Course Companion

A-Level Computer Science

Every topic handout in one volume

全部专题讲义合集

exercises.lol

Contents 目录

1. Information representation 3

2. Communication 19

3. Hardware 40

4. Processor Fundamentals 62

5. System Software 81

6. Security, privacy and data integrity 95

7. Ethics and Ownership 109

8. Databases 121

9. Algorithm Design and Problem-solving 137

10. Data Types and Structures 152

11. Programming 170

12. Software Development 195

13. Data Representation 211

14. Communication and internet technologies 230

15. Hardware and Virtual Machines 244

16. System Software 261

17. Security 278

18. Artificial Intelligence (AI) 290

19. Computational thinking and Problem-solving 306

20. Further Programming 328

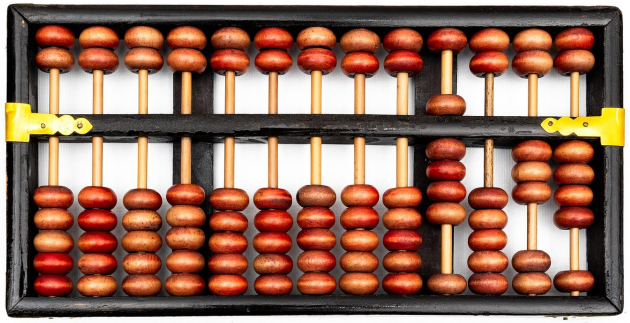
Information representation

A-Level Computer Science

Number systems

The three **number systems** 数制 you must use:

- **denary** 十进制 (decimal, base 10) – uses digits 0–9. Place values are powers of ten.
- **binary** 二进制 (base 2) – uses 0 and 1. Place values are powers of two. Every **byte** 字节 is 8 **bits** 位.
- **hexadecimal** 十六进制 (base 16) – uses 0–9 then A–F for 10–15. Each hex **digit** 数位 stands for exactly 4 bits.



An abacus represents numbers by place value – the same idea behind decimal, binary and hexadecimal

Conversions

Denary → binary: keep dividing by 2 and record the remainders, read bottom-up. Or subtract the largest **place value** 位值 (power of 2) that fits.

Example: 558_{10}: $558 = 512 + 32 + 8 + 4 + 2 = 2^9 + 2^5 + 2^3 + 2^2 + 2^1$. In 12 bits: `0010 0010 1110`.

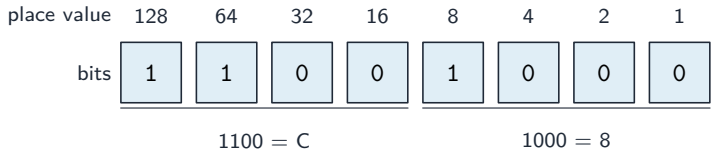
Binary → hex: group the bits into **nibbles** 半字节 (4 bits) from the right and convert each. `0010 0010 1110` → `2 2 E` → `22E`.

Hex → binary: replace each hex digit with its 4-bit pattern. **Hex → denary**: multiply each digit by its place value. `22E` $= 2 \times 256 + 2 \times 16 + 14 = 558$.

exercises.lol exercises.com.cn

Worked example. Convert denary 200 to 8-bit binary, then to hexadecimal.

$200 = 128 + 64 + 8$, so the binary is `11001000`. In nibbles, `1100 1000` = 12 and 8, i.e. C and 8, so the hexadecimal is `C8`.



$$200 = 128 + 64 + 8 = \texttt{11001000}_2 = \texttt{C8}_{16}$$

Reading 200 from its place values, then grouping the bits into nibbles to get hex C8

How many bits?

Exam questions fix the **register width** 寄存器宽度 (8, 12 or 16 bits). Pad with leading zeros to that width: 558 in 12 bits is `0010 0010 1110`, never `10 0010 1110`.

To find the **minimum number of bits** that can store a value, ask which place values you need:

- an unsigned integer from 0 to $2^n - 1$ needs n bits: 200 needs 8 bits (the top is 255), 1000 needs 10 bits (the top is 1023), 16 needs 5 bits (4 bits stop at 15).
- a signed two's-complement integer from -2^{n-1} to $2^{n-1} - 1$ needs n bits: -200 needs 9 bits, because 8 bits stop at -128.
- one hexadecimal digit needs 4 bits, one BCD digit needs 4 bits, and one ASCII character needs 7 bits (8 for extended ASCII).

Binary vs decimal prefixes

Two prefix families look similar but differ – decimal (powers of 10) and binary (powers of 2):

Decimal (SI)	Binary (memory)
kilo $= 10^3$	kibi (Ki) $= 2^{10} = 1024$
mega $= 10^6$	mebi (Mi) $= 2^{20}$
giga $= 10^9$	gibi (Gi) $= 2^{30}$
tera $= 10^{12}$	tebi (Ti) $= 2^{40}$

So a tebibyte (TiB) is slightly more than a terabyte (TB). A "1 TB" drive holds 10^{12} bytes, but an operating system that reports in TiB shows a smaller number.

Binary arithmetic

Binary addition

Add column by column from the right, carrying as in denary:

carries	1	1	1						
	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

Bit A	Bit B	Carry in	Sum bit	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	1	0	0	1
1	1	1	1	1

Overflow 溢出 happens when the result needs more bits than the **register** 寄存器 can hold – the carry-out of the leftmost column is the overflow bit.

> **Worked example.** Add the 8-bit unsigned integers 10110101 and 01101100, and comment on the result.

10110101 + 01101100 = 1 00100001. The answer needs 9 bits, so it does not fit in an 8-bit register: overflow has occurred. A full answer names the error and says why, using the word size the question gave: "Overflow: the true result (289) is larger than the largest value an 8-bit register can hold (255), so the carry out of the most significant bit is lost and the stored result (00100001 = 33) is wrong."

Binary subtraction

The usual way is **two's complement** 补码 addition: to do $A - B$, form the two's complement of B (invert every bit and add 1), then add, and discard any final carry-out.

To subtract 00011110 from 01100100 (unsigned 8-bit):

- two's complement of 00011110: invert → 11100001, add 1 → 11100010.
- add to 01100100: result 1 01000110 (9 bits) – discard the leading 1 → 01000110 = 70_{10}. Check: 100 − 30 = 70. ✓

Two's complement signed integers

In an n-bit two's-complement number:

- the **most significant bit** 最高有效位 (MSB) is the **sign bit** 符号位: 0 = positive, 1 = negative.
- to read a negative number: invert every bit, add 1, then negate.

So 11100010 is negative; invert → 00011101, add 1 → 00011110 = 30, so it is −30. This is a **signed integer** 有符号整数 (unlike an **unsigned** 无符号 one). The range for n bits is -2^{n-1} to $+2^{n-1} - 1$; for 8 bits, −128 (10000000) to +127 (01111111).

The same bits mean different numbers depending on the agreed reading. As an **unsigned** integer every bit is a place value, so 8 bits run from 0 to 255; as a **signed** two's-complement integer the top bit is the sign, so the same 8 bits run from −128 to +127. The pattern 11111111 is 255 read one way and −1 read the other – nothing in the bits themselves says which.

8 bits	as unsigned every bit a place value	as signed top bit is the sign
00000000	0	0
01111111	127	+127
10000000	128	−128
11111111	255	−1
range for n bits	$0 \ldots 2^n - 1$	$-2^{n-1} \ldots 2^{n-1} - 1$

The same byte read as unsigned and as signed: only the agreed interpretation tells them apart  *8-bit two's complement: the sign bit splits the range into negative (−128 to −1) and positive (0 to 127)*

> **Worked example.** What denary value does the 8-bit two's-complement number 10110100 represent?

The MSB is 1, so it is negative. Invert → 01001011, add 1 → 01001100 = 76, so the value is −76. Check with place values: −128 + 32 + 16 + 4 = −76.

> **Worked example.** Write −108 as a 12-bit two's-complement integer.

Start from +108 in 12 bits: 108 = 64 + 32 + 8 + 4, so `0000 0110 1100`. Invert every bit: `1111 1001 0011`. Add 1: `1111 1001 0100`. Check with place values, where the top bit is worth $-2^{11} = -2048$: −2048 + 1024 + 512 + 256 + 128 + 16 + 4 = −108. ✓

For 12 bits the range is −2048 (`1000 0000 0000`) to +2047 (`0111 1111 1111`). Questions that ask for the smallest and largest values want these two patterns, so learn

exercises.lol exercises.com.cn

the rule: the most negative number is a 1 followed by zeros; the most positive is a 0 followed by ones.

An **arithmetic shift** 算术移位 moves every bit left or right but keeps the sign: a shift right by one place halves the value and copies the sign bit into the empty space on the left, so a negative number stays negative (`1111 1001 0100` shifted right three places is `1111 1111 0010`, which is −14: $-108/8 = -13.5$, and a shift right rounds down). A shift left doubles the value. Shifts belong to the assembly instruction set in topic 4, but this question is asked with the number work here.

Overflow in signed arithmetic happens when the true result falls outside this range – spotted when the sign bit flips wrongly (two positives giving a negative, or two negatives giving a positive).

One's complement

Before two's complement, an older scheme called **one's complement** 反码 represented a negative number by simply **inverting every bit** of the positive – there is no "add 1" step.

- $+30 = 00011110$, so in one's complement $-30 = 11100001$ (just the inverse).
- Drawback: it has **two zeros** – 00000000 (+0) and 11111111 (−0) – which wastes a bit pattern and makes arithmetic awkward.

Two's complement (invert **and** add 1) removes the negative zero: it has a single zero and lets addition and subtraction use the same circuit. That is why modern computers store signed integers in two's complement, not one's complement.

Binary Coded Decimal (BCD)

In **BCD** 二进码十进数, each denary digit is written as its own 4-bit pattern. The number 93 is `1001 0011` in BCD – not binary 93 (01011101). Each nibble uses only 0–9; patterns 1010–1111 are invalid.

BCD reading: `0010 0111 0101` → 2, 7, 5 → 275.

Use: calculators, digital clocks, and devices that show denary digits – each digit drives a **7-segment display** 七段显示器. Currency code often uses BCD to avoid the rounding errors of converting fractions like 0.1 to binary.

A "justify" answer must link the use to a property of BCD: each denary digit has its own 4 bits, so a digit can be sent straight to its display, or added digit by digit, with no conversion of the whole number; and a decimal fraction such as 0.10 is stored exactly, which a binary fraction cannot do.

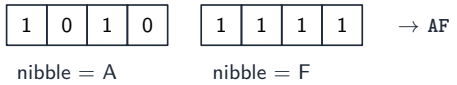
exercises.lol exercises.com.cn



A seven-segment display shows one denary digit, often driven by BCD

Hexadecimal – practical uses

Hex is a compact way to write binary (1 hex digit = 4 bits):



A byte is two nibbles; each nibble is one hex digit

- **memory addresses** 内存地址 in low-level programming – `0x7FFE`.
- **colour values** in HTML/CSS – `#FF8800`.
- **MAC addresses** – `AC:DE:48:00:11:22`.

Hex does not change the stored data – it just makes binary easier for humans.

Character codes

Computers store text as numbers; each character has a numeric **code point** 码点 set by a **character set** 字符集.

ASCII

- **ASCII** uses 7 bits – 128 code points. Basic Latin letters, digits, punctuation, and control codes.
- **Extended ASCII** uses 8 bits – 256 code points; the lower 128 match ASCII, the upper 128 vary by region.

character	code (denary)	binary
A	65	01000001
a	97	01100001
0	48	00110000
(space)	32	00100000

Each character is stored as a number – a few ASCII code points in denary and binary

Unicode

- **Unicode** is a universal character set covering almost every script, plus symbols and emoji.
- common **encodings** 编码: **UTF-8** (1–4 bytes, ASCII-compatible), **UTF-16** (2 or 4 bytes), **UTF-32** (fixed 4 bytes).

Why Unicode beats ASCII

- it represents **far more characters** (every script, emoji); ASCII covers only basic English.
- files are **portable** with no code-page confusion, and allow **multilingual** text in one document.
- trade-off: Unicode files are usually **larger** for English-only text.

When a question asks for **differences**, give them in pairs with numbers: ASCII uses 7 bits (extended ASCII 8), so 128 (256) characters; Unicode uses up to 32 bits (UTF-8 uses 1 to 4 bytes), so more than a million code points. ASCII covers basic English only; Unicode covers every script, and its first 128 code points are the ASCII ones. In UTF-8 an English letter still takes 1 byte, so a 40-letter English file name is 40 bytes in ASCII and in UTF-8 alike, while a Chinese character takes 3 bytes.

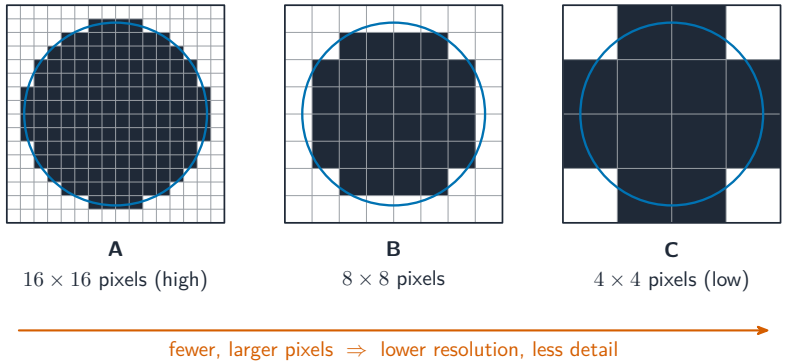
Bitmap images

A **bitmap** 位图 image (also called a **bitmapped image**) stores the colour of every **pixel** 像素 in a grid. At the start of the file a **file header** 文件头 records the image's metadata – its width, height and colour depth – so software knows how to read the pixel data that follows.

- **image resolution** 图像分辨率: the bitmap's own size, width × height in pixels (e.g. 1920 × 1080).
- **screen resolution** 屏幕分辨率: the width × height the **display** can show. If an image's resolution is larger than the screen it is scaled down to fit; a low-resolution image looks blocky when stretched onto a higher-resolution screen.

exercises.lol exercises.com.cn

- **colour depth** 颜色深度 (**bit depth** 位深度): bits per pixel. 1 bit → black/white; 8 bits → 256 colours; 24 bits → 16.7 million ("true colour").

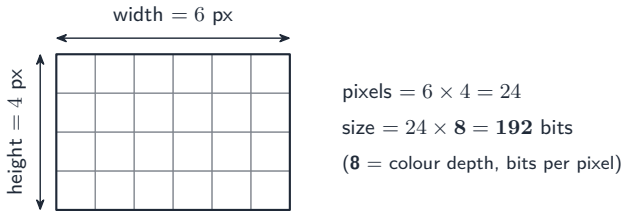


The same image stored at three resolutions, from high (A) to low (C): fewer, larger pixels mean less detail

File size

$$\text{size in bits} = \text{width} \times \text{height} \times \text{bit depth}.$$

Divide by 8 for bytes, by 1024 for KiB, etc. Example: a 3000 × 2000 image at 24 bpp is $3000 \times 2000 \times 24 = 1.44 \times 10^8$ bits ≈ 17.2 MiB.



The same formula on small numbers: count the pixels, then multiply by the colour depth

State the units you used. The mark scheme accepts 1 MB = 10^6 bytes (the SI prefix) or 1 MiB = 1024 × 1024 bytes (the binary prefix), as long as your working shows which one; the same image is 18.0 MB or 17.2 MiB. Add the size of the file header if the question gives one.

A video is a sequence of bitmap images, each one a **frame** 帧. Before compression its size is the size of one frame × the **frame rate** 帧率 (frames per second) × the duration in seconds: 30 frames per second of 1920 × 1080 pixels at 24 bits is 30 ×

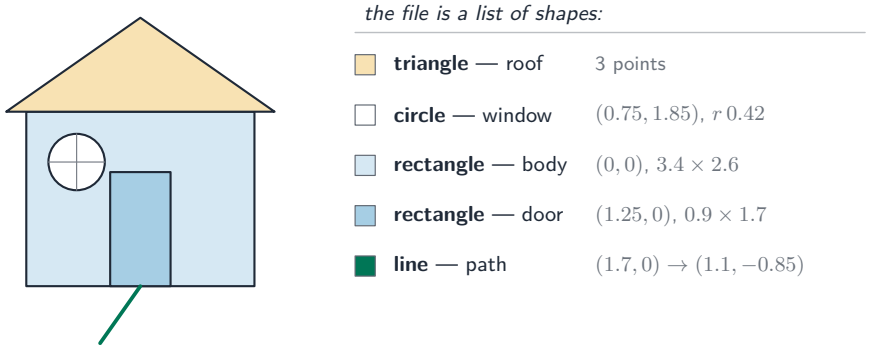
$1920 \times 1080 \times 24 \approx 1.5 \times 10^9$ bits, about 187 MB, for every second. That is why video is always compressed.

Changing settings

- **lower resolution** → smaller file, less detail (looks blocky when enlarged).
- **lower colour depth** → smaller file, but smooth shades show banding.
- **higher** of either → larger file, better quality.

Vector graphics

A **vector graphic** 矢量图形 stores the **instructions** to draw the image as a **drawing list** 绘图列表 – an ordered list of **drawing objects** 绘图对象 (geometric **primitives** 图元: lines, curves, polygons, circles). Each drawing object has **properties** 属性 such as colour, fill, line width and position (coordinates). To show it, the program **renders** 渲染 the drawing list at any resolution needed.



A vector image is built from labelled geometric shapes, each with attributes

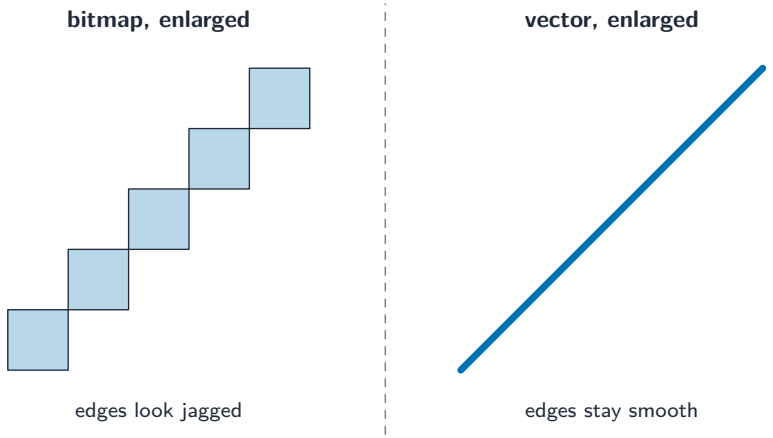
Bitmap vs vector

Task	Better choice	Why
Photograph	Bitmap	Complex pixel-level detail can't be described as shapes.
Logo, icon, sign	Vector	Sharp edges; scales to any size without blur.
Engineering drawing	Vector	Precise geometry and scaling.
Painting, texture	Bitmap	Smooth tonal detail per area.

exercises.lol exercises.com.cn

Vector advantage: it **scales without losing quality** – a vector logo stays sharp at any size, while a bitmap blurs when enlarged. **Vector disadvantage**: it cannot describe arbitrary pixel detail (photographs).

