**Reed-Solomon code (RS code)**

Reed-Solomon code is one of the powerful error correction methods used to protect data during transmission in error-prone environments (e.g. digital communications, CD/DVD and other media). This method was developed by James Reed and Gustav Solomon in the 1960s. It is a special case of another encoding - BCH codes (cyclic Bose-Chaudhuri-Hocquenghem codes). Unlike other codes, such as **Hamming (see previous lecture)** , Reed-Solomon does not work with individual bits, but with entire blocks of data, such as bytes. BCH codes are designed to correct arbitrary errors, but Reed-Solomon specializes in correcting blocks of data, which makes it more applicable to tasks such as audio, video, and digital data transmission.

Reed-Solomon codes are non-binary perfect systematic linear block codes that belong to the class of cyclic codes.

**main idea**

When we send data, redundant information is added to it, which allows us to correct possible errors when receiving data. This additional data (check characters) allows the receiver not only to detect errors, but also to correct them, and sometimes even to restore lost or completely damaged data.

**The key idea of the Reed-Solomon code** is that it adds redundancy to the original data. For example, if the message is 5 symbols long, the Reed-Solomon code can add 3 more symbols, so that the receiver receives 8 symbols. These additional symbols allow up to *t* errors to be corrected, where *t* = ( *n* − *k* )/2 ( *n* is the total number of symbols, *k* is the number of original symbols).

**Mathematical basis**

The Reed-Solomon code uses a mathematical structure called **a Galois field** (or finite field) to process data, denoted as GF(2^m)), where m is the number of bits used to represent one symbol. **The Galois field** is a special mathematical system where addition, multiplication, and division operations can be performed, which are necessary to create check symbols and correct errors. With this code, data can be protected with high efficiency.

For example, to work with bytes we use the GF(2^8) field, which contains 256 possible values (0-255). This allows us to encode information with large blocks of data, not just bits, which increases the ability to detect and correct errors.

**How does the Reed-Solomon code work?**

**- Coding** :

Imagine your data is a polynomial *D* . A polynomial is just a mathematical expression, but in the context of Reed-Solomon codes it can be a set of characters or bytes.

To protect this data, we multiply it by the generator polynomial *G* . This polynomial is known to both the sender and the receiver. After multiplication, we get the code word *C* - it contains both the original data and the added control symbols.

- **Decoding** :

When the receiver receives the codeword *C* , he divides it by the generator polynomial *G* .

If there is no remainder when dividing, then the data has been transmitted successfully, without errors. But if there is a remainder, this indicates an error. Depending on the type of error and the number of check characters, the Reed-Solomon code can not only detect that there is an error, but also correct it.

In Hamming codes, the control bits controlled only those information bits that were on the right side of them and ignored all their "left-hand" comrades.

In Reed-Solomon codes, the control bits extend their influence to all information bits, and therefore, with an increase in the number of control bits, the number of recognized/correctable errors also increases.

It is precisely due to the latter circumstance that the popularity of Reed-Solomon correction codes is due.

**Relationship between polynomial degrees and errors**

* If the degree of the generator polynomial *G* is greater than the degree of the data polynomial *D* by two or more, then the code can correct errors. The greater the difference in degrees, the more errors can be corrected.
* For example, if the difference between the degrees of the polynomials is two, the code can correct one error. If the difference is four, two errors can be corrected. In general, the number of correctable errors *t* is related to the degree of the polynomial by the formula: k = *2* ⋅ *t* , where *k* is the number of control symbols.

**Why is ordinary arithmetic not suitable?**

When we work with Reed-Solomon codes, we need to use a special arithmetic called finite field (Galois field) arithmetic. This is a special mathematical system in which we can safely perform addition, subtraction, multiplication, and division operations without going beyond the limits of values.

For example, when dividing or multiplying, we must not lose precision or go beyond the limits of acceptable values. Galois arithmetic guarantees that the result of any mathematical operation will be within acceptable limits, which is very important for correct encoding and decoding of data.

**Types of Reed-Solomon Encoders**

There are two types of encoders:

- Non-systematic encoder.

