

[5] Munir, Rinaldi. 2020. Teori Bilangan (Bagian 1): Bahan Kuliah IF2120 Matematika Diskrit. Merupakan slide bahan ajar perkuliahan. Diakses pada tanggal 10 Desember 2022

[6] https://dictionary.cambridge.org/dictionary/english/ diakses pada 10 Desember 2022

[7] Munir, Rinaldi. 2020. Sistem Persamaan Linier (Bagian 1): Bahan Kuliah IF2123 Aljabar Linier dan Geometri. Merupakan slide bahan ajar perkuliahan. Diakses pada tanggal 11 Desember 2022.

[8] Munir, Rinaldi. 2020. Sistem Persamaan Linier (Bagian 2): Bahan Kuliah IF2123 Aljabar Linier dan Geometri. Merupakan slide bahan ajar perkuliahan. Diakses pada tanggal 11 Desember 2022.

## PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Desember 2022

Farhan Nabil Suryono
13521114