Learners' Space 2025 – Coding Theory
Week 1 Handout

Aryan Prakash & Nirav Bhattad, MnP Club, IIT Bombay

June 2025

Contents

§1. Source Coding

1.1. The Problem of Communication: An Early Scenario

In the early 1900s, imagine two neighbors wanting to communicate at night without speaking. Suppose they could see each other through their windows and owned flashlights. With no sound, how could they send messages? One idea was to use the flashlight to signal letters by blinking: `A` $\to$ 1 blink, `B` $\to$ 2 blinks, ... , `Z` $\to$ 26 blinks.

Example 1.1. To send `I LOVE CODING THEORY`, they would use

$I(9) + L(12) + O(15) + V(22) + E(5) + C(3) + O(15) + D(4) + I(9) + N(14) + G(7) + T(20) + H(8) + E(5) + O(15) + R(18) + Y(25) = 206$ blinks

The blinking sequence might look like this:

```
111111111 11111111111 11111111111111 1111111111111111111111 11111 111 111111111111111 1111
111111111 1111111111111 1111111 1111111111111111111 11111111 11111 111111111111111
11111111111111111 11111111111111111111111
```

1.2. Encoding and Decoding

A **message** is the original data to be sent, made up of symbols. An **encoded message** is the actual data sent, made up of **codewords**. The process of turning messages into encoded messages is **encoding**. The reverse process is **decoding**.

Encoding: `I LOVE CODING THEORY` $\to$ blinking pattern above

Decoding: blinking pattern above $\to$ `I LOVE CODING THEORY`

1.3. Designing a Code

A **code** is a system for transferring information. In the flashlight example, we used base-1 unary codes. However, we can improve efficiency by mapping frequent letters to shorter codes.

1.3.1. Frequency-Based Codes

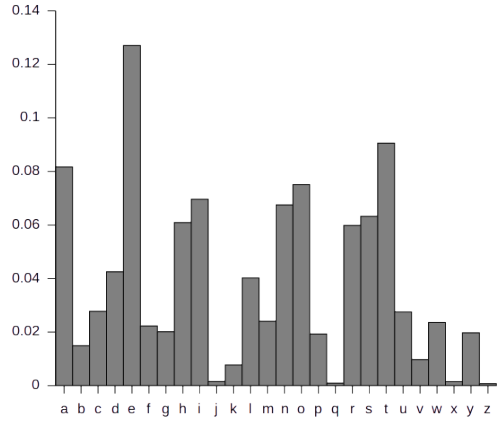


Figure 1: Frequency of Alphabets in the message

If we define a new scheme where `E` $\to 1$, `T` $\to 2$, `A` $\to 3$, $\ldots$, we reduce the blink count significantly.

Example 1.2. `I LOVE CODING THEORY` $= 138$ blinks.

```
11111 11111111111 1111 11111111111111111111 1 1 11111111111 1111 1111111111 11111 111111
1111111111111111 11 11111111 1 1111 111111111 1111111111111111
```

This is still base-1 encoding. A better scheme is Morse Code.

1.4. Morse Code

We now switch to base-2 encoding. Morse Code uses **dots** (.) $\to 1$ blink and **dashes** (–) $\to 3$ blinks. Letters are represented using sequences of dots and dashes. Spaces are used to separate letters and words.

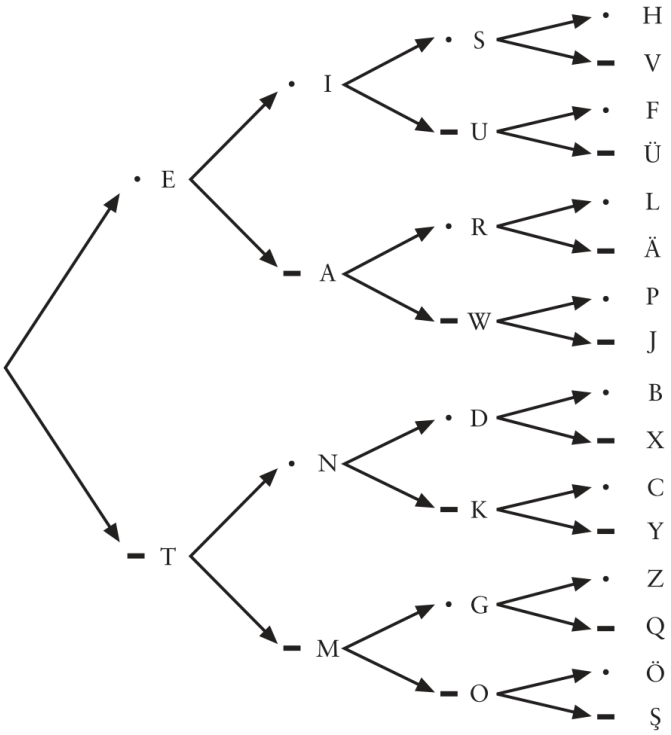


Figure 2: Decoding of Alphabets

The first image above shows the tree structure of Morse Code. Each left branch corresponds to a dot (.) and each right branch corresponds to a dash (-). Note that Morse Code is not a Prefix-Free coding scheme.

