Data representation

IGCSE Computer Science

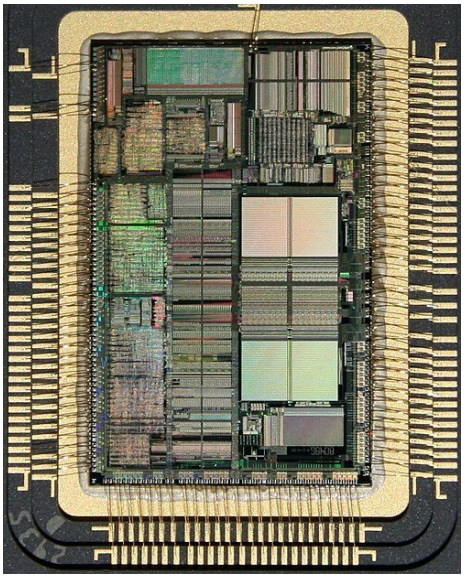
Why computers use binary

Digital computers represent data using two logical states, written as 1 and 0. A system that uses only two digits 数字 is called **binary** 二进制 (base 2).



Computers represent all data – numbers, text, sound and images – as binary strings of 0s and 1s

Every kind of data 数据 – numbers, text, sound and images – must be changed into binary before a computer can use it. The computer processes this binary using **logic gates** 逻辑门, and stores it in **registers** 寄存器 (small, fast stores inside the processor 处理器).



A microprocessor holds millions of tiny transistors, each a switch that is on (1) or off (0) – the physical basis of binary

Number systems

A **number system** 数制 is a way of writing numbers using a fixed set of digits. You need three of them.

System	Base	Digits used
Denary	10	0–9
Binary	2	0 and 1
Hexadecimal	16	0–9 then A–F

- **denary** 十进制 is the normal counting system (also called decimal).
- **binary** uses only 0 and 1.
- **hexadecimal** 十六进制 (hex) uses sixteen digits: 0–9, then A, B, C, D, E, F stand for 10, 11, 12, 13, 14, 15.

The **base** 基数 tells you how many different digits a system uses.

Place value

Each column in a number has a **place value** 位值. In binary the place values double from right to left. For an 8-bit number they are:

```
128  64  32  16  8  4  2  1
```


exercises.lol exercises.com.cn

place value	128	64	32	16	8	4	2	1
bits	**1**	0	0	**1**	0	**1**	**1**	0

$150 = 128 + 16 + 4 + 2 \Rightarrow$ `10010110`

An 8-bit place-value chart: the 1s sit under the values that add up to 150

One **bit** 位 is a single 0 or 1. Eight bits make one **byte** 字节. Four bits (half a byte) is a **nibble** 半字节.

■ Converting between number systems

Denary → binary. Write the place values. Put a 1 under each value you need so they add up to your number; put 0 under the rest.

Example: change denary 150 to binary. $150 = 128 + 16 + 4 + 2$.

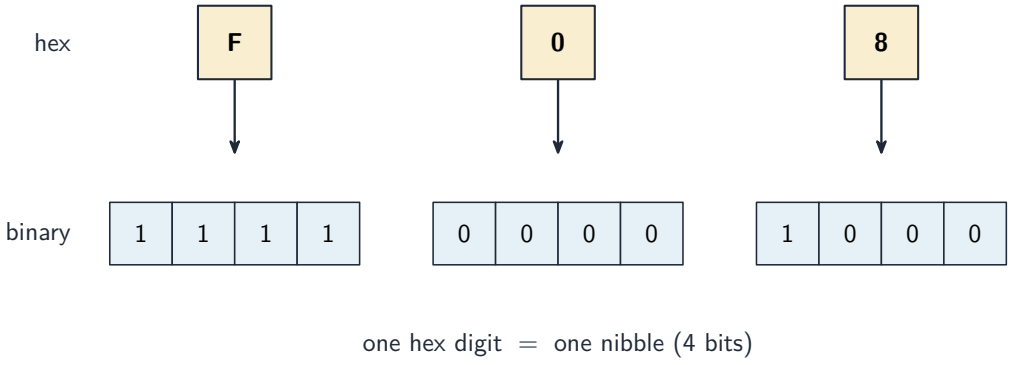
```
128 64 32 16 8 4 2 1
  1  0  0  1 0 1 1 0
```

So 150 = `10010110`.