In this case, the code word is completely different from the original. That is, the data is completely changed, and in order to use it, it is necessary to perform full decoding, even if there are no errors.

- Systematic Encoder:

In this case, the original data remains unchanged, and only additional characters (check characters) are added to the end of the data. This is a more convenient method because the data remains understandable and does not need to be decoded if there are no errors.

**Bug fixes**

The main task of the Reed-Solomon code is to correct errors that occur during data transmission. The code's ability to correct errors directly depends on the number of **control characters** (bytes) that are added to the data:

* If you add *r* check bytes, the code will be able to detect **up to *r* corrupted bytes** .
* It is also guaranteed to be able to correct **up to r/2 errors** . That is, if you add 4 check bytes, you can correct up to 2 errors.

The Reed-Solomon code not only corrects errors, but can also recover **erased** or completely lost symbols, making it very useful in systems where data may be partially damaged or lost (for example, in optical storage media or when transmitted over unstable channels).

**Encoder architecture**

Reed-Solomon encoders are often implemented using shift registers. **A shift register** is a sequence of memory cells that store data symbols. Symbols are shifted from one cell to another, and check symbols are calculated based on them. Shift registers can operate either sequentially (one symbol per cycle) or in parallel (several symbols at once). In hardware implementation, parallel shifts can be used, which makes encoding faster.

**Software implementation**

In software implementations, working with shift registers is usually limited by the capabilities of the computer, so parallel shifts are not always possible. This reduces the efficiency of the Reed-Solomon code in software compared to hardware.

In order for **the Reed-Solomon coding algorithm** to be understandable, we will analyze it step by step with detailed explanations. The main task is to create a code word containing both the original data and control symbols that will help correct errors in data transmission.

**Step 1: Add zeros to the original data**

Imagine that we have **an initial information word** *D* , which consists of a set of characters (bytes). The length of this word is *m* characters.

To prepare the data for further coding steps, we **add *k* zeros to the right of this word** . *k* is the number of control characters that we are going to calculate later. These zeros are needed so that the data can be correctly divided into parts.

Now the length of the information word increases and becomes equal to *n* = *m* + *k* . This is important because ***n*** is the total length of the code word (information word + check symbols).

**What it looks like:**

Let's say we have an information word *D* =[ *d* 1 , *d* 2 , *d* 3 ], and we add two zeros to it ( *k* =2):

*D* ′=[ *d* 1 ​, *d* 2 ​, *d* 3 ​,0,0]

Now the length of the word *D* ′=5 characters.

**Step 2: Dividing the polynomial**

Now we represent the information word D′′ as **a polynomial** . This is simply a way to write our data, where each symbol of the word is a coefficient of the polynomial. We then **multiply** our word by *X k* , which shifts it *k* positions to the left, as if increasing its order.

The next step is **to divide** this new polynomial by **the generator polynomial** *G* . The generator polynomial is a key mathematical expression that defines the structure of Reed-Solomon codes and helps calculate the check symbols.

As a result of division we get two values:

1. **Private** *Q* : We don't need it, so we ignore it.
2. **Remainder** *R* : this is what we are interested in. The remainder of the division is exactly the **control characters** that we need to protect the data.

**What it looks like mathematically:**

Let's represent the division as an equation:

*X k* ⋅ *D* = *G* ⋅ *Q* + *R*

Where:

* *X k* ⋅ *D* is an information word shifted by *k* positions.
* *G* is the generating polynomial.
* *Q* is a particular that does not interest us.
* *R* is the remainder, which are our control characters.

**Step 3: Add the remainder to the original data**

Now that we have the remainder *R* , we add it to the original data word *D* . This is **the encoded word** - it consists of the original data and the check symbols.

Important: The check characters are added separately from the original data. This is called **systematic coding** because both the original data and the check characters are visible in the code word.