A	•–
B	–•••
C	–•–•
D	–••
E	•
F	••–•
G	––•
H	••••
I	••

J	•–––
K	–•–
L	•–••
M	––
N	–•
O	–––
P	•––•
Q	––•–
R	•–•

S	•••
T	–
U	••–
V	•••–
W	•––
X	–••–
Y	–•––
Z	––••

Figure 3: Encoding of Alphabets

The second image lists the morse code for each letter A – Z.
Example 1.3. `I LOVE CODING THEORY` $\rightarrow 91$ blinks.

```
..  .-..  --  ...-  .  -.-.  --  -..  ..  -.  -.  -  ....  .  --  .-.  -.-
```

Pauses in Morse Code

In Morse transmission (e.g., via blinking light or beeps), timing plays a critical role in decoding:

- A **dot** is 1 time unit (1 blink or short beep).
- A **dash** is 3 time units (3 blinks or long beep).
- The **gap between elements of a letter** (dots and dashes) is 1 unit.
- The **gap between letters** is 3 units (i.e., a slightly longer pause).
- The **gap between words** is 7 units (i.e., a significantly longer pause).

This timing ensures that messages can be decoded unambiguously, even without explicit delimiters. Morse code is thus both compact and self-synchronizing.

1.5. Challenges in Communication Codes

Storage and Delimiters. When storing or transmitting messages, we must distinguish where one codeword ends and another begins. If codewords overlap or are prefixes of each other, decoding becomes ambiguous.

Prefix Problem. For instance, if:

$$T \rightarrow 1$$
$$N \rightarrow 10$$
$$D \rightarrow 100$$
$$X \rightarrow 1001$$

then codewords are prefixes of longer ones. This causes problems during decoding.

1.6. Solutions to the Prefix Problem

1.6.1. Longer Delimiters

Use longer pauses or markers to separate codewords, but this adds overhead.

1.6.2. Fixed-Length Codes

Assign the same number of bits to each symbol. For example, ASCII uses 8 bits per character.

ASCII Example: `I LOVE CODING THEORY`

```
01001001 00100000 01001100 01001111 01010110 01000101 00100000 01000011
01001111 01000100 01001001 01001110 01000111 00100000 01010100 01001000
01000101 01001111 01010010 01011001
```

1.6.3. Prefix-Free Codes

Use codes where no codeword is a prefix of another. Such codes are **instantaneously decodable**.

1.6.4. Huffman Coding

Huffman Coding builds an optimal prefix-free code given the frequency of each symbol. Symbols with higher frequency get shorter codewords.