A "justify" answer links the choice to the task. "The logo must appear on a business card and on a billboard, so it should be a vector graphic: it is stored as drawing objects and is re-rendered sharply at any size, whereas a bitmap would show its pixels when enlarged." For a photograph the argument runs the other way: there are no shapes to describe, so every pixel's colour must be stored.

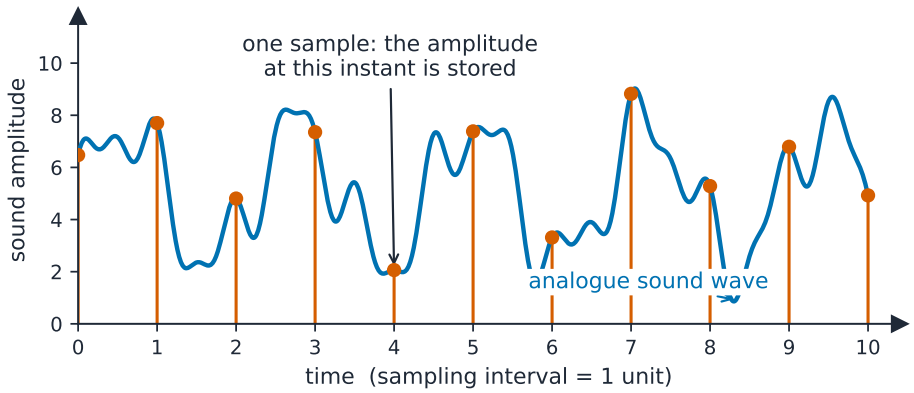


Enlarged, a bitmap's pixels turn jagged; a vector stays smooth at any size

Sound

A continuous wave of **analogue data** 模拟数据 (the sound) is converted into **digital data** 数字数据 by **sampling** 采样:

- **sampling rate** 采样率 – samples per second (Hz). CD quality is 44.1 kHz.
- **sampling resolution** 采样分辨率 (bit depth) – bits per sample's **amplitude** 振幅. CD quality is 16 bits.



Sampling a sound wave: its amplitude is read at each time interval

File size

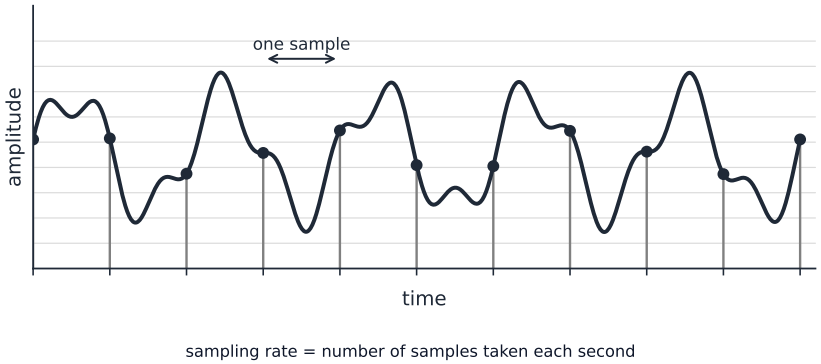
$$\text{size in bits} = \text{sampling rate} \times \text{resolution} \times \text{duration} \times \text{channels}.$$

A 10-second stereo CD clip: $44100 \times 16 \times 10 \times 2 = 14\,112\,000$ bits ≈ 1.68 MiB.

Changing settings

- **higher sampling rate** → captures higher pitches, larger file.
- **higher sample resolution** → finer amplitude steps, less **quantisation** 量化 noise, larger file.
- **lower** of either → smaller file, clear quality loss.

(The sampling rate must be at least twice the highest frequency you want to keep.)



The sampling rate is the number of samples taken each second

exercises.lol exercises.com.cn

Compression

Compression 压缩 reduces file size, saving storage and transmission **bandwidth** 带宽. Two kinds:

- **lossless** 无损 – the original data is recovered exactly (text, programs, ZIP/PNG).
- **lossy** 有损 – some detail is dropped for much smaller files (JPEG, MP3, video).

When to use which

- **lossless** for documents, source code, medical images – anything needing exact data.
- **lossy** for streaming media. **Real-time video streaming** uses lossy compression because it must send huge amounts of data in real time over limited bandwidth; lossless would not shrink it enough. Raw HD video is gigabytes per minute, so without compression the picture would keep freezing.

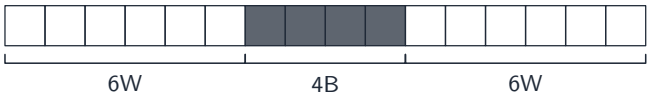
A "justify" answer names the method, then the reason from the situation: "Lossless, because the spreadsheet must be restored exactly; a single changed value would make the accounts wrong." Or: "Lossy, because the photographs are viewed on a phone screen where the dropped detail is not visible, and the smaller files upload faster and use less storage."

Lossless methods

- **run-length encoding** 行程编码 (RLE): store "the next n values are x" instead of repeating x. Great for flat areas; useless for noisy data.
- **dictionary methods** 字典编码 (ZIP, PNG): replace repeated byte sequences with a short reference. Good for text and code.
- **Huffman coding** 霍夫曼编码: give short codes to common symbols and long codes to rare ones, bringing the average code length near the data's **entropy** 熵.

How each kind of file is compressed:

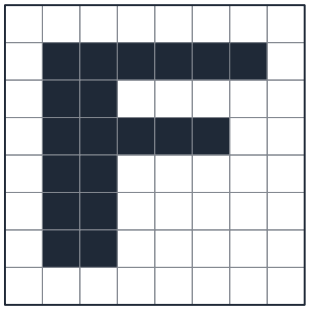
- **text file**: dictionary methods and Huffman coding turn repeated words and common characters into short codes. Text must stay lossless, because one changed character changes the meaning.
- **bitmap image**: RLE for runs of identical pixels (icons, diagrams, black-and-white scans); lossy JPEG for photographs, or a lower colour depth or resolution.
- **vector graphic**: the drawing list is already small; remove drawing objects that are not needed, store coordinates to fewer decimal places, or apply a lossless method such as ZIP to the file.
- **sound file**: lossy MP3 or AAC removes what the ear cannot hear; a lower sampling rate or resolution is also lossy; lossless formats keep every sample and shrink the file much less.



16 pixels
⇒ **6W 4B 6W**
(3 runs, not 16)
W = white, B = black

Run-length encoding on a single row: 16 pixels become 3 runs

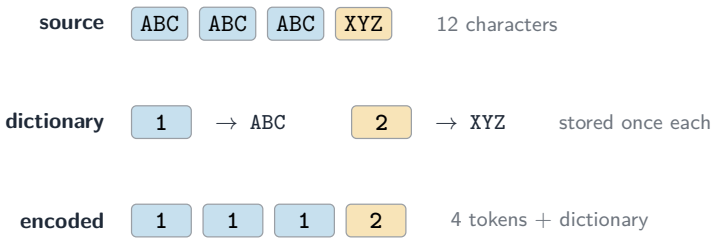
8×8 bitmap



binary	RLE code
00000000	8W
01111110	1W 6B 1W
01100000	1W 2B 5W
01111100	1W 5B 2W
01100000	1W 2B 5W
01100000	1W 2B 5W
01100000	1W 2B 5W
00000000	8W

Key: 1 = black (B), 0 = white (W). RLE stores *count* + *colour*: e.g. 01111110 → 1W 6B 1W

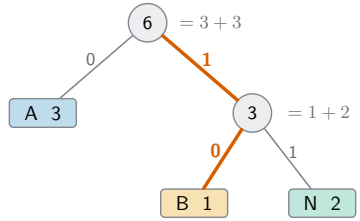
Run-length encoding of the letter F in an 8 × 8 black-and-white grid



The decoder rebuilds the dictionary as it reads, so the original returns exactly — **lossless** (ZIP, PNG).

Dictionary coding: each repeated sequence is stored once, and every occurrence becomes a short index

exercises.lol exercises.com.cn



Walk **down from the root**, writing a bit at each branch: left = 0, right = 1.

A = 0
B = 10 right, then left
N = 11

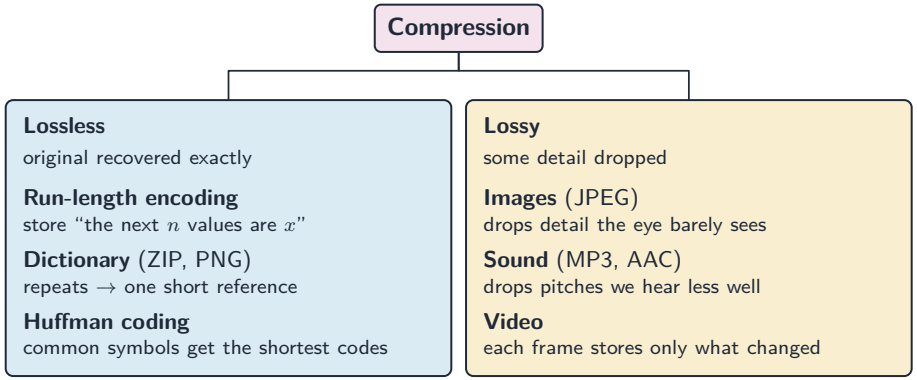
each circle joins the two smallest counts

BANANA becomes 10 0 11 0 11 0 — **10 bits**, against 12 for fixed 2-bit codes: the commonest letter is nearest the root.

Huffman coding: the commonest symbol gets the shortest code, so BANANA needs 10 bits instead of 12

Lossy methods

- **images** (JPEG): drop fine detail and colour differences the eye barely sees.
- **sound** (MP3, AAC): drop pitches we hear less well, and quiet sounds hidden by louder ones.
- **video** combines **spatial** 空间 compression (within each frame, like JPEG) with **temporal** 时间 compression (most frames store only the differences from the previous frame).



Compression methods: lossless versus lossy, with common examples

■ Logical and arithmetic shifts; IP address sizes

A **logical shift** moves the bit pattern and fills vacated positions with 0. In a **right arithmetic shift** of a two's-complement integer, the sign bit is copied into the new leftmost positions. For eight-bit `11110100` (−12), logical right by two gives `00111101` (61); arithmetic right by two gives `11111101` (−3). Discarded bits are lost; overflow and rounding can prevent shifts behaving like exact multiplication or division.

An **IPv4 address** has 32 bits, often shown as four eight-bit decimal groups. An **IPv6 address** has 128 bits, usually shown as eight groups of four hexadecimal digits before permitted abbreviation. These are address lengths, not numbers of characters in the printed address.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
bit	a single binary digit, 0 or 1
byte	a group of 8 bits
binary prefix	a multiplier that is a power of 2 (kibi = 1024) rather than a power of 10 (kilo = 1000)
two's complement	a way of representing signed integers in which the most significant bit has a negative place value
overflow	the result of a calculation is too large to be represented in the number of bits available
Binary Coded Decimal	each denary digit is stored as its own 4-bit binary pattern
character set	the set of characters a computer can represent, each with its own binary code
pixel	the smallest element of a bitmap image, storing one colour value
image resolution	the number of pixels in an image, given as width by height
screen resolution	the number of pixels a display can show, given as width by height
colour depth	the number of bits used to store the colour of one pixel
sampling rate	the number of samples of the sound taken per second

exercises.lol exercises.com.cn

Term	Definition
sampling resolution	the number of bits used to store the amplitude of one sample
lossless compression	compression from which the original data can be recovered exactly
lossy compression	compression that permanently removes some data, so the original cannot be recovered
run-length encoding	replacing a run of repeated values with one value and a count

Exam tips

- Show working for base conversions: denary → binary by place values, binary → hexadecimal in **nibbles** (groups of 4 bits).
- For **two's complement** the MSB is negative; to negate, **invert and add 1**; watch for **overflow** when the sign bit flips wrongly.
- Distinguish **bitmap** (pixels; file size = width × height × colour depth) from **vector** (drawing commands; scales without loss).
- Sound file size depends on **sample rate** × **bit depth** × **time** – more of each means better quality but a bigger file.
- Compare **lossless vs lossy** compression and give a use for each.

■ Common mistakes

- Explaining an overflow with "the answer was greater than 255" or "it has 9 bits". State the word size the question gave, then say the result cannot be represented in it.
- Making a negative number by setting the top bit to 1 and leaving the rest (sign and magnitude). Two's complement means invert every bit of the positive value, then add 1.
- Forgetting to pad a converted number to the register width the question asks for.
- Mixing bits and bytes in a file-size calculation. Work in bits, divide by 8 once, and say whether you used 1000 or 1024.
- Answering "describe" in everyday words ("the picture gets worse"). Use the syllabus terms: fewer colours, banding, lower image resolution, larger pixels.

Communication

A-Level Computer Science

Networks: purpose and benefits

A **network** 网络 is a set of computing devices connected so they can communicate and share resources. Benefits:

- **sharing resources** (printers, file servers, internet) – cheaper than equipping each computer.
- **sharing data** – many users access the same files.
- **central management** – install software, manage users and back up once on a server.
- **communication** – email, video calls, messaging.
- **remote access** – work from anywhere.

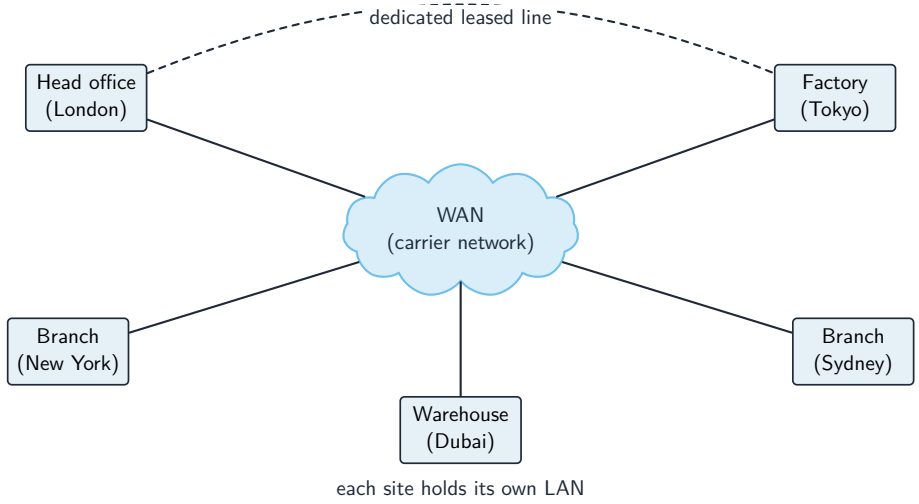
LAN vs WAN

A **local area network** 局域网 (LAN) covers a **small area** – a home, office or school, usually owned by the organisation, with high data rates and low **latency** 延迟.

A **wide area network** 广域网 (WAN) covers a **large area** – a city, country, or the world (the internet is the largest WAN). It uses telecom-company infrastructure – often the **Public Switched Telephone Network** 公共交换电话网 (PSTN), leased lines or fibre – with lower data rates and higher latency. A WAN connects LANs together.

For "give two characteristics of a LAN": it covers a small geographical area (one site or building); the hardware is owned by the organisation, not leased from a telecom company; it connects through its own switches, cables and access points. For "two ways a WAN is different": it covers a large geographical area; it uses third-party (leased or public) infrastructure; data rates are lower and latency higher; it usually joins several LANs. A school on one site is a LAN; a company with offices in two cities needs a WAN, with a leased line or the internet between the sites. Justify the choice with the area covered and who owns the links.

exercises.lol exercises.com.cn

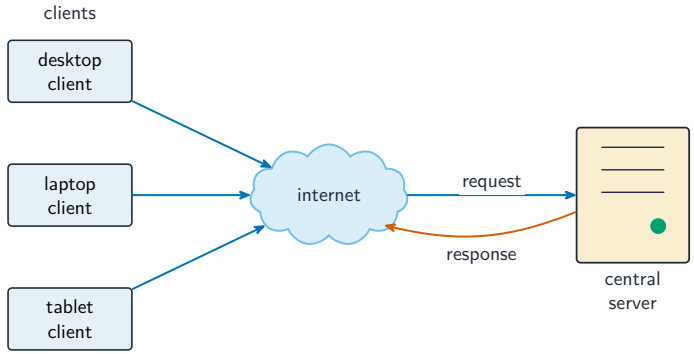


A wide-area network links many systems across a large area

Client-server and peer-to-peer

Client-server

- powerful machines act as **servers** 服务器, providing services (files, web pages, email).
- other machines are **clients** 客户端 that request services.
- central and easy to manage, but the server is a single point of failure unless backed up.

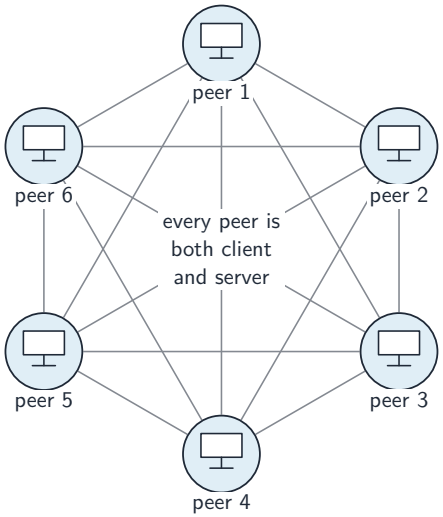


In a client-server network, clients request services from a central server

Peer-to-peer (P2P)

- all machines are equal **peers**; each can be both client and server (**peer-to-peer** 对等网络).
- resources are spread across the peers – no central server. Robust to one failure, but harder to keep secure and consistent.

Choosing a model. Client-server suits a school or a business: files are stored and backed up centrally, a user logs in with one account from any machine, software and security are managed once, and the server can be a powerful machine. The drawbacks are the cost of the server and of a technician, and that the server is a single point of failure. Peer-to-peer suits a few friends sharing files or a game: no server to buy, easy to set up, and each user keeps control of their own machine. The drawbacks the scheme lists: files are spread across many machines, so they are hard to back up and a file is unavailable when its owner's machine is off; each machine must be secured separately; and a peer that serves the others slows down. An online game played through a web browser with other users is the client-server model: the browser is the client, and the game and its shared virtual world run on the company's server, which keeps every player's view consistent.

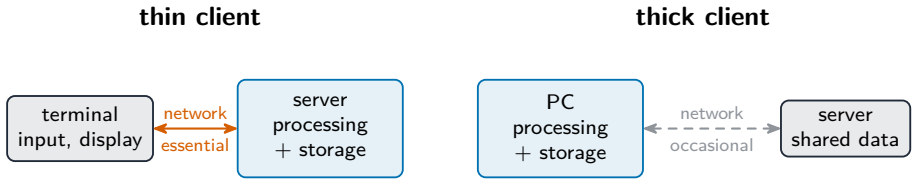


In a peer-to-peer network, every node is both client and server

exercises.lol exercises.com.cn

Thin and thick clients

A **thin client** 瘦客户端 does **little processing locally** and relies on a powerful server (web terminals, remote desktops). A **thick client** 胖客户端 has **strong local processing and storage** and runs full applications itself (a normal desktop PC).