Binary → denary. Add the place values where there is a 1. `10010110` $= 128 + 16 + 4 + 2 = 150$.

Hexadecimal → binary. Change each hex digit into its own 4-bit group (a nibble).

Example: hex F08. $F = 1111$, $0 = 0000$, $8 = 1000$, so F08 = `1111 0000 1000`.



Each hex digit maps to its own 4-bit nibble – F08 = 1111 0000 1000

Binary → hexadecimal. Group the bits into nibbles of 4, starting from the right. Change each nibble to one hex digit.

Denary → hexadecimal. The easy way is to change to binary first, then binary to hex.

exercises.lol exercises.com.cn

This table helps with the hex letters:

Denary	Binary	Hex
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Cambridge questions use binary numbers up to 16 bits long.

> **Worked example.** Convert denary 100 to 8-bit binary, then to hexadecimal.

$100 = 64 + 32 + 4$, so the binary is `01100100`. Splitting into nibbles, `0110 0100` = 6 and 4, so the hexadecimal is `64`.

> **Worked example (hexadecimal → denary).** Convert hex `2C` to denary. The hex place values are 16 and 1. `2` is worth $2 \times 16 = 32$ and `C` is 12, so `2C` $= 32 + 12 = 44$. Check through binary: `0010 1100` $= 32 + 8 + 4 = 44$.

■ Why hexadecimal is used

Hex is shorter than binary and easier for people to read and write. One hex digit replaces 4 binary digits, so you make fewer mistakes. The value does not change – hex is just a shorter way to show the same binary.

Computer scientists use hex for:

- MAC addresses and IPv6 addresses
- colour codes in HTML (for example `#FF0000` is red)
- **memory addresses** 内存地址 and error codes
- showing the contents of memory (a "memory dump")

Binary addition

You can add two 8-bit binary numbers, column by column from the right, just like denary. The rules for one column are:

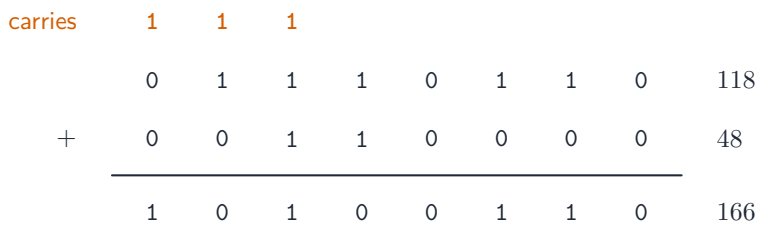
A	B	Result bit	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

When a carry also comes in, $1 + 1 + 1 = 1$ with a carry of 1.

exercises.lol exercises.com.cn

Example: add `01110110` (118) and `00110000` (48).

```
  0 1 1 1 0 1 1 0    (118)
+ 0 0 1 1 0 0 0 0    (48)
--------------------
  1 0 1 0 0 1 1 0    (166)
```

