

Topic 1: Information Representation

Smarter compression, self-correcting data and codes that never glitch.

Optional stretch material. This goes beyond the 9618 syllabus, so you will not be asked about it directly in the exam. Its purpose is to deepen ideas you already know. **Builds on:** 1.1 Data representation, 1.3 Compression. Try the challenge questions on the last pages before you read the answers.

Huffman coding: shorter codes for common symbols

Run-length encoding squeezes repeated values. **Huffman coding** squeezes *frequent* values. Instead of giving every character the same number of bits, common characters get short codes and rare ones get long codes. The codes are **prefix-free**: no code is the start of another, so a stream of bits can be decoded without any separators.

Building the tree: list the symbols with their frequencies, then repeatedly merge the two lowest-frequency nodes into a new node whose frequency is their sum. Label left branches 0 and right branches 1. Ties can be broken either way; the total length comes out the same.

Symbol (from ABRACADABRA)	Frequency	Code	Bits used
A	5	0	$5 \times 1 = 5$
B	2	110	$2 \times 3 = 6$
R	2	111	$2 \times 3 = 6$
C	1	100	$1 \times 3 = 3$
D	1	101	$1 \times 3 = 3$

The 11 characters need **23 bits** instead of 88 bits in 8-bit ASCII (or 33 bits with a fixed 3-bit code). The catch: the receiver needs the code table (or tree), so Huffman works best on larger inputs. It is **lossless**, and it is one stage inside real formats such as ZIP/DEFLATE and JPEG.

Hamming code: fixing errors, not just spotting them

A parity bit can *detect* a single flipped bit but cannot say *which* bit flipped. **Hamming(7,4)** adds 3 parity bits to 4 data bits so that a single-bit error can be located and corrected. Bit positions are numbered 1 to 7. Parity bits sit at the power-of-two positions (1, 2, 4); data bits fill positions 3, 5, 6, 7. Each parity bit checks a different group of positions (even parity):

Parity bit	Positions it checks
p1 (position 1)	1, 3, 5, 7
p2 (position 2)	2, 3, 6, 7
p4 (position 4)	4, 5, 6, 7

Worked example. Data bits 1 0 1 1 go to positions 3, 5, 6, 7. p1 covers 1,0,1 so p1 = 0. p2 covers 1,1,1 so p2 = 1. p4 covers 0,1,1 so p4 = 0. Sent: 0110011.

Receiving. Suppose position 6 flips: 0110001. Recheck each group: group 1 passes, group 2 fails, group 4 fails. Reading the failures as a binary number (p4 p2 p1 = 110) gives **6**: the faulty position. Flip it back. This is how ECC memory in servers repairs single-bit errors from cosmic rays.

Gray code: one bit at a time

In ordinary binary, $011 \rightarrow 100$ changes three bits at once. A sensor reading a rotating dial at that instant could briefly see 111 or 000: a wild glitch. **Gray code** orders values so that neighbours differ by exactly one bit, so a misread is off by at most one step.

Decimal	Binary	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

To convert binary to Gray, XOR the number with itself shifted right one place: $\text{gray} = b \text{ XOR } (b \gg 1)$. Gray code is used in rotary encoders and also appears in Karnaugh map axes (topic 15).

Challenge questions

1. Build a Huffman code for MISSISSIPPI ($M = 1, I = 4, S = 4, P = 2$). How many bits does the message need, compared with 8-bit ASCII?
2. Encode the data bits 1 1 0 0 as Hamming(7,4) (data in positions 3, 5, 6, 7). The word 0111101 is then received. Which position is wrong?
3. Convert binary 110 to Gray code, then decode the Gray code 111 back to binary. Why do rotary encoders use Gray code?

Worked answers

Only look once you have had a proper go at each question.

1. Merge $M(1)+P(2) = 3$. Merge 3 + one of the 4s = 7. Merge the other 4 + 7 = 11. One valid code: $S = 0$, $I = 10$, $M = 110$, $P = 111$.

Bits: $S \ 4 \times 1 + I \ 4 \times 2 + M \ 1 \times 3 + P \ 2 \times 3 = 4 + 8 + 3 + 6 = \mathbf{21 \ bits}$. ASCII needs $11 \times 8 = 88$ bits.

2. p_1 covers positions 3,5,7 = 1,1,0 so $p_1 = 0$. p_2 covers 3,6,7 = 1,0,0 so $p_2 = 1$. p_4 covers 5,6,7 = 1,0,0 so $p_4 = 1$. Codeword: 0111100.

Received 0111101: group 1 (positions 1,3,5,7 = 0,1,1,1) has odd parity: fail. Group 2 (2,3,6,7 = 1,1,0,1) fail. Group 4 (4,5,6,7 = 1,1,0,1) fail. 111 in binary = **position 7**, so the last bit flipped.

3. $110 \text{ XOR } 011 = \mathbf{101}$.

Decoding 111: the top bit stays 1; next bit = $1 \text{ XOR } 1 = 0$; last bit = $0 \text{ XOR } 1 = 1$, giving **101** (5).

Only one bit changes between neighbouring positions, so a reading taken mid-transition is wrong by at most one step instead of a random value.