Thin or thick depends on where the processing happens.

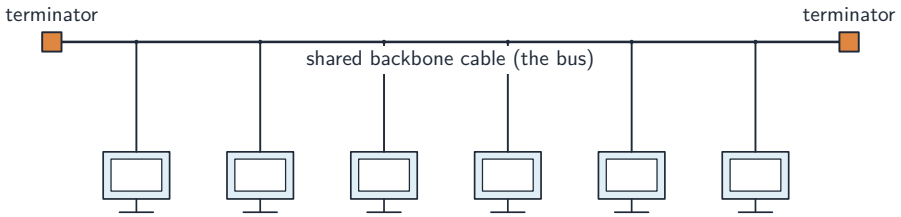
Feature	Thin client	Thick client
Local processing	minimal	substantial
Local storage	minimal	substantial
Reliance on network	high	lower
Server load	high	lower

The roles: in a thin-client model the server does the processing and stores the data, and the client only sends input and shows the output. A cheap terminal is enough, and everything is backed up and updated on the server, but nothing works if the network or the server fails. In a thick-client model the client runs the software and stores files itself, so it can work with no network connection and puts less load on the server, at the cost of more powerful (and more expensive) clients that must each be updated and secured. A school computer room can run thin clients (cheap, centrally managed); a video editor needs a thick client.

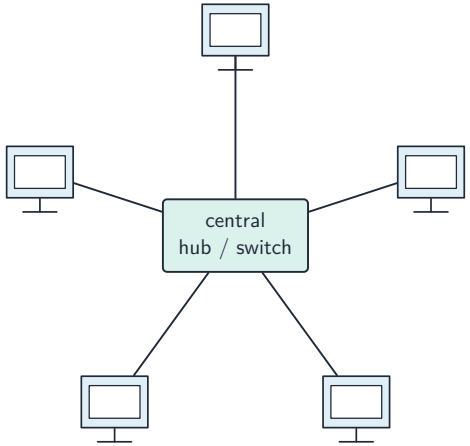
Network topologies

The **topology** 拓扑 is how the nodes and links are arranged.

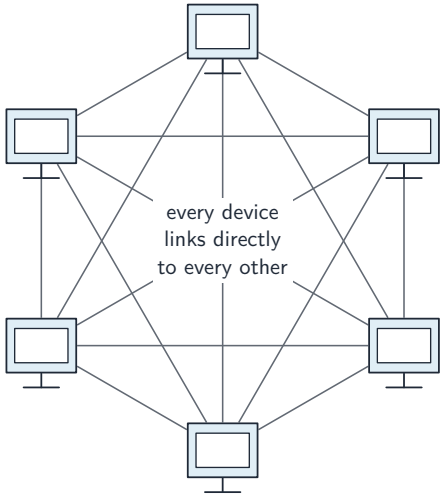
- **bus** 总线 – all devices on one shared cable. Cheap; the whole LAN fails if the bus fails; performance drops as more devices share the **bandwidth** 带宽.
- **star** 星形 – every device connects to a central switch. One device failing does not affect others; the switch failing brings all down. Most common today.
- **mesh** 网状 – every device links directly to others, with many paths. Very **fault-tolerant** 容错 (traffic reroutes) but needs lots of cabling.
- **hybrid** – a mix (a star in each office, mesh links between offices).



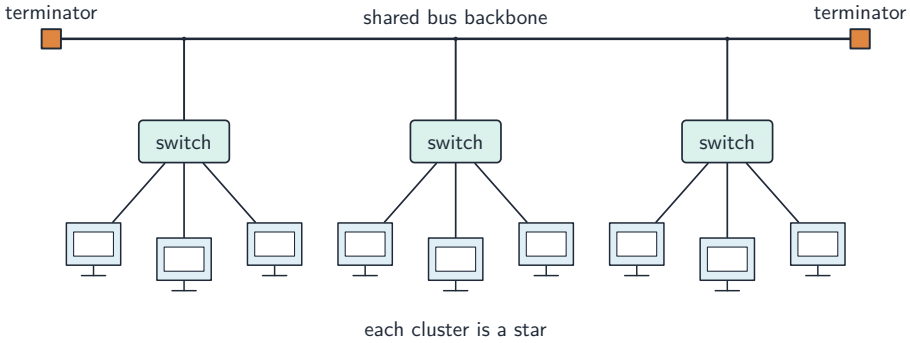
Bus topology: all devices share one cable with a terminator at each end



Star topology: every device connects to a central hub or switch



Mesh topology: every device links directly to the others



Hybrid topology: star clusters joined by a central bus

How packets travel in each topology

Bus: the sending device puts the packet on the shared cable; every device sees it, and only the one whose address matches accepts it. Only one device can transmit at a time, so collisions happen (CSMA/CD, below). **Star**: the sender passes the packet to the central switch, which reads the destination address and forwards it only down the cable to that device; two other devices can talk at the same time. **Mesh**: the packet is passed from node to node along one of several possible routes until it reaches the destination; if a link fails, another route is used.

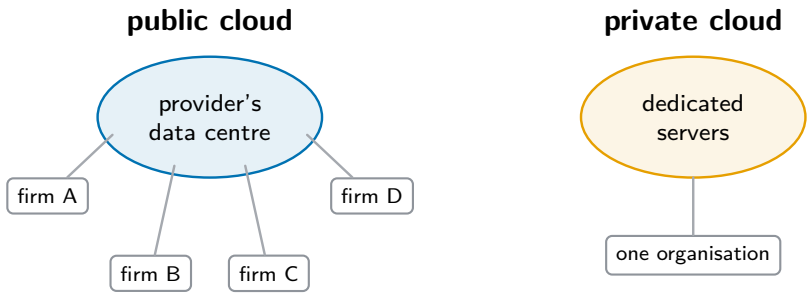
To justify a topology: a star for a classroom or an office (a failed cable affects one

exercises.lol exercises.com.cn

device; a device is easy to add; with a switch there are no collisions); a mesh where reliability matters most (a hospital, the internet's backbone); a bus only where cost matters and few devices share it. "Draw the star topology" means: the switch in the middle, one line from the switch to each computer, and the server (and the router, if there is one) on their own lines to the switch.

Cloud computing

Cloud computing 云计算 delivers computing services (servers, storage, software) **over the internet**, hosted by a third party. Benefits: **scalability** 可扩展性 (pay for what you need), lower cost, access from anywhere, and reliable redundant data centres. Drawbacks: needs internet, your data is held by a third party, and possible vendor lock-in.



Public cloud is shared; private cloud serves one organisation.

For the one-mark definition: cloud computing is **on-demand computing services (storage, processing, software) provided over the internet by a third party**. A **public cloud** 公有云 is owned by a provider and shared by many customers over the internet; a **private cloud** 私有云 is dedicated to one organisation, on its own hardware or hosted for it alone. Benefits the scheme accepts: files are accessible from any device with an internet connection; storage scales up and down as needed; the provider handles the hardware, backups and security updates; there is no local server to buy or maintain. Drawbacks: no access without an internet connection; the data is on a third party's hardware, so security and privacy depend on the provider; an ongoing subscription cost; the provider could fail or be attacked; large files may be slow to transfer. A "why does the company use a public cloud" answer says that they need no hardware of their own, pay only for what they use, and their users can reach it from anywhere.

exercises.lol exercises.com.cn

Wired vs wireless

- **wired** (**Ethernet** 以太网 over **twisted-pair** 双绞线 or **fibre-optic** 光纤): higher speed, lower latency, fewer errors, more secure.
- **wireless** (Wi-Fi, Bluetooth, cellular): no cables, devices can move, but slower, prone to interference and eavesdropping.

For the same generation, wired wins on speed and reliability; wireless wins on convenience.

Transmission media

Medium	Characteristics
copper cable (twisted pair, coaxial)	cheap and easy to install; carries an electrical signal; affected by electromagnetic interference; the signal weakens with distance, so repeaters are needed; lower bandwidth than fibre
fibre-optic cable	light pulses in a glass core; very high bandwidth; long distances without repeaters; immune to interference; hard to tap, so secure; expensive and needs skilled installation
radio waves (including WiFi)	no cable, so devices can move; a range of tens of metres, weakened by walls; a shared frequency, so interference and lower speed; can be intercepted, so needs encryption
microwaves	higher-frequency radio for point-to-point links; needs a line of sight; affected by rain and buildings; high bandwidth
satellites	reach remote areas and the whole globe; a long delay (latency), because the signal travels to orbit and back; affected by weather; expensive

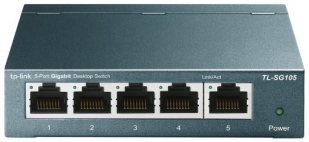
The exam asks for the comparison in both directions. Wired beats wireless on speed, reliability (no interference), security (a cable must be physically tapped) and consistency; wireless beats wired on mobility, the cost of installation, and adding a device without cabling. Allowing both lets students move around with laptops and phones while the fixed desktops keep the faster, more secure connection, and a device with no network port can still connect. Satellite instead of copper reaches places no cable can, but with more delay, weather interference and higher cost.

LAN hardware

- **network interface card** 网络接口卡 (NIC) – lets a device send and receive on the network; has a unique **MAC address** MAC 地址 (a 48-bit hardware address). A wireless device uses a **wireless network interface card** 无线网络接口卡 (WNIC).

- **switch** 交换机 – forwards Ethernet frames only to the port for the destination MAC address.
- **hub** 集线器 – a simpler device that copies traffic to all ports (now obsolete).
- **wireless access point** 无线接入点 (WAP) – lets wireless clients join a wired LAN.
- cabling – twisted-pair for short runs; fibre-optic for longer, faster runs.
- **server** – a computer that provides a service to the other devices: files, printing, web pages, email storage.
- **bridge** 网桥 – joins two LAN segments into one network, passing traffic between them.
- **repeater** 中继器 – receives a weakened signal and retransmits it at full strength, to extend a cable's reach.

A WNIC's functions, for a four-mark describe: it converts the data into radio signals and back; it carries the device's unique MAC address; it connects the device to a wireless access point and follows the wireless protocol (which channel and frequency to use); and it decodes the incoming signals for the device. Two devices that can physically connect thirty computers with NICs: a switch, or a hub.

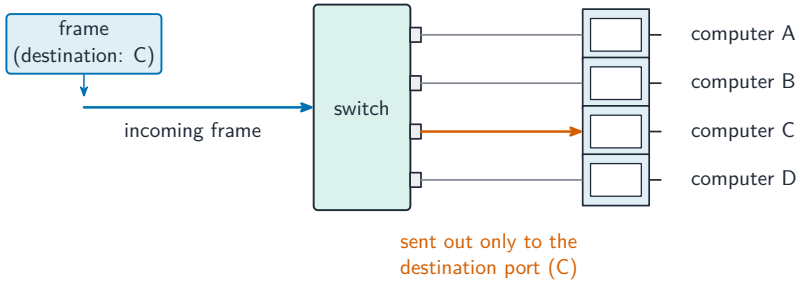


A network switch: each device's cable plugs into one of its ports



An RJ-45 plug on a twisted-pair Ethernet cable

exercises.lol exercises.com.cn



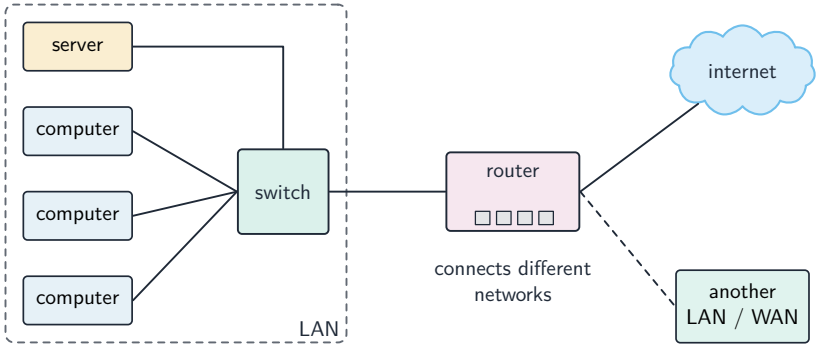
A switch sends each frame only to the port for its destination

Routers

A **router** 路由器 connects **different networks** and forwards data between them – usually at the boundary of a LAN and the internet. It does:

- **forwarding** – reads each **packet** 数据包's destination **IP address** IP 地址 and sends it out the right port, using a **routing table** 路由表.
- **network address translation** 网络地址转换 (NAT) – lets many private LAN addresses share one public IP.
- **DHCP** 动态主机配置协议 – hands out private IP addresses to LAN devices.
- **firewall** 防火墙 – blocks unwanted incoming traffic.

In **packet switching** 分组交换 a message is split into packets that are sent independently. Each router reads a packet's destination IP address, looks up the next hop in its routing table and forwards it, so the packets of one message may take different routes and are reassembled in order at the destination. A router does receive packets, forward them between networks and hand out IP addresses; it does **not** find the IP address for a URL (DNS does that) and it does not store web pages. A home router also contains the modem and the wireless access point, so one box connects the LAN to the internet.



A router connects a LAN to the internet or another network

Ethernet and CSMA/CD

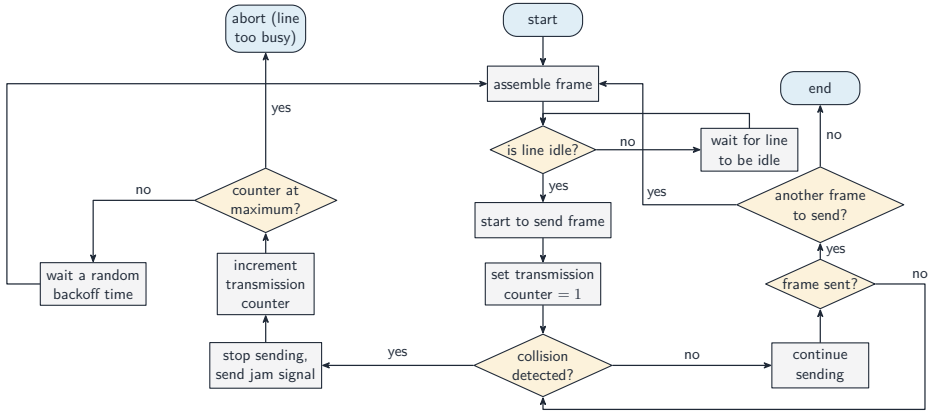
Ethernet is the standard (protocol) for wired LANs: devices are joined by twisted-pair or fibre cable, data is sent in frames that carry the source and destination MAC addresses, and a shared medium uses CSMA/CD to deal with collisions. On shared media a **collision** 冲突 can happen when two devices send at once. The protocol is **CSMA/CD** 载波侦听多路访问/冲突检测 (Carrier Sense Multiple Access with Collision Detection):

1. **carrier sense** – listen before sending; wait if the cable is busy.
2. **multiple access** – many devices share the medium.
3. **collision detection** – keep listening while sending; a clash is a collision.
4. on a collision, both stop, send a brief "jam" signal, then wait a **random backoff** time before retrying.

The three tasks, in the scheme's words: the device listens (senses the carrier) before transmitting; it keeps checking for a collision while it transmits; on a collision it stops, sends a jam signal, waits a random time and retransmits.

Modern switched Ethernet uses **full-duplex** 全双工 point-to-point links, so collisions no longer happen.

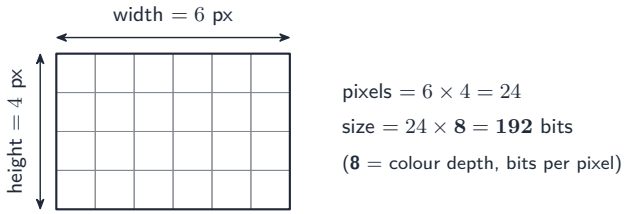
exercises.lol exercises.com.cn



The CSMA/CD process for handling collisions on shared media

Bit streaming

Bit streaming 流式传输 sends multimedia as a continuous stream that the receiver plays as it arrives, instead of downloading the whole file first.



The same formula on small numbers: count the pixels, then multiply by the colour depth

- **real-time** (live): captured and streamed as it happens (live sport, video calls). You cannot rewind; low latency is vital.
- **on-demand**: pre-recorded on a server (YouTube, Netflix). You can pause and rewind; the server can **buffer** 缓冲 ahead.

Real-time streaming works as a short pipeline:

1. **capture and sample** the source (a camera or microphone).
2. **encode** it, using **compression** 压缩 to shrink the data.
3. **send** it across the network as packets.
4. the receiver buffers a little, then plays it live – **dropping any packet that arrives late**, because a live stream cannot wait for it.

Lossy 有损 compression is used here: moving pictures hide small losses, and the stream must be small enough to fit the bandwidth.

Why a video is compressed before real-time streaming: the uncompressed stream would need more bandwidth than the connection has, so frames would arrive late and the playback would stall. Compression cuts the number of bits, so the **bit rate** 比特率 stays below the broadband speed, the delay stays small, and less storage and cost are needed at both ends. The bit rate must be lower than the connection's speed: a higher bit rate gives better quality but needs a faster connection, and if the data arrives more slowly than it is played, the buffer empties and the video freezes. On-demand streaming can buffer more of the file ahead, so it copes with a slower connection; real-time streaming cannot.

> **Worked example.** A camera streams live video at 1920 × 1080 pixels, 24-bit colour, 30 frames per second, over a 50 Mbit/s connection. Can it stream uncompressed, and by what factor must it be compressed?