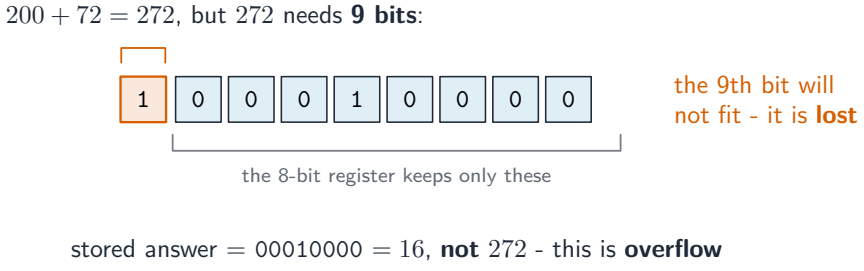


Adding column by column; the carries ripple to the left. 118 + 48 = 166

Overflow

When interpreted as an **unsigned** integer, an 8-bit register holds values from 0 to 255. If an addition gives a result above 255, the answer needs a 9th bit. The register cannot hold this extra bit, so it is lost. This is called **overflow** 溢出 (an overflow error). It happens when a value goes outside the limit the register can store.

Example: `11001000` (200) + `01001000` (72) = 272. In binary that is `1 00010000`, which needs 9 bits. The leading 1 will not fit in 8 bits, so the stored answer is wrong.



Adding 200 and 72 needs 9 bits, but an 8-bit register drops the ninth, so the answer is wrong

Logical binary shift

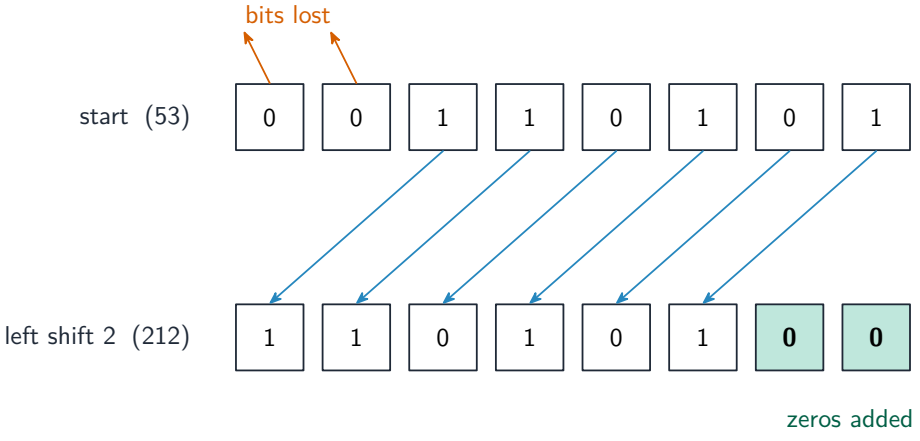
A **logical binary shift** 逻辑二进制移位 moves all the bits left or right by a number of places.

- Bits that move off the end of the register are **lost**.
- **Zeros** are added at the empty end.

For an unsigned integer, a **left shift** multiplies by 2 for each place moved **if no non-zero bit is lost**. A logical **right shift** gives the whole-number result of division by 2 for each place; the right-most bits (the **least significant bit(s)** 最低有效位) are lost.

Example: left shift `00110101` (53) by 2 places.

```
start:          0 0 1 1 0 1 0 1
left shift 2:   1 1 0 1 0 1 0 0
```



A left shift of 2: every bit moves 2 places left, the top bits are lost and zeros fill the right

The result is `11010100` (212), which is 53×4. The two left-most bits were lost and two zeros came in on the right. If a 1 is pushed off the end, that information is gone for good.

Worked example (right shift). Shift `10110100` (180) **right** by 3 places. Give the result in binary and denary, and say what effect the shift has.

```
start:          1 0 1 1 0 1 0 0
right shift 3:  0 0 0 1 0 1 1 0
```

The three right-most bits (`100`) are lost and three zeros enter on the left: `00010110` $= 16 + 4 + 2 = 22$. Shifting right 3 places divides by $2^3 = 8$: $180 \div 8 = 22.5$, and the shift keeps only the whole-number part, 22. The 0.5 was carried by the lost 1 bit.

> **Worked example (a left shift that goes wrong).** Shift `01100100` (100) left by 2 places. The result is `10010000` = 144, not 400: a 1 was pushed off the left end, and 400 is too large for 8 bits anyway. A left shift multiplies correctly only while no 1 bits are lost.

