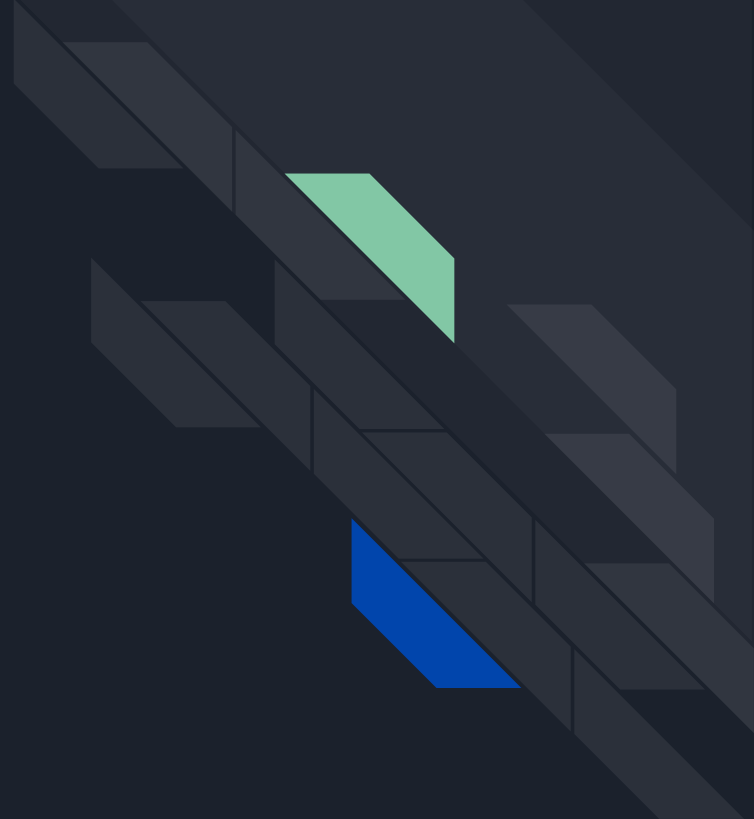




How Reed-Solomon Encoding Works

by Thomas Manning

Part 1: Coding Theory





Messages

Messages are strings of symbols, i.e. "001010", "Hello, World", or "0xDEADBEEF".

The symbols come from a set of elements. Also known as an alphabet. Notationally we'll use " Σ ".

A string of K symbols is represented as Σ^K .

Binary strings have $\Sigma = \{0, 1\}$. With this, a byte can be thought of as Σ^8

Messages with a single byte as their symbol (i.e. bytestrings) would have $\Sigma = \{0, 1, \dots, 255\}$. An `int64` is Σ^8 when Σ is a byte.



Erasures and Errors

When a data is transmitted/stored, two types of faults can occur:

- Erasure: A known loss (or corruption) of a symbol.
 - You know the position of the fault.
 - Example: An HDD/SSD dies, but we know it's unavailable and what symbols it was supposed to store.
 - Example: A physical encoding where a voltage of -1 is "0", 1 is "1" and 0 is "missing".
- Error: An unknown corruption of a symbol.
 - You do not know the position of the fault.
 - A bit flip.
 - A scratch on a CD causing a mis-reading.



2x Redundancy

A basic scheme for handling errors and erasures: redundancy. We take a message and modify it to form a "codeword" which is what we store/transmit.

If we repeat every element twice, we can:

- *Correct* one erasure per symbol, because we have 2 copies and we know which one is good.
- *Detect* one error per symbol, because we know there is a mismatch (but we don't know which is wrong).

Message	Codeword	w/Erasures	w/Errors
ABC	AABBCC	A__BC_	ABABCA
		ABABCA	



3x Redundancy

What if we want to also correct errors? We need more redundancy!

If we repeat every element *thrice*, we can:

- *Correct* two erasures per symbol, because we have 3 copies and we know which ones are good.
- *Correct* one error per symbol, because we can see there is a mismatch AND we can pick the most common symbol (i.e. "A" if it's "AAB").
- *Detect* one XOR two errors per symbol (2 if the errors go to different symbols).

Message	Codeword	w/Erasures	w/Errors
ABC	AAABBBCCC	_A___BC__	AABAABCAB



The Problem with Redundancy

It's expensive.

Can we achieve something similar but with less *additional* data?



Parity Bit

For binary messages, we can XOR all the bits in the message and get one extra "parity" bit. With this we can

- *Correct* a single erasure in the message by XORing all the other bits (including the parity) together.
- *Detect* a single error per message by repeating the process and seeing if the parity matches between what's received and the calculation.

Message
0101

Codeword
01010

w/Erasures
0_010

w/Errors
11010



Generalizing Parity

- Can we have more than one parity bit?
- Does parity work for non-binary messages?
- Can we find a code that adds the least number of symbols to correct a given number of errors/erasures?

YES!

But first, some theory...



Linear Block Codes

Injective Mapping: $C: \Sigma^K \rightarrow \Sigma^N$

Σ : An alphabet (set of symbols); $|\Sigma| = q$.

Σ^K : A string of K symbols from Σ . $\Sigma^K = \text{"message"}$; $\Sigma^N = \text{"block" or "codeword"}$

Example:

Message	Codeword
abc	abca
abb	abbc
bab	babc
cca	ccab

$\Sigma = \{a, b, c\}$

$\rightarrow = \text{map } a \rightarrow 0, b \rightarrow 1, c \rightarrow 2$

sum modulo 3

unmap and append result to message

ex. $abc \rightarrow (0 + 1 + 2) \% 3 = 0 = a: abc \rightarrow abca$



Mappings

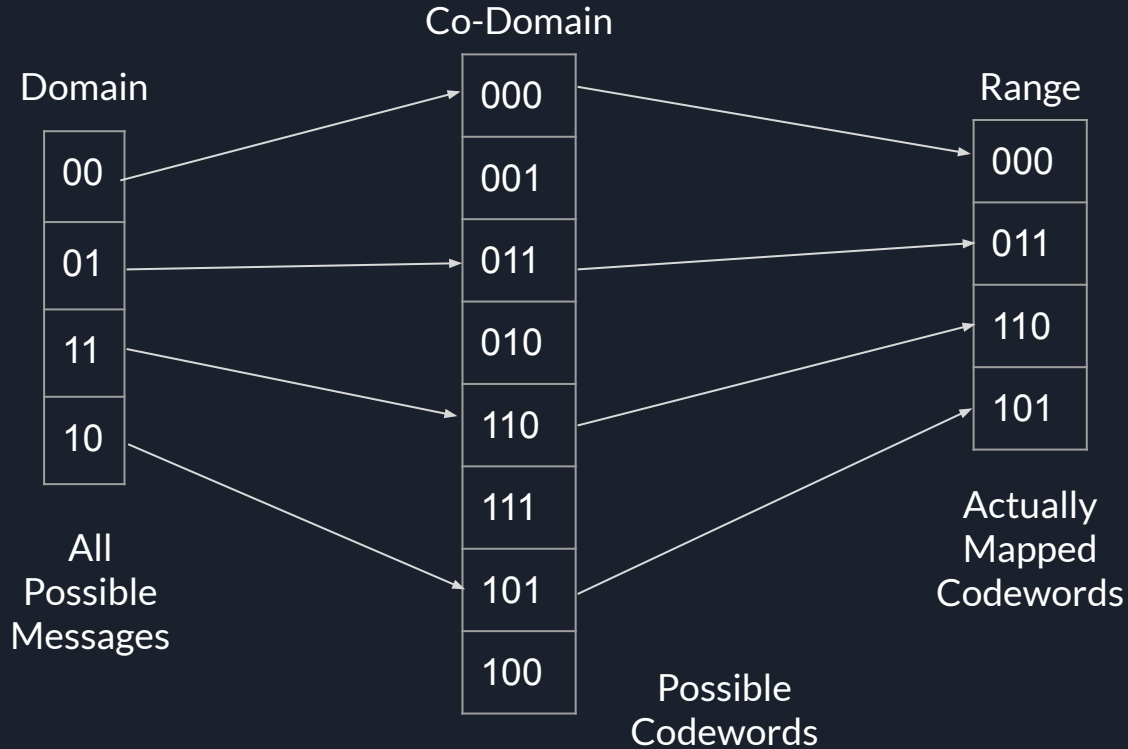
Mappings have three sets:

- Domain, this is the set that is mapped from. In our case, it's the set of all possible messages or Σ^K
- Co-Domain, this is the set of possible elements that could be mapped to (but aren't necessarily). In our case, it's the set of all possible codewords, or Σ^N
- Range, this is the non-proper subset of the co-domain that *is* mapped to. In our case, it's the set of codewords actually mapped to.

Injective means that:

- Every element in the domain is mapped *from*
- Not necessarily every element in the co-domain is mapped *to*.
 - Which is the same as saying the range can be smaller than the co-domain.

Simple Parity Coding

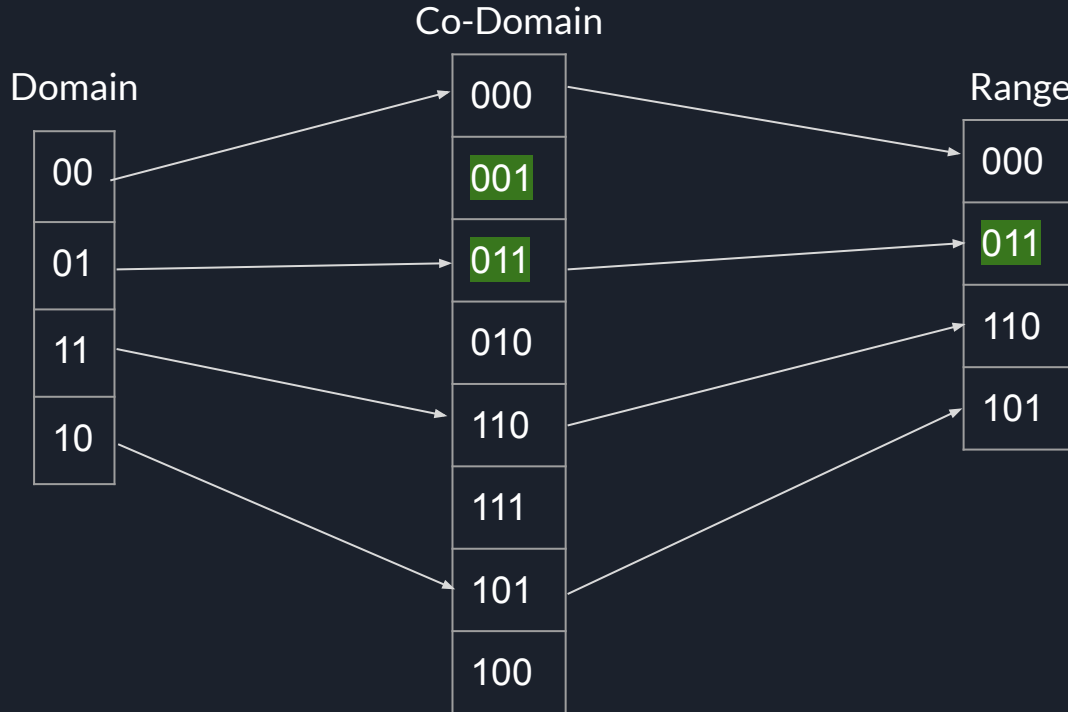




Correcting Erasures

We can *correct* an erasure by ignoring the erased symbol in the codeword and every codeword in the *range* and seeing if there is a single match.

Simple Parity Coding - Erasure



Received Codeword
0_1



Hamming Distance

Distance(Σ^K, Σ^K) $\rightarrow \mathbb{N}$ The substitution-only distance between two strings

Distance(abc, abc) $\rightarrow 0$

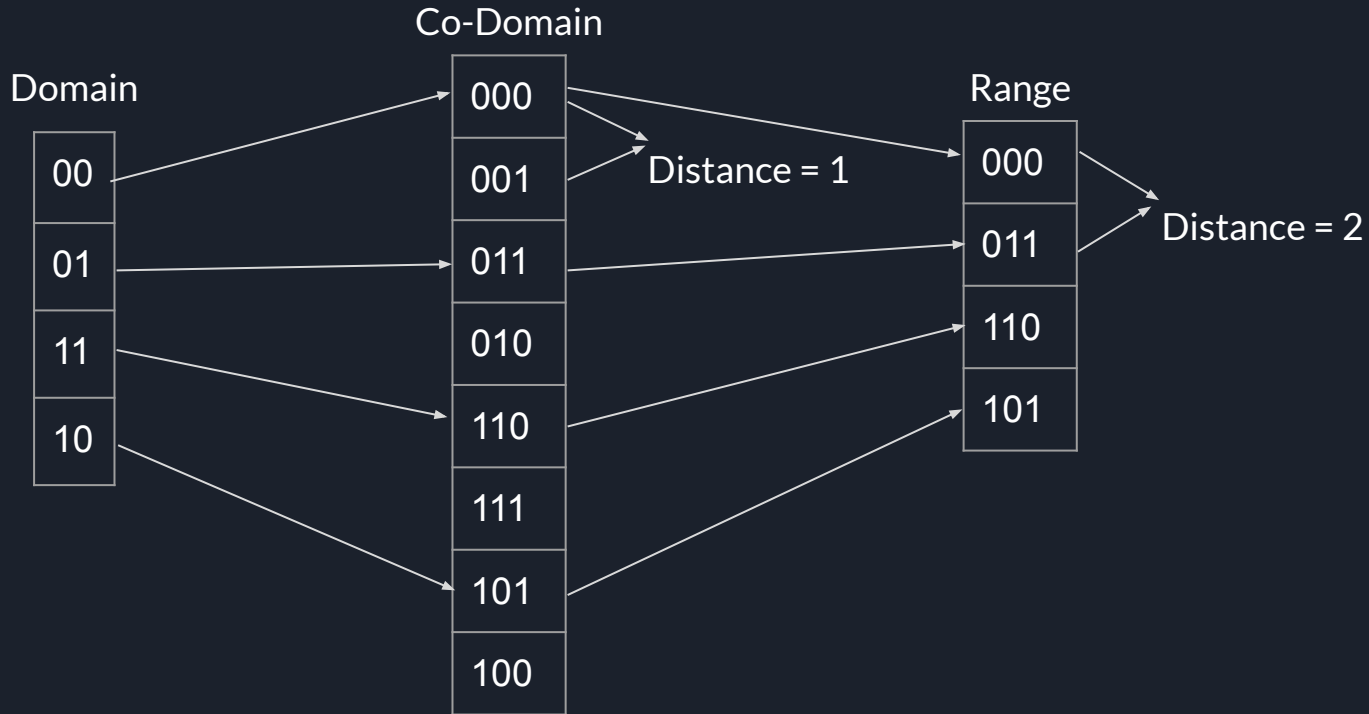
Distance(abc, aac) $\rightarrow 1$ (in position 2)

Distance(abc, bac) $\rightarrow 2$ (in position 1 and 2)

Distance(abc, cab) $\rightarrow 3$ (in all positions)

In a binary string, the hamming distance is the number of bitflips between 2 strings.


Simple Parity Coding - Distance





Hamming Distance Visualized


- I'll be using a list for most demos, but the true way of thinking about distance is as a graph where all codewords in the domain are vertices and there's an edge between any vertices distance 1 away from each other.
- We can represent this graph geometrically and see some interesting results.
- For simplicity, let's just consider binary codes.