Bits in one frame:

$$\begin{aligned}\text{bits per frame} &= \text{pixels} \times \text{colour depth} \\ &= 1920 \times 1080 \times 24 = 49\,766\,400 \text{ bits}\end{aligned}$$

Bit rate of the uncompressed stream:

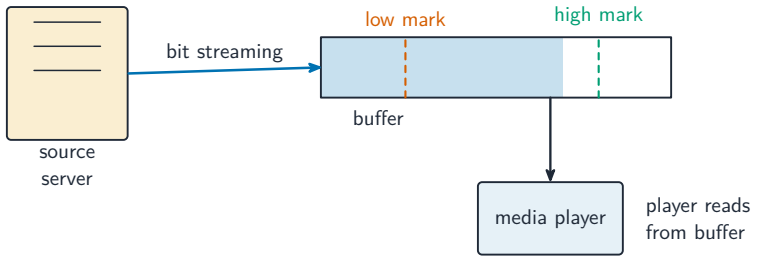
$$\begin{aligned}\text{bit rate} &= \text{bits per frame} \times \text{frame rate} \\ &= 49\,766\,400 \times 30 \\ &= 1\,492\,992\,000 \text{ bits/s} \approx 1493 \text{ Mbit/s}\end{aligned}$$

Compression needed:

$$\text{factor} = \frac{\text{bit rate}}{\text{connection speed}} = \frac{1493 \text{ Mbit/s}}{50 \text{ Mbit/s}} \approx 30$$

The uncompressed stream needs about 30 times the connection's speed, so it cannot be sent as it is: it must be compressed to **less than** $\frac{1}{30}$ of its size, with room to spare for other traffic and for changes in speed. This is why live video uses lossy compression. Use 1 Mbit = 10^6 bits for transmission speeds unless the question says otherwise.

exercises.lol exercises.com.cn

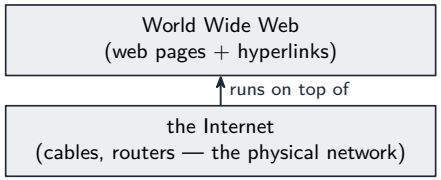


Data streams from the server into a buffer before the media player reads it

The internet and the World Wide Web

The **internet** 互联网 is a global **network of networks** using a common **protocol** 协议 suite (TCP/IP). The **World Wide Web** 万维网 (WWW) is a **service** that runs over it: hyperlinked documents identified by URLs, viewed in browsers via HTTP/HTTPS. Email and file transfer are other internet services that are not part of the WWW.

Webmail uses **both**: the WWW, because the mailbox is a web page reached through a URL in a browser over HTTP; and the internet, because the email itself travels across the network of networks (email is a separate internet service from the web).

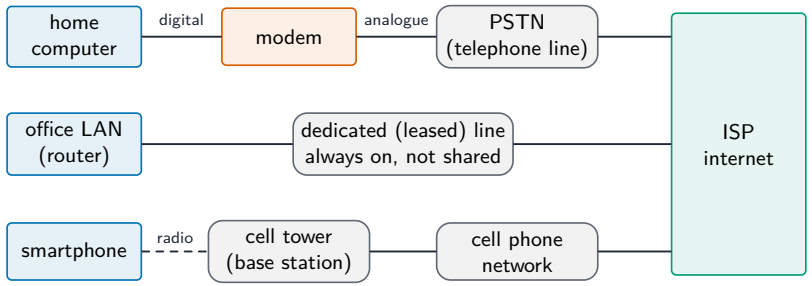


The Web is one service running on top of the Internet

Hardware that supports the internet

- **modem** 调制解调器 – converts the computer's digital signal into an analogue signal for a telephone line, and back again at the other end (modulation and demodulation).
- **PSTN** – the public telephone network of exchanges and lines; a dial-up or DSL connection carries internet data over it.
- **dedicated line** 专线 – a leased line between an organisation and its ISP: always on, with a fixed bandwidth that is not shared, so faster and more reliable, but expensive.

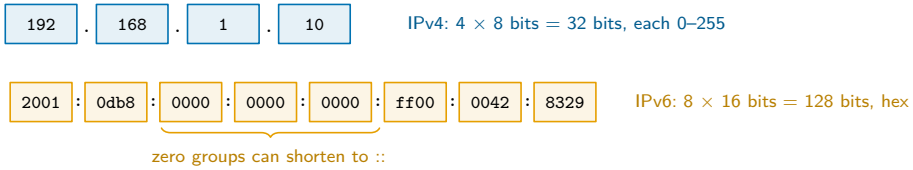
- **cell phone network** 蜂窝网络 – the phone sends data by radio to the nearest cell tower (base station); the towers are linked to the phone company's network, which routes the data to the internet; as the phone moves, it is handed over from one cell to the next.



Three ways to reach the internet: a modem and the PSTN, a dedicated line, and the cell phone network

IP addresses

An IP address uniquely identifies a device.



Count the groups, the separators and the digit base.

- **IPv4** – 32-bit, four denary numbers 0–255 (`192.168.1.10`); about 4.3×10^9 addresses (now exhausted).
- **IPv6** – 128-bit, eight groups of four hex digits; about 3.4×10^{38} addresses.

IPv4 is written as **four** groups of denary numbers separated by **dots**; each group is an 8-bit number, so it runs from **0 to 255**. IPv6 is written as **eight** groups of **four hexadecimal digits** separated by **colons**, `2001:0db8:0000:0000:0000:ff00:0042:8329`, and a run of zero groups can be shortened to `::`. So `192.168.3.2` is not IPv6: it has four groups, not eight, separated by dots rather than colons, and its groups are denary, not hexadecimal. `256.0.0.A` is not a valid address of either kind: an IPv4 group cannot exceed 255 and cannot be a letter, and IPv6 would need colons and eight groups.

exercises.lol exercises.com.cn

Worked example: valid or not?

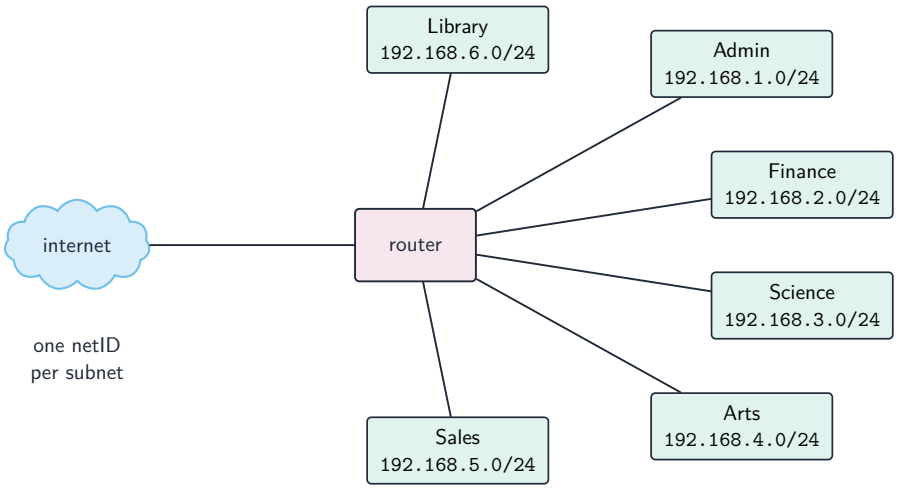
Address	Valid?	Reason
`10.0.0.1`	yes, IPv4	four denary groups, each 0–255 (a private address)
`172.16.5`	no	only three groups; IPv4 needs four
`192.168.1.256`	no	256 is more than 8 bits can hold
`2001:db8::ff00:42:8329`	yes, IPv6	five groups are written, so `::` stands for three zero groups: `2001:0db8:0000:0000:0000:ff00:0042:8329`
`2001:db8::1::2`	no	`::` may appear only once, or the number of missing groups would be unclear
`fe80:0:0:0:g1:0:0:1`	no	`g` is not a hexadecimal digit

To expand a shortened IPv6 address, add leading zeros to make each group four digits, then replace `::` with as many `0000` groups as are needed to make eight.

Subnetting

A network can be split into **subnets** 子网. The IP address splits into a network part and a host part, given by a **subnet mask** 子网掩码 (e.g. `255.255.255.0` = first 24 bits are network). Subnetting improves management, cuts broadcast traffic, and improves security.

The two parts of an address in a subnetwork: the **network ID** (the first bits, the same for every device in that subnet, given by the ones in the mask) and the **host ID** (the remaining bits, unique to each device). Benefits of subnetting, for "describe two benefits": less traffic on each part, because broadcasts stay inside their subnet; better security, because one department's traffic is kept from the others; easier management and fault-finding; more efficient use of the addresses. Two devices with the mask `255.255.255.0` are in different subnets when their first three groups differ.



Splitting a network into subnets, one netID per department

Public vs private addresses

- **private** addresses are used within a LAN and are not routable on the internet (e.g. `192.168.0.0/16`).
- a **public IP address** is globally unique and routable, assigned by an **ISP** 互联网服务提供商.

Devices behind NAT with private addresses are not directly reachable from the internet, giving some protection.

The descriptions the tables want: a **public** address is visible on the internet and unique across it, allocated by the ISP; a **private** address is visible only inside the LAN, is reused by many LANs, and needs NAT to reach the internet. A **static** address never changes (set by hand or reserved, as a server needs); a **dynamic** address is allocated by DHCP each time the device connects and may change.

Static vs dynamic

- a **static IP address** is fixed; used for servers that must be found at a known address.
- a **dynamic IP address** is assigned by DHCP and may change; easier for client devices and uses a limited address pool efficiently.

exercises.lol exercises.com.cn

> **Worked example.** A host has IP address `192.168.10.130` with subnet mask `255.255.255.192`. Which network is it on, and is `192.168.10.200` on the same one? The mask's last octet, `192`, is `11000000` in binary, so the first **26** bits are the network part and the last 6 bits address the host. That makes the subnets step in blocks of $256 - 192 = 64$: `.0`, `.64`, `.128`, `.192`. The address `130` falls in the block starting at `.128`, so the host is on network **`192.168.10.128/26`**, whose usable hosts run `.129` to `.190` (`.191` is the broadcast address). `200` falls in the **next** block (`.192`), so it is on a **different** subnet and traffic between the two must pass through a **router**. Get the block size from the mask first (256 minus the mask octet) - guessing from the first three octets is what makes these go wrong.

URL and DNS

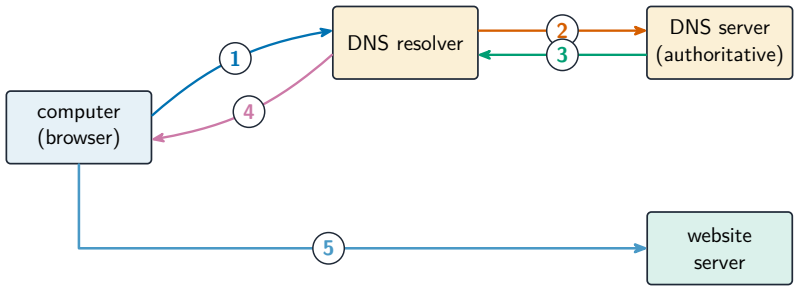
A **URL** 统一资源定位符 (Uniform Resource Locator) locates a resource on the WWW:

```
https://www.example.com/about/contact.html
protocol     domain name        path
```

- **protocol**: `http`, `https`, etc.
- **domain name** 域名: a readable server address.
- **path**: the resource on that server.

The **Domain Name System** 域名系统 (DNS, also called the Domain Name Service) is a distributed set of servers that turns **domain names into IP addresses**. When you type a URL, the browser asks a DNS resolver for the IP, which queries DNS servers (root → top-level → authoritative) until it finds it; the browser then connects to that IP and requests the path. DNS saves humans from memorising IP addresses and lets a site change server without changing its name.

For "explain how the browser uses the URL": the browser splits the URL into the protocol, the domain name and the path; it sends the domain name to a DNS server, which returns the matching IP address (a cache on the computer or at the ISP may answer first); it opens a connection to that IP address using the protocol (HTTPS on port 443); it sends a request for the path; and the web server returns the page, which the browser renders. If the DNS lookup fails, the browser reports that the server cannot be found.



1 computer asks the resolver for the IP **2** resolver queries a DNS server
3 server returns the IP address **4** resolver passes the IP back to the computer
5 browser connects to that IP and requests the page

How DNS finds a website's IP address before the browser connects

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly.

Term	Definition
LAN	a network covering a small geographical area, usually one site, whose hardware is owned by the organisation
WAN	a network covering a large geographical area, joining LANs through third-party (leased or public) links
client-server	a model in which client computers request services from a central, more powerful server that provides them
peer-to-peer	a model in which every computer is equal and can act as both client and server, with no central server
thin client	a client that does little processing or storage itself and depends on the server for both
thick client	a client that does its own processing and storage and can work without the server
mesh topology	a topology in which each device is connected directly to many others, giving more than one route between two devices
cloud computing	on-demand computing services (storage, processing, software) provided over the internet by a third party
Ethernet	the standard protocol for wired LANs, sending data in frames and using CSMA/CD on a shared medium

exercises.lol exercises.com.cn

Term	Definition
switch	a device that forwards each frame only to the port of its destination MAC address, within a LAN
router	a device that connects networks and forwards packets between them by their destination IP address
bit streaming	sending a continuous stream of bits so that the receiver plays the media as it arrives, without downloading the whole file first
internet	the global network of networks that uses the TCP/IP protocols
World Wide Web	the collection of hyperlinked web pages, identified by URLs and accessed over the internet through a browser
URL	the address that locates a resource on the web: protocol, domain name and path
DNS	the service that translates a domain name into the IP address of the server that holds the resource

Exam tips

- Distinguish **LAN vs WAN** and **client-server vs peer-to-peer** by who stores and controls the resources.
- Match each **topology** (bus, star, mesh) to its advantages and drawbacks (cost, reliability, collisions).
- Know the job of each device: a **switch** directs within a LAN by MAC address, a **router** routes between networks by IP.
- Explain **bit streaming** and why buffering is needed (data arrives at a different rate from playback).
- Distinguish **IPv4 vs IPv6** and public vs private addresses; DNS turns a URL into an IP address.

Common mistakes

- Saying a switch works by IP address. A switch forwards by MAC address inside the LAN; the router forwards by IP address between networks.
- Treating the internet and the World Wide Web as the same thing. The web is one service that runs over the internet; email and file transfer are others.
- Giving "faster" as the whole comparison of wired and wireless. Say faster **and** more reliable and more secure, and give the wireless side (mobility, no cabling) when the question asks for a comparison.

- Writing that a router finds the IP address for a URL. DNS does that; the router forwards packets to it.
- Describing IPv6 with dots and denary groups. Eight groups of four hexadecimal digits, separated by colons.
- Drawing a star topology as a ring or a chain. Every device has its own line to the switch in the middle.

exercises.lol exercises.com.cn

Hardware

A-Level Computer Science

Computers and their components

A general-purpose computer has four building blocks:

- **input devices** 输入设备 – get data in (keyboard, mouse, microphone, scanner, sensors).
- **output devices** 输出设备 – give results out (monitor, speakers, printer, actuators).
- **primary memory** 主存储器 – fast memory the **processor** 处理器 (CPU) reaches directly (RAM and ROM). Holds the running program and its data.
- **secondary storage** 辅助存储器 – slower, larger, keeps programs and data when not in use (hard disk, SSD, optical disc, USB stick).

The syllabus asks why each is **needed**. Input devices are needed because the computer can only work on data and instructions that have been entered. Output devices are needed to present the results in a form people can use. Primary memory is needed because the processor can only execute instructions and use data that are held in memory it can address directly, and it must reach them fast. Secondary storage is needed because primary memory is volatile and small: programs and data must survive the power being switched off, in a larger and cheaper store, and **removable** storage lets data be moved between computers or kept as a backup.



A keyboard: a common input device for typing text and commands



A mouse: a pointing input device



A flatbed scanner: an input device that turns a paper page into a digital image



A monitor: a common output device that displays the screen image

Embedded systems

An **embedded system** 嵌入式系统 is a computer built **into another device** to do one fixed job (washing machine, microwave, car engine unit, thermostat).

- **benefits**: optimised for one task, so it is small, uses little power and is cheap to make in volume; reliable, because it runs one fixed program with few chances to go wrong; starts quickly and needs no user set-up; easy to use through a simple interface.
- **drawbacks**: limited to its one task, so it cannot be upgraded to do more; hard to update (its **firmware** 固件 may need special tools or cannot be changed at all); difficult to troubleshoot, and usually the whole device must be replaced when it fails; if it is connected to a network it can be a security weakness, because its software is rarely patched.

A "describe the drawbacks" question wants each drawback as a full point: what the limitation is and what it means for the user, for example "the firmware cannot be updated, so a security fault found later cannot be fixed".

Principal hardware devices

Laser printer

A **laser printer** 激光打印机 scans the page image onto a charged photosensitive **drum** 感光鼓. **Toner** 墨粉 sticks to the charged areas, transfers to the paper, and is melted on by a fuser. Fast, sharp, high-volume.



A laser printer: fast, sharp printing using a charged drum and toner

How it works, in the steps the mark scheme lists:

1. The data for the page is sent to the printer's buffer.
2. The **drum** is given a uniform electrostatic charge.
3. A laser, reflected off a rotating mirror, scans the page image onto the drum, removing the charge where it strikes, so the charge left on the drum matches the image.
4. **Toner**, a charged powder, is attracted to the charged parts of the drum only.
5. The paper is given the opposite charge and rolled against the drum, so the toner transfers onto it.
6. The **fuser** 定影器, a pair of heated rollers, melts the toner into the paper. The drum is then discharged and cleaned for the next page.

3D printer

A **3D printer** 3D 打印机 builds an object **layer by layer**: FDM melts plastic filament through a nozzle; stereolithography cures liquid resin with a UV laser. Used for prototypes and custom medical parts.

exercises.lol exercises.com.cn



An FDM 3D printer builds an object layer by layer by melting plastic filament

How it works:

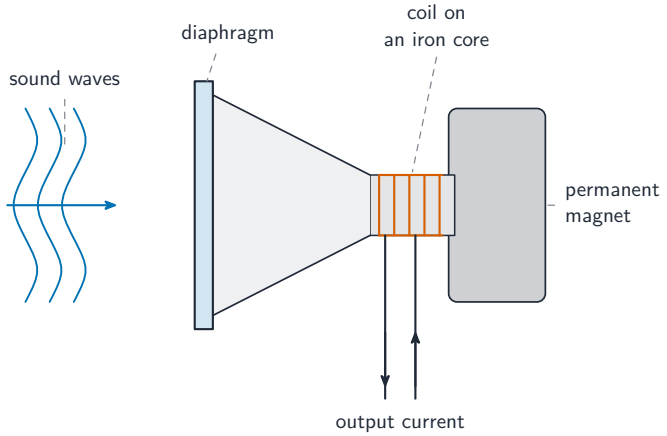
1. A model of the object is designed in **CAD** software (or scanned).
2. Slicing software divides the model into thin horizontal layers and produces the instructions for each one.
3. The printer builds the object one layer at a time: an FDM printer melts plastic **filament** 塑料丝 and lays it down through a moving nozzle; a resin printer cures liquid resin with a laser or UV light; a powder printer fuses powder with a laser.
4. Each layer bonds to the layer below, and the platform (or nozzle) moves by one layer's thickness.
5. When the last layer is done, any support material is removed. Uses include prototypes, custom medical parts such as prosthetics, and spare parts printed on demand.