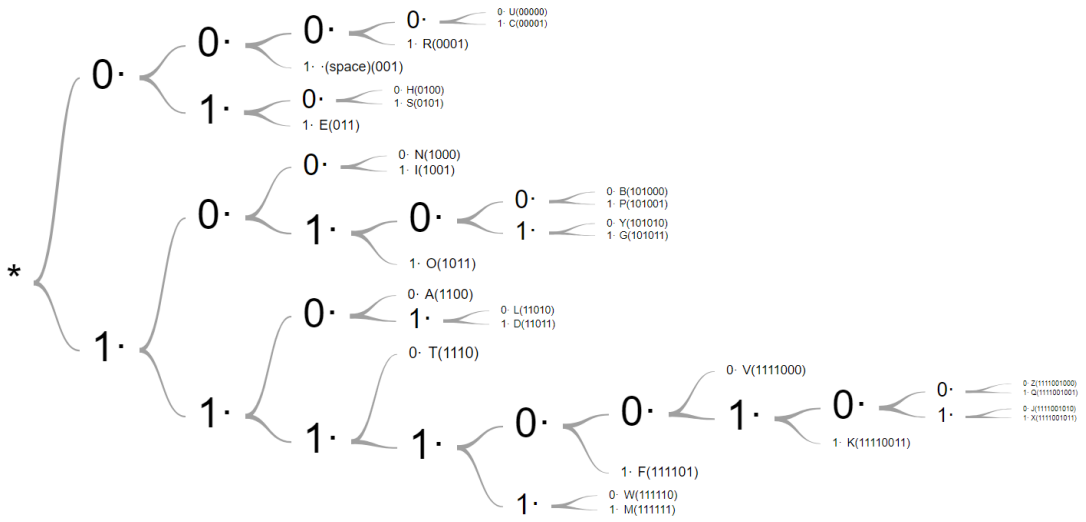


Figure 4: Example of a Huffman Encoding

A	1100	N	1000
B	101000	O	1011
C	00001	P	101001
D	11011	Q	1111001001
E	011	R	0001
F	111101	S	0101
G	101011	T	1110
H	0100	U	00000
I	1001	V	1111000
J	1111001010	W	111110
K	11110011	X	1111001011
L	11010	Y	101010
M	111111	Z	1111001000
		<space>	001

Figure 5: Example of a Huffman Encoding

Example Huffman Output (binary):

```
1001001110101011111100001100100001101111011100110001010110011111001000111011000110101 0
```

We will be discussion more about Huffman Encoding in Section 3.

§2. Channel Coding

So far, we've looked at how to encode messages efficiently. But what if something goes wrong during transmission?

Imagine you're again using flashlights to communicate. This time, you're far apart, and there's fog in the air. Sometimes a blink doesn't get noticed, or worse, a blink appears even though you didn't send one! This is the problem of **noise** – interference that corrupts the signal.

2.1. What Can Go Wrong?

You send: `1 1 1 0 0 1`
They receive: `1 1 1 1 0 1`

One blink was changed – perhaps your hand slipped, or the receiver miscounted. To ensure your message is understood correctly, you need to protect it.

2.2. Idea: Add Extra Information

Just like when we double-check our work in exams, we can double-check our signals. The idea is to send extra blinks that help the receiver detect and possibly fix mistakes.

A Simple Trick: Repetition Instead of sending each bit once, send it three times:

$$1 \to 111, \quad 0 \to 000$$

So, the message `1 0 1` becomes:

$$111\ 000\ 111$$

Even if one blink flips by mistake (e.g., `110` instead of `111`), the receiver can guess what you meant by taking the majority: two 1s vs one 0 → likely a 1.

2.3. Fixing vs Detecting Errors

There are two goals here:

- **Detecting** errors: knowing that something went wrong.
- **Correcting** errors: figuring out what the original message was.

Some systems are only meant to detect errors. For example, if you're copying a number from a form and it doesn't add up, you can just request it again. But if asking again is costly—say, in deep space communications—you want to correct it on your own.

2.4. Smarter Than Repeating

Repeating bits works, but it's wasteful. If every bit takes 3 times the space, your messages become long and slow.

There are better ways to add just enough extra blinks to spot or fix mistakes—clever patterns that can tell you where the error happened, without needing to resend the whole message. These patterns form the basis of what we now call **error-correcting codes**.

2.5. Everyday Use

These ideas are used all around us:

- QR codes and barcodes include extra bits so that even if part is smudged, your phone still reads it.
- Computers use extra bits when storing data so that minor hardware glitches don't corrupt your files.
- Streaming videos and music rely on these techniques to prevent static or glitches even when internet connections drop briefly.

Even the best-encoded message is useless if it gets scrambled in transmission. Channel coding gives us a way to guard against this – by adding smart redundancy, we make our messages not only compact, but also reliable.

The next challenge is to design these clever patterns – codes that not only carry the message, but also carry a backup plan.

§3. Huffman Encoding

Huffman coding is a fundamental algorithm in information theory that produces prefix-free binary codes for a set of symbols based on their frequencies. We rigorously present the Huffman encoding algorithm, demonstrate its correctness, and prove its optimality among all prefix-free binary codes with respect to expected code length.

Let $\Sigma = \{s_1, s_2, \ldots, s_n\}$ be a set of symbols, each occurring with frequency (or probability) $f_i > 0$ such that $\sum_{i=1}^{n} f_i = 1$. A binary prefix code assigns a binary string $C(s_i)$ to each symbol s_i, such that no codeword is a prefix of another (ensuring uniquely decodable codes).

The objective is to find a prefix code minimizing the expected codeword length:

$$L(C) = \sum_{i=1}^{n} f_i \cdot |C(s_i)|$$

where $|C(s_i)|$ denotes the length of the binary codeword for symbol s_i.