Hamming Distance Visualized - Binary Codes

Binary Codewords Length 1



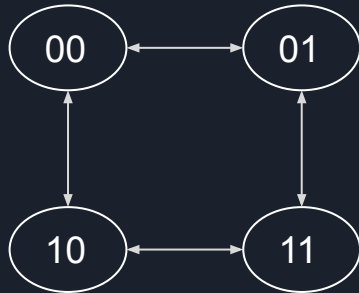


Hamming Distance Visualized - Binary Codes

Binary Codewords Length 1



Binary Codewords Length 2

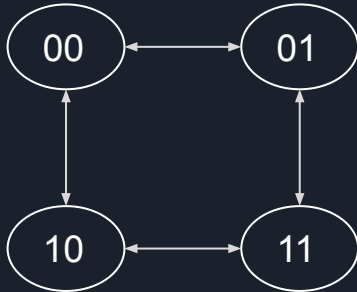


Hamming Distance Visualized - Binary Codes

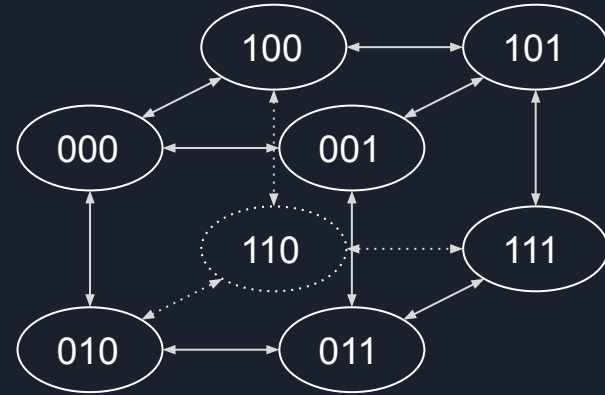
Binary Codewords Length 1



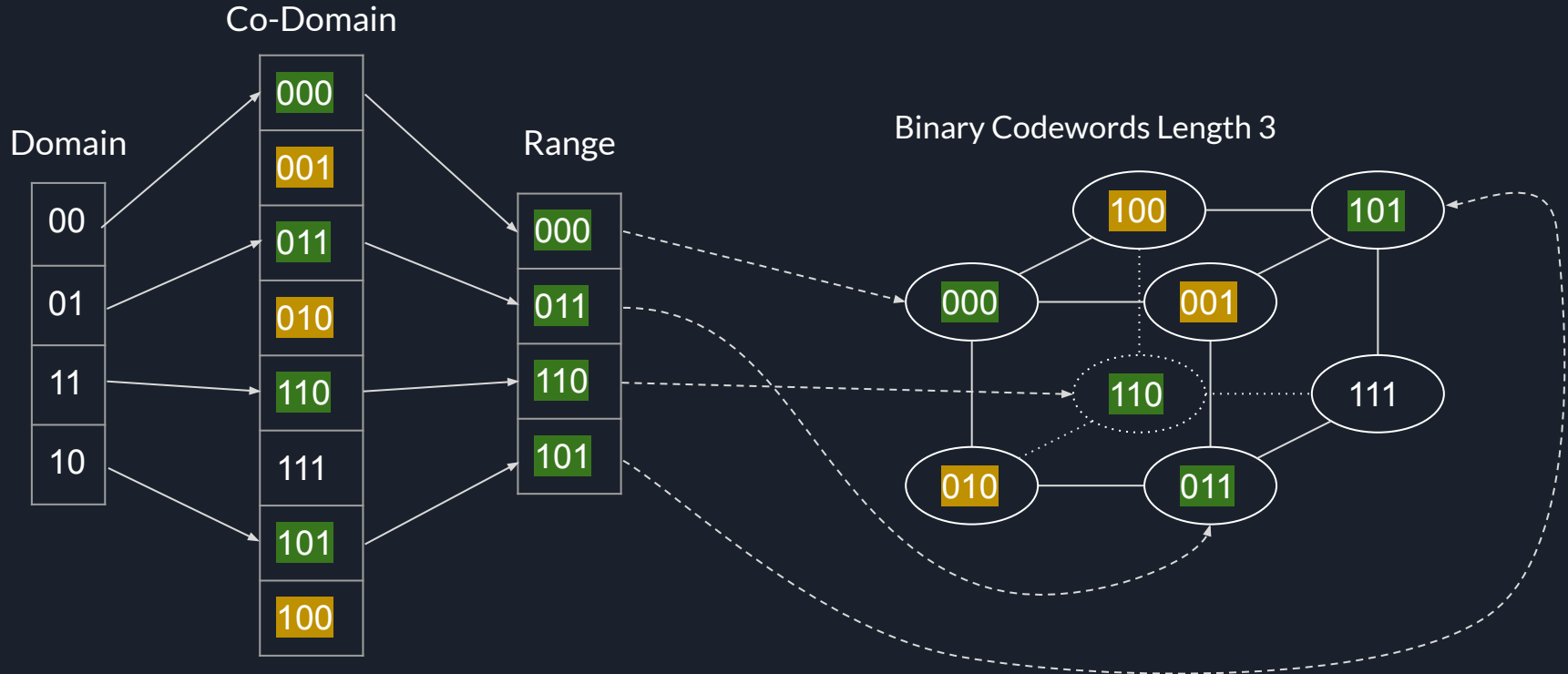
Binary Codewords Length 2




Binary Codewords Length 3



Simple Parity Coding - Distance Visualized

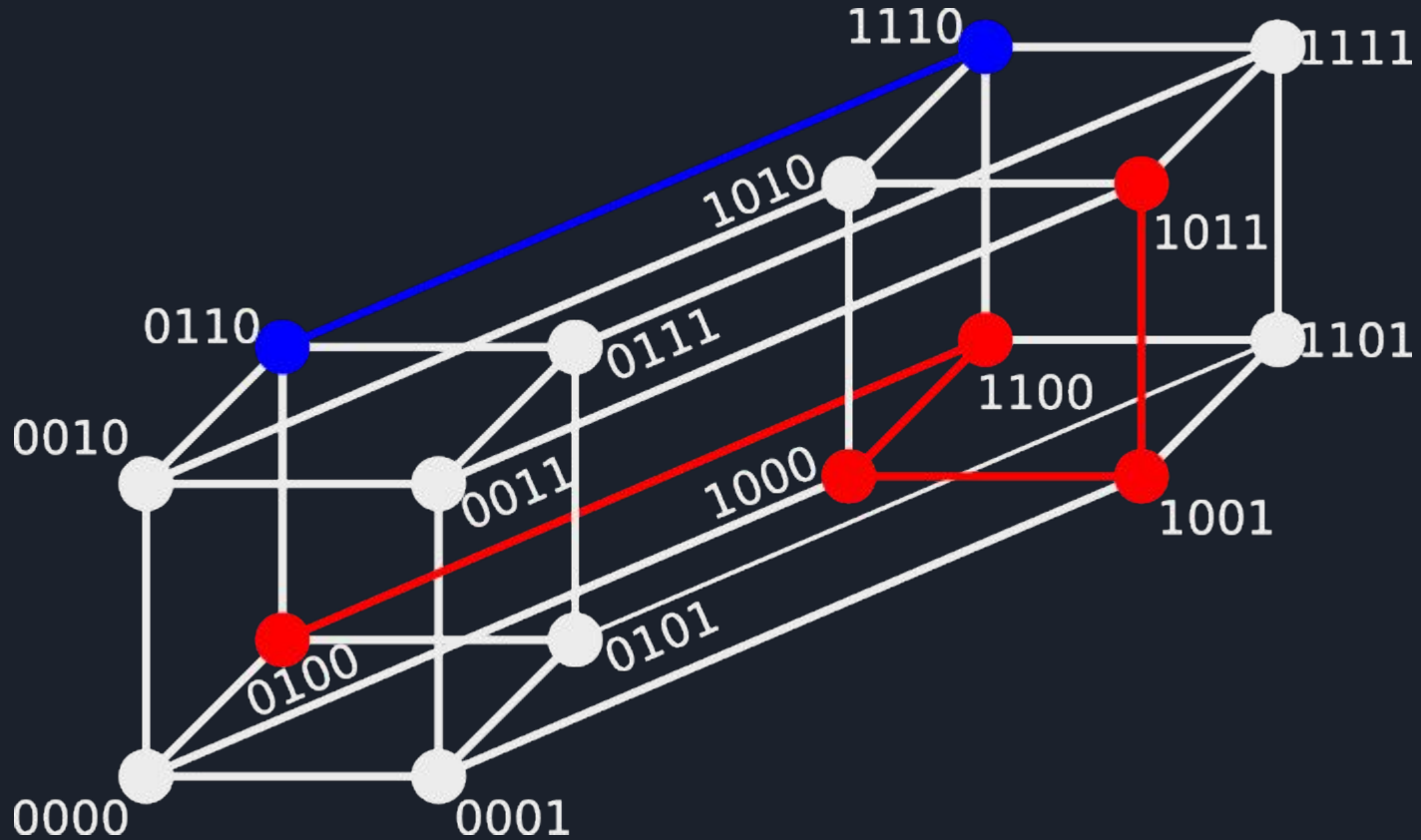




Hamming Distance Visualized - Binary Codes

- In general, binary codewords of length N form a hypercube of N dimensions.
- All codeword graphs have q^K vertices (all possible codewords) each with $(q-1) * k$ edges (all possible single substitutions).

Hamming Distance Visualized - Paths

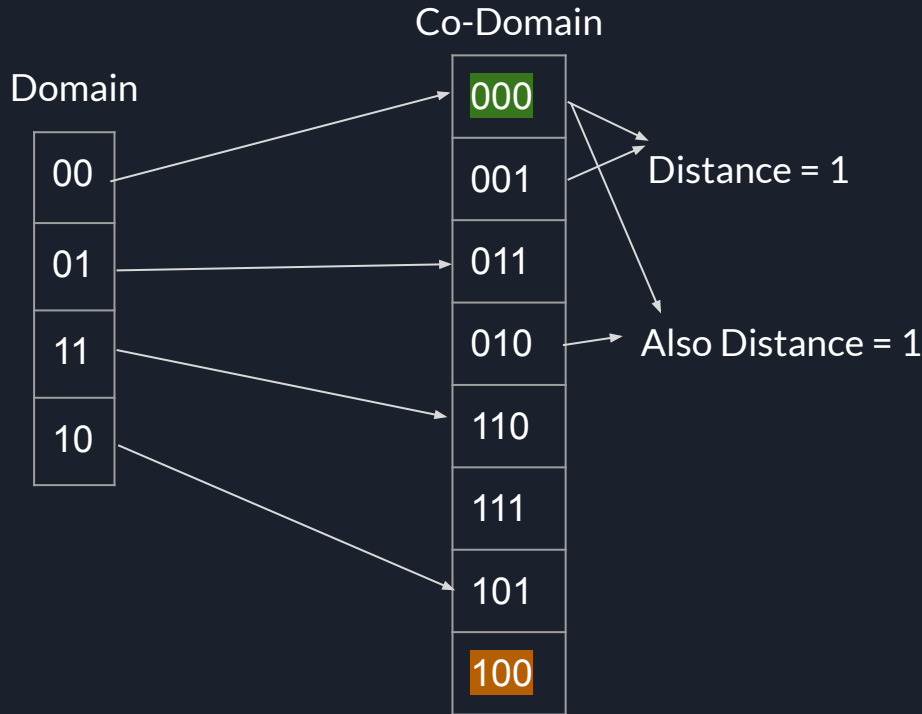




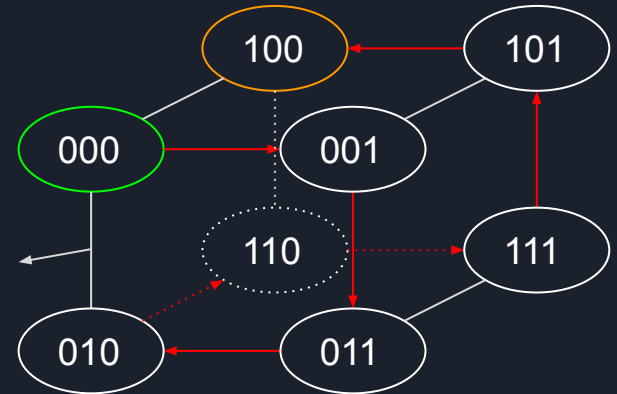
Gray Codes

- An ordering of binary numbers such that any 2 numbers differ by only one symbol.
- Equivalent to a Hamiltonian path through the codeword graph.

Simple Parity Coding - Hamiltonian



Binary Codewords Length 3

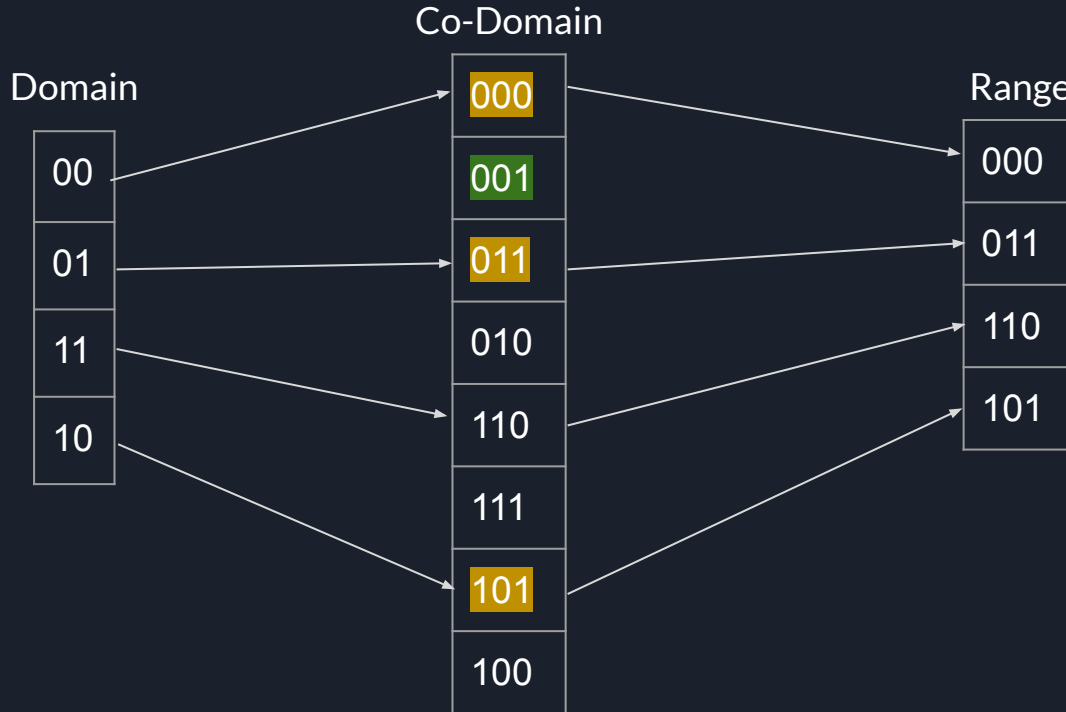




Detecting Errors

We can *detect* an error if we receive a codeword that is not in the range.