Two's complement

So far the numbers were positive. **Two's complement** 补码 lets an 8-bit register hold negative numbers too.

In two's complement, the left-most bit (the **most significant bit** 最高有效位, or MSB) has a *negative* place value:

```
-128  64  32  16  8  4  2  1
```

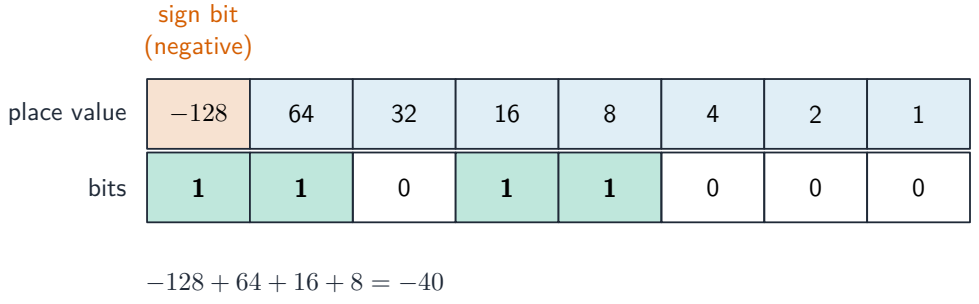
- If the MSB is 0, the number is zero or positive.
- If the MSB is 1, the number is negative.

To make a positive number negative: write the positive binary, flip every bit (0↔1), then add 1.

Example: make −40.

- +40 = `00101000`
- flip the bits = `11010111`
- add 1 = `11011000`

So −40 = `11011000`. Check by adding the place values: $-128 + 64 + 16 + 8 = -40$.



The most significant bit is worth −128, so 11011000 = −128 + 64 + 16 + 8 = −40

To read a negative two's complement number, just add the place values (the MSB counts as −128). The range of an 8-bit two's complement number is −128 to +127.

> **Worked example.** Convert the two's complement number `10011010` to denary. The MSB is 1, so the number is negative: $-128 + 16 + 8 + 2 = -102$. Check by the other method: flip the bits (`01100101`) and add 1 (`01100110` = 102), so the number is −102.

exercises.lol exercises.com.cn

Representing text

Computers store text by giving every character a number, then storing that number in binary. The set of characters a computer can use, together with their numbers, is a **character set** 字符集.

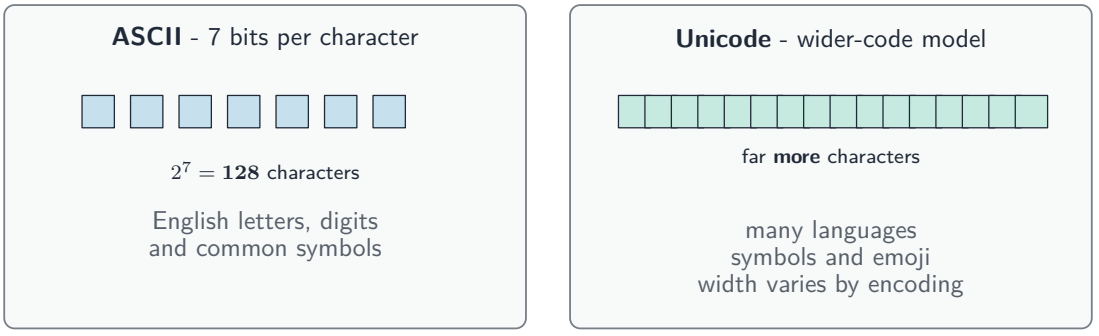
ASCII uses 7-bit character codes, giving $N = 2^b = 2^7 = 128$ different codes. It includes English letters, digits and common symbols; not all 128 codes are letters. **Unicode** represents a much greater range of characters, including different languages and **emoji** 表情符号.