**What it looks like:**

Let's take our information word *D* = [ *d* 1 , *d* 2 , *d* 3 ] and add to it the calculated control symbols *R* = [ *r* 1 , *r* 2 ]. The resulting code word is:

*C* =[ *d* 1 , *d* 2 , *d* 3 , *r* 1 , *r* 2 ]

Now we have a complete codeword of length *n* , which contains both the original data and the check symbols.

**Step 4: Write the final code word**

At this stage, we can write the final code word as a mathematical formula:

*T* = *X k* ⋅ *D* + *R* = *G* ⋅ *Q*

Here *T* is the coded word that we will transmit. If the data gets corrupted during transmission, we can use the check characters to correct the errors.

**Example with specific values:**

Let's say we have **an information word** D=[1,2,3], and we add two zeros (k=2) to prepare for encoding. Let's represent this as a step-by-step process:

1. **Add zeros:**
   * We have the original data D=[1,2,3], and we add two zeros to it to make the word length m+k=3+2=5. We get:

D′=[1,2,3,0,0]

This is our new information word D′′, which we will use for further coding steps.

1. **We divide by the generating polynomial:**
   * Now we divide this word D′D'D′ by **the generator polynomial** G. For the sake of this example, let's assume that the generator polynomial G=[1,0,1] is just an example of a polynomial that we will be working with.
   * As a result of division we get **a remainder** R, which is two control characters (bytes). Let the result of division be:

R=[4,5]

These control characters r1=4 and r2=5 will be used for error correction.

***Explanation***

- We represent information in the form of polynomials

The original information word D′=[1,2,3,0,0] can be written as a polynomial:

*D* ′( *x* )=1 ⋅ *x* 4 +2 ⋅ *x* 3 +3 ⋅ *x* 2 +0 ⋅ *x* +0= *x* 4 +2 *x* 3 +3 *x* 2

The generating polynomial G=[1,0,1] can be written as a polynomial:

*G* ( *x* ) = *x* 2 +1

**Dividing a polynomial by a polynomial**

1. We take the leading term of the polynomial *D* ′( *x* ) — this is *x* 4 , and divide it by the leading term *G* ( *x* ), which is *x* 2 . We get:

Multiply the result by *G* ( *x* ):

*x* 2 ⋅ ( *x* 2 +1) = *x* 4 + *x* 2

Now we subtract this from *D* ′( *x* ):

( *x* 4 +2 *x* 3 +3 *x* 2 )−( *x* 4 + *x* 2 )=2 *x* 3 +2 *x* 2

1. **Step 2** : Now we divide 2 *x* 3 by *x* 2 . We get:

Multiply the result by *G* ( *x* ):

2x ⋅ (x 2 +1) = 2x 3 +2x

We subtract this from the remaining polynomial:

(2x 3 +2x 2 )−(2x 3 +2x)=2x 2 −2x

1. **Step 3** : Now we divide 2x 2 by x 2 . We get:

Multiply the result by G(x):

2 ⋅ (x 2 +1) = 2 *x* 2 +2

We subtract this from the remaining polynomial:

(2x 2 −2x)−(2x 2 +2)=−2x−2

The remainder of the division is −2x−2. In systems with finite fields, such as the Galois field GF(7), where we can use remainders modulo 7, we transform this result into positive values:

-2≡5 (mod 7)

The notation −2≡5 (mod 7) means that when calculating modulo 7, the negative number −2 is equivalent to the positive number 5, since adding 7 to −2 gives 5, which is in the range from 0 to 6.

So, the remainder R=[5,5]

We divided the information word by the generator polynomial, and the division resulted in a remainder R=[−2,−2]. However, in Reed-Solomon codes we work with numbers in **a finite Galois field** . This means that all calculations are performed modulo, for example, GF(7), where the modulus is 7.

**Transition to positive values**

When we get negative numbers -2 as part of the remainder (check symbols), we must convert them to positive values that lie within the range of valid values of the finite field GF(7). In a finite field with modulus 7, the valid values are the numbers from 0 to 6.

To convert -2 to a positive number modulo 7, you add 7 to -2:

-2+7=5

Thus, −2 in GF(7) is equivalent to 5.