3.1. The Huffman Encoding Algorithm

The Huffman algorithm proceeds as follows:

Step 1. Initialize a forest of n trees, each containing one symbol s_i with weight f_i.

Step 2. While there is more than one tree in the forest:

(a) Select the two trees T_1 and T_2 with the smallest weights w_1 and w_2.

(b) Merge them into a new tree T with root having weight $w_1 + w_2$ and children T_1 and T_2.

Step 3. The final tree defines the code: assign 0 to the left edge and 1 to the right edge recursively. The codeword for each symbol is the path from root to the leaf representing it.

3.2. Proof of Optimality

We now prove that Huffman coding yields a prefix-free code of minimum expected length.

Theorem 3.1. The Huffman algorithm produces a prefix code of minimum expected length among all prefix codes for given symbol frequencies.

Proof Outline (Greedy Exchange Argument)

We proceed by induction and greedy exchange.

Base Case: For $n = 2$, the only prefix code has one symbol encoded as 0 and the other as 1. This is trivially optimal.

Inductive Step: Assume optimality holds for $n - 1$ symbols. Let x and y be the two symbols with least frequencies. Any optimal prefix code must assign them the longest codewords and these codewords must differ only in the last bit (as they must be siblings in the code tree).

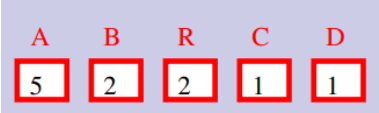
Why? Because assigning longer codewords to more frequent symbols would increase expected length.

Now, merge x and y into a single symbol z with frequency $f_z = f_x + f_y$, and find an optimal code for the reduced set. By the inductive hypothesis, the Huffman procedure produces the optimal code for this smaller set. Expanding z back to x and y by adding a bit distinguishes them, giving an optimal code for the original set.

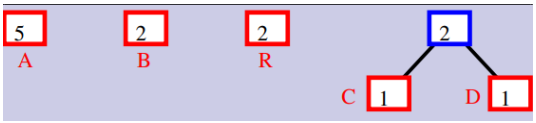
Thus, by induction, the Huffman algorithm produces an optimal prefix-free code.

□

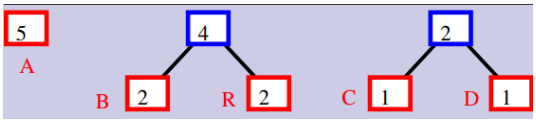
We now present an example which shows the algorithm in action.Consider the string $ABRACADABRA$. We demonstrate the algorithm by encoding this string. First we calculate the frequency of each of these characters in the string, we notice that A appears 5 times, B appears 2 times, C appears 1 time and D appears 1 time. No step 1 asks us to make a forest of trees, each with one symbol and their weights being their frequencies. So initially our tree looks like this :



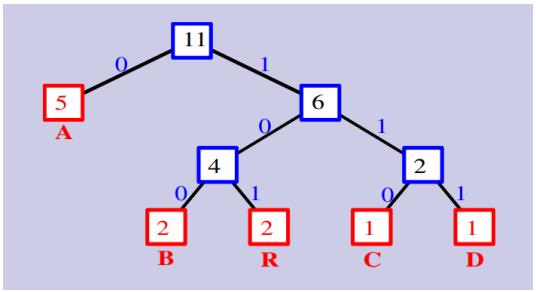
Next in step 2, we must choose, the 2 smallest weighted trees in the forest and merge them until the forest has only 1 tree. These are C and D both with weight 1. These 2 trees are merged and the forest then looks like this :



Now notice the forest has 3 trees with the minimum weight of 2, so which 2 shoudl we choose? Turns out, no matter which 2 you choose, the final encoding of the message will have the same number off bits. So we choose to merge B and R, here for the sake of explaining the algorithm.The forest now looks like this :



Now we merge the tress with weight 4 and 2, and then finally merge the tree with weight 5, and the previously formed tree of weight 6, to get the final tree which looks something like this(note this is the tree after applying step 3, where all left edges have been 0 and right edges have assigned 1) :