Simple Parity Coding - Error Detection



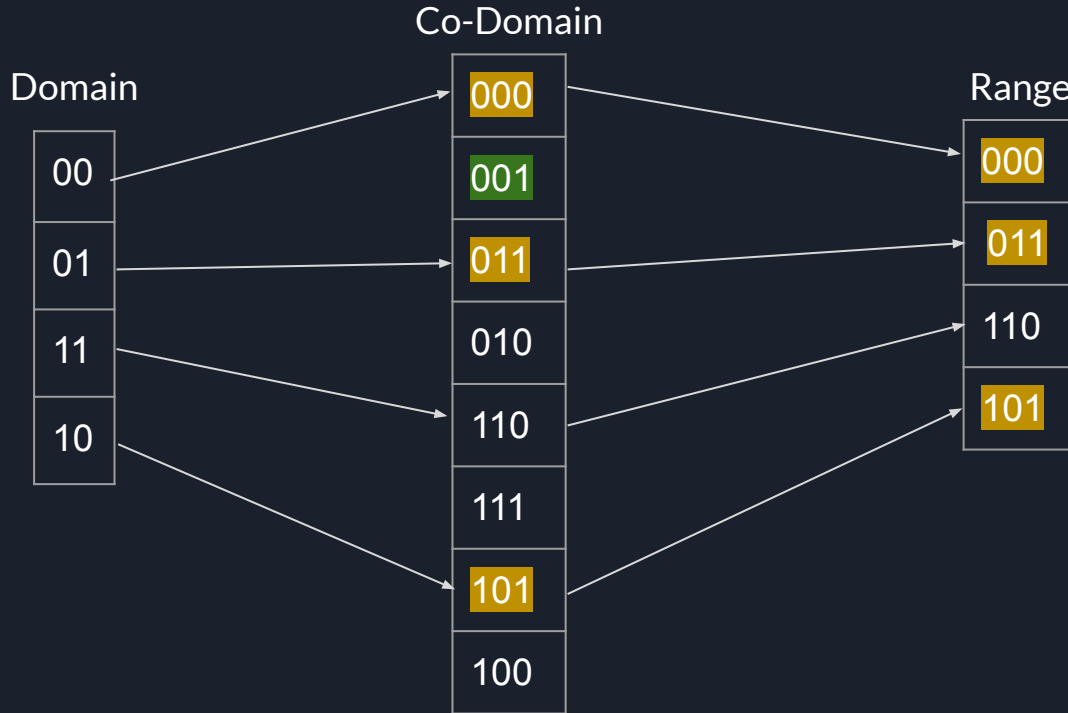
Received Codeword
001



Correcting Errors

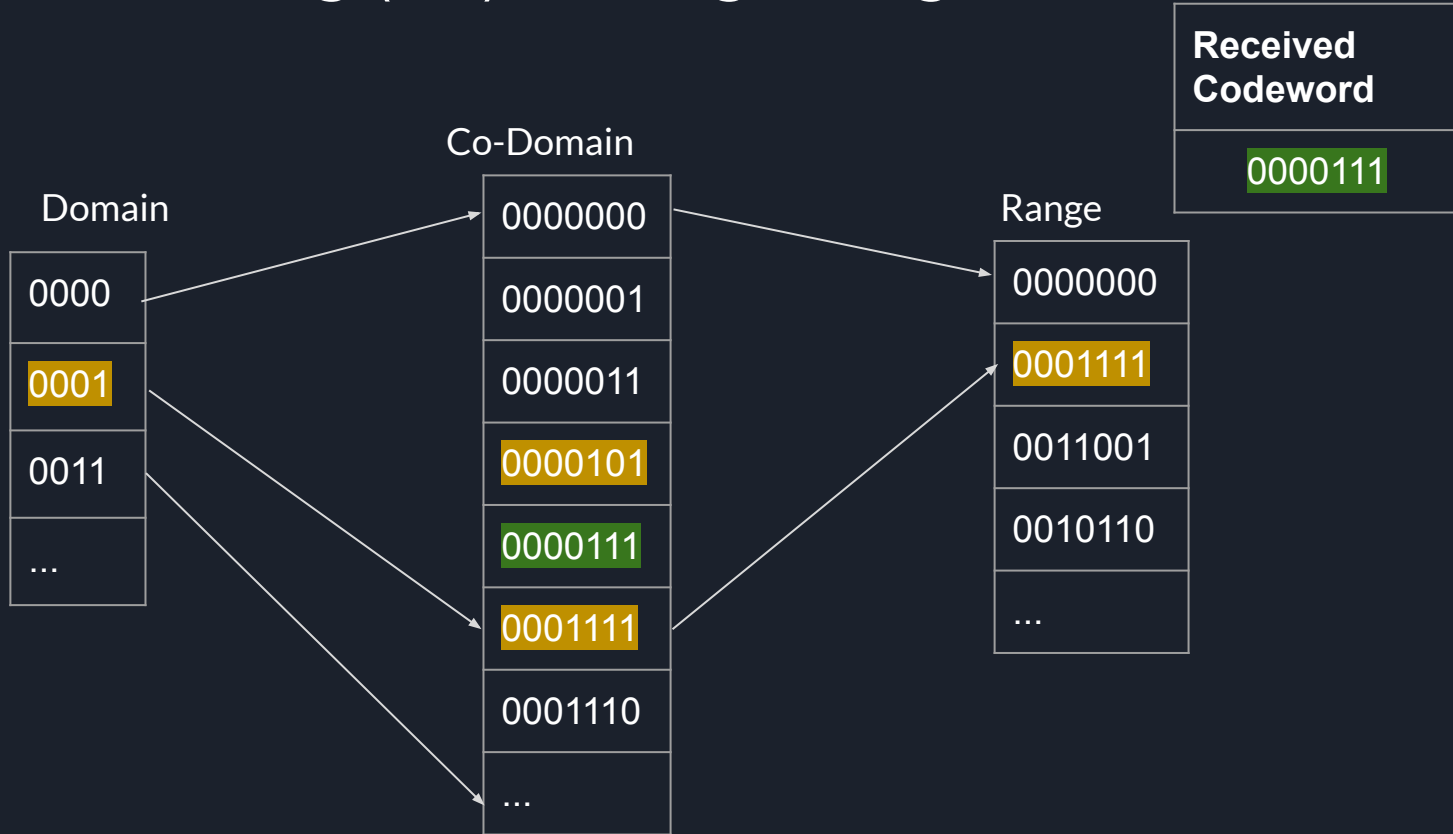
We can *correct* a single error if there is a single closest codeword (by hamming distance) to what we received in the range.

Simple Parity Coding - Error

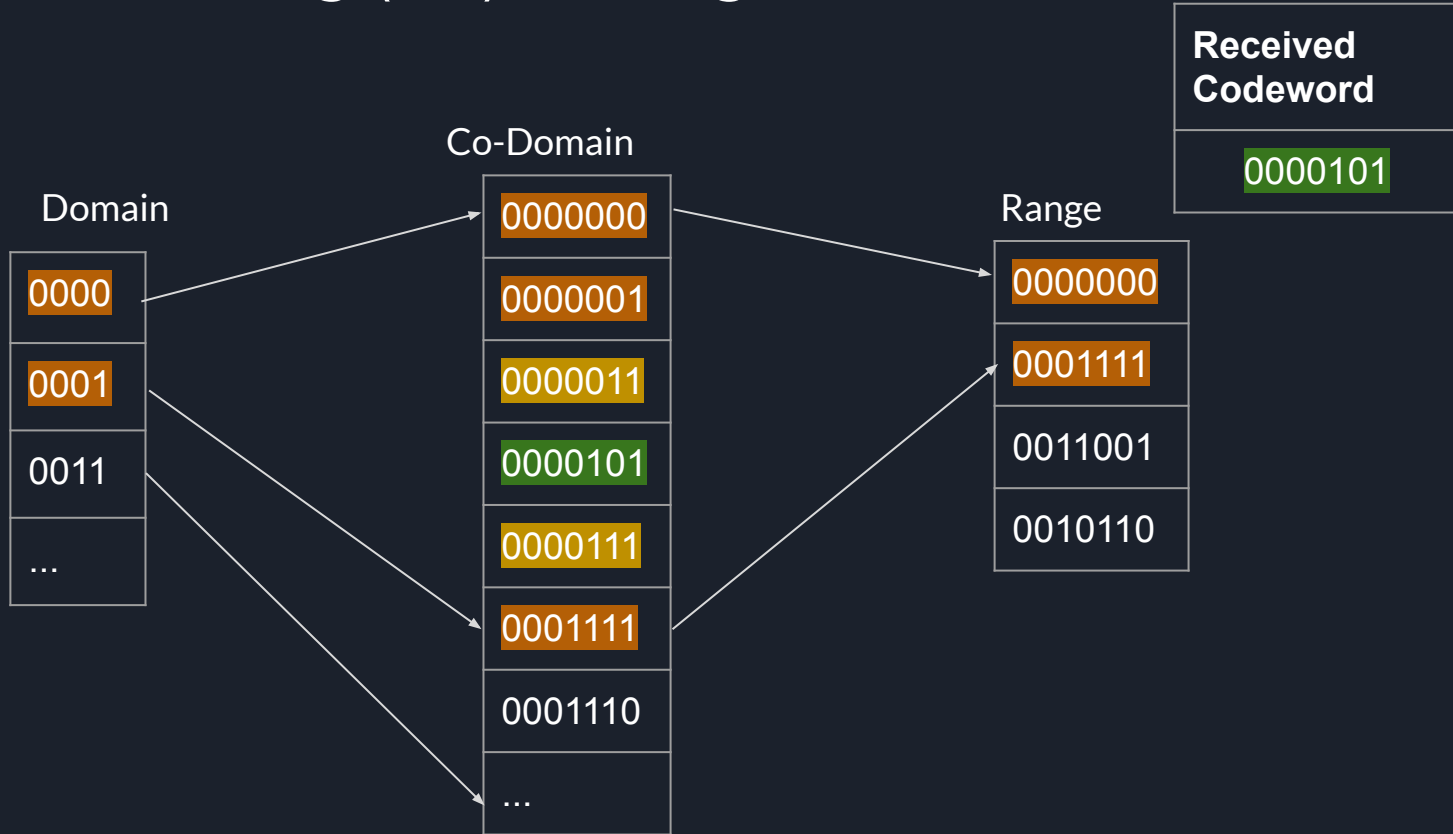


Received Codeword
001

Hamming (7,4) Coding - Single Error



Hamming (7,4) Coding - Double Error



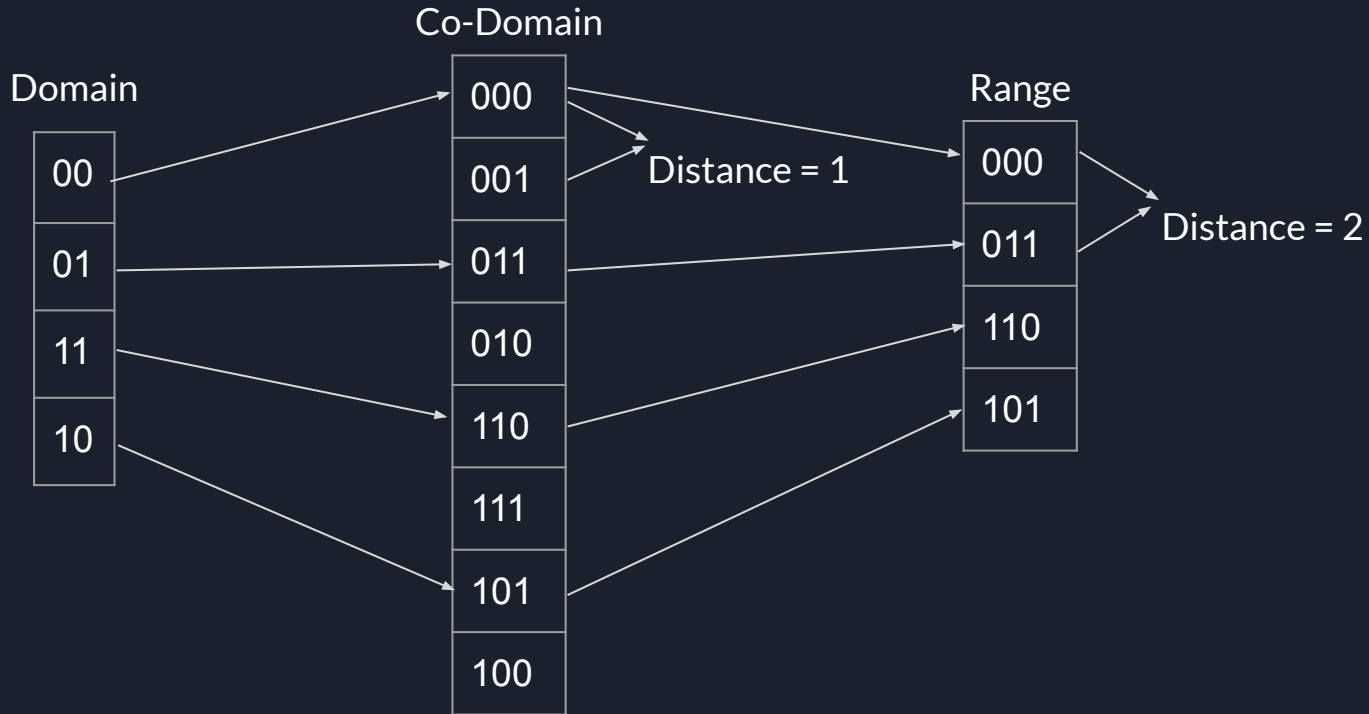


Hamming Distance, Errors, and Erasures

Let's say that "d" is the minimum Hamming distance between any two possible strings for a set of strings C. That is,

$$d = \min(\text{Distance}(a, b) \mid \forall a, b \mid a \neq b, a \in C, b \in C)$$

Simple Parity Coding - Distance





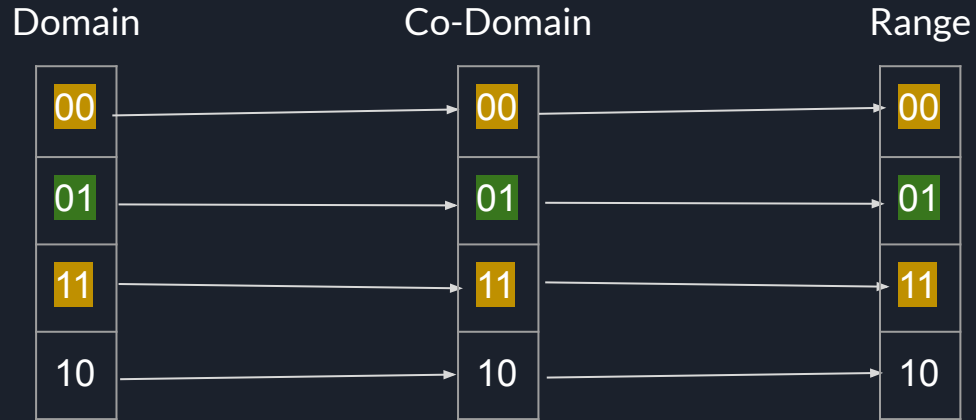
Codewords and Hamming Distance

Why do we care about minimum Hamming distance, d ?