Microphone and speakers

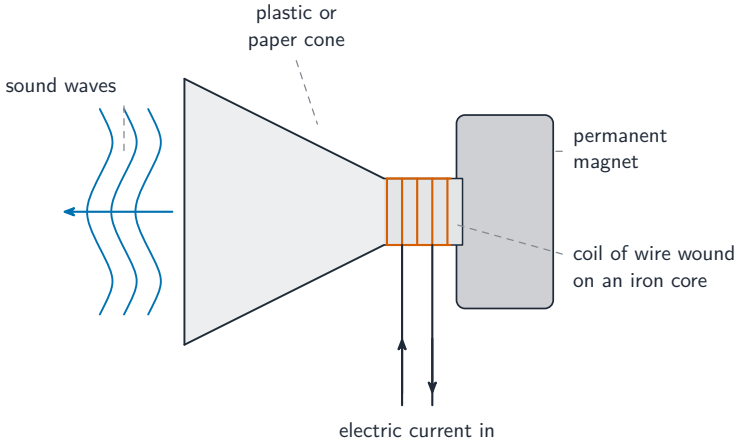
A **microphone** 麦克风 turns **sound into an electrical signal** (a diaphragm vibrates, changing **capacitor** 电容器 charge or coil position); the signal is digitised by an **analogue-to-digital converter** 模数转换器 (ADC). A speaker does the reverse – a varying signal drives a coil in a magnetic field, moving a cone to make sound.



A microphone turns sound into an electrical signal



Inside a microphone: sound vibrates the diaphragm and coil to produce a current



Inside a loudspeaker: a varying current in the coil moves the cone to make sound

How a microphone works: sound waves make a **diaphragm** 膜片 vibrate; in a dynamic microphone a coil attached to the diaphragm moves in a magnetic field, so a varying current is induced in it, and in a condenser microphone the diaphragm is one plate of a capacitor whose capacitance changes as it moves; the varying analogue signal is then sampled by an ADC and stored as digital data. A speaker runs the chain backwards: a **digital-to-analogue converter** 数模转换器 (DAC) produces a varying current, the current in the coil creates a changing magnetic field that pushes against the permanent magnet, the coil and cone move in and out, and the cone's movement makes pressure

exercises.lol exercises.com.cn

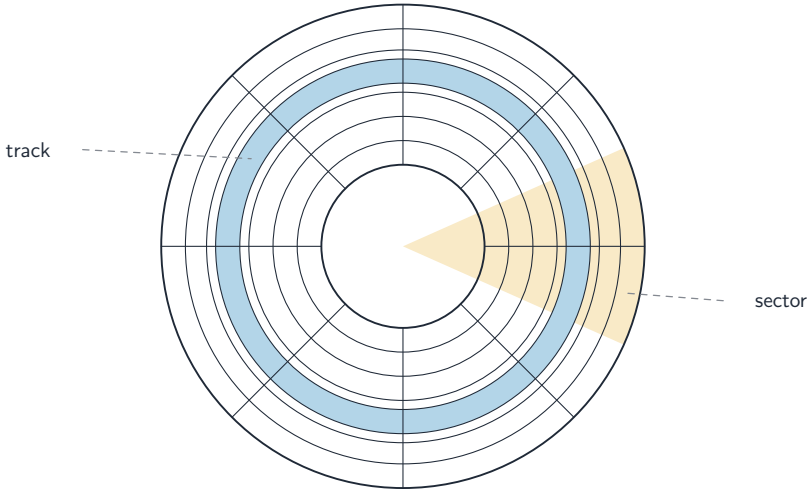
waves in the air.

Magnetic hard disk (HDD)

A **hard disk** 硬盘 stores data on spinning platters coated with magnetic material. Each platter has **tracks** 磁道 divided into **sectors** 扇区. A **read/write head** 读写头 floats just above and magnetises tiny regions (write) or senses them (read). Cheap per gigabyte, but slower than SSDs and has moving parts.



An opened hard disk: the actuator arm carries the read/write head over a platter

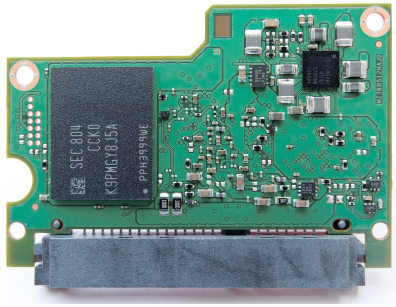


Tracks and sectors on a hard disk platter

How it works: the platters spin at high speed (thousands of revolutions per minute); each surface is divided into concentric tracks and each track into sectors; read/write heads on **actuator arms** 磁头臂 move across the platters to the right track; to write, the head magnetises a tiny region with one of two polarities, representing 0 or 1; to read, it detects the polarity as the region passes beneath it. The delays, waiting for the arm to reach the track and for the sector to spin round, are why a hard disk is slower than an SSD.

Solid-state (flash) memory

A **solid-state drive** 固态硬盘 stores data as charge in **transistors** 晶体管, with no moving parts. Faster random access than HDDs, tougher, lower power, but dearer per gigabyte; each cell wears out after many writes.



Inside an SSD: data is stored in flash memory chips, with no moving parts (compare the hard disk above)

How it works: each cell is a **floating-gate transistor** 浮栅晶体管; a charge trapped on the floating gate represents a bit and stays there when the power is off; a controller chip maps each address to a cell and spreads writes across the cells, because a cell survives only a limited number of writes.

	Magnetic hard disk	Solid-state drive
Moving parts	platters and heads	none
Speed	slower: seek and rotation delays	much faster random access
Cost per gigabyte	lower	higher
Robustness	damaged by knocks; noisy; more power	shock-resistant; silent; less power
Lifetime	many rewrites; wears mechanically	limited write cycles per cell

A "why a server uses hard disks rather than SSDs" question wants the left column: cheaper per gigabyte for very large capacities, a long life under constant rewriting, and easier data recovery.

exercises.lol exercises.com.cn

Optical disc

A laser detects reflections from tiny pits on an **optical disc** 光盘 (CD, DVD, Blu-ray). The drive is an **optical disc reader/writer**: writing uses a stronger laser to change the surface's reflectivity.

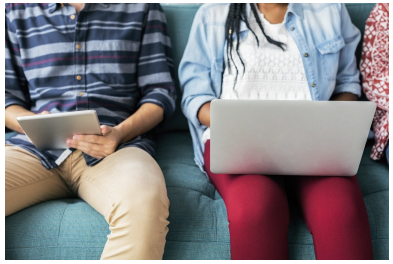


An optical disc drive: a laser reads tiny pits on a CD, DVD or Blu-ray disc

How it works: the disc carries one long spiral track of **pits** 凹坑 and **lands** 平台 (the flat areas between them); the disc spins and a laser is focused on the track; light reflected from a land differs from light reflected at the edge of a pit, and a light sensor reads each change as a 1 and no change as a 0. Writing uses a stronger laser to change the reflectivity of a dye or alloy layer. A Blu-ray uses a blue laser with a shorter wavelength, so its pits are smaller and closer together, which is why it holds more data than a DVD.

Touchscreen

A **touchscreen** 触摸屏 senses contact. **Resistive** 电阻式: two conductive layers pressed together; works with anything but is less accurate. **Capacitive** 电容式: a finger disturbs a charge field; accurate, multi-touch, used in phones.



A touchscreen senses where a finger touches the glass

How it works: a resistive screen has two thin conductive layers separated by spacers; pressing pushes the top layer onto the bottom one, closing a circuit at that point, and the controller reads the voltage to find the coordinates. A capacitive screen has a glass layer coated with a transparent conductor that holds a charge; a finger touching it draws a tiny current, the current is measured at each corner, and the controller works out the touch position from the differences. Capacitive screens respond to a light touch and to several fingers at once, but not to a gloved finger or an ordinary stylus.

Virtual reality headset

A **virtual reality** 虚拟现实 (VR) headset has two small displays (one per eye) and motion sensors (**accelerometer** 加速度计, **gyroscope** 陀螺仪) that track head movement so the scene shifts as you look around.



A virtual reality headset: two small displays and motion sensors track the head

How it works: each eye sees its own display through a lens, and the two images differ slightly, so the brain sees depth; sensors (accelerometer, gyroscope, sometimes cameras) report where the head is and which way it points; the computer re-renders the scene from that viewpoint many times a second, so turning the head turns the view; headphones give sound that matches the direction. Used for games, for training such as flight or surgery simulators, and for viewing designs before they are built.

Buffers

A **buffer** 缓冲 is memory that **holds data temporarily** while it moves between devices of different speeds. Example: the CPU writes a document to a printer buffer quickly, then is free to do other work while the printer prints from the buffer at its own pace. Buffers stop the fast device waiting for the slow one (also used in streaming, the keyboard, and disk access).

"State why a 3D printer needs a buffer": the computer sends the print data much faster than the printer can build the layers, so the data is held in the buffer until the printer is ready for it, and the processor is freed to do other work. When the buffer runs low the printer sends an **interrupt** 中断 to ask for more (topic 4). A video stream works the same way: the buffer fills ahead of playback so a short drop in the network speed does not stop the picture.

exercises.lol exercises.com.cn

RAM and ROM

- **RAM** 随机存取存储器 (Random Access Memory) – **volatile** 易失性 (loses data without power). Holds the OS, running programs and their data; read and written constantly.
- **ROM** 只读存储器 (Read-Only Memory) – **non-volatile** 非易失性 (keeps data without power). Usually written once; holds firmware needed at start-up (the BIOS / boot loader).

RAM
volatile — lost at power off
read *and* write
holds running programs & data

ROM
non-volatile — kept at power off
read-only
holds start-up instructions

RAM is volatile and read/write; ROM is non-volatile and read-only

ROM starts the system; RAM then holds the active work.

	RAM	ROM
Volatile?	yes: contents lost when the power is off	no: contents kept without power
Read/write?	read and written constantly	read only in normal use
Holds	the operating system, running programs and their data	the firmware and bootstrap program that start the computer
Size	large, and can usually be increased	small and fixed
Typical use	the main memory of a computer or phone	the start-up code of a PC; the whole program of an embedded system such as a washing machine

More RAM lets a computer hold more programs and data at once, so it swaps less between memory and disk and runs faster; that is the answer to "explain why the computer with more RAM performs better".

exercises.lol exercises.com.cn



A RAM module (DIMM) plugs into the motherboard as the computer's fast main memory

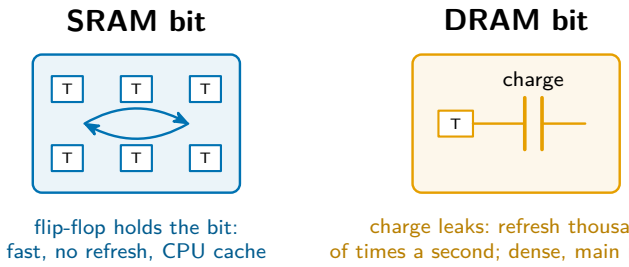
The same memory split matters in a wearable device: its fixed program must remain available after power off, while its live readings change during use.



A wrist-worn heart-rate monitor

SRAM vs DRAM

- **SRAM** 静态 RAM (Static RAM) stores each bit in a **flip-flop** 触发器 of several transistors. Fast, but expensive and not dense. Used for CPU **cache** 高速缓存.
- **DRAM** 动态 RAM (Dynamic RAM) stores each bit as charge on a tiny capacitor. Cheaper and denser but slower, and must be **refreshed** 刷新 (rewritten) thousands of times a second. Used for main memory.



A flip-flop needs no refresh; a leaking capacitor does.

Use SRAM for small fast memory (cache); DRAM for large main memory.

	SRAM	**DRAM**
Each bit stored in	a flip-flop of several transistors	one capacitor and one transistor
Needs refreshing?	no	yes, thousands of times a second
Speed	faster	slower
Density and cost	fewer bits per chip, more expensive	more bits per chip, cheaper
Power	uses less power when idle	uses more, because of the refresh
Used for	processor cache	main memory, including in embedded systems

"Explain why the embedded system uses DRAM": it needs a large amount of memory at low cost in a small space, and its speed requirement is modest, so the cheaper, denser DRAM is the right choice; SRAM is kept for the small cache where speed matters most.

PROM, EPROM and EEPROM

ROM variants you can program after manufacture:

- **PROM** (Programmable ROM) – written **once** (fuses burned by a programmer); cannot be changed.
- **EPROM** (Erasable Programmable ROM) – erased by **strong UV light** through a window, then rewritten (whole chip at once).
- **EEPROM** (Electrically Erasable Programmable ROM) – erased and rewritten **electrically**, a byte at a time, in circuit. Flash memory is a derivative optimised for block erase.

exercises.lol exercises.com.cn



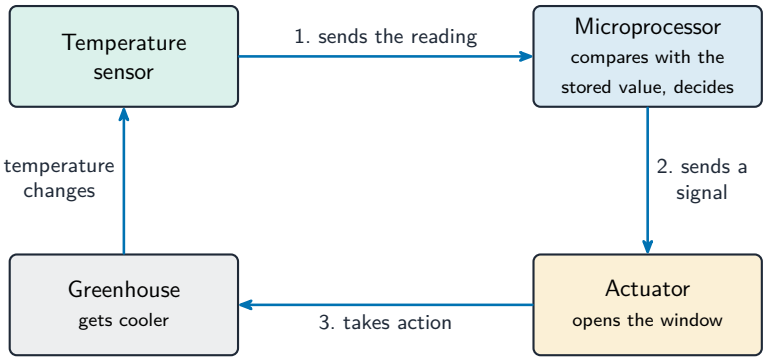
An EPROM chip with a window for ultraviolet erasure

	PROM	**EPROM**	**EEPROM**
Written	once, by the user with a programmer	many times	many times
Erased by	cannot be erased	ultraviolet light through a quartz window	an electrical signal
Erases	nothing	the whole chip at once	a byte or block at a time
Must be removed from the circuit to reprogram?	not applicable	yes	no

"Give two differences between EPROM and EEPROM" wants two rows of this table, each stated for both types.

Monitoring and control systems

Both read **sensors**; the difference is what they do next.



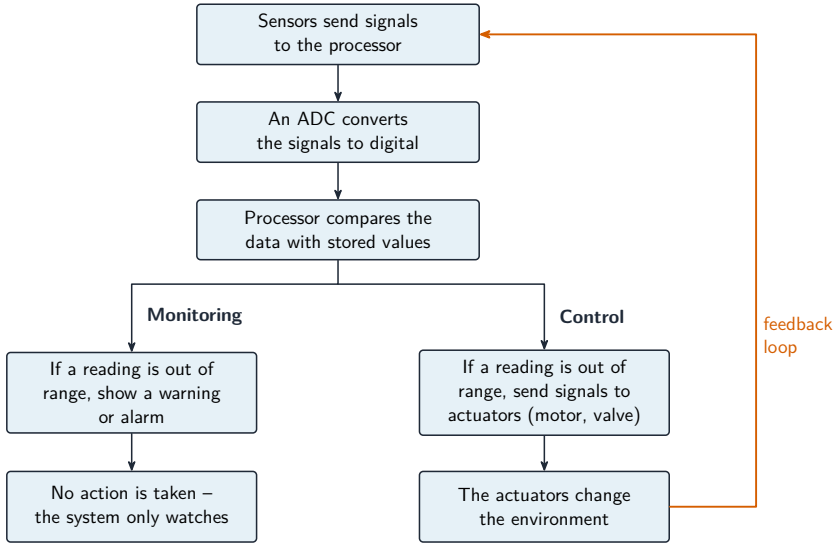
A sensor feeds the microprocessor, which compares the reading with a stored value and signals an actuator; the loop then repeats

- **monitoring** 监控 – collects and reports data but **takes no action** (a weather station logging readings).
- **control system** 控制系统 – uses sensor data to **decide and act** through actuators, usually in a feedback loop (a thermostat turning a boiler on/off).

The three-mark "describe the differences" answer: a monitoring system only measures, records or displays the readings, and at most raises a warning; a control system compares each reading with a **preset value** 预设值 and, if it is outside the range, sends signals to actuators that change the physical process; the change is then measured again, so a control system contains feedback and a monitoring system does not. Whether a given system is one or the other is decided by that test: a bridge system that measures a vehicle's height and switches on a warning sign is monitoring, because nothing it does changes the vehicle; a system that lowers a barrier is control.

> **Worked example.** Describe how an automated system opens a door when a person is within 2 metres and closes it when nobody is.

An infra-red or ultrasonic sensor measures the distance to anything in front of the door; the analogue reading is converted to digital by an ADC and sent to the processor; the processor compares the distance with the preset 2 metres; if it is less, the processor sends a signal to the actuator (a motor) to open the door; the sensor keeps measuring, and when no reading below 2 metres is received the processor signals the motor to close the door. The repeated measuring after each action is the feedback that stops the door opening and closing at the wrong times.



Monitoring reports data; a control system acts through a feedback loop

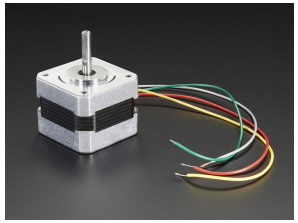
Sensors and actuators

A **sensor** 传感器 turns a physical quantity into a signal: temperature (a **thermistor** 热敏电阻 or thermocouple), pressure (strain gauge), infra-red, sound. Analogue signals need an ADC first. An **actuator** 执行器 does the reverse – turns a signal into an action (a motor, valve, heater, buzzer).



A thermistor: a temperature sensor whose resistance changes with heat

exercises.lol exercises.com.cn



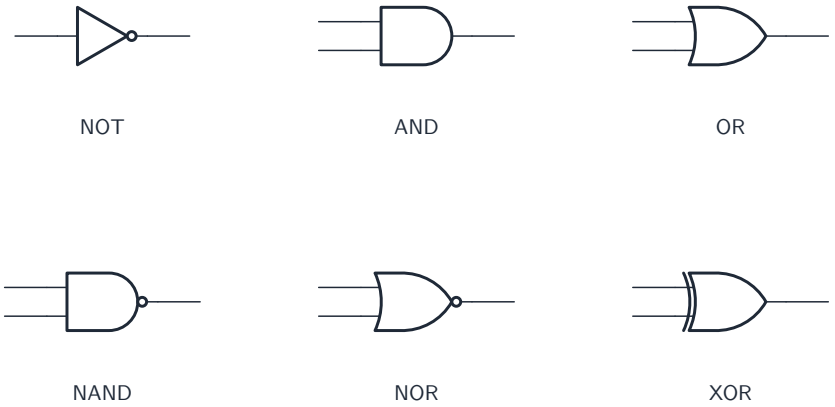
A small electric motor: an actuator that turns a signal into movement

Feedback

In a control system the actuator changes the environment, which the sensors then re-measure – a **feedback** 反馈 loop. Without feedback the system cannot correct itself or know when to stop (a thermostat with no temperature feedback would heat forever).

Logic gates

A **logic gate** 逻辑门 is a small circuit that does one **Boolean** 布尔 operation. Inputs and outputs are 0 (false, low) or 1 (true, high). Know the symbol, function and **truth table** 真值表 for each gate.