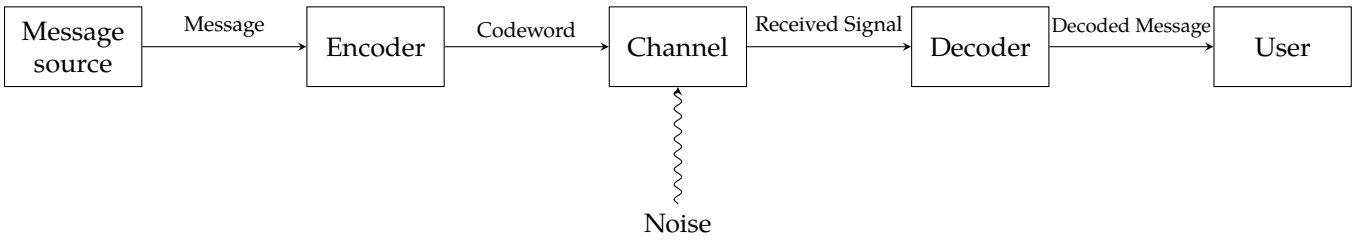
Notice that this gives how each symbol is encoded. A is encoded as 0, B is encoded as 100, R as 101, C as 110 and D as 111. We can see that this encoding ensures that no symbol's code is a prefix of the others. For the word $ABRACDABRA$, we get the encoded message as 01001010110011101000010, which has 23 bits in total, which is indeed optimal. Try out the steps yourself if we decide to merge the R-tree with the C, D-tree in the second step, and check if you get the same number of bits.

§4. Introduction to Codes

Communication is a crucial part of modern life, and one of the remarkable features of languages like English is the inherent redundancy that allows people to understand each other even when there are errors, such as accents or mispronunciations. This redundancy makes it possible to recover information even when small errors occur. Similarly, in digital communication, redundancy is used to handle errors that might occur during data transmission, processing, or storage.

Error-correcting codes (or simply "codes") are techniques that introduce redundancy into data to allow the original message to be recovered even if parts of it are corrupted. These codes are vital in scenarios where retransmission is not possible, such as satellite communication, deep space exploration, and cellular networks. Beyond communication, codes are used in data storage technologies like CDs and DVDs, which remain functional even with scratches due to the error-correcting codes.

In a typical communication setup, the sender encodes a message into a longer sequence of symbols (called a codeword) and transmits it over a noisy channel. The receiver then decodes the received data to recover the original message. The challenge in coding theory lies in balancing redundancy and error correction: more redundancy generally means better error correction but also more resources. The key question is how much redundancy is needed to correct a given number of errors while minimizing the overhead.



Definition 4.1 (Code)**.** A code C of block length n over an alphabet Σ is a subset of Σ^n.

Usually the elements of Σ are called alphabets and the elements of C are called codewords. We define q as the alphabet size $|\Sigma|$.

Definition 4.2 (Dimension of a Code)**.** Given a code $C \in \Sigma^n$, its dimension is defined by

$$k \coloneqq \log_q |C|$$

Definition 4.3 (Rate of a Code)**.** The rate of a code with dimension k and block length n is given by

$$R \coloneqq \frac{k}{n}$$

The dimension of a code is the number of bits in the original message. The higher the dimension, the more information can be transmitted. The rate basically means the ratio of the original message to the transmitted message. The higher the rate, the more efficient the code is.

Example 4.4 (Parity Code)**.** The *Parity Code*, denoted by $C_\oplus$, is one of the most basic error-correcting codes. Given a t-bit binary string $(x_1, x_2, x_3, \ldots, x_t) \in \{0,1\}^4$, its corresponding codeword is given by

$$C_\oplus(x_1, x_2, x_3, \ldots, x_t) = (x_1, x_2, x_3, x_4, x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_t)$$

where $\oplus$ denotes the XOR operation. This code has a dimension of t and a rate of $\frac{t}{t+1}$. Note that such a code uses the minimum amount of non-zero redundancy.

Example 4.5 (Repetition Code)**.** The *Repetition Code*, is another basic error-correcting code. Given a t-bit binary string $(x_1, x_2, x_3, \ldots, x_t) \in \{0,1\}^t$, its corresponding codeword is given by

$$C_{3,rep}(x_1, x_2, x_3, \ldots, x_t) = (x_1, x_1, x_1, x_2, x_2, x_2, x_3, x_3, x_3, \ldots, x_t, x_t, x_t)$$

This code has a dimension of t and a rate of $\frac{t}{3t} = \frac{1}{3}$.

4.1. Error Correction

Before we formally define what Error Correction means, we first define the notion of encoding and decoding.

Definition 4.6 (Encoding Function)**.** Let $C \subseteq \Sigma^n$. An equivalent description of the code C is an injective mapping $E\colon [|C|] \to \Sigma^n$ called the *encoding function*.