When d for a set is (such as the range):

- $d = 1$, A single error is *undetectable*, a single erasure is *uncorrectable*

No Change Coding - $d=1$



Received
Codeword

01



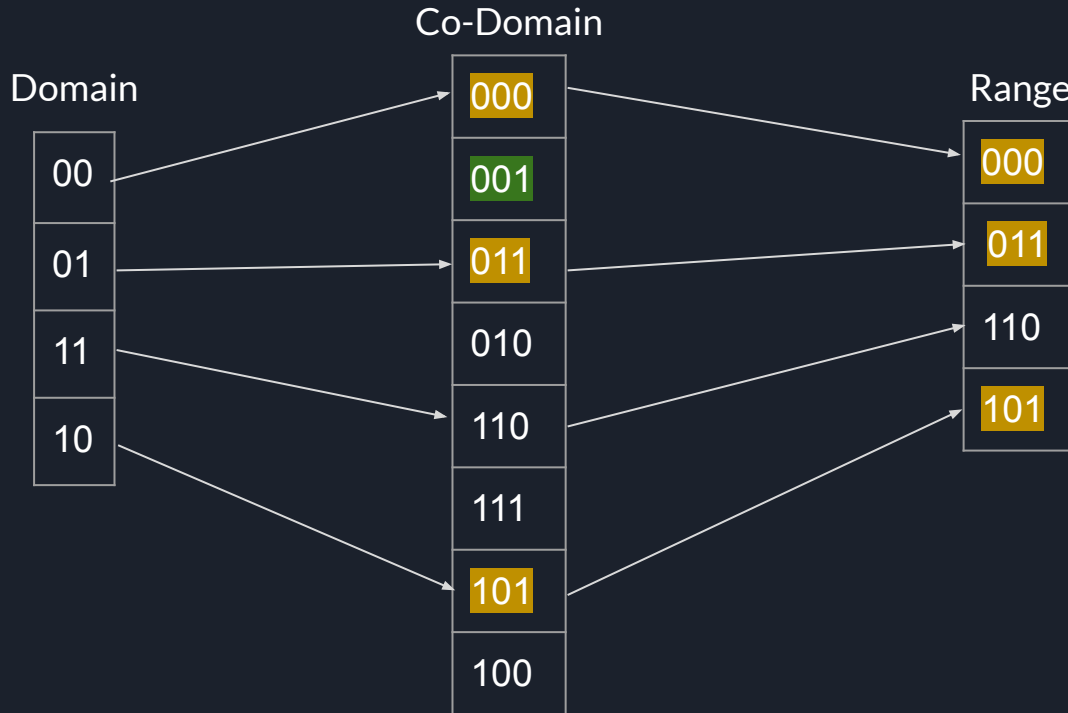
Codewords and Hamming Distance

Why do we care about minimum Hamming distance, d ?

When d for a set is (such as the range):

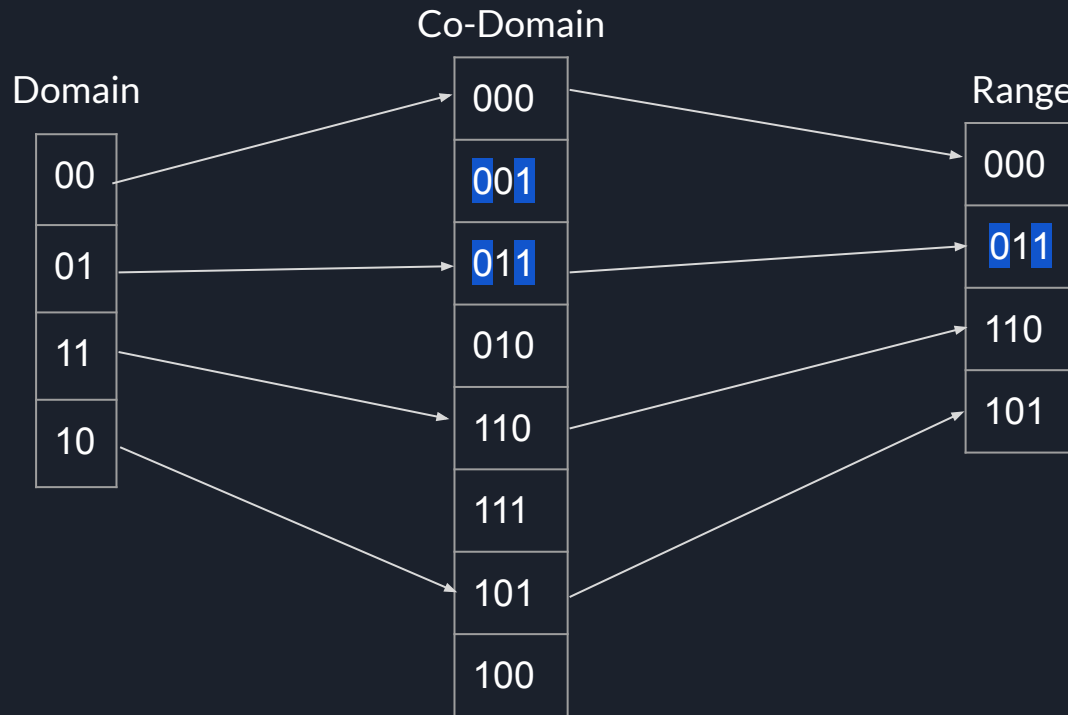
- $d = 1$, A single error is *undetectable*, a single erasure is *uncorrectable*
- $d = 2$, A single error is *detectable*, but not *correctable*. A single erasure is *correctable*

Simple Parity Coding - d=2 - Error



Received Codeword
001

Simple Parity Coding - d=2 - Erasure



Received
Codeword

0_1



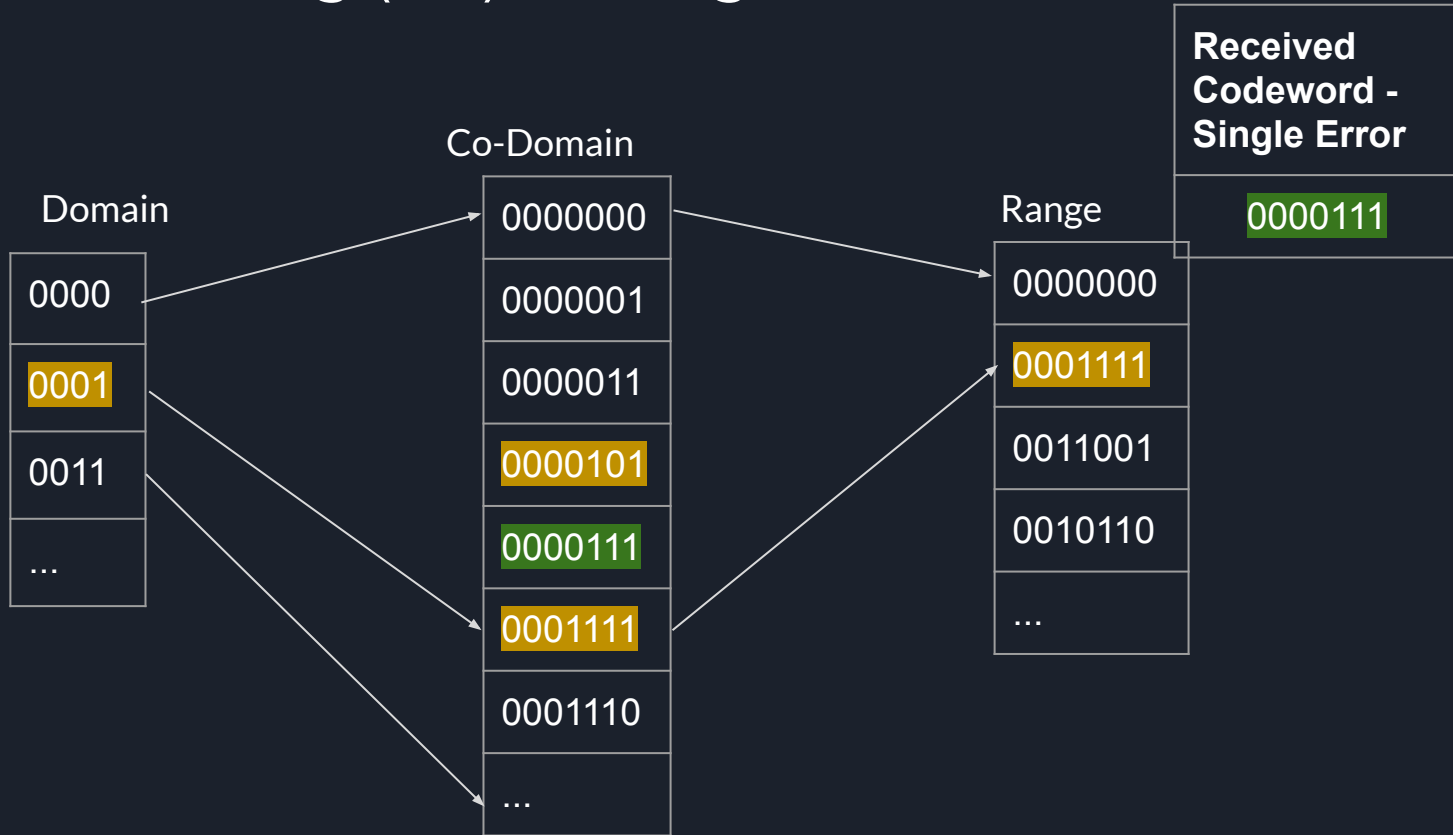
Codewords and Hamming Distance

Why do we care about minimum Hamming distance, d ?

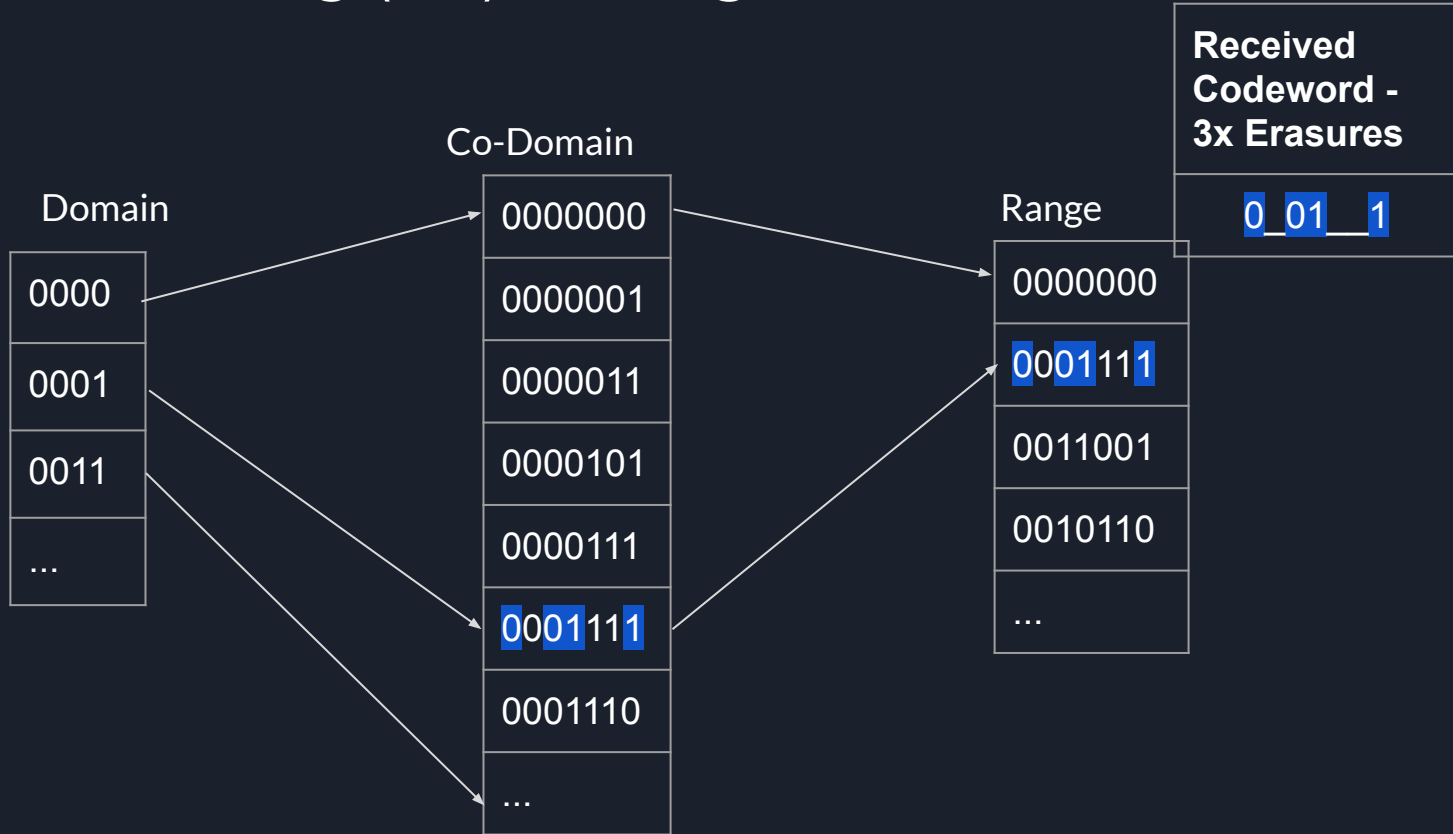
When d for a set is (such as the range):

- $d = 1$, A single error is *undetectable*, a single erasure is *uncorrectable*
- $d = 2$, A single error is *detectable*, but not *correctable*. A single erasure is *correctable*
- $d = 3$, Two errors are *detectable*, one is *correctable*. Two erasures are *correctable*
- $d = 4$, Three errors are *detectable*, one error is *correctable*. Three erasures are *correctable*