Try it – encode a short message. This extract from ASCII gives A = 65, B = 66, C = 67 and space = 32. Write the four 7-bit groups for `CAB ` , including its final space. Then decode the groups `1000010 1000001 1000011`.

Worked answer. Look up each character, then convert its code to binary:

Character	Code	Seven bits
C	67	1000011
A	65	1000001
B	66	1000010
space	32	0100000

So `CAB ` is `1000011 1000001 1000010 0100000`. The second message decodes to `BAC`. A space has a code too; it is not an absence of stored data. The sender and receiver must use the same character-code mapping to interpret the message correctly.



ASCII has 128 codes. Unicode supports a much greater range; the wider row illustrates the syllabus model.

Why a larger range matters. ASCII cannot directly represent a Chinese name. Unicode includes Chinese characters as well as English characters and many other languages. For the syllabus comparison, explain that a greater range needs more distinct binary codes, requiring more bits per character than the 7-bit ASCII model.

Try it – compare stated storage models. Store the four characters `CAB ` using (A) 7 bits per character and (B) 16 bits per character. Assume the bits are packed with no padding or extra file information. Find the text data size in each model. The

exercises.lol exercises.com.cn

16-bit value is an assumption for this example, not a rule that all Unicode characters take 16 bits.

Worked answer. With the same number of characters, multiply by the stated bits per character:

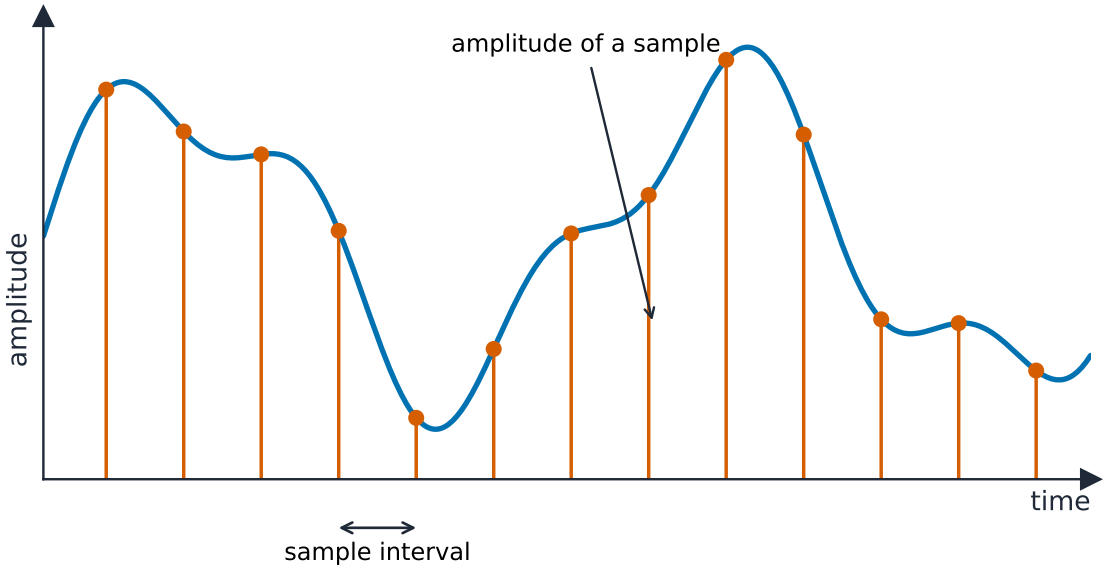
$$S = nb = 4 \times 7 \text{ bits} = 28 \text{ bits}$$

$$S = nb = 4 \times 16 \text{ bits} = 64 \text{ bits}$$

Model B uses more **storage** 存储 for this example because its stated code width is larger. Use the width given in an exam question. In real files, the encoding also matters: UTF-8 stores ASCII characters in the same bytes as an ASCII file, so Unicode text does not always occupy more bytes. You do not need the detailed UTF-8 rules for this syllabus comparison.