Definition 4.7 (Decoding Function)**.** Let $C \subseteq \Sigma^n$ be a code. A mapping $D\colon \Sigma^n \to C$ is called a *decoding function* for C.

Definition 4.8 (Hamming Distance)**.** Given 2 vectors $u, v \in \Sigma^n$, the *Hamming Distance* between u and v, denoted by $\Delta(u, v)$, is the number of positions in which u and v differ. We also define the *relative Hamming Distance*, denoted by $\delta(u, v)$, as $\delta(u, v) = \frac{\Delta(u,v)}{n}$.

One can easily verify that Hamming distance is a metric on the set Σ^n. In particular, it satisfies that Triangle Inequality.

The notion of relative Hamming Distance is important because if normalizes the distance $\delta(u, v)$ so that it always lies in the interval $[0, 1]$ for every n, Σ and every $u, v \in \Sigma^n$. This normalization will prove to be useful when we study the asymptotic behaviour of codes as $n \to \infty$.

4.2. Hamming Codes

Before we move on, we must first define what the **distance** of a code means.

Definition 4.9 (Minimum Distance)**.** Let $C \subseteq \Sigma^n$. The *minimum distance* (or just *distance*) of C , denoted $\Delta(C)$, is defined to be

$$\Delta(C) = \min_{u,v \in C, u \neq v} \Delta(u, v)$$

We also define the *relative minimum distance* of C to be $\delta(c)$, is defined to be

$$\delta(C) = \min_{u,v \in C, u \neq v} \delta(u, v)$$

We denote a code with block length n, dimension k and minimum distance d as a $[n, k, d]$ code. It is easy to check that the $C_{3,rep}$ mentioned in Example 4.5, has a distance of 3. We then ask ourselves the following question : Does there exist a code with distance and rate, $R > \frac{1}{3}$? If yes, this code would have a lower redundancy that $C_{3,rep}$, which is what we desire. An example of such a code is the Hamming code.

On a 4-bit vector, that is a message of length 4, where $\Sigma = 0, 1$, the Hamming code, denote by C_H, encodes, the message x_1, x_2, x_3, x_4, in the following way :

$$C_H(x_1, x_2, x_3, x_4) = (x_1, x_2, x_3, x_4, (x_2 \oplus x_3 \oplus x_4), (x_1 \oplus x_3 \oplus x_4), (x_1 \oplus x_2 \oplus x_4))$$

It is easy to see that for C_H, since $k = 4$ and $n = 7$, $R = \frac{4}{7}$. But we still do not know the distance of C_H. To find this, we must define what is meant by the Hamming weight of a code.

Definition 4.10. Let $q \geqslant 2$. Given any vector $\mathbf{v} \in \{0, 1, 2, 3, \ldots, q-1\}^n$, its Hamming weight, denoted by $\mathrm{wt}(\mathbf{v})$, is the number of non-zero symbols in $\mathbf{v}$.

Theorem 4.11. C_H has a distance of 3.

The proof is a bit long, but not very difficult to figure out and is thus left as an exercise to the reader. A hint that can be used is that first try to prove that the minimum Hamming weight of any code in C_H is 3. Then show that the minimum Hamming weight of a Hamming code is the same as its minimum distance.In fact, this is the case for every binary linear code. We will look into this when we define what a linear code is.

So we have found a code with a rate better than $C_{3,rep}$, with distance 3. Now the next question that naturally arises is, can we still do better?

4.3. Hamming Bounds

Well, if you guessed that this is the best we can do, you were right. But how do we show that we cannot obtain a better value of rate for distance 3? For the proof we require the definition, of something called the Hamming Ball.

Definition 4.12 (Support, Hamming Weight)**.** For $x = (x_1, x_2, \dots x_n) \in \{0, 1, \dots q-1\}^n$, the *support* of x is the set

$$\operatorname{supp}(x) \coloneqq \{i \in [n] \mid x_i \neq 0\}$$

and the *Hamming Weight* of x is the non-negative integer

$$\operatorname{wt}(x) \coloneqq |\operatorname{supp}(x)| = \big|\{i \in [n] \mid x_i \neq 0\}\big|$$

Definition 4.13 (Hamming Ball)**.** For any vector $x \in \{0, 1, \dots q-1\}^n$,

$$B(x, e) \coloneqq \{y \in \{0, 1, \dots q-1\}^n \mid \Delta(x, y) \leqslant e\}$$

is called the *Hamming Ball* of radius e around x.

Next we give an upper bound for the dimension of ***any*** code with distance 3.