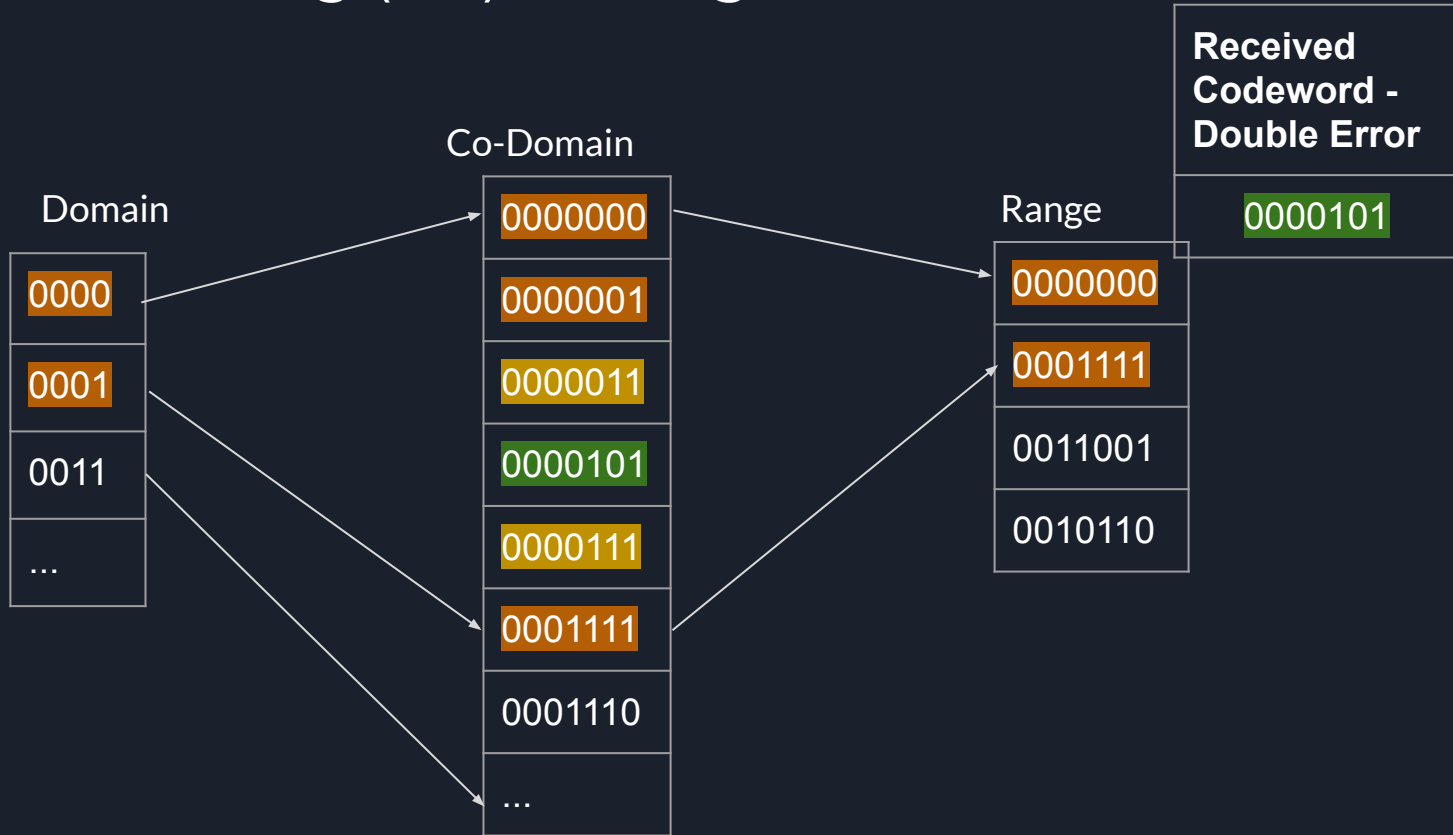
Hamming (7,4) Coding - d=4



Hamming (7,4) Coding - d=4



Hamming (7,4) Coding - $d=4$





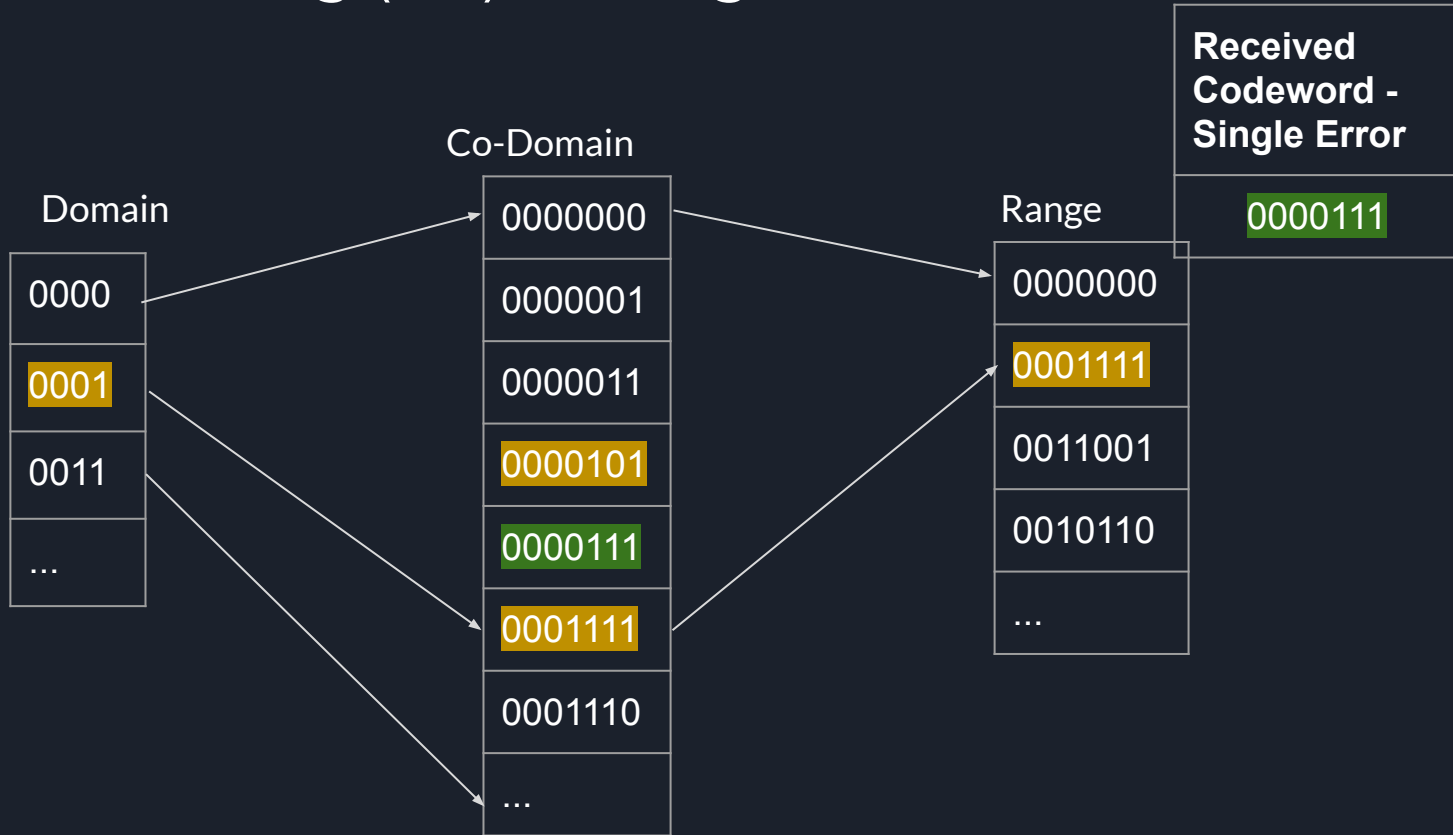
Codewords and Hamming Distance

Why do we care about minimum Hamming distance, d ?

When d for a set is (such as the range):

- $d = 1$, A single error is *undetectable*, a single erasure is *uncorrectable*
- $d = 2$, A single error is *detectable*, but not *correctable*. A single erasure is *correctable*
- $d = 3$, Two errors are *detectable*, one is *correctable*. Two erasures are *correctable*
- $d = 4$, Three errors are *detectable*, one error is *correctable*. Three erasures are *correctable*
- $d = 5$, Four errors are *detectable*, two errors are *correctable*. Four erasures are *correctable*
- ...
- In general:
 - $d-1$ errors are *detectable*
 - $d-1$ erasures are *correctable*
 - $\lfloor (d-1)/2 \rfloor$ errors are *correctable*

Hamming (7,4) Coding - $d=4$





Optimal Codes

Great! So we know we can use the minimal Hamming distance between codewords in the range to predict how many errors/erasures we can detect and correct.

But in general, do codes exist where the minimal distance between codewords in the range is maximized? That is, are there optimal codes?



Singleton Bound

All coding schemes follow the Singleton bound!

Singleton Bound: $k + d \leq n + 1$ equivalently: $d \leq n - k + 1$ equivalently: $n \geq k + d - 1$

k = message length

d = minimum Hamming distance between any 2 codewords

n = codeword length

Implications (in English):

- Every codeword differs in at least one position.
- To increase the distance between codewords by 1, the codeword length needs to be increased by *at least* 1.



MDS Codes

"Maximum Distance Separable" codes. Codes that map input messages as far apart as possible.

Codes that improve upon the Singleton bound by changing the inequality to an equality.

MDS Bound: $k + d = n + 1$

Implications in English:

- Every additional symbol in the codeword increases the minimum distance by 1.
- One cannot generate a better code than an MDS code (without violating the singleton bound).
- Any code that is an MDS code is optimal in terms of codeword Hamming distance.



Trivial MDS Codes

Do Nothing Code: $\Sigma^K \rightarrow \Sigma^K$ (aka $\lambda x: x$)

In this case, $k = n$ and because every different message must differ in at least one place
 $k + d = n + 1$ (because $d = 1$ and $k = n$)

No Information Code: $()^1 \rightarrow ()^1$ (aka a code that has a single message/codeword)

Same as above, but now $k = n = 1$.



Trivial MDS Codes - Continued

Single Parity Symbol: $\Sigma^K \rightarrow \Sigma^{K+1}$

- If $q = 2$, this is a parity bit (remember that $q = |\Sigma|$) which you can get by XORing all the message bits together (because $q = 2$, bits can represent the symbols).
- If $q > 2$ and q is prime, then you can sum all the symbols together modulo q .

We have $d = 2$ and $n = k + 1$, so $k + 2 = (k + 1) + 1$

Single Symbol Replication: $\Sigma^1 \rightarrow \Sigma^R$

A code that just replicates a single symbol message R times. $n = r = d$ since every different message is now different in r places and also r times as long. $k = 1$ by definition

$k + r = r + 1$ (or: $1 + r = r + 1$)



3x Redundancy - Revisited: Is it MDS?

MDS Bound: $k + d = n + 1$

3x Redundancy is a coding scheme where $\Sigma^K \rightarrow \Sigma^{3K}$, so let's plug in the numbers!

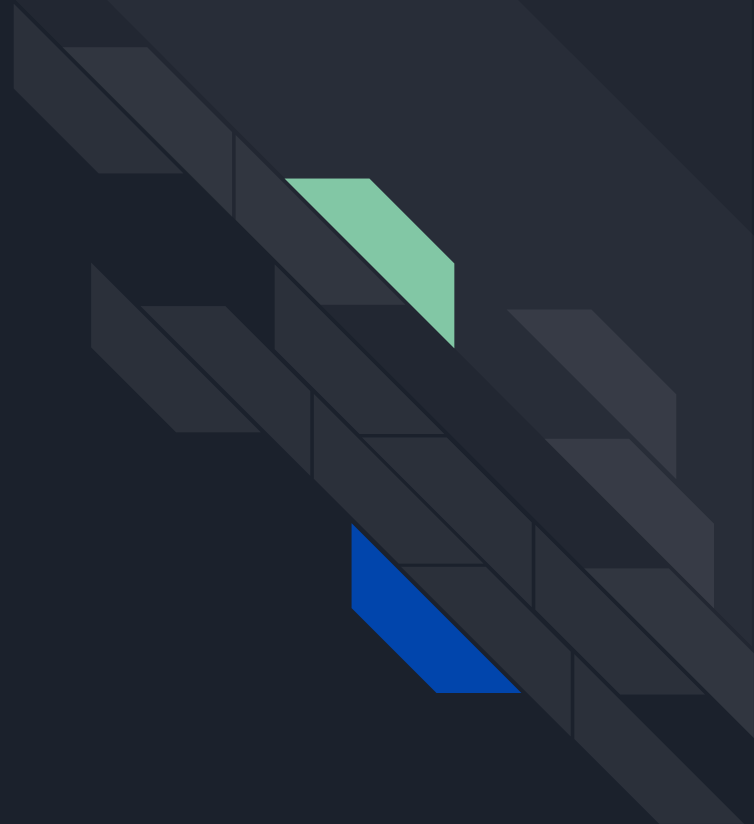
$k = k, n = 3k$

What about d ? $d = 3$ because without redundancy, $d = 1$, now we've repeated every symbol 3 times, so the minimum distance is also 3x.

Is it optimal? Does $k + 3 = 3k + 1$ (for all k)? NO!

3x redundancy *can* in some cases detect/correct more errors, but in general it can only guarantee to detect 3 and correct 1 (aka $\text{floor}(d-1/2)$)!

Part 2: Reed Solomon Codes

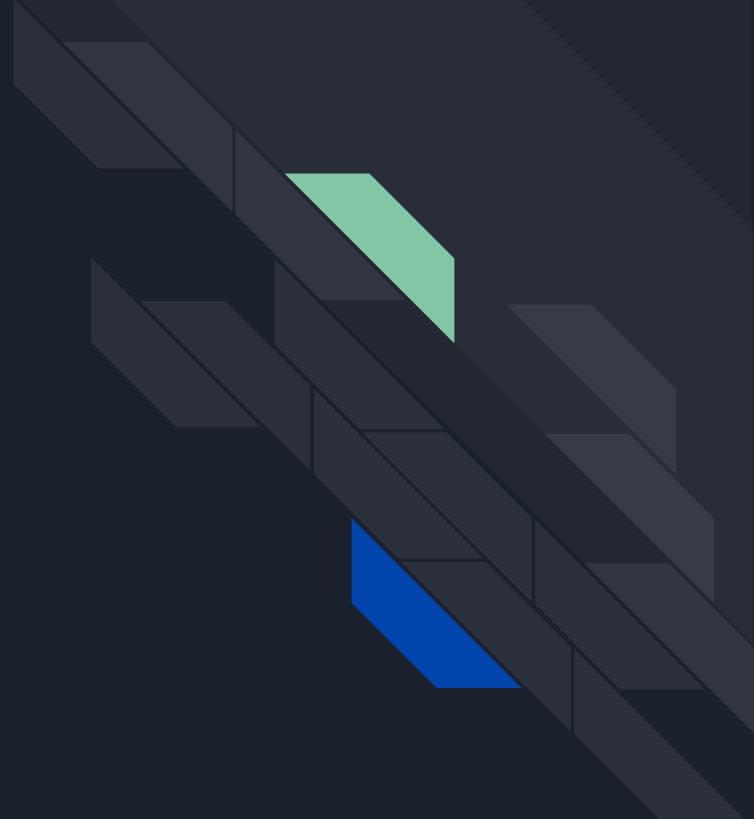




What is it?

[Wikipedia](#): "Reed–Solomon codes are a group of error-correcting codes that were introduced by Irving S. Reed and Gustave Solomon in 1960. They have many applications, the most prominent of which include consumer technologies such as CDs, DVDs, Blu-ray discs, QR codes, data transmission technologies such as DSL and WiMAX, broadcast systems such as satellite communications, DVB and ATSC, and storage systems such as RAID 6."

Polynomials





Polynomials - Definition

An expression consisting of variables and coefficients using only the operations of addition, subtraction, multiplication, division, and non-negative integer exponents on the variables.

We'll only concern ourselves with polynomials over a single variable.

Notationally, we'll say a polynomial is a function over x with the form:

$$f(x) = a_0x^0 + a_1x^1 + a_2x^2 + \dots + a_nx^n \quad \text{more simply:} \quad f(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

Polynomials represent everything that can be done with just addition and multiplication.

Polynomial - Representations

Polynomials can be represented in many different ways:

- As a function:
 - $f(x) = 0$
 - $f(x) = -7$
 - $f(x) = 3 + 2x - 8x^2$
 - $f(x) = 9x^4$
 - $f(x) = (2 - x)(x^2 + 1) = 2 - x + 2x^2 - x^3$
- A set of points:
 - $\{(0, -4), (2, 0), (4, 12)\} = f(x) = x^2 - 4 = -4 + x^2$
- A graph
- A list of coefficients:
 - $[0, -1, 0, 5] \Rightarrow 5x^3 - x$



$$\{(-4, 0), (-1, 0), (2, 0)\}$$
$$f(x) = (x+4)(x+1)(x-2)$$
$$f(x) = x^3 + 3x^2 - 6x - 8$$
$$f(x) = x(x(x + 3) - 6) - 8$$



Polynomials - From Points

It's clear how the functions are polynomials (by notation).

It's clear how to turn the list of coefficients into a function.

What about the points (or the graph)?

- Can we *always* construct a polynomial from a set of points?



Polynomials - From Points

What if you have the empty set? $\{\}$

- Map it to the "zero" polynomial, i.e. 0.

What if you have a set of just one point? $\{(x_0, y_0)\}$

- Just make the function always return y_0 !
- $f(x) = y_0$



Polynomials - From Points

What if you have a set of 2 points? $\{(x_0, y_0), (x_1, y_1)\}$

- $f(x_0) = y_0$
- $f(x_1) = y_1$

But what's $f(x)$? How can we combine the two equations?



Polynomials - From Points

Just add them?

$$f(x) = y_0 + y_1$$

Does $f(x_0) = y_0 + y_1$? No...