The symbols for the six logic gates

exercises.lol exercises.com.cn

NOT (inverter)

A	NOT A
0	1
1	0

AND – output 1 only if all inputs are 1

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

OR – output 1 if at least one input is 1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

NAND (NOT AND) – output 0 only when all inputs are 1

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

NOR (NOT OR) – output 1 only when all inputs are 0

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

■ **XOR (Exclusive OR, also called EOR) – output 1 if the inputs are different**

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

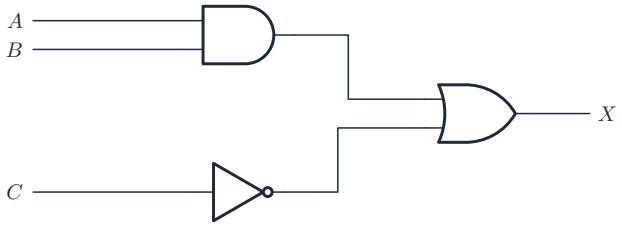
Logic circuits

A **logic circuit** 逻辑电路 is a network of gates that carries out a Boolean expression. You should be able to move between a problem statement, a logic expression, a truth table, and a circuit diagram.

The paper writes expressions in words, `X = (A AND NOT B) OR (B AND C)`, and accepts the algebraic form $X = A\overline{B} + BC$ where a dot (or nothing) is AND, a plus is OR, and a bar is NOT. Use whichever the question uses.

■ From expression to circuit

Draw one gate per operator and wire them up. For $X = (A \text{ AND } B) \text{ OR } (\text{NOT } C)$: a NOT gate on C, an AND gate on A and B, then an OR gate on the two results.



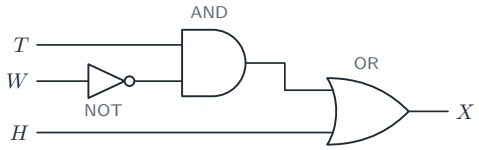
Gates wired together to carry out a Boolean expression

■ From circuit to expression

Work forwards from the inputs, labelling each gate's output, until you reach the final output.

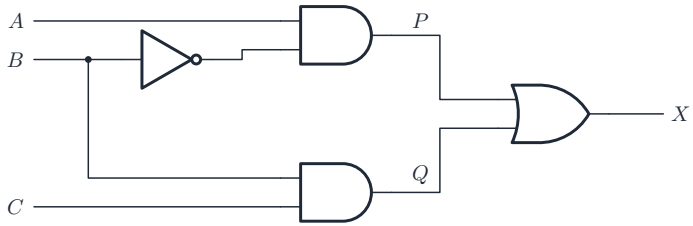
exercises.lol exercises.com.cn

$X = (T \textbf{ AND NOT } W) \textbf{ OR } H$



The greenhouse fan: one NOT, one AND and one OR gate

Worked example. Write the expression for the circuit below, then complete its truth table.



Label every intermediate output: here P is A AND NOT B and Q is B AND C, so X is P OR Q

Label the gate outputs: $P = A$ AND NOT B, $Q = B$ AND C, so $X = P$ OR $Q = (A \text{ AND NOT } B) \text{ OR } (B \text{ AND } C)$. Then give the truth table a column for each intermediate output, so every row can be checked one gate at a time:

A	B	C	NOT B	P	Q	X
0	0	0	1	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	0	1	1	1	0	1
1	1	0	0	0	0	0
1	1	1	0	0	1	1

Drawing a circuit from an expression is the same walk in reverse: start from the innermost brackets, draw one gate per operator, draw a NOT gate on the wire of any input that appears with NOT, keep the inputs on the left and the single output on

the right, and label the output with its letter. Every line must end at a gate input or the output; a line that goes nowhere loses the mark.

From circuit to truth table

For n inputs there are 2^n rows. List every input combination; for each, work out the internal gates then the output.

From truth table to expression (sum of products)

For each row that outputs 1, write an AND of the inputs (with NOT on any input that is 0 in that row); OR these together. Example: a table that is 1 only on $(A=0, B=1)$ and $(A=1, B=0)$ gives $\overline{A}B + A\overline{B}$, which is A XOR B.

From a problem statement

Turn the English into a Boolean expression first: "A and B" → A AND B; "A or B or both" → A OR B; "exactly one of A and B" → A XOR B; "neither A nor B" → A NOR B; "not both" → A NAND B.

> **Worked example.** A machine's alarm X sounds when the guard is open ($A=1$) **and** either the motor is running ($B=1$) or the temperature is high ($C=1$). Write the Boolean expression, and give the rows where $X=1$. Turn the English into logic one clause at a time: "either B or C" is $B+C$, and "A **and** that" is $X = A \cdot (B+C)$. For the rows, $X=1$ needs $A=1$ **and** at least one of B, C equal to 1 - so $(A,B,C) = (1,0,1)$, $(1,1,0)$ and $(1,1,1)$, three rows out of eight. Notice $A=0$ can never sound the alarm, whatever B and C do. **Bracket the OR** before ANDing it: $X = A \cdot B + C$ is a different circuit altogether, one that would sound the alarm on a high temperature even with the guard **closed**.

Flash cells and light sensors

In a floating-gate flash cell, an insulated floating gate traps charge without continuous power. A voltage on the **control gate** is used to program or read the cell; trapped charge changes the voltage threshold at which the transistor conducts, so the read circuit distinguishes stored states. Erasing removes the trapped charge. **NAND flash** connects cells in strings and is suited to dense block storage; **NOR flash** allows direct access to individual locations, useful for firmware. Neither requires moving parts.

A **light sensor** measures light intensity, for example to control a street lamp when daylight falls. Its analogue signal can pass through an ADC before the processor compares it with a threshold. State the measured physical quantity, the conversion if needed, and the resulting control action.

exercises.lol exercises.com.cn

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
embedded system	a computer system with a dedicated function built into a larger device
buffer	an area of memory that temporarily stores data while it is transferred between devices working at different speeds
RAM	volatile memory that can be read from and written to, holding the programs and data in use
ROM	non-volatile memory whose contents cannot be changed in normal use, holding the start-up instructions
SRAM	static RAM that stores each bit in a flip-flop and needs no refreshing
DRAM	dynamic RAM that stores each bit as a charge on a capacitor and must be refreshed continually
monitoring system	a system that uses sensors to measure and report on a physical process without changing it
control system	a system that uses sensor readings to decide on and carry out actions, through actuators, that change a physical process
sensor	a device that measures a physical quantity and converts it into a signal for the computer
actuator	a device that converts a signal from the computer into a physical action
feedback	the output of a control system being measured and fed back as input so that the system can correct itself
logic gate	an electronic circuit that performs a Boolean operation on one or more binary inputs to give one binary output
truth table	a table listing every combination of inputs to a logic circuit with the output for each

Exam tips

- Distinguish **RAM (volatile, read/write)** from **ROM (non-volatile, holds the bootstrap)**; SRAM (cache, faster) from DRAM (main memory, needs refreshing).
- For a **logic circuit**, build the Boolean expression gate by gate, then a truth table covering **every** input combination.
- Learn the symbol, expression and truth table for each gate (AND, OR, NOT, NAND, NOR, XOR).
- Explain a **buffer** (a temporary store bridging two different speeds) and the role of an interrupt.

Common mistakes

- Naming the device instead of describing its operation. "It uses a laser" earns nothing; the steps (charge the drum, laser removes charge, toner attracted, transferred, fused) earn the marks.
- Saying a monitoring system "controls" something. If nothing changes the physical process, it is monitoring; add the actuator and the feedback and it becomes control.
- Writing that RAM "stores files permanently" or that ROM "stores the user's data". RAM is volatile working memory; ROM holds the fixed start-up instructions.
- A truth table with fewer than 2^n rows, or rows in a random order. Count in binary from 000 to 111 so no combination is missed.
- Drawing two lines from one output of a gate to be safe, or leaving a wire that ends nowhere. Draw exactly the connections the expression needs.

exercises.lol exercises.com.cn

Processor Fundamentals

A-Level Computer Science

Von Neumann architecture

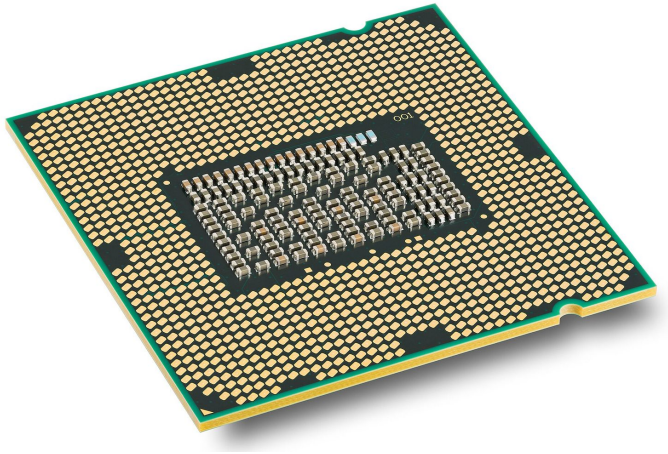
The **Von Neumann architecture** 冯 · 诺依曼体系结构 underlies almost every general-purpose computer:

- a **single memory** – the **Immediate Access Store** 立即存取存储器 (IAS) – holds both **program instructions** and **data** (the **stored program** 存储程序 concept).
- a **processor** 处理器 (CPU) fetches instructions from memory and runs them one at a time.
- instructions run **in order** unless a branch changes the flow.

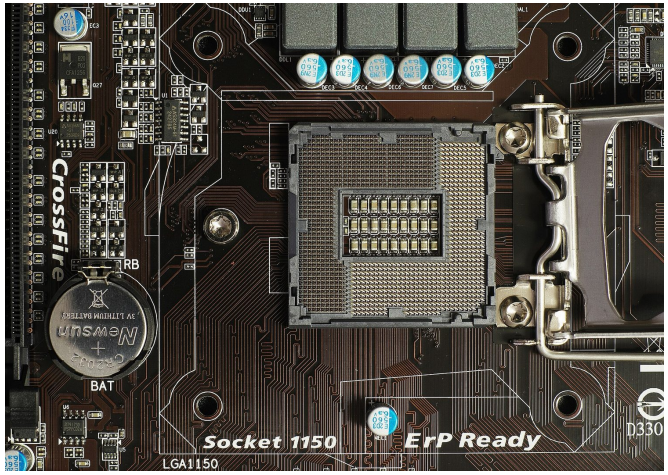
The stored-program idea is what makes a computer flexible: change the program and you change what it does, with no rewiring.

The CPU's main parts

All of these parts sit inside one small chip. The diagram later in this section shows how they connect; the photo below shows the real thing.



A modern CPU: the whole processor is one small chip (here seen from below, showing the contacts)



The matching CPU socket on the motherboard: the chip's contacts press onto these pins

Arithmetic and Logic Unit (ALU)

The **ALU** 算术逻辑单元 does the arithmetic (add, subtract, …) and logic (AND, OR, comparisons). It takes operands from **registers** 寄存器 and puts results back in a register.

Control Unit (CU)

The **control unit** 控制单元 **decodes** each instruction and sends the **control signals** to carry it out – opening data paths, telling the ALU what to do, and controlling memory reads and writes.

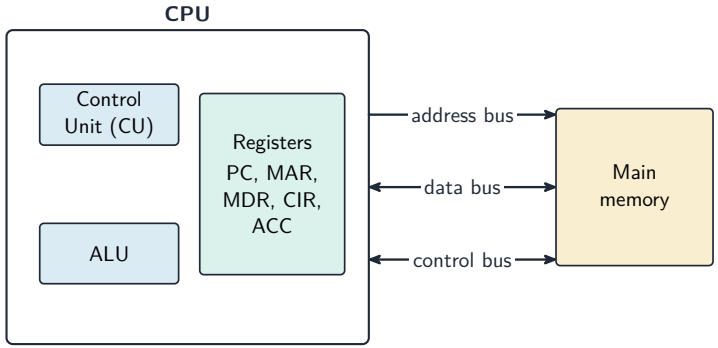
System clock

The clock sends a steady stream of pulses that keep the CPU in step. Each instruction takes a fixed number of cycles, and the **clock speed** 时钟频率 (e.g. 3.8 GHz) is one factor in performance.

"Explain how the CU and the system clock work together": the clock emits pulses at a fixed frequency; the control unit uses each pulse to move the fetch-execute cycle on by one step, sending its control signals in time with the pulses, so every part of the processor changes state together. A faster clock means more steps per second, up to the point where the circuits cannot settle between pulses.

Registers

Registers are tiny, very fast stores inside the CPU. The **special purpose registers** 专用寄存器 each have a fixed job in the cycle:



The CPU holds the control unit, the ALU and the registers; three buses link it to main memory

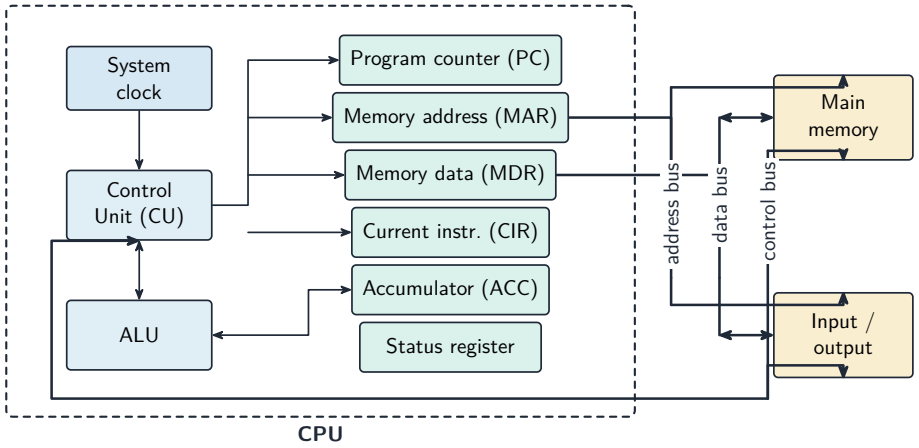
- **Program Counter** 程序计数器 (PC) – the address of the **next** instruction.
- **Memory Address Register** 内存地址寄存器 (MAR) – the address being read or written.
- **Memory Data Register** 内存数据寄存器 (MDR) – the data going to or from memory.
- **Current Instruction Register** 当前指令寄存器 (CIR) – the instruction being decoded.
- **Accumulator** 累加器 (ACC) – the value the ALU is working on.
- **Status Register** 状态寄存器 – holds **flags** 标志 (carry, zero, negative, overflow) used by branches. Each flag is one bit, set or cleared by the ALU after an operation: the zero flag after a comparison that matched, the carry flag when an addition overflowed the register, the negative flag when a result is negative. A conditional jump reads the flags to decide whether to branch, and an overflow flag can raise an interrupt.
- **Index Register** 变址寄存器 – an offset added to an address in indexed addressing; incrementing it steps through an array one element at a time.

The "complete the table describing the role of each register" question wants one precise sentence per register in these terms: the PC holds the address of the next instruction to be fetched; the MAR holds the address of the location being read from or written to; the MDR holds the data or instruction just read from, or about to be written to, that location; the CIR holds the instruction currently being decoded and executed; the ACC holds the result of the last arithmetic or logic operation.

General-purpose registers 通用寄存器 are used by the programmer for temporary

exercises.lol exercises.com.cn

values during a calculation. Movements of data between registers and memory are written in **register transfer** 寄存器传送 notation – e.g. `MAR ← [PC]` ("copy the contents of PC into MAR").



The Von Neumann CPU: registers, control unit and ALU linked by buses

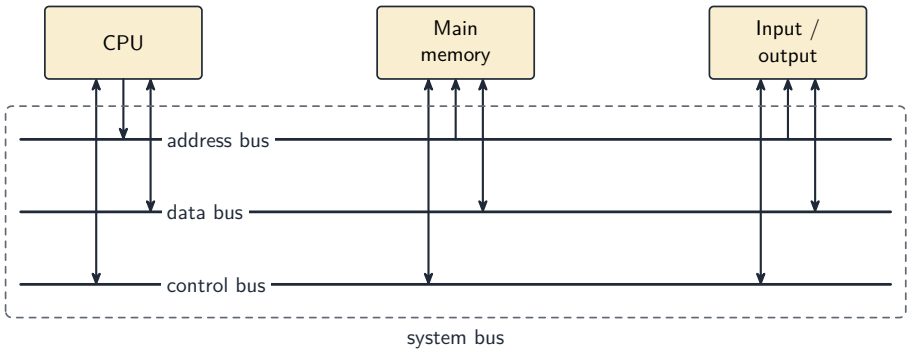
Buses

Three internal **buses** 总线 (sets of parallel wires) connect the parts:

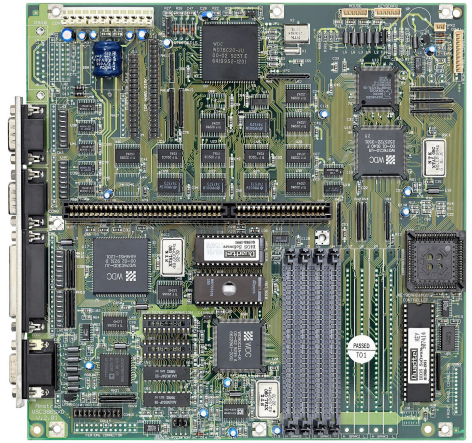
- **address bus** 地址总线 – carries the memory address. **One-way** (CPU → memory).
- **data bus** 数据总线 – carries the data. **Two-way**.
- **control bus** 控制总线 – carries control signals (read, write, interrupt). **Two-way**.

An n-bit address bus can reach 2^n memory locations. The data-bus width sets how many bits move per access (often the word size).

exercises.lol exercises.com.cn



The three system buses connecting the CPU, memory and input/output

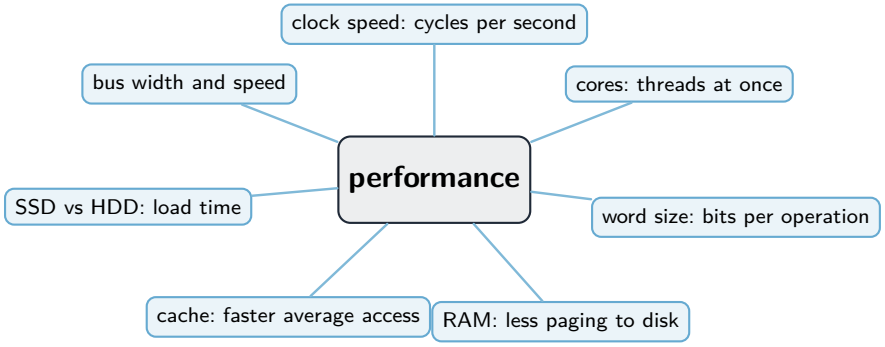


A motherboard: the CPU, memory and I/O all sit on one set of buses – the printed tracks running between them

What affects performance

- **clock speed** – more cycles per second.
- **number of cores** 核心 – a multi-core CPU runs several threads at once.
- **word size** 字长 – a 64-bit CPU handles 64-bit chunks per cycle and can address far more memory than a 32-bit one.
- **amount of RAM** 随机存取存储器 – more RAM holds more of the working set; too little forces the OS to **page** 页 to disk.
- **cache memory** 高速缓存 size – more cache cuts average memory access time.

- **secondary storage** 辅助存储器 type – an SSD loads programs far faster than an HDD.
- **bus width and speed** – wider/faster buses move data more quickly.