Theorem 4.14 (Hamming Bound for $d = 3$)**.** Every binary code of block length n, dimension k and distance $d = 3$ satisfies

$$k \leqslant n - \log_2(n+1)$$

Proof. Given any 2 codewords $c_1, c_2 \in C$, we have $\Delta(c_1, c_2) \geqslant 3$. This implies that the Hamming Balls of radius 1 around c_1 and c_2 are disjoint, since they contain all codewords at distance at most 1 from c_1 and c_2 respectively. Now note that for all $x \in \{0,1\}^n$, $|B(x, 1)| = n + 1$ (why?). Now consider the union of all Hamming Balls of radius 1 around all codewords in C; their union is a subset of $\{0,1\}^n$. In other words, we have:

$$\left|\bigcup_{c \in C} B(c, 1)\right| \leqslant 2^n$$

Since the Hamming Balls of distinct codewords of radius 1 are disjoint, we have:

$$\begin{aligned}\left|\bigcup_{c \in C} B(c, 1)\right| &= \sum_{c \in C} |B(c, 1)| \\ &= \sum_{c \in C} (n+1) \\ &= |C|(n+1) \\ &= 2^k(n+1)\end{aligned}$$

Hence, we have:

$$2^k(n+1) \leqslant 2^n \implies k \leqslant n - \log_2(n+1)$$

□

Theorem 4.15 (Generalized Hamming Bound)**.** For every $(n, k, d)_q$ code, we have

$$k \leqslant n - \log_q \left(\sum_{i=0}^{\lfloor \frac{d-1}{2} \rfloor} \binom{n}{i} (q-1)^i \right)$$

Definition 4.16 (Perfect Code)**.** Codes that meet the Hamming Bound are called *Perfect Codes*.

In other words, a perfect code leads to the following perfect "packing": if one constructs Hamming Balls of radius $\left\lfloor \frac{d-1}{2} \right\rfloor$ around all the codewords, then we would cover the entire ambient space, i.e. every possible vector will lie in one of these Hamming balls. One example of perfect code is the Hamming Code described at the start of this section.

§5. Introduction to Finite Fields

Definition 5.1 (Group)**.** A *Group* $(G, \star)$ is a set G with a binary operation $\star \colon G \times G \to G$ that satisfies the following axioms:

(i) *Closure:* For every $a, b \in G$, the element $a \star b \in G$

(ii) *Associative:* For every $a, b, c \in G$, $(a \star b) \star c = a \star (b \star c)$

(iii) *Identity Element:* There is an identity element $e \in G$ such that for every $a \in G$, $e \star a = a \star e = a$

(iv) *Inverse Element:* For every $a \in G$, there exists an inverse element $(-a) \in G$ such that $a \star (-a) = (-a) \star a = 0$

Some basic examples of groups are the integers under addition, the set of nonzero integers under multiplication, the set of invertible matrices under matrix multiplication, and the symmetric group on a set. For a group G, the order of the group is the number of elements in the group. The order of an element $a \in G$ is the smallest positive integer n such that $a^n = e$, where e is the identity element of the group.

Definition 5.2 (Subgroup)**.** A *Subgroup* of a group is a subset of the group that is itself a group under the same operation.

For example, the set of even integers is a subgroup of the integers under addition. The set of invertible matrices is a subgroup of the set of all non-singular matrices under matrix multiplication. The set of all permutations of a set is a subgroup of the symmetric group on that set.

Definition 5.3 (Abelian Group)**.** A group $(G, \cdot)$ is said to be an *Abelian group* (or *commutative group*) if the group operation is commutative; that is, for all $a, b \in G$, we have

$$a \cdot b = b \cdot a.$$

Definition 5.4 (Field)**.** A *Field* $(F, +, \cdot, 0, 1)$ is a set F with two binary operations $+ \colon F \times F \to F$ and $\cdot \colon F \times F \to F$ that satisfy the following axioms:

(i) $(F, +)$ is an abelian group

(ii) $(F \setminus \{0\}, \cdot)$ is an abelian group

(iii) *Distributive Laws:* For every $a, b, c \in F$, $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(a + b) \cdot c = a \cdot c + b \cdot c$

Some standard examples of a field are the rational numbers, real numbers, complex numbers, and finite fields. The *characteristic of a field* is the smallest positive integer p such that $p \cdot 1 = 0$, where 1 is the multiplicative identity of the field. If no such integer exists, the field is said to have characteristic zero.