Polynomials - From Points

Ah, use a conditional!

$$f(x) = \text{if } (x == x_0) \text{ then } y_0 \text{ else } y_1$$

But how can we use just addition and multiplication without "if"s?



Polynomials - From Points

Can we at least see if x is x_0 ?

$x - x_0 = 0$ only if x is x_0

And multiplication by 0 is 0...so we can "switch off" a term...



Polynomials - From Points

Can we do it by "switching off" terms?

If we want to turn off y_1 when $x = x_0$, then...

$$f(x) = y_0 + (x-x_0)y_1$$

If we want to turn off y_0 when $x = x_1$ too, then...

$$f(x) = (x-x_1)y_0 + (x-x_0)y_1$$

Does this work now?

$$f(x_0) = (x_0-x_1)y_0 + 0y_1 = (x_0-x_1)y_0 \neq y_0 \quad \dots\text{No, that's a multiple of } y_0, \text{ but not } y_0 \text{ itself}$$



Polynomials - From Points

Can we "normalize" the "switched on" term, i.e. make the "switch" become 1?

- Anything divided by itself is 1
- $10/10 = 1$

The y_0 switch is $(x-x_1)$, and it's only "on" when $x = x_0$, so to normalize it when it's on we make it...

$$\frac{x - x_1}{x_0 - x_1}$$

Altogether:

$$f(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0}$$



Polynomials - From Points

Did we do it?

$$f(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0}$$

Let's try it...

$$f(x_0) = y_0 \frac{x_0 - x_1}{x_0 - x_1} + y_1 \frac{x_0 - x_0}{x_1 - x_0} = y_0 * 1 + y_1 * 0 = y_0$$

$$f(x_1) = y_0 \frac{x_1 - x_1}{x_0 - x_1} + y_1 \frac{x_1 - x_0}{x_1 - x_0} = y_0 * 0 + y_1 * 1 = y_1$$



Polynomials - From Points

What we achieved was like "boolean" logic, but with mathematical operators.

In bash-like notation we might have said:

$$f(x) = x \neq x_1 \ \&\& \ y_0 \ || \ x \neq x_0 \ \&\& \ y_1$$

- We do inequality testing with "-" instead of "!=" which gives us 0 or not 0.
 - We then normalize this to 0 and 1
 - Thus 0 and 1 become our "booleans"
- Multiplication becomes our &&: $x \ \&\& \ \text{false} = \text{false}$, similarly $x * 0 = 0$
- Addition becomes our ||: $x \ || \ \text{false} = x$, similarly $x + 0 = x$
- There's more to this intuition in later slides...



Polynomials - From Points

Whew! But now what if you have a set of 3 points? $\{(x_0, y_0), (x_1, y_1), (x_2, y_2)\}$

- $f(x_0) = y_0$
- $f(x_1) = y_1$
- $f(x_2) = y_2$

Can we extend the switch?

- We want y_0 when x is not x_1 and not x_2
- $(x - x_1)$ switches off (becomes 0) when x is x_1 , it's similar for $(x - x_2)$
- Let's just "and" them together, i.e. multiply them
- $(x - x_1)(x - x_2)$ is 0 when the value is x_1 or x_2 but it's not zero when x_0
- Then let's normalize when $x=x_0$ same as before by dividing by $(x_0 - x_1)(x_0 - x_2)$



Polynomials - From Points

Whew! But now what if you have a set of 3 points? $\{(x_0, y_0), (x_1, y_1), (x_2, y_2)\}$

- $f(x_0) = y_0$
- $f(x_1) = y_1$
- $f(x_2) = y_2$

What are we left with?

$$f(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$



Polynomials - From Points

I'm not done yet! What if you have a set of n points? $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$

Can we generalize the switch?

For point (x_i, y_i) we want to "turn off" for any other point:

$$\prod_{j \neq i} (x - x_j)$$

Normalized:

$$\prod_{j \neq i} \frac{(x - x_j)}{(x_i - x_j)}$$




Polynomials - From Points

I'm not done yet! What if you have a set of n points? $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$

Can we put it all together?

$$\sum_{i=0}^n y_i \cdot \prod_{j \neq i} \frac{(x - x_j)}{(x_i - x_j)}$$




Polynomials - From Points

It's clear how the functions are polynomials.

It's clear how to turn the list of coefficients into a function.

What about the points (or the graph)?

- Can we *always* construct a polynomial from a set of points?
 - Yes



Polynomials - From Points

It's clear how the functions are polynomials.

It's clear how to turn the list of coefficients into a function.

What about the points (or the graph)?

- Can we *always* construct a polynomial from a set of points?
 - Yes
- Is the polynomial we construct unique?



Polynomials - Degrees

First, an aside about degrees of polynomials

The degree is the value of the largest exponent in the polynomial.

- $\text{degree}(x^3+2) = 3$
- $\text{degree}(5) = 0$
- $\text{degree}(0) = -1$ (by convention)
- $\text{degree}(a_n x^n + \dots + a_0 x^0) = n$



Polynomials - Degrees

But what is the degree of a polynomial generated from a set of points?

It's a tricky question. We could imagine that any number of points were all selected from a line, in which case there is a degree 1 polynomial that matches.

However, we can't *know* that. So what about the polynomial algorithm from before?

$$\text{degree}(\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}) == n-1$$

Our equation took n points and for each y_i multiplied $n-1$ terms, each containing x , together (excluding the one we didn't want to "switch off"). Then it summed the terms (which cannot change the degree). The degree must thus be $n-1$.



Polynomials - From Points

It's clear how the functions are polynomials.

It's clear how to turn the list of coefficients into a function.

What about the points (or the graph)?

- Can we *always* construct a polynomial from a set of points?
 - Yes
- Is the polynomial we construct *from n points* unique *among polynomials of degree $n-1$* ?



Polynomials - From Points - Uniqueness

A root of a polynomial is an input such that the output is 0. $f(x) = 0$ means x is a root.

Theorem: The zero polynomial is the only polynomial over \mathbb{R} (the set of real numbers) of degree (at most) n which has more than n distinct roots.

- Alternatively: Every non-zero degree n polynomial has, counted with multiplicity, exactly n complex roots.
- This is the "Fundamental Theorem of Algebra"

Uniqueness Proof: Suppose we can form 2 polynomials from a set of n points, g and f . The degree of both would be (at most) $n-1$. We can form a new polynomial $(g-f)(x) = g(x) - f(x)$. This polynomial is 0 in n points (all input points since they're the same), but its degree is $n-1$ and by the theorem it must thus be the zero polynomial. Thus g and f are the same.



Polynomials - From Points

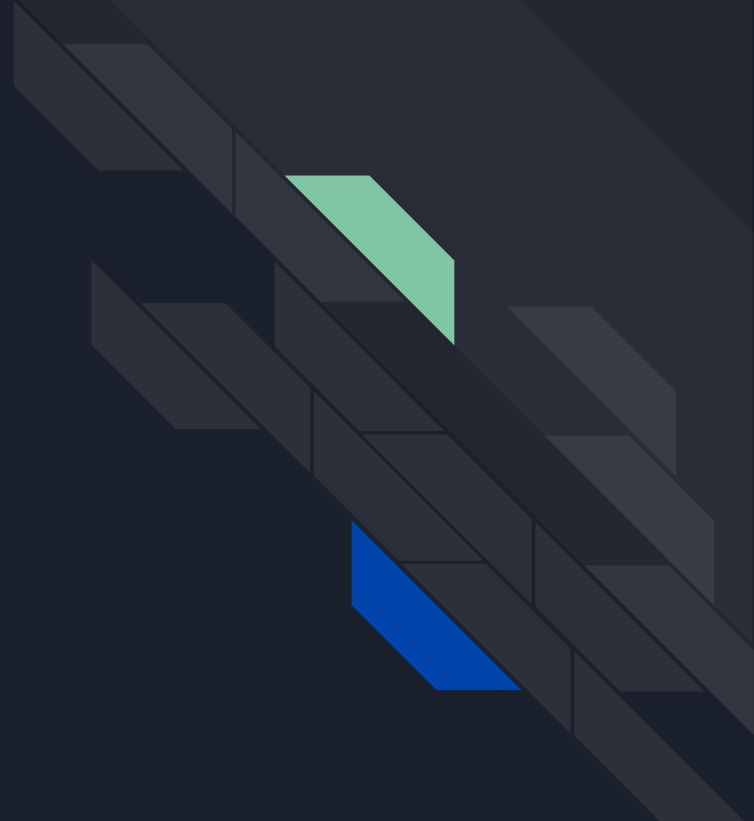
It's clear how the functions are polynomials.

It's clear how to turn the list of coefficients into a function.

What about the points (or the graph)?

- Can we *always* construct a polynomial from a set of points?
 - Yes
- Is the polynomial we construct *from n points* unique *among polynomials of degree $n-1$* ?
 - Yes

Reed Solomon





Reed Solomon - What we know

- We know we can take N points and form a unique polynomial of degree $N-1$
- Thus if we have any N points which came from a polynomial (of degree $N-1$) we can (re-)generate the polynomial with them.
 - If we have 2 different sets of N points from the (degree $n-1$) polynomial they'd generate the same polynomial since a set of N points generates a *unique* polynomial,.
- If we took a few more points along the polynomial we'd have some redundancy.
 - Specifically, if we take M additional points (so we have $N+M$) then we can lose any M points and still have N left to re-generate the polynomial.



Reed Solomon - An Algorithm

An algorithm for RS:

- Take N symbols encoded as integers: $[y_0, y_1, y_2, \dots, y_{n-1}]$
- Turn them into points by using their index as the x value:
 $\{(0, y_0), (1, y_1), (2, y_2), \dots, (n-1, y_{n-1})\}$ i.e.
- Generate a polynomial (of degree $n-1$) with the N points.
- Generate a few more points along the polynomial for redundancy.
 - e.g. find $[f(n), f(n+1), f(n+2), \dots, f(n+m)]$ where m is the number of additional points.
- If you lose a few points, reconstruct the polynomial with any N remaining points.
 - This requires you to remember the 'x' value for the points.
 - This could be labels for your HDDs/SSDs.



Reed Solomon - Devil's in the Details

Some problems with our algorithm:

- All that polynomial stuff is really complicated and thus slow.
- It uses polynomials over \mathbb{R} , but computers don't like real numbers and floats are no substitute.
- It's not clear how to turn our symbols into integers in general. What if the new points we generate do not map back?

We'll look next how to address these problems...



Improving on Lagrangian Interpolation

We know all polynomials can be represented in the form $a_0x^0 + a_1x^1 + \dots + a_nx^n$

What if we represent our set of points like this:

$$y_0 = a_0x_0^0 + a_1x_0^1 + \dots + a_{n-1}x_0^{n-1}$$

$$y_1 = a_0x_1^0 + a_1x_1^1 + \dots + a_{n-1}x_1^{n-1}$$

...

$$y_{n-1} = a_0x_{n-1}^0 + a_1x_{n-1}^1 + \dots + a_{n-1}x_{n-1}^{n-1}$$

Now we have a system of N equations, we have N unknowns (the coefficients, we know the x and y values of the points)...can we use matrices?

Improving on Lagrangian Interpolation

First, let's separate out the coefficients from the x points and rewrite it using matrix notation:

$$\begin{array}{l} y_0 = a_0x_0^0 + a_1x_0^1 + \dots + a_nx_0^n \\ y_1 = a_0x_1^0 + a_1x_1^1 + \dots + a_nx_1^n \\ \dots \\ y_n = a_0x_n^0 + a_1x_n^1 + \dots + a_nx_n^n \end{array} \rightarrow \begin{bmatrix} x_0^0 & x_0^1 & \dots & x_0^n \\ x_1^0 & x_1^1 & \dots & x_1^n \\ x_2^0 & x_2^1 & \dots & x_2^n \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$$

N.B. It should be "n-1" everywhere we have "n". My bad.



Improving on Lagrangian Interpolation

We can write the matrix equation more generally as:

$$XA = Y$$

And in theory we can solve for A like so:

$$X^{-1}XA = X^{-1}Y$$

$$\rightarrow IA = X^{-1}Y$$

Note: "I" is the identity matrix. It's like "1" for matrix multiplication.

$$\rightarrow A = X^{-1}Y$$

BUT, can we guarantee that X is always invertible?

Vandermonde Matrices

Vandermonde matrices are those of the form

$$\begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^n \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^n \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ x_n^0 & x_n^1 & x_n^2 & \cdots & x_n^n \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}$$

Just like our X matrix from before.



Vandermonde Matrices

Vandermonde matrices have a key property. The determinant is:

$$\det(V) = \prod_{1 \leq i < j \leq n} (x_j - x_i)$$

As long as every x_i value is distinct, then no term will be 0 and thus the determinant will not be 0.

Non-zero determinant means that a matrix is invertible! Because our X matrix from before was a vandermonde matrix, and because we're using N *distinct* points (i.e. their x values must all be different), it *must* be invertible.



Vandermonde Matrices

Amended RS encoding algorithm - generating the polynomial:

$$A = X^{-1}Y$$

- If we want to encode a message of N symbols, generate an $N \times N$ Vandermonde matrix.
 - The x values can just be $[0, N)$, so we only need to build the matrix once.
- Invert the matrix. This can be done with Gaussian Elimination (see wikipedia).
 - Again, because X is constant, we only need to invert once.
- To generate a polynomial from N symbols, multiply the inverted Vandermonde matrix by the N symbols to generate the polynomial's coefficients.
 - The inverted matrix is fixed, so for a different set of N symbols, you just need to do the matrix multiplication.



Vandermonde Matrices

Amended RS encoding algorithm - generating the parity symbols:

$$Y' = X'X^{-1}Y$$

- Make a larger Vandermonde matrix M^*N (extra rows for the extra symbols to generate).
 - $M = N + P$ where P is the number of parity symbols to add.
 - Let's call it X'
- If we multiply X' by A we get Y' , that is, our original symbols plus the new parity symbols.
- We know that $A = X^{-1}Y$ and $Y' = X'A$. Thus $Y' = X'X^{-1}Y$
 - X' is constant and X^{-1} is constant and if we pre-multiply them the result is constant.
- We can form an encoding matrix $E = X'X^{-1}$, then we generate all our points with the equation: $Y' = EY$.
 - Multiplying X' by X^{-1} will cause the top (square) part of X' to be the identity matrix!
 - That means we do not modify the input data, we just generate new parity symbols!

Vandermonde Matrices

Amended RS encoding algorithm - generating the parity symbols:

$$\begin{array}{|c|c|c|c|} \hline 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \\ \hline X_0 & X_1 & X_2 & X_3 \\ \hline X_4 & X_5 & X_6 & X_7 \\ \hline \end{array} \times \begin{array}{|c|} \hline d_0 \\ \hline d_1 \\ \hline d_2 \\ \hline d_3 \\ \hline \end{array} = \begin{array}{|c|} \hline d_0 \\ \hline d_1 \\ \hline d_2 \\ \hline d_3 \\ \hline p_0 \\ \hline p_1 \\ \hline \end{array}$$

Vandermonde Matrices

Amended RS decoding algorithm:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \theta & 4 & \theta & \theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \times_0 & \times_1 & \times_2 & \times_3 \\ X_4 & X_5 & X_6 & X_7 \end{bmatrix} \times \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} d_0 \\ e_1 \\ d_2 \\ d_3 \\ p_0 \\ p_1 \end{bmatrix}$$

Vandermonde Matrices

Amended RS decoding algorithm:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ X_4 & X_5 & X_6 & X_7 \end{bmatrix} \times \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_2 \\ d_3 \\ p_1 \end{bmatrix}$$

Vandermonde Matrices

Amended RS decoding algorithm:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ X_4 & X_5 & X_6 & X_7 \end{bmatrix}^{-1} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ X_4 & X_5 & X_6 & X_7 \end{bmatrix} \times \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ X_4 & X_5 & X_6 & X_7 \end{bmatrix}^{-1} \times \begin{bmatrix} d_0 \\ d_2 \\ d_3 \\ p_1 \end{bmatrix}$$

Vandermonde Matrices

Amended RS decoding algorithm:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ X_4 & X_5 & X_6 & X_7 \end{bmatrix}^{-1} \times \begin{bmatrix} d_0 \\ d_2 \\ d_3 \\ p_1 \end{bmatrix}$$



More problems...