We convert both negative -2 numbers to positive, and both become 5. So the remainder we calculated becomes:

R=[5,5]

**Why do we work with a module?**

The finite field GF(7) means that all calculations are performed modulo 7. Any negative or excessively large numbers are automatically "rolled back" into the valid range of 0 to 6. This ensures correct results and ensures that all numbers in the field remain within that range.

1. **Add the remainder:**
   * Now we add the check symbols R=[5,5] to our original data word D=[1,2,3] to get **the code word** :

C=[1,2,3, 5 ,5]

It is an encoded word that contains both the original data and the control characters.

Now this code word C=[1,2,3,4,5] can be transmitted over the communication channel. If errors occur during transmission (for example, one or two bytes are damaged), the receiver can use control symbols 4 and 5 to detect and correct these errors.

Thus, thanks to the check symbols, the Reed-Solomon code can not only detect but also correct errors, which makes data transmission more reliable.

**Decoding** of the obtained word T is carried out in exactly the same way as described earlier. If, when dividing T (which is actually the product of G and Q) by the generating polynomial G, a remainder is formed, then the word T is distorted and, accordingly, vice versa.

**BUT** in integer arithmetic division is not defined for all pairs of numbers (in particular, 2 cannot be divided by 3, and 9 cannot be divided by 4, without loss of significance, of course). As for floating-point numbers, their accuracy is catastrophically insufficient for the effective use of Reed-Solomon codes), in addition, this arithmetic is quite complex in hardware implementation. In this case, it is better to use special arithmetic - the arithmetic of finite groups called **Galois fields** . The advantage of this arithmetic is that the operations of addition, subtraction, multiplication and division are defined for all members of the field (naturally, excluding the situation of division by zero), and the number obtained as a result of any of these operations is necessarily present in the group! That is, when dividing any integer A belonging to the set 0...255 by any integer B from the same set (naturally, B should not be equal to zero), we obtain a number C included in this set. Therefore, there is no loss of significance, and no uncertainty arises.

Thus, Reed-Solomon correcting codes are based on polynomial operations in Galois fields and require the programmer to have knowledge of several aspects of higher mathematics from the number theory section. Galois fields are an abstraction that cannot be visualized or "touched" with your hands. It simply needs to be accepted as a set of axioms, without trying to understand its meaning, it is enough to just know that it works.

**Let us construct** a prototype of a Reed-Solomon encoder/decoder that operates according to the rules **of ordinary integer algebra** . Naturally, due to the inevitable expansion of the bit grid in this case, it will be very difficult for such an encoder/decoder to find practical application.

We will assume that if *g* = 2 *n* + 1, then for any *a* from the range 0…2 *n* , the product *a* ∗ *g* = *c* (where c is the code word) will represent, in essence, a complete jumble of the bits of both original numbers.

Let's say n = 2, then g = 3. It's easy to see: no matter what we multiply g by – 0, 1, 2, or – 3 – the result obtained is divisible by g if and only if none of its bits are inverted (that is, simply put, there are no single errors).

The remainder of the division clearly indicates the position of the error (provided that the error is single, this algorithm is not capable of correcting group errors). More precisely, if the error occurred at position x, then the remainder of the division k will be equal to k = 2x. To quickly determine x by k, you can use a trivial table algorithm. However, to restore the faulty bit, it is not at all necessary to know its position, it is enough to make R=e ^k, where e is the corrupted codeword, ^ is the XOR operation, and R is the recovered codeword.

*The simplest example of implementing a Reed-Solomon encoder/decoder that works using conventional arithmetic (i.e. with an unjustified expansion of the bit grid) and corrects any single errors in one 8-bit information word (however, the program can easily be adapted for 16-bit information words). Note that the encoder is implemented almost an order of magnitude simpler than the decoder. In a real Reed-Solomon decoder, capable of correcting group errors, this gap is even more significant*

C language

#include <stdio.h>

// width of input information symbol (bits)

#define SYM\_WIDE 8

// input data (one byte)