Representing sound

A **sound wave** 声波 is smooth and always changing. To store it, the computer measures the height of the wave at regular moments. This is called **sampling** 采样, and each measurement is a sample.



Sampling records the wave's height (amplitude) at regular moments

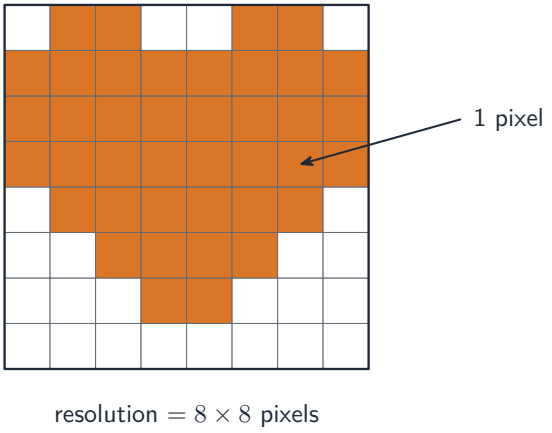
- **sample rate** 采样率 is the number of samples taken each second (measured in Hz).
- **sample resolution** 采样分辨率 is the number of bits used for each sample. The height of the wave at a sample point is its **amplitude** 振幅.

exercises.lol exercises.com.cn

A higher sample rate and a higher sample resolution give a more accurate recording, but a larger file.

Representing images

A computer image is made of a grid of small dots called **pixels** 像素.



A bitmap is a grid of pixels; the resolution is how many pixels it has

- **resolution** 分辨率 is the number of pixels in the image (for example 1920×1080).
- **colour depth** 颜色深度 is the number of bits used to store the colour of each pixel.

A higher resolution and a higher colour depth give a better-quality image, but a larger file.

The colour depth decides how many different colours each pixel can be: with n bits there are 2^n colours. A colour depth of 1 bit gives 2 colours (black and white), 8 bits give $2^8 = 256$ colours, and 24 bits give over 16 million.

> **Worked example.** An image uses 64 different colours. What is the smallest colour depth it can have? Find the smallest n with $2^n \geqslant 64$: $2^6 = 64$, so **6 bits** per pixel. If it used 65 colours it would need 7 bits, because 6 bits can only make 64 different patterns.

Measuring data storage

Data storage is measured in the units below. A nibble is 4 bits and a byte is 8 bits; from the kibibyte upward, each unit is 1024 times the one before it (because $1024 = 2^{10}$, which fits binary).

exercises.lol exercises.com.cn

Unit	Equals
bit	a single 0 or 1
nibble	4 bits
byte	8 bits
kibibyte (KiB)	1024 bytes
mebibyte (MiB)	1024 KiB
gibibyte (GiB)	1024 MiB
tebibyte (TiB)	1024 GiB
pebibyte (PiB)	1024 TiB
exbibyte (EiB)	1024 PiB



Hard-disk platters: storage is measured in bytes – knowing file size needs width × height × colour depth for images

Calculating file size

Image file size (in bits) = resolution × colour depth = width × height × colour depth.

Example: an image is 1024 × 1024 pixels with a colour depth of 2 bytes (= 16 bits).

- bits = $1024 \times 1024 \times 16 = 16\,777\,216$ bits
- bytes = $\div 8 = 2\,097\,152$ bytes
- KiB = $\div 1024 = 2048$ KiB
- MiB = $\div 1024 = 2$ MiB

exercises.lol exercises.com.cn



1024 × 1024 pixels × 16 bits; divide by 1024, not 1000, after the bytes

Bits to bytes is divide by 8; every step after that is divide by 1024

Sound file size (in bits) = sample rate × sample resolution × length in seconds.

These formulas count uncompressed data without file headers. The sound formula is for one channel; multiply by the channel count if a question specifies more channels.