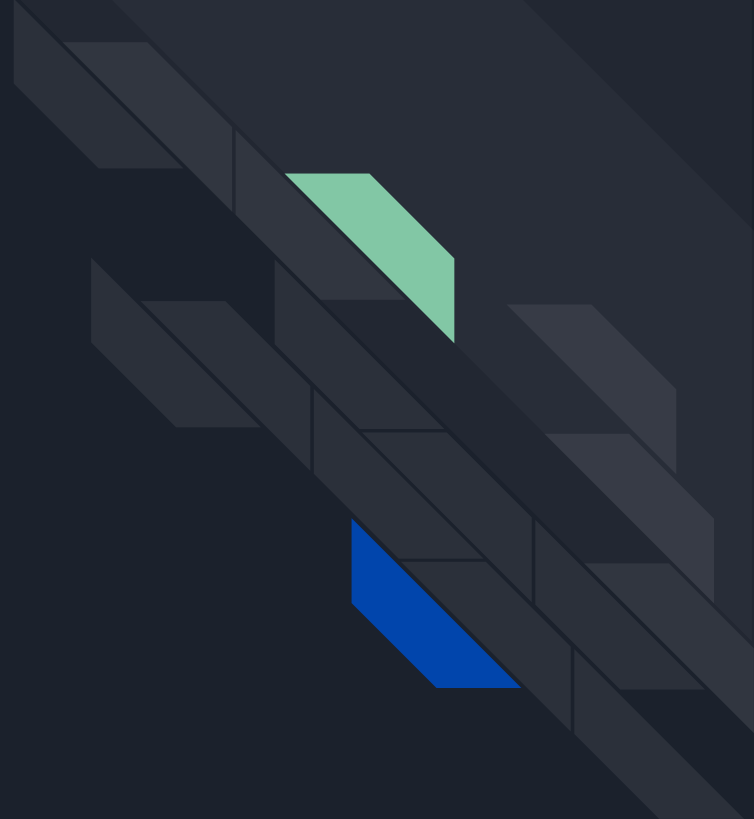
We've solved the problem of using slow polynomials by replacing them with fast matrices...but we still have two problems:

- We use matrices (polynomials) over \mathbb{R} , but computers don't like real numbers and floats are not a good substitute.
- It's not clear how to turn our symbols into integers in general. What if the new points we generate do not map back?

So, we need something other than real numbers, \mathbb{R} , but that *thing* must support addition, subtraction, multiplication, division, and exponentiation (by non-negative integers).

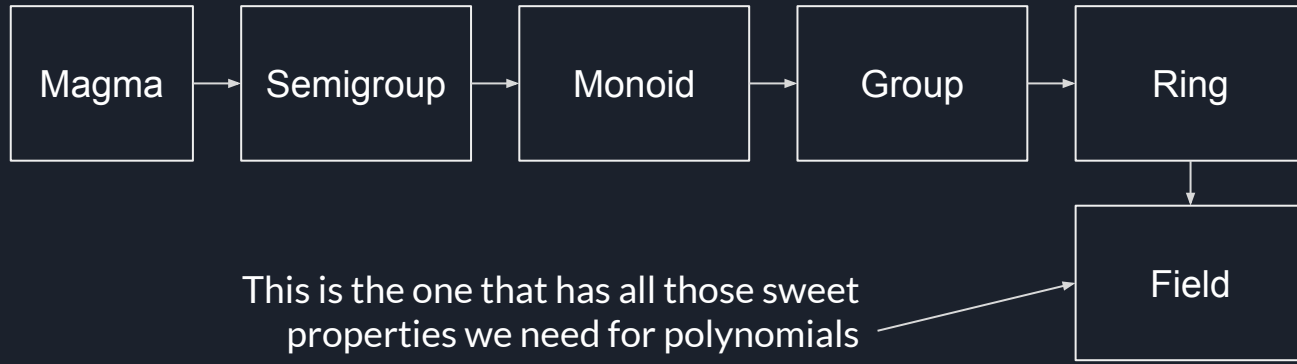
Hmm...

Finite Fields



Abstract Algebra

- Goal: Find the important properties of algebra and abstract them over things other than numbers.
- Many structures, but we'll just introduce a few.
- Think of each like an "interface" (with each building on the last).





Magma

- A set S with a binary operation: \bullet .
- The operation is "closed" (i.e. $x \bullet y \in S$)
 - Does not have to be associative.
 - Does not have to be commutative.

- Example: The set of integers under the addition (+) operator
 - Add any two integers and you get an integer.
 - However, integers with addition is more than just a magma.

- Example: vector cross product of \mathbb{R}^3 (aka 3D vectors)
 - This operation is closed, but not associative, commutative, has no identity, etc.



Semigroup

- A magma, but where the binary operation is associative.
 - Does not have to be commutative.
- $x \bullet y \bullet z = (x \bullet y) \bullet z = x \bullet (y \bullet z)$

- Example: The set of positive integers under addition.
 - Addition is closed and associative.

- Example: The set of integers under either "max" or "min".
 - $\max(x, y)$ is closed, so is $\min(x, y)$
 - Associative, $\max(x, \max(y, z)) = \max(\max(x, y), z)$
 - Also commutative.
 - No identity element, though.



Monoid

- A semigroup, but with an identity element, 0 or e.
 - Still does not have to be commutative.
- $x \bullet 0 = 0 \bullet x = x$

- Example: The set of integers under addition.
 - Addition is closed and associative and "0" is the identity.
- Example: The set of integers under multiplication.
 - Multiplication is closed and associative and "1" is the identity.
- Example: String concatenation
 - String concatenation is closed and associative and "" is the identity.
 - It is not commutative.
- Example: max of natural numbers with "0" as the identity.



Group

- A monoid, but with inverses for every element (i.e. x^{-1} or $-x$).
 - Still does not have to be commutative. An "abelian" group is a group with a commutative operation.
- $x \bullet (-x) = (-x) \bullet x = 0$ (i.e. the identity, i.e. "e")

- Example: The set of integers under addition.
 - Addition is closed and associative, "0" is the identity, and negation provides the inverse.
- **NON**-Example: the set of integers under multiplication.
 - Multiplication is closed and associative and "1" is the identity, but integer reciprocals do not exist.
- Example: The set of rational numbers under multiplication.
 - Same as with the set of integers, but now we can do $x^{-1} = 1/x$



Ring

- A group, but with a second associative binary operation that has its own identity.
 - Generalizes the notion of addition and multiplication.
 - The first operation ("addition") must have inverses, but not the second.
- $x \times y \in S$

- Example: Integers with addition and multiplication.
 - Division is not defined, but addition, subtraction (addition of inverses), and multiplication is. Two identity elements, 0 and 1.
- Example: Booleans with \wedge (XOR) and $\&\&$ (AND).
 - \wedge and $\&\&$ are both closed (produce booleans)
 - False is the identity for \wedge (False \wedge x = x) and True is the identity for $\&\&$ (True $\&\&$ x = x)
 - For XOR, every element is its own inverse (x \wedge x = 0)
 - Also works for sets of sets with disjoint union (analog to XOR) and intersection (analog to AND).



Field

- A ring, but where every element has an inverse under the second binary operator.
 - Generalizes the notion of division; inverse is not necessarily defined for the zero of the first binary operator.
- $x \times x^{-1} = 0$ ("0" is the identity of the second operator)

- Example: Rational, real, and complex numbers
 - All define addition, subtraction (additive inverse), multiplication, and division (multiplicative inverse). All are closed and associative.
 - 0 and 1 are the respective identities.
- Example: Integers modulo a prime number
 - Now division can be defined as the element y such that $x \times x^{-1} = 1$
 - This is guaranteed to exist thanks to Bezout's Identity



Finite Fields

Because computers tend to not like infinite precision, we'd like to use finite numbers like integers in our computations.

Our polynomials (our our matrices) require addition, additive inverses, multiplication, and multiplicative inverses...so the polynomials require a Field!

Real numbers are fields, but we can construct a finite field too.

Even better, we may be able to construct finite fields with the same number of elements as we have symbols so we can map them 1:1!



Finite Fields

It's easy to construct an additive *group* by just doing addition and taking the result modulo some number. Take hours (in 24h format) for example:

- $22:25 + 8:00 \text{ hours} == 30:25 \text{ MOD } 24 = 6:25$
- Additive inverses work too: $6:25 - 8:00 == -1:35 \text{ MOD } 24 = 22:25$

Doing multiplication modulo some number works too:

- $5 * 5 \text{ MOD } 24 = 25 \text{ MOD } 24 = 1$
- $4 * 8 \text{ MOD } 24 = 32 \text{ MOD } 24 = 8$



Finite Fields

But what about multiplicative inverses?

Remember a multiplicative inverse is defined as x^{-1} such that $x * x^{-1} = 1$

For every element, another element (possibly the same) must exist such that $x * y = 1$, thus $y = x^{-1}$

Let's take the presumptive "field" with 4 elements....

Elements: {0, 1, 2, 3}

Uh oh, no such element x^{-1} exists for $x=2$.

Multiplication: * MOD 4

Why? Because 2 is not coprime to the field size.

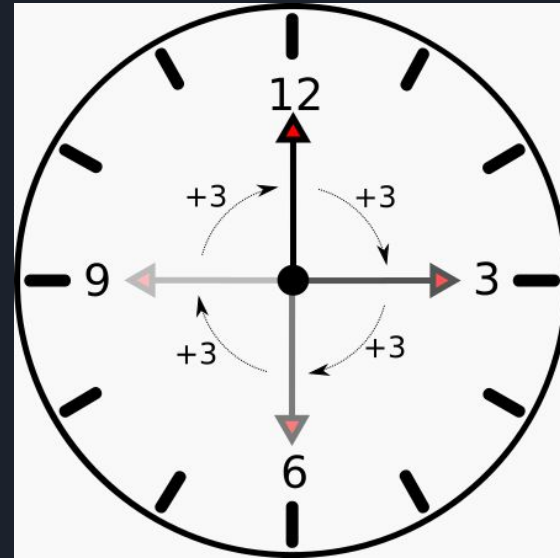
- $2 * 0 \text{ MOD } 4 = 0$
- $2 * 1 \text{ MOD } 4 = 2 \text{ MOD } 4 = 2$
- $2 * 2 \text{ MOD } 4 = 4 \text{ MOD } 4 = 0$
- $2 * 3 \text{ MOD } 4 = 6 \text{ MOD } 4 = 2$

Coprimality

Let's illustrate why co-primality is necessary with a clock...

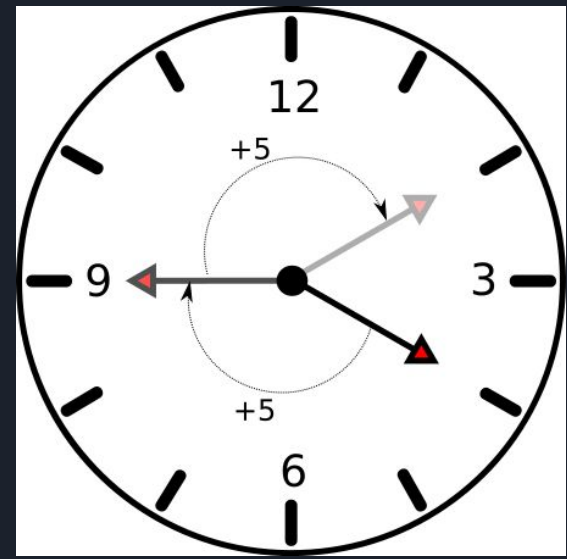
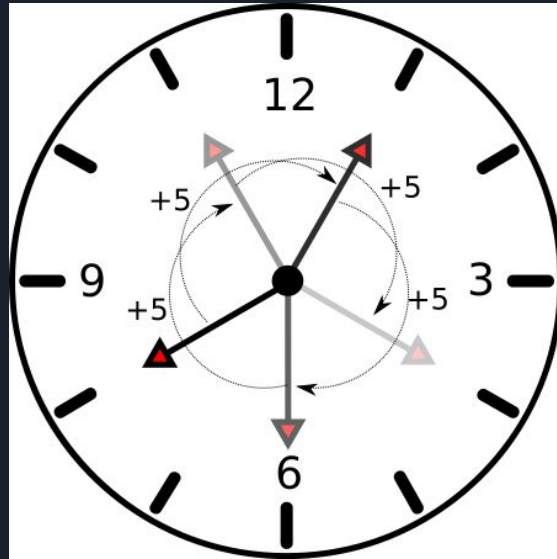
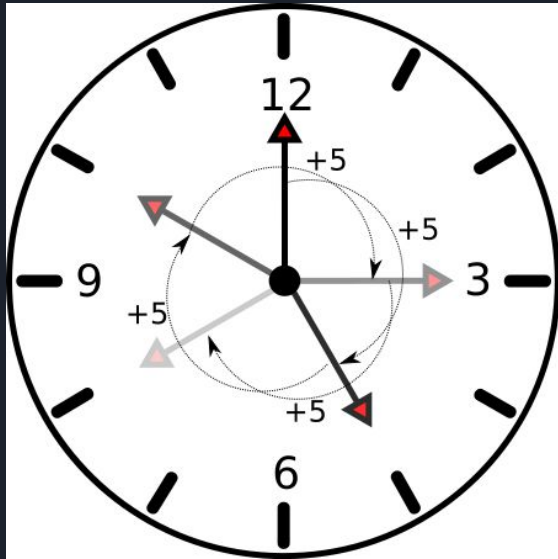
Here are multiples of 3 MOD 12:

$\text{GCD}(12, 3) = 3$, so the two are NOT coprime



Coprimality

Here are multiples of 5 MOD 12: $\text{GCD}(5, 12) = 1$





Finite Fields - Primes

Okay, so for x to be guaranteed to have a multiplicative inverse it must be coprime to the field size. That is, $\text{GCD}(x, q) = 1$.

This must be true for all elements in the field.

Which numbers are coprime to all numbers less than them? You guessed it!

Primes!



Finite Fields - Prime Powers

Integers with all operations modulo a prime are finite and they are fields. Yay!

But integers modulo a prime power (e.g. 2^N) are also finite and also fields!

- 2^N generates a lot of numbers that programmers like, so this is cool.

Prime-power finite fields are commonly called Galois Fields after [Évariste Galois](#).

- Technically finite fields == Galois fields, but I usually see the term "Galois fields" when dealing with fields of size q^{2^+} and "Finite fields" when dealing with fields of size q^1 .



Galois Fields

How do we make a field with q^N elements (also written: $GF(2^N)$)? Do we still mod? Let's try with 2^2 ...