#define DATAIN 0x69

// bit number to be corrupted by crash

#define ERR\_POS 3

// irreducible polynomial

#define MAG (1<<(SYM\_WIDE\*1) + 1<<(SYM\_WIDE\*0))

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

// determining the position of the error x by the remainder k from the division

// code word to the polynomial k = 2^x, where "^" is the exponentiation

// to the power; the function takes k and returns x

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

int pow\_table[9] = {1, 2, 4, 8, 16, 32, 64, 128, 256};

int lockup(int x) {

int a;

for (a = 0; a < 9; a++)

if (pow\_table[a] == x) return a;

return -1;

}

int main() {

int i, g, c, e, k;

fprintf(stderr, "simplest Reed-Solomon encoder/decoder by Kris Kaspersky\n\n");

// input data (information word)

i = DATAIN;

// irreducible polynomial

g = MAG;

printf("i = %08x (DATAIN)\ng = %08x (POLYNOM)\n", i, g);

// REED-SOLOMONON CODER (the simplest, but still somehow working).

// Calculate the code word to be transmitted

c = i \* g;

printf("c = %08x (CODEWORD)\n", c);

// end of CODER

// we transmit with distortions

e = c ^ (1 << ERR\_POS);

printf("e = %08x (RAW RECEIVED DATA+ERR)\n\n", e);

/\* ^^^^ distort one bit, simulating a transmission error \*/

// REED-SOLOMON DECODER

// check for transmission errors

// (in fact, this is the simplest Reed-Solomon decoder)

if (e % g) {

// errors found, trying to fix

printf("RS decoder says: (%x) error detected\n{\n", e % g);

// k = 2^x, where x is the position of the faulty bit

k = (e % g);

printf("\t0 to 1 err position: %x\n", lockup(k));

printf("\trestored codeword is: %x\n}\n", (e ^= k));

}

printf("RECEIVED DATA IS: %x\n", e / g);

// end of DECODER

}

In Python

# width of input information symbol (bits)

SYM\_WIDE = 8

# input data (one byte)

# Here we use data byte 0x69 (in binary: 01101001)

DATAIN = 0x69

# bit number to be corrupted by crash

# Specify the bit that will be changed to simulate an error. In this case, it is the 3rd bit.

ERR\_POS = 3

# irreducible polynomial

# This is the polynomial that is used for encoding. In this case, the polynomial is represented as the sum of two shifts: 1 << (SYM\_WIDE \* 1) and 1 << (SYM\_WIDE \* 0),

# which ultimately gives us the polynomial 256 + 1 = 257.

MAG = (1 << (SYM\_WIDE \* 1)) + (1 << (SYM\_WIDE \* 0))

# Power table 2. Used to find the position of the error.

# pow\_table stores 2^x values for x from 0 to 8.

pow\_table = [1, 2, 4, 8, 16, 32, 64, 128, 256]

# Function to find the error position by the remainder k

# It takes a number x and returns its position in the table of powers of 2.

# If the remainder k matches any value in pow\_table, its index (error bit position) is returned.

def lockup(x):

for a in range(9):

if pow\_table[a] == x:

return a

return -1 # If not found, return -1

# The main function of the program

def main():

print("Simplest Reed-Solomon encoder/decoder \n")

# Input data (information word)

i = DATAIN # This is our data word that we will encode

# Irreducible polynomial

g = MAG # Generator polynomial that is used for encoding and decoding

print(f"i = {i:08x} (DATAIN)\ng = {g:08x} (POLYNOM)")

# REED-SOLOMON CODER

# Coding: multiply the information word i by the generator polynomial g, obtaining the code word c

c = i \* g

print(f"c = {c:08x} (CODEWORD)")

# end CODER

# We transmit data with distortions

# We change one bit in the code word to simulate a transmission error

# XOR (^) changes one bit which we specify with shift 1 << ERR\_POS

e = c ^ (1 << ERR\_POS)

print(f"e = {e:08x} (RAW RECEIVED DATA+ERR)\n")

# This is a simulation of data transmission with an error in the 3rd bit.

# REED-SOLOMON DECODER

# Check for errors in received data

# If the remainder of the division (e % g) is not 0, then there was an error

if e % g != 0:

# Error detected. We are trying to find and fix it.

print(f"RS decoder says: ({e % g:x}) error detected\n{{")

# k = e % g, where k is the remainder of the division. This number corresponds to 2^x, where x is the position of the bad bit.

k = e % g

# lockup(k) searches for the error position corresponding to the remainder k

print(f"\t0 to 1 err position: {lockup(k):x}")

# Correct the codeword using XOR to cancel the error

e ^= k

print(f"\trestored codeword is: {e:x}\n}}")

# Data obtained after fixing the error

# Here we divide the corrected codeword e by the generator polynomial g to obtain the original data

print(f"RECEIVED DATA IS: {e // g:x}")

# Launching the main program block

if \_\_name\_\_ == "\_\_main\_\_":

main()

***Explanations***

DATAIN : This is the raw data (information word) that we want to transmit.

ERR\_POS : This parameter specifies the bit that will be changed to simulate an error in data transmission .

MAG : This is the generator polynomial that is used for encoding and decoding. It determines how the data will be encoded.

pow\_table : A table of powers of two used to find the position of the remainder error.

lockup ( x ): This function finds which bit was corrupted (where the error is) using a power-of-2 table.

Main function:

- First, the information word is encoded by multiplying it by the generating polynomial.

- Then there is a transmission with an error (one bit changes).

- After this, the code tries to detect and correct the error.

Encoding: Multiplying the original data by a polynomial yields the encoded word.

Decoding: The error is detected by checking the remainder of the division. If the remainder is not zero, then there is an error that we can find and fix.

The result of the simplest Reed-Solomon encoder/decoder. Note: the distorted bit was successfully corrected, but to do this, three bits had to be added to the original information word instead of two (if you take the maximum permissible 8-bit value 0xFF as the input word, the code word will be equal to 0x1FE00, and since 2 10 = 10000, there are not enough free bits and the bit grid has to be increased to 2 11 , while the least significant bits of the code word actually remain unused and the “correct” encoder must “loop” them, roughly speaking, closing the processed bits like a ring.

i = 00000069 (DATAIN)

g = 00000200 (POLYNOM)

c = 0000d200 (CODEWORD)

e = 0000d208 (RAW RECEIVED DATA+ERR)

RS decoder says: (8) error detected

{

0 to 1 err position: 3

restored codeword is: d200

}

RECEIVED DATA IS: 69

**Advantages of the Reed-Solomon code**

1. High error tolerance: The code can correct a large number of errors, especially when transmitting data in blocks (e.g. bytes).
2. Wide range of applications:
   1. CD, DVD and Blu-ray: To correct errors when reading data from discs.
   2. Mobile communications and digital television: To correct errors during data transmission over unstable channels.
   3. QR Codes: The Reed-Solomon code is used to repair partially damaged QR codes.
   4. Internet protocols (e.g. IEEE 802.16): For transmitting data with high reliability.
3. Data recovery: The code is capable of recovering not only erroneous but also completely lost characters.

**Example of code running**

Let's say we have a 5 byte message that we want to protect against errors. We add 3 check bytes to protect that data. So we now have 8 bytes on the output.

1. **Encoding** : The original message is converted into a mathematical polynomial. The check bytes are calculated by dividing this polynomial by a generator polynomial that is defined for a particular Reed-Solomon code. The remainder of the division is the check characters.
2. **Data transmission** : The message along with the control characters is sent over the communication channel. If several bytes are damaged, the control bytes can be used at the receiving end to detect and correct these errors.
3. **Decoding** : First, the receiver checks the integrity of the message using check symbols. If errors are detected, the Reed-Solomon code uses mathematical methods (such as error syndromes and the Berlekamp-Massey algorithm) to correct the data and restore the original message.

**Conclusion**

The Reed-Solomon code is a powerful and widely used error correction method that is used in various areas of digital technology. It not only corrects errors, but also restores lost data, providing high reliability in data transmission under interference and failure conditions.