Always divide by 1024 (not 1000) to change to KiB, MiB and so on. Give your answer in the unit the question asks for.

> **Worked example.** A sound is recorded for 30 seconds at a sample rate of 8,000 Hz with a sample resolution of 16 bits. Find the file size in kibibytes (KiB).

- bits = $8\,000 \times 16 \times 30 = 3\,840\,000$ bits
- bytes = $3\,840\,000 \div 8 = 480\,000$ bytes
- KiB = $480\,000 \div 1024 \approx 469$ KiB

Compression

Compression 压缩 makes a file smaller. A smaller file:

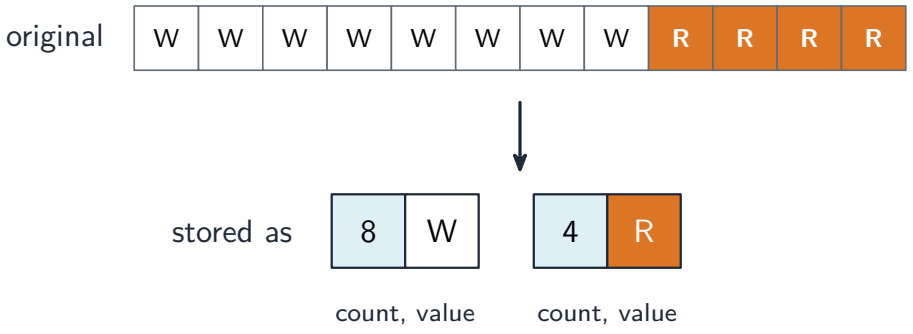
- uses less storage space,
- needs less **bandwidth** 带宽 (the amount of data a connection can carry),
- takes a shorter time to send (a shorter **transmission** 传输 time).

There are two types.

■ Lossless compression

Lossless 无损 compression makes the file smaller with **no permanent loss** of data. The original file can be rebuilt exactly.

One method is **run-length encoding** 行程编码 (RLE). It replaces a run of repeated values with one copy of the value and a count of how many times it repeats. For example `WWWWWWWW` (8 whites) is stored as "8 W". This works well when data has many repeats.



Run-length encoding stores each run once as a count and a value

> **Worked example.** One row of a black-and-white image is `B B B B B W W B B B B B B B B B` (16 pixels). Compress it with RLE, and say whether it saved space if each count and each colour is stored in one byte.

The runs are 5 black, 2 white, 9 black, so the RLE data is `5 B 2 W 9 B`: 6 bytes instead of 16 bytes for the original (one byte per pixel). It saves space because the row has long runs. A row that changes colour at every pixel, `B W B W …`, would become `1 B 1 W 1 B 1 W …`, **twice** as large as the original: RLE only helps when data repeats.

■ Lossy compression

Lossy 有损 compression makes the file much smaller by **permanently removing** some data. The removed data cannot be got back. For example:

- reducing the resolution or colour depth of an image,
- reducing the sample rate or sample resolution of a sound.

Use lossless when you must keep every detail (text and program files). Use lossy for photos, music and video, where a small loss of quality is worth a much smaller file.

Exam tips

- Convert denary → binary by subtracting the place values (128, 64, 32 …); binary → denary by adding the place values that hold a 1.
- To convert to hex, group the binary into **nibbles** of 4 bits from the right; each nibble is exactly one hex digit.
- **Overflow** happens when a result is outside the chosen representation's range. Eight-bit unsigned integers hold 0–255; eight-bit two's complement holds −128–127.
- File size in bits: for an image, width × height × colour depth; for sound, sample rate × resolution × seconds. Divide by 8 for bytes, then by 1024 for each larger unit.

exercises.lol exercises.com.cn

- **Lossless** compression keeps every bit (text; run-length encoding); **lossy** permanently removes data (photos, music) for a much smaller file.

exercises.lol exercises.com.cn