Seven factors, each worth a reason in the answer.

Match the specs to the workload: a quad-core beats a dual-core on parallel work, but higher per-core speed wins on single-threaded work.

Each factor is a two-mark answer with a reason attached:

- **More cores**: each core can fetch and execute its own instruction at the same time, so several programs, or the threads of one program, run in parallel. But a program must be written to use more than one core, so doubling the cores does not double the speed.
- **Higher clock speed**: more fetch-execute cycles per second, so more instructions per second; the limit is the heat produced.
- **Wider bus**: a wider data bus moves more bits in each transfer, so fewer transfers are needed for the same data; a wider address bus can address more memory locations.
- **Cache memory**: a small, fast memory inside or next to the processor that keeps the instructions and data used most recently or most often. Reading them from cache is much faster than from RAM, so the processor spends less time waiting.

"Explain why the new computer performs better" is answered by comparing the two specifications line by line: a higher clock speed executes more instructions per second, more cores run more tasks at once, more cache means fewer slow accesses to RAM, and more RAM means fewer transfers to disk.

Ports

A **port** 端口 is a physical socket for connecting a **peripheral** 外围设备:

- **USB** (Universal Serial Bus) – general-purpose (keyboards, drives, phones).
- **HDMI** (High Definition Multimedia Interface) – digital video and audio to a screen.
- **VGA** (Video Graphics Array) – older analogue video output to a monitor.
- **Ethernet (RJ-45)** – wired LAN. Audio jacks – headphones/microphone.

Different ports use different signals, so an HDMI cable will not fit a USB socket. USB-C is unusual in carrying video, data and power.

"Explain how the computer connects to the monitor through HDMI": the HDMI port sends the video and the audio as one digital signal down a single cable, so no conversion to analogue is needed and the picture is not degraded; the cable carries high-definition resolutions and the monitor's own port decodes the signal. A USB device is **plug-and-play**: when it is connected the computer detects it, identifies it, loads or installs the driver it needs, and can supply it with power, all without a restart.

Fetch-Execute cycle

The CPU repeats the **fetch-execute cycle** 取指-执行周期, one run per machine instruction.

Fetch

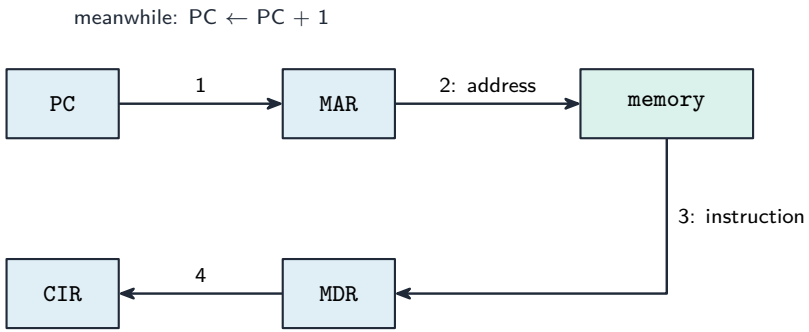
1. the PC's address is copied to the MAR.
2. the PC is **incremented** to point to the next instruction.
3. a **read** signal goes over the control bus.
4. memory puts the instruction on the data bus.
5. it is copied into the MDR, then into the CIR.

The exam asks for these steps in **register transfer notation** 寄存器传送记法, where `[X]` means the contents of register X and `[[MAR]]` means the contents of the memory location whose address is in the MAR:

```
MAR ← [PC]          the address of the next instruction goes to the MAR
PC  ← [PC] + 1      the PC now points to the following instruction
MDR ← [[MAR]]       the instruction at that address is read into the MDR
CIR ← [MDR]         the instruction is copied into the CIR for decoding
```

The order matters: the PC is incremented straight after its address has been copied, so that a jump executed later can still overwrite it. During execution the same notation describes each instruction; for `LDD 200`, for example, `MAR ← 200`, `MDR ← [[MAR]]`, `ACC ← [MDR]`.

exercises.lol exercises.com.cn



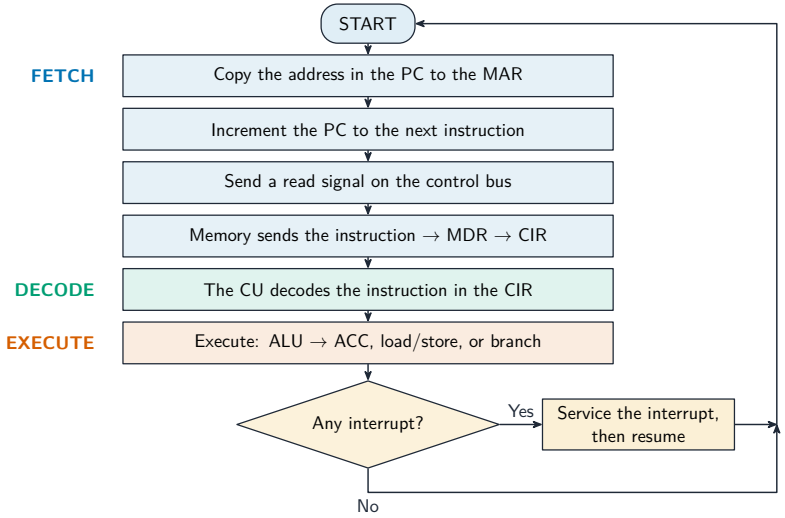
The register transfers in a fetch: PC → MAR → memory → MDR → CIR, with the PC incremented

Decode

The CU decodes the instruction in the CIR – what operation, and which operands or addresses.

Execute

The CU carries it out: arithmetic/logic goes to the ALU (result to the ACC); a load/store moves data between memory and a register; a branch changes the PC. Then the cycle repeats.

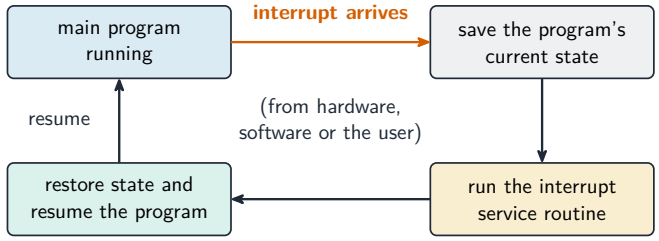


The fetch-execute cycle, with a check for interrupts each time

exercises.lol exercises.com.cn

Interrupts

An **interrupt** 中断 is a signal that **pauses** the normal cycle so the CPU can handle an urgent event (a key press, a packet arriving, a hardware fault, division by zero, the OS timer).



On an interrupt the CPU saves its place, runs the interrupt service routine, then restores its place and carries on

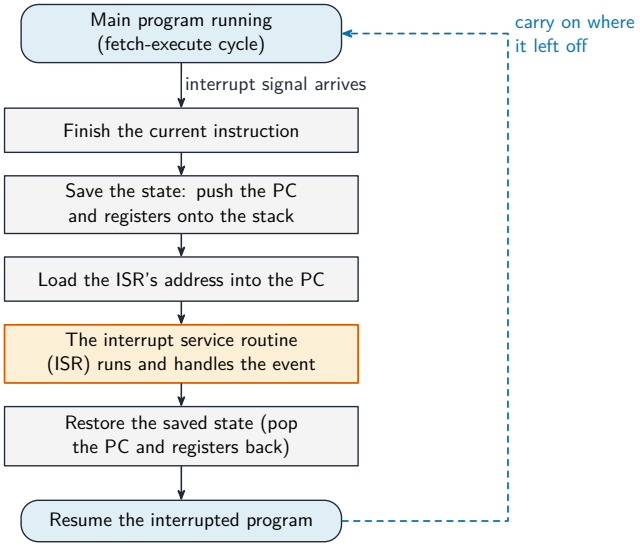
Handling one:

1. finish the current instruction.
2. **save the state** (PC and registers).
3. load the address of the **interrupt service routine** 中断服务程序 (ISR) into the PC and run it.
4. the ISR handles the event.
5. **restore** the saved state and carry on.

Interrupts let the system respond promptly without the CPU constantly checking devices, and are how the OS multitasks.

"Explain how an interrupt from an input device is detected and handled in the F-E cycle" is a four-mark answer with these points: the device sends an interrupt signal that sets the interrupt flag in the **interrupt register** 中断寄存器; the processor checks that register at the **end of every fetch-execute cycle**, after the current instruction has finished executing; if a flag is set and the interrupt has a higher priority than the current task, the contents of the PC and the other registers are saved onto the **stack** 栈; the address of the interrupt service routine is loaded into the PC and the routine runs; when it finishes, the saved values are restored from the stack and the interrupted program continues from where it stopped.

Causes worth naming: a hardware interrupt from a device (a key pressed, a printer buffer empty, a network packet arriving), a software interrupt from a fault (division by zero, an illegal instruction, arithmetic overflow), a timer interrupt from the operating system marking the end of a time slice, and a power failure warning.



How an interrupt fits into the fetch-execute cycle

Assembly language and machine code

The CPU actually runs **machine code** 机器码 – bit patterns, specific to one architecture. **Assembly language** 汇编语言 is a readable form, with one instruction per machine instruction, written using **mnemonics** 助记符 like `LDD`, `ADD`, `JMP`. An **assembler** 汇编器 translates it to machine code.



An assembler turns mnemonics into machine-code bit patterns

Two-pass assembler

A two-pass assembler reads the source twice:

- **pass 1** builds a **symbol table** 符号表: each time a **label** 标签 (like `LOOP:`) appears, record its address; no code yet.
- **pass 2** generates code: translate each instruction, and when one refers to a label (like `JMP LOOP`), look up its address in the symbol table.

Two passes handle **forward references** 前向引用 (a jump to a label defined later).

exercises.lol exercises.com.cn

Worked example. Apply the two-pass process to this program, whose first instruction is stored at address 100.

```
        LDD  COUNT
LOOP:   DEC  ACC
        CMP  #0
        JPN  LOOP
        END
COUNT:  5
```

Pass 1 reads each line, counts the address it will occupy, and records every label in the symbol table: `LOOP` = 101 (the `DEC` line) and `COUNT` = 105 (the data line). No code is produced. Pass 2 reads the program again and translates each line into machine code, replacing each mnemonic by its **opcode** 操作码 and each symbolic address by the number from the symbol table: `LDD COUNT` becomes the opcode for `LDD` with **operand** 操作数 105, and `JPN LOOP` becomes the opcode for `JPN` with operand 101. The jump back to `LOOP` could have been resolved in one pass, but a jump *forward* to a label not yet seen could not, which is why the assembler makes two.

Example instruction set

Cambridge uses a small generic set, printed in the paper's reference table, with one general-purpose register, the accumulator (ACC), and an index register (IX). An operand written `#n` is a denary number, `Bn` a binary number and `&n` a hexadecimal number; `<address>` is a location number or a label.

Group	Instruction	What it does
Data movement	`LDM #n`	load the number n into ACC (immediate)
	`LDD <address>`	load the contents of the address into ACC (direct)
	`LDI <address>`	the address holds another address; load the contents of that one into ACC (indirect)
	`LDX <address>`	add IX to the address and load the contents of the result into ACC (indexed)
	`LDR #n`	load the number n into IX
	`MOV <register>`	copy ACC into the named register (IX)
	`STO <address>`	store the contents of ACC at the address

Group	Instruction	What it does
Input and output	`IN`	read a key press and put its ASCII code in ACC
	`OUT`	output the character whose ASCII code is in ACC
Arithmetic	`ADD <address>` / `ADD #n`	add the contents of the address, or the number, to ACC
	`SUB <address>` / `SUB #n`	subtract from ACC
	`INC <register>` / `DEC <register>`	add 1 to, or subtract 1 from, ACC or IX
Compare	`CMP <address>` / `CMP #n`	compare ACC with the contents of the address, or with n, and set the flag
	`CMI <address>`	compare ACC with the contents of the address held at the address (indirect)
Jump	`JMP <address>`	jump to the address unconditionally
	`JPE <address>` / `JPN <address>`	jump if the last compare was equal / not equal
Bit manipulation	`AND`, `OR`, `XOR` with `#n`, `Bn`, `&n` or `<address>`	bitwise operation on ACC
	`LSL #n` / `LSR #n`	shift ACC logically n places left or right
	`END`	end the program

The "assembly language instructions are grouped" question wants the group names, and an instruction from each: data movement, input and output, arithmetic, unconditional and conditional jumps, compare, and bit manipulation.

Addressing modes

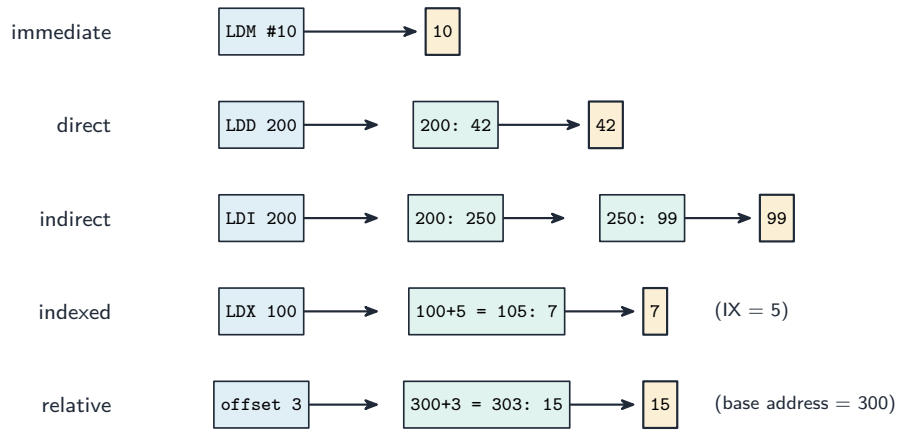
The **addressing mode** 寻址方式 (the **modes of addressing**) says how the CPU finds the operand:

- **immediate addressing** 立即寻址 – the operand is the value in the instruction. `LDM #10` loads 10.
- **direct addressing** 直接寻址 – the instruction holds an address; the operand is the value there. `LDD 200`.

exercises.lol exercises.com.cn

- **indirect addressing** 间接寻址 – the instruction holds an address that holds **another** address, which is the data. `LDI 200`.
- **indexed addressing** 变址寻址 – effective address is `address + index register`; used for arrays. `LDX 100` with IX = 5 reads address 105.

(**Relative addressing** 相对寻址 gives the address as an offset from the PC – used for jumps.)

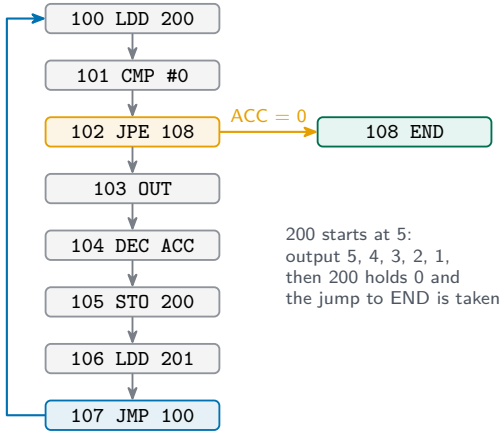


How each addressing mode reaches its operand – immediate, direct, indirect and indexed

> **Worked example.** Memory holds: location `200` = `250`, location `250` = `99`, location `105` = `7`. The index register holds `5`. What is in the accumulator after each of `LDM #200`, `LDD 200`, `LDI 200` and `LDX 100`? Follow how far each mode has to look. `LDM #200` is **immediate** - the operand *is* the number written in the instruction, so the accumulator holds **200**. `LDD 200` is **direct** - go to location 200 and take what is there: **250**. `LDI 200` is **indirect** - location 200 holds 250, which is *another address*, so go on to location 250: **99**. `LDX 100` is **indexed** - add the index register to the address, $100 + 5 = 105$, and read location 105: **7**. Count the hops to keep them apart: immediate 0, direct 1, indirect 2, indexed 1 (once the index has been added).

Tracing an assembly program

To **trace** it: make a table with columns for the PC, ACC, index register, each variable and any flags. Step through the instructions, updating the table after each; follow branches when they change the PC; stop at `END`. A common pattern is a loop over an array using indexed addressing.



Follow the jumps to see the loop before you trace it.

Worked example. Trace this program. Address 200 holds 5 and address 201 holds 0.

```
100   LDD  200
101   CMP  #0
102   JPE  108
103   OUT
104   DEC  ACC
105   STO  200
106   LDD  201
107   JMP  100
108   END
```

Write one row for each instruction executed, filling in only the columns that change:

Instruction	ACC	200	201	Output
start		5	0	
LDD 200	5			
CMP #0				
JPE 108				not taken
OUT				character with code 5
DEC ACC	4			
STO 200		4		
LDD 201	0			
JMP 100				

exercises.lol exercises.com.cn

Instruction	ACC	200	201	Output
LDD 200	4			

and so on, until `LDD 200` loads 0, the compare sets the equal flag, `JPE 108` is taken and the program ends. Three things the examiner checks: a `CMP` changes no register, only a flag; a jump not taken still counts as executed; and `OUT` outputs a character, so it goes in the output column, not the ACC column. "State the effect of changing `LDD 10` to `LDM #10`": the ACC would hold the number 10 instead of the contents of address 10.

Binary shifts

A **logical shift** 逻辑移位 moves all the bits left or right by some places, filling new positions with 0.

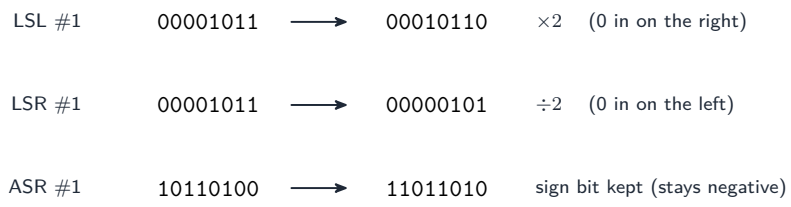
- **left shift by 1** (`LSL #1`) – bits move left, a 0 enters on the right; for an unsigned number this is $\times$ **2**.
- **right shift by 1** (`LSR #1`) – bits move right, a 0 enters on the left; for an unsigned number this is **integer** $\div$ **2**.

Shifting by n places multiplies or divides by 2^n. Example: `00001011` (11) `LSL #1` $\rightarrow$ `00010110` (22).

Bits shifted off the end are lost, so the multiplication is only correct while they were zeros. `LSL #2` on the two's-complement integer `11001010` gives `00101000`: the two 1s that fell off the left are gone, the sign bit has changed, and the result is no longer four times the original.