In this subsection, we will discuss some basic concepts of Finite Fields, which are used extensively in Coding Theory. Finite Fields are also known as Galois Fields, in honour of Évariste Galois. It is possible to get quite far treating finite fields as "black-boxes" that allow the field operations to be performed efficiently as atomic steps, along with just one important mantra:

A non-zero polynomial of degree d with coefficients from a field $\mathbb{F}$ has at most d roots in $\mathbb{F}$.

But it is nevertheless desirable to have a good working knowledge of the basics of the theory of finite fields, so here are some excellently done notes on finite fields, written by G. David Forney and available on MIT's OpenCourseWare for the course 6.451 Principles of Digital Communication II. These notes rigorously prove everything that we would need (and more!) from first principles, in a nice sequence.

Collected below are some basic results about finite fields, for quick reference. All these facts are proved in the above linked notes.

1. For every prime p, there is a unique finite field of size p that is isomorphic to $\mathbb{F}_p$ which is the set $\{0, 1, \ldots, p-1\}$ under mod-p addition and multiplication.
2. For each prime p, positive integer $m \geqslant 1$, and polynomial $g(X)$ with coefficients in $\mathbb{F}_p$ of degree m that is *irreducible* (in $\mathbb{F}_p[X]$), the set of polynomials in $\mathbb{F}_p[X]$ of degree at most $m-1$ with addition and multiplication of the polynomials defined modulo $g(X)$ is a finite field (denoted $\mathbb{F}_{g(X)}$) with p^m elements.
3. Every finite field is isomorphic to such a field, and therefore must have p^m elements for some prime p and positive integer m.
4. For every prime p and integer $m \geqslant 1$, there exists an irreducible polynomial $g(X) \in \mathbb{F}_p[X]$ of degree m. Therefore, there is a finite field with p^m elements for every prime p and positive integer m.
5. Additively, a finite field with p^m elements has the structure of a vector space of dimension m over $\mathbb{F}_p$.
6. The multiplicative group of a finite field (consisting of its non-zero elements) is cyclic. In other words, the non-zero elements of a field $\mathbb{F}$ can be written as $\{1, \gamma, \gamma^2, \ldots, \gamma^{|\mathbb{F}|-2}\}$ for some $\gamma \in \mathbb{F}$.
   - A γ with such a property is called a *primitive element* of the field $\mathbb{F}$.
   - A field $\mathbb{F}$ has $\varphi(|\mathbb{F}|-1)$ primitive elements, where $\varphi(\cdot)$ is the Euler's totient function.
7. All fields of size p^m are isomorphic to $\mathbb{F}_{g(X)}$ for an arbitrary choice of degree m irreducible polynomial $g(X) \in \mathbb{F}_p[X]$.

   The finite field with p^m elements is therefore unique up to isomorphism field and will be denoted by $\mathbb{F}_{p^m}$.

   Remark: *While one can pick any irreducible $g(X)$ to represent the field $\mathbb{F}_{p^m}$ as $\mathbb{F}_{g(X)}$, sometimes a special choice can be judicious. For example, the complexity of multiplication is better if $g(X)$ is sparse (i.e., has very few non-zero coefficients).*
8. The elements of $\mathbb{F}_{p^m}$ are the p^m distinct roots of the polynomial $X^{p^m} - X \in \mathbb{F}_p[X]$.
9. For each k dividing m, the field $\mathbb{F}_{p^m}$ has a unique subfield of size p^k, which consists of the roots of the polynomial $X^{p^k} - X$.
10. The polynomial $X^{p^m} - X$ is the product of all monic irreducible polynomials in $\mathbb{F}_p[X]$ whose degree divides m, with no repetitions.

References

[Gho17] Sudhir R. Ghorpade. Aspects of Coding Theory. Lecture notes for the AIS/IST on Gröbner Bases and Their Applications, IIIT Delhi, 2017, 2017. Lecture series from December 11–23, 2017. Adapted from lectures at the Advanced Instructional School on Algebraic Combinatorics, ISI Bangalore, 2013.

[GRS23] Venkatesan Guruswami, Atri Rudra, and Madhu Sudan. *Essential Coding Theory*. October 2023. Available at https://cse.buffalo.edu/faculty/atri/courses/coding-theory/book/web-coding-book.pdf.

[Hil86] Raymond Hill. *A First Course in Coding Theory*. Oxford University Press, Oxford, UK, 1986.

[Rat25] Param Rathour. Coding theory: A playful introduction. https://docs.google.com/presentation/d/1nhN2k6yBmH2nXc7uRguEb_kZ41vBd7c5/edit?usp=sharing, 2025. Presentation slides.