Elements: $\{0, 1, 2, 3\}$

Multiplication: $* \text{ MOD } 4$

- $1 * 2 \text{ MOD } 4 = 2$
- $2 * 2 \text{ MOD } 4 = 4 \text{ MOD } 4 = 0$
- $2 * 3 \text{ MOD } 4 = 6 \text{ MOD } 4 = 2$
- $3 * 2 \text{ MOD } 4 = 6 \text{ MOD } 4 = 2$

Uh oh... What's the multiplicative inverse of 2? i.e. what X exists such that $2 * X = 1$? If we don't have a multiplicative inverse it's not a field!



Galois Fields

How do we make a field with 2^2 ? Here's a crazy idea... what if we just pretend it's 2 separate field elements of size 2?

So our elements $\{0, 1, 2, 3\}$ become:

- 0:

0	0
---	---
- 1:

0	1
---	---
- 2:

1	0
---	---
- 3:

1	1
---	---

This might seem familiar...

Galois Fields - Addition

How do we do addition with our new elements? Well, each element just maps to 2 different elements, so add them independently the way we did before, + MOD 2?

So our elements {0, 1, 2, 3} become:

- 0:

0	0
---	---
- 1:

0	1
---	---
- 2:

1	0
---	---
- 3:

1	1
---	---

$$\begin{array}{r} 2 \\ +3 \\ \hline =1 \end{array}$$

	1	0
+	1	1
=	0	1

$$\begin{array}{r} 1 \\ +3 \\ \hline =2 \end{array}$$

	0	1
+	1	1
=	1	0

It's like normal addition, but now when we go "over" (i.e. $1+1 = 2$) there's no carry as we just do MOD within the single element...so let's call it carryless addition.



Galois Fields - Addition Table

Here's the full addition table:

Remember that the additive inverse is $-x$ such that $x + -x = 0$. Here, every element is its own additive inverse!

+	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Galois Fields - Multiplication

Now, how do we do multiplication with our new elements? Well, multiplication is just repeated (and shifted) addition, so let's try that?

- 0:

0	0
---	---
- 1:

0	1
---	---
- 2:

1	0
---	---
- 3:

1	1
---	---

		1	0
	x	1	1
	+	1	0
	1	0	0
=	1	1	0

Uh oh... we got an element not in the field, but multiplication must be a closure.

Can we somehow "mod" it back into the field?



Galois Fields - Polynomials

The interpretation of a q^N field as N separate elements in a field of size q makes the elements behave a lot like a polynomial...

A polynomial looks like: $f(x) = a_0x^0 + a_1x^1 + a_2x^2 + \dots + a_nx^n$

If we want to add two polynomials, we have to do it term by term. That is, the x^0 s go together and the x^1 s go together, etc.

If you multiply two polynomials, you "shift" the degree up. $2x^3 * 3x^2 = 6x^5$

So Galois fields of size q^N are like polynomials of degree $N-1$, where the coefficients are evaluated MOD q .



Galois Fields - Polynomials

Great, they're polynomials, but when we multiply two of them together, they get too big still!

Well, polynomials have something just like prime numbers called irreducible polynomials. These are polynomials that cannot be formed by multiplying two other polynomials together.

In 2^N here are some:

- x
- $x+1$
- x^2+x+1
- x^3+x+1

In general we can always find an irreducible polynomial of degree N . Similar to with regular primes, an irreducible polynomial is "coprime" to all other polynomials less than it.



Galois Fields - Modulo Polynomials

What does it mean to find a polynomial "modulo" another polynomial?

With positive integer numbers, division of X by Y does two things:

1. Tells you how many times you can subtract Y from X and have a result >0 (or, until $X < Y$)
2. Tells you what the result is (the "remainder").

Modulo is the operation that gives you just the remainder after subtracting Y as many times as possible.

So, if X and Y are polynomials and you want $X \text{ MOD } Y$, just subtract Y from X until $X < Y$ (or until $\text{degree}(X) < \text{degree}(Y)$).

In $\text{GF}(2^N)$ subtraction and addition are both XOR!

Galois Fields - Modulo Polynomials

GF(2^N) example 10 * 12

Multiplication Step:

10:

1	0	1	0
---	---	---	---

12:

1	1	0	0
---	---	---	---

			0	0	0	0
--	--	--	---	---	---	---

		0	0	0	0	
--	--	---	---	---	---	--

	1	0	1	0		
--	---	---	---	---	--	--

1	0	1	0			
---	---	---	---	--	--	--

Result:

1	1	1	1	0	0	0
---	---	---	---	---	---	---

Galois Fields - Modulo Polynomials

GF(2^N) example: 10 * 12
 Irreducible: x⁴+x+1 (10011)
 Reduction Step

2.

0	1	1	0	0	1	0
	1	0	0	1	1	
		1	0	1	0	0

1.

1	1	1	1	0	0	0
1	0	0	1	1		
	1	1	0	1	0	0

3.

0	0	1	0	1	0	0
		1	0	0	1	1
			0	1	1	1

10 * 12 = 7



Galois Fields - Division

Now we have one operation remaining: division. Remember, this is equivalent to finding the multiplicative inverse for a number. That is, $X/Y = X * Y^{-1}$.

There are (at least) 4 ways to find Y^{-1} :

- Use the [polynomial extended Euclidean algorithm](#) (which uses Bezout's identity).
- Loop through every element of the (finite) field and break when you find an element Y^{-1} such that $Y * Y^{-1} = 1$. That is, brute force.
 - If building a multiplication table, this can be done at the same time.
- Work in log-space, that is $\log(x / y) = \log(x) - \log(y)$ (see appendix for more).
- Build an inverses table from a method above.



Galois Fields - Summary

So, to have finite fields with q^N elements, i.e. $GF(q^N)$ we:

- Pretend there are N different elements in a field of size q .
- Equivalently, we say the elements are polynomials.
- Addition and subtraction are carryless (it's polynomial addition/subtraction MOD q).
 - In $GF(2^N)$ both addition and subtraction are XOR.
- Multiplication is normal multiplication with carryless addition...but then we reduce the result with some irreducible polynomial of degree N .
 - It can be any polynomial of degree N . Usually there will be several irreducible polynomials.
 - The result polynomial should be degree $N-1$
- Division is multiplication by an element's multiplicative inverse.
 - Because we use an irreducible polynomial, all elements less than it must be "coprime" to it and thus we know there will be an element x^{-1} for each x such that $x^{-1} * x = 1$.



Galois Fields And Others Bits

Because we're working with computers, we likely want to work with things like bits and bytes. Luckily for us bits correspond to $GF(2)$ and bytes correspond to $GF(2^8)$!

Thus, we can do our Reed Solomon computations using $GF(2^N)$ for some N (8 or 16 are common). Finally we solved our last 2 problems:

- We use matrices (polynomials) over \mathbb{R} , but computers don't like real numbers and floats are not a good substitute.
 - Not anymore! We use finite fields now.
- It's not clear how to turn our symbols into integers in general. What if the new points we generate do not map back?
 - Well, if the symbols we want to work with are binary in nature (i.e. 2^N) then we can just use $GF(2^N)$ and have a 1:1 mapping.



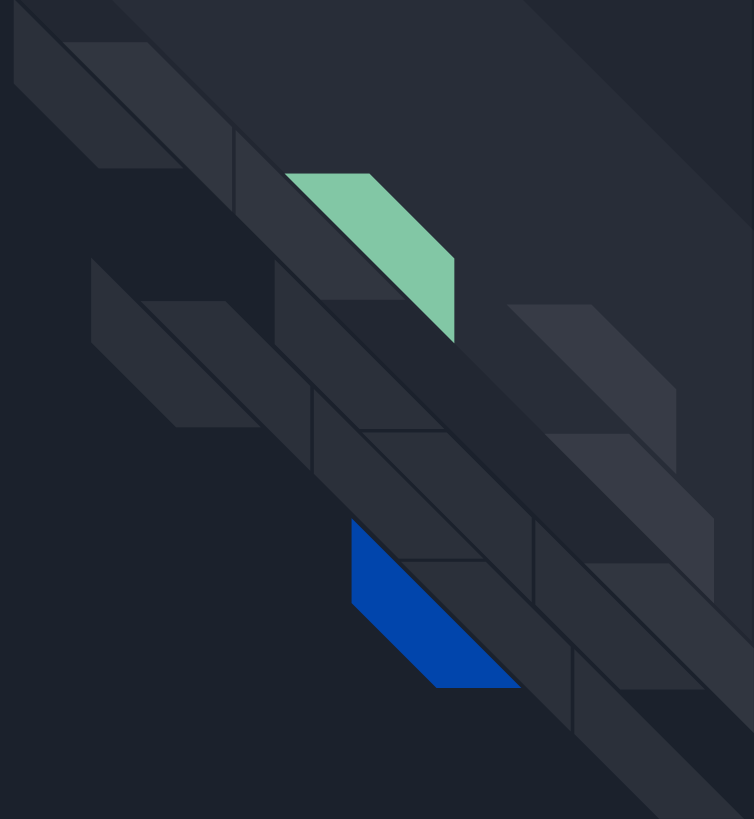
Reed Solomon - Summary

- Reed Solomon is generating a polynomial from a set of input data "points" and then generating redundancy by finding a few more points along the polynomial.
- We can generate a matrix to do the same thing, but much faster than futzing with polynomials.
- Polynomials work with any kind of field. Because our computers are finite, we want to use finite fields.
- Specifically, because we usually work with binary data (e.g. bytes), we use Galois fields with 2^N elements, that is, $GF(2^N)$ for our points and our operations on them.

It's as simple as that!

The next few slides (the appendix) has some more details on optimizations and other random things I think are interesting.

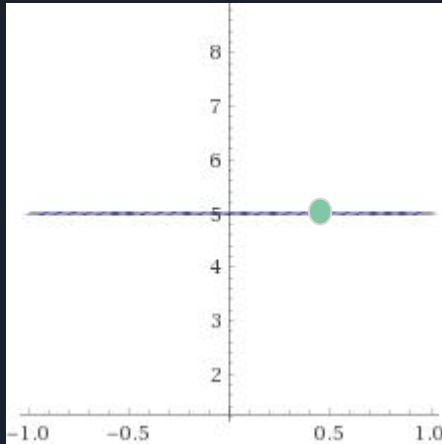
Appendix



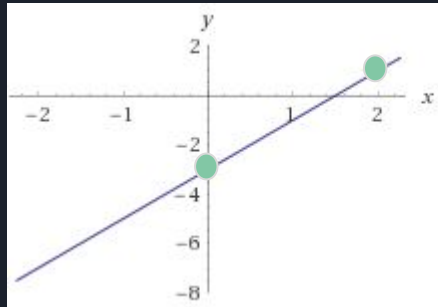
Polynomials - Another Intuition for Degrees

On a previous slide I said that with N points we can form a polynomial of degree $N-1$...let's consider how this looks for a few values of N ...

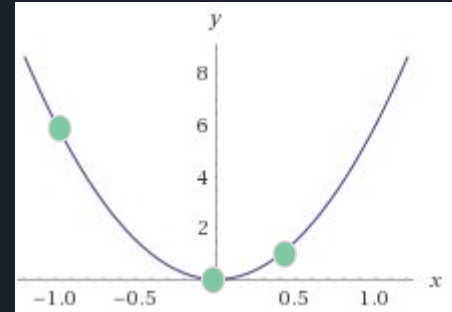
$\{(0.5, 5)\}$



$\{(0, -3), (2, 1)\}$



$\{(-1, 6), (0, 0), (0.5, 1.5)\}$





$rs=1.2 \implies r=3?$

This is a question from a coworker: Does $rs=1.2$ do the same thing as $r=3$?

Yes! It's really cool how we can prove this. We know that with n points we'll construct a polynomial of degree $n-1$, so if we have 1 (data) point, then we'll construct a polynomial of degree 0. A polynomial of degree 0 is constant (the highest x term is x^0 which is always 1, and thus x is ignored). And, the constructed polynomial must satisfy $f(x_0) = y_0$. So, it's constant and at least one point is (x_0, y_0) so we know that $f(x)$ over all x values MUST be y_0 , and thus we've just done replication (but with RS).

Note too that we had a proof much further up that replicating a single symbol is an MDS code (and RS is an MDS code), so this is not a surprise.



Faster Galois Field Arithmetic

As I noted (at least in the speaker notes) in an earlier slide, when we're using $GF(2^N)$, addition becomes XOR. If you do y XOR y you always get 0, thus every element is its own additive inverse, so subtraction is also XOR.

However, multiplication required multiple XORs and a final reduce step which is fairly slow. Can we do better? Yes!

- We can pre-compute multiplication tables if the field isn't too large; then we just need to do a lookup.
- We can work in "log space" where multiplication becomes addition.
 - That is, take a "generator" element (an irreducible polynomial, so " $x+1$ " works) and raise it to all powers $[0..N)$. Map each number to its generator power and vice-versa.
 - To multiply 2 numbers, you map to their exponential number, then add them, then map back.
 - This requires 3 table lookups, but you only need $2N$ elements in your table instead of N^2



CPU Intrinsics for faster field arithmetic

- XOR is a single CPU instruction to do carryless addition, but what about multiplication?
 - So far we've had to implement carryless multiplication in software using either repeated carryless addition (i.e. long form multiplication) and/or multiplication/exponent tables.
- Enter CLMUL
 - Carry-less multiplication, i.e. multiplication in a 2^N field.
 - https://en.wikipedia.org/wiki/CLMUL_instruction_set
- There are also fancy CPU instructions for doing many scalar operations across a vector of data at once: [SIMD](#).
 - Combine these and you can do entire matrix dot products (where the multiplication step is CLMUL and the addition step is XOR) with a small number of CPU instructions.



Cauchy Matrices

The generator matrix comes from the equation $Y' = X'X^{-1}Y$

We really only need 2 properties from our X matrix:

1. X is invertible
2. Any N rows from X' are linearly independent (invertible).

Vandermonde matrices satisfy these properties, but so do [Cauchy matrices](#). Square Cauchy matrices are *always* invertible (by their construction) and every submatrix of a Cauchy matrix is also a Cauchy matrix meaning a square submatrix is invertible.

Some RS algorithms use Cauchy matrices because they allow for different matrices to be used and some of those matrices actually guarantee less XORs are needed when doing multiplication! You can read briefly about it [here](#) or see [a full paper about it here](#).



Why did I write this?

It started as I was reading the second chapter of "[A Programmer's Introduction to Mathematics](#)". It discussed polynomials and at the end of the chapter presented a programming-related application, [Shamir's Secret Sharing algorithm](#). While reading that, I realize it was very similar to Reed Solomon which I was familiar with in practice (a few years ago I tried learning how it worked, but then got scared off when I saw the word "polynomial" :P). I was looking for a programming project and thought making a Reed-Solomon encoder would be fun. Little did I know I needed to learn a lot more (about fields and block codes) to do it right. I implemented something that didn't work, then spent lots of time learning about the mathematics of Fields. I also spent a lot of time moving from the Lagrangian interpolation method of constructing polynomials presented in the book towards understanding the mathematics behind doing it via Vandermonde and Cauchy matrices. Once I finally finished my encoder, I thought it'd be cool to give a talk given how much time I spent learning this stuff and how interesting I find the math. So here we are.



Additional Resources

- [A Gentle Introduction to Erasure Codes](#)
- [Finite Field Arithmetic and Reed-Solomon Coding](#) (by the author of RE2)
- [Hamming's Code](#)
- [The Codes of Solomon, Reed, and Muller](#)
- [Reed-Solomon error-correcting code decoder](#)
- [Error Correction with Reed-Solomon](#)
- [Backblaze Video on Reed Solomon](#)
- [Backblaze open source implementation \(in Java\) of RS](#)
- [Implementation of RS in Go by klauspost](#) (inspired by Backblaze implementation)