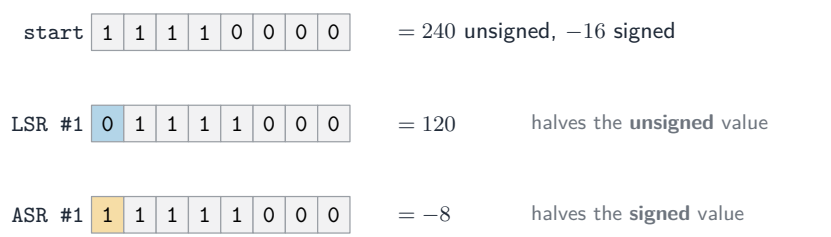
An **arithmetic right shift** keeps the sign bit so a negative signed number stays negative. A **cyclic shift** 循环移位 (rotate) feeds the bit that drops off one end back in at the other end, so no bits are lost.

"Show the result of an arithmetic right shift of 3 places on `10011110`": copy the sign bit into each vacated place, `11110011`. The same shift on `01011100` gives `00001011`. A cyclic left shift of 1 on `10000110` gives `00001101`: the leading 1 reappears on the right.



Logical left ($\times$2), logical right ($\div$2) and arithmetic right (keeps the sign bit)

The difference between the two right shifts is a single bit. Take `11110000`, which is 240 read as unsigned and −16 read as signed. `LSR #1` brings in a 0 and gives `01111000` = 120, which is the correct half of 240. `ASR #1` copies the sign bit instead and gives `11111000` = −8, which is the correct half of −16. Neither is wrong – each halves the value under one reading.



The two results differ in **one bit** — the one that entered on the left. **LSR** always brings in a 0; **ASR** copies the sign bit, so a negative number stays negative and $-16 \div 2$ comes out right.

Logical and arithmetic right shift on the same byte: only the bit that enters on the left differs

■ Bit manipulation for monitoring/control

Embedded devices often use one **bit** 位 of a register per signal (e.g. bit n = LED n). Using a **mask** 掩码 – **bit masking** – you can:

- **set** bit n: `R = R OR` a mask with bit n set.
- **clear** bit n: `R = R AND` a mask with bit n clear and the rest set.
- **toggle** bit n: `R = R XOR` a mask with bit n set.
- **test** bit n: `R AND` the mask, then check if the result is non-zero.

set bit 2		clear bit 6		toggle bit 3	
	01001000		01001000		01001000
OR	00000100	AND	10111111	XOR	00001000
=	01001100	=	00001000	=	01000000

bits are numbered 7 to 0 from the left, so bit 0 is the rightmost bit

Set a bit with OR, clear it with AND, toggle it with XOR – each using a mask

Bit manipulation is fast, uses little memory, and lets one byte hold up to 8 on/off states.

In the exam's instruction set these are `AND`, `OR` and `XOR` with a mask written as a denary, binary or hexadecimal operand. With the ACC holding `10101100`:

Instruction	Mask	Result in ACC	Effect
`AND B00001111`	00001111	00001100	keeps only the low four bits (clears the others)
`OR #1`	00000001	10101101	sets the least significant bit, leaving the rest unchanged
`XOR &FF`	11111111	01010011	inverts every bit
`AND B00001000` then `CMP #0`	00001000	00001000	tests bit 3: the compare is not equal, so bit 3 was set
`LSL #2`		10110000	shifts left two places, losing the top two bits
`LSR #3`		00010101	shifts right three places, zeros entering on the left

"Write the instruction that sets the least significant bit to 1 and leaves the others unchanged": `OR #1`, or `OR B00000001`. To clear a bit use `AND` with a mask that has a 0 in that place and 1s elsewhere; to test a bit, `AND` with a mask that has a 1 only in that place, then compare the result with zero. In a monitoring device, one bit of a register per sensor lets a single `AND` check whether a particular sensor is on, and one `OR` switches an actuator's control bit on without disturbing the others.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
stored program concept	the program instructions and the data are both held in main memory, and instructions are fetched and executed one at a time
register	a small, very fast storage location inside the processor with a specific purpose
Program Counter	the register holding the address of the next instruction to be fetched
Memory Address Register	the register holding the address of the memory location being read from or written to

exercises.lol exercises.com.cn

Term	Definition
Memory Data Register	the register holding the data or instruction just read from, or about to be written to, memory
Current Instruction Register	the register holding the instruction currently being decoded and executed
Accumulator	the general-purpose register holding the result of the last arithmetic or logic operation
cache memory	small, fast memory close to the processor holding frequently used instructions and data
interrupt	a signal from a device or program that causes the processor to pause the current task and run an interrupt service routine
assembly language	a low-level language in which each mnemonic instruction corresponds to one machine-code instruction
immediate addressing	the operand is the value written in the instruction
direct addressing	the operand is the contents of the address written in the instruction
indirect addressing	the address in the instruction holds the address of the operand
indexed addressing	the operand's address is the address in the instruction plus the contents of the index register
relative addressing	the operand's address is given as an offset from the address of the current instruction
logical shift	every bit moves the given number of places and zeros fill the vacated places

Exam tips

- Learn the **fetch-execute cycle** in register-transfer terms (PC, MAR, MDR, CIR, ACC) and what increments the PC.
- Name each register's job; the **address bus is one-way**, the data bus is two-way.
- Distinguish the **addressing modes** (immediate, direct, indirect, indexed) – a frequent question.
- Explain how clock speed, number of cores, cache size and word length affect **performance**.

exercises.lol exercises.com.cn

- For a **binary shift**, state whether it is logical or arithmetic; a left shift multiplies by 2, a right shift divides by 2.

Common mistakes

- Saying the PC holds the current instruction, or the MDR holds an address. The PC holds the address of the *next* instruction; the MDR holds data or an instruction, never an address.
- Leaving the increment of the PC out of the fetch, or putting it after the execute. It happens as soon as the address has been copied to the MAR.
- Reading `LDD 10` as "load 10". `LDD 10` loads the contents of address 10; `LDM #10` loads the number 10.
- Putting a value in the ACC column for `CMP` or `OUT`. A compare sets a flag only; an output goes to the output column.
- Saying an interrupt is handled "immediately". The processor finishes the current instruction and checks for interrupts at the end of the cycle.
- Using a logical right shift on a negative two's-complement number. Only an arithmetic shift keeps the sign bit.

System Software

A-Level Computer Science

Operating systems

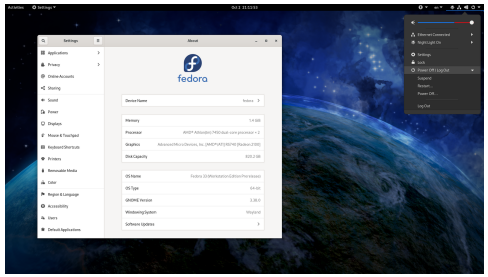
Why a computer needs an OS

Hardware on its own can only fetch and run instructions – it knows nothing about files, programs, networks or users. The **operating system** 操作系统 (OS) is the **software layer** that:

- **manages the hardware** (**processor** 处理器, memory, I/O, storage) for the running programs.
- **provides services** (file system, network, user accounts) through a clear interface, so programs need not talk to the hardware directly.
- **provides a user interface** (command line, GUI, touch).
- **lets several programs share the hardware safely** – each gets fair CPU time and is kept out of the others' memory.

Without an OS, every program would need its own drivers, and only one program could safely run at a time.

"Describe the purpose of an OS" – the five-mark list. The OS (1) provides an **interface** between the user and the hardware; (2) **hides the complexity** of the hardware from the user and from application programs; (3) **manages the hardware resources** – processor time, memory, storage and input/output devices – and shares them between programs; (4) loads application software into memory and runs it, giving every program the same platform to run on; (5) lets several programs run at once (**multitasking** 多任务处理) while keeping them, and the users' data, secure. Give five *different* points; "it runs the computer" or "it manages resources" alone earns nothing.



A desktop operating system manages the screen, files and programs for the user

exercises.lol exercises.com.cn

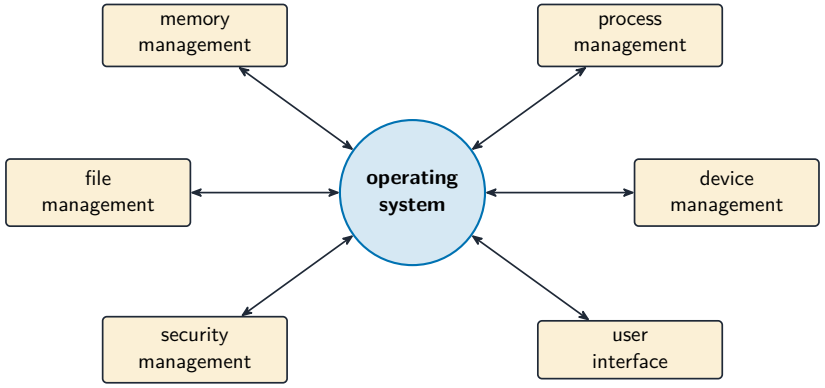


A phone needs one just as much: this is a mobile operating system, Android

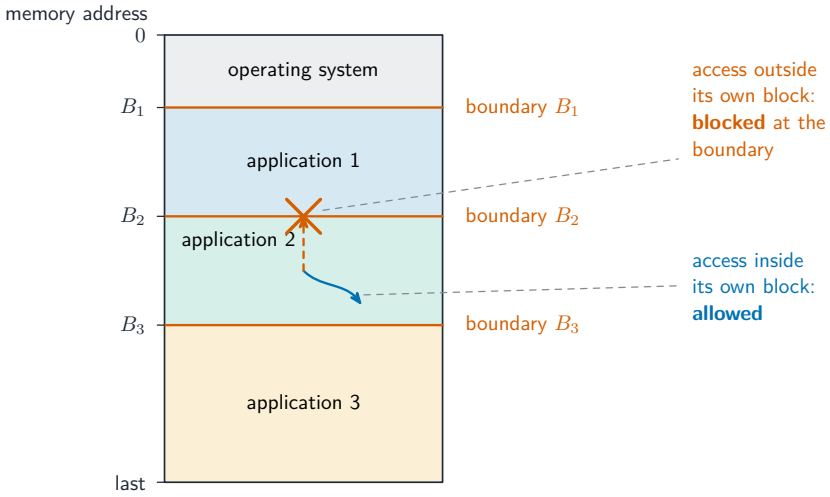
Key management tasks

The syllabus names five. Each point below is one thing the OS actually does, which is what a "describe" question wants.

- **memory management** 内存管理 – allocates memory to each program when it is loaded, keeps every program's memory separate (**memory protection** 内存保护) so one cannot overwrite another, frees the memory when a program ends, and swaps pages between **RAM** 随机存取存储器 and **secondary storage** 辅助存储器 (the disk) (**virtual memory** 虚拟内存 / **paging** 分页) so more programs can be open than physical memory allows.
- **process management** 进程管理 – a running program is a **process** 进程. The OS creates and ends processes, decides which process gets the CPU next (**scheduling** 调度) and for how long (a **time slice** 时间片), switches between them, resolves conflicts when two want the same resource, and can kill one that stops responding.
- **hardware management** (input/output and peripherals) – talks to each device through its **device driver** 设备驱动, queues and buffers data going to slow devices such as a printer, responds to **interrupts** 中断 from devices, and shares one device between several programs.
- **file management** – creates, names, copies, moves and deletes files and folders, keeps the **directory** 目录 structure and a record of where each file is stored on the disk, allocates disk space to files, and enforces **access rights** 访问权限 (read / write / execute) for each user.
- **security management** – user accounts and passwords (**authentication** 身份验证), access rights, encryption of stored data, a firewall, automatic security updates, and a log of who did what.



The main jobs the operating system manages



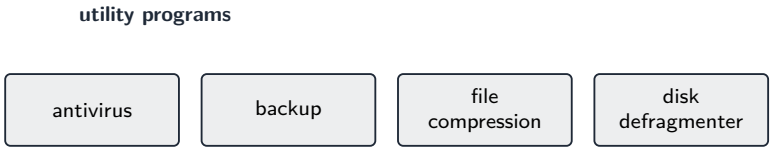
Memory protection keeps each application in its own block of memory

How memory and process management support multitasking (a four-mark favourite). Memory management loads several programs into memory at the same time, each in its own protected area, and keeps track of which addresses belong to which; process management shares the processor between them – each process runs for a time slice, the OS saves its state and switches to the next, and the switching is so fast that all the programs appear to run together. Interrupts let the OS take the processor back from a process whenever a device needs attention.

Interrupts. A **hardware interrupt** 硬件中断 comes from a device: a key pressed, a mouse click, a printer out of paper, a disk finishing a transfer, a power failure. A **software interrupt** 软件中断 comes from a program: division by zero, an invalid instruction, an attempt to use memory it does not own, or a request for an OS service. The OS's **interrupt handler** 中断处理程序 saves the state of the running process, deals with the interrupt, then restores the process (topic 4 covers the fetch–execute detail).

Utility software

Utility programs 实用程序 are system software that maintain, repair or optimise the computer rather than doing a user's task; the examiner accepts "performs a specific maintenance task that improves performance or security". Most OSes bundle these:



Utility programs: antivirus, backup, compression and defragmenter

- **disk formatter** – prepares a new disk (or wipes an old one) for use: sets up its **file system** and partitions, deleting any existing data.
- **virus checker** (**antivirus** 杀毒软件) – scans files and memory, compares code against a database of known virus **signatures** 签名 and watches for suspicious behaviour, then quarantines or deletes what it finds; runs on a schedule and on every download, and needs updating as new viruses appear.
- **defragmentation software** (**disk defragmenter** 碎片整理) – a hard disk stores a file in whatever free blocks it finds, so after many saves and deletes a file is scattered (**fragmented** 碎片化) across the platter and the read/write head must jump between the pieces. The defragmenter moves the pieces of each file next to each other and gathers the free space into one region, so files load faster and new files are not fragmented. Not needed on an SSD, which has no moving head.
- **disk contents analysis / disk repair software** – shows what is using the disk space (large, duplicate or temporary files) so they can be removed; finds and repairs bad sectors, lost clusters and file-system errors.
- **file compression** (**compression** 压缩) – shrinks files so they need less storage and transfer faster; archiving bundles many files into one.
- **back-up software** (**backup** 备份) – copies files to another medium (external disk, network, cloud) on a schedule so data can be restored after loss, corruption or a ransomware attack; a full copy is followed by **incremental backups** 增量备份 of only what changed.

exercises.lol exercises.com.cn

- a **firewall** 防火墙 (filters network traffic by rules) and encryption tools, for security; a **system monitor** and automatic **updates**.

Bundling these with the OS saves the user installing each one.

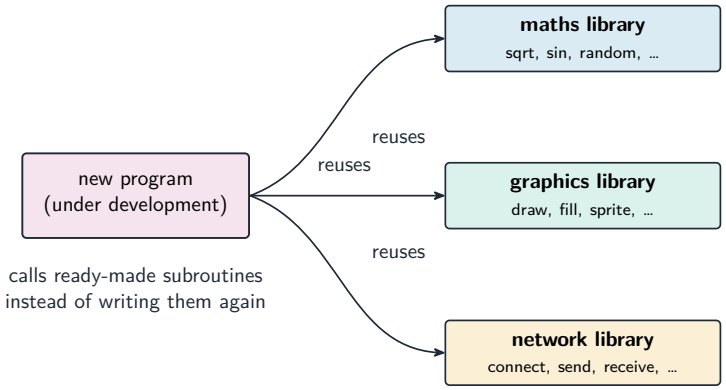
Which utility does what. *Performance:* defragmentation (faster file access), disk repair (a disk with errors is slow or fails), disk contents analysis (free space by deleting junk), compression (more fits on the disk). *Security:* virus checker, firewall, encryption, and backup (the only recovery from ransomware). A "draw one line" question pairs each utility with exactly one purpose – learn the pairs above and use the syllabus names.

> **Worked example.** Explain how defragmentation can improve the performance of a computer (3 marks).

Over time a file is stored in blocks scattered across the hard disk, so reading it needs many movements of the read/write head. The defragmenter rearranges the blocks so each file is stored **contiguously** and the free space is together. Files are then read with fewer head movements, so they load faster, and new files can be written into one continuous space.

Program libraries

A **program library** 程序库 is pre-written code (**subroutines** 子程序, classes, modules) that programs reuse instead of writing it themselves – e.g. a maths library, a network library, a graphics library.

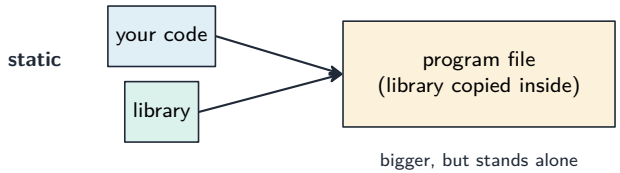


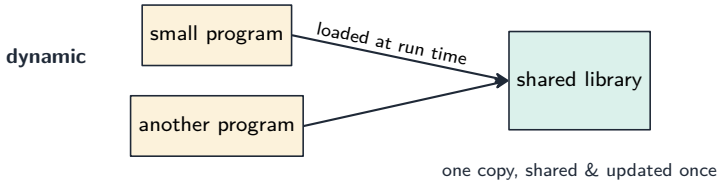
A new program reusing ready-made routines from libraries

Benefits: **saves time** (off-the-shelf code), **reliable** (well-tested, widely used), and **standardised** (consistent behaviour).

The examiner's benefit list, for the developer. The **library routines** 库例程 are already written and tested, so development is faster and cheaper; they are reliable and, being used by many programs, largely error-free; the developer needs no expertise in that area (graphics, compression, encryption, path-finding); the program is easier to maintain because common code lives in one place; and a whole team can use the same routines, giving consistent results. **Drawbacks:** a routine may not do exactly what you need and you cannot change it; your program depends on the library being available, correct and secure – a bug or a security hole in the library is a bug in your program; and you must learn how to call it.

- a **static library** 静态库 is copied into the executable at compile time (stands alone, but larger and needs rebuilding to update).
- a **dynamic library** 动态库 (DLL, Dynamic Link Library; `.so`) is loaded at run time (smaller executables, shared by many programs, updated once for all).





Static: the library is copied into the executable. Dynamic: a shared library file is loaded at run time

Dynamic Link Library (DLL) files. A DLL is a library that is loaded into memory only when a program calls it, at **run time**, and stays as a separate file rather than being copied into the executable. *Benefits:* the executable is smaller; several running programs share one copy of the DLL in memory; a DLL can be updated (bug fix, new device) without recompiling the programs that use it; and memory is used only while the routine is needed. *Drawbacks:* the program will not run if the DLL is missing, moved or the wrong version; an updated DLL can break a program that relied on the old behaviour; and a fake DLL put in its place runs with the program's rights.

> **Worked example.** A team writing the software for a restaurant robot uses a program library that includes a routine to find the shortest path between tables. Explain two benefits and one drawback to the team.

exercises.lol exercises.com.cn

Benefits: the routine is already written and tested, so the team saves time and can trust the result; the team need not understand path-finding algorithms themselves and can spend the time on the robot's own features. *Drawback:* the routine may not handle the restaurant's exact needs (moving chairs, one-way aisles) and the team cannot alter it, so they may have to work around its limits.

Language translators

You write source code; the computer runs **machine code** 机器码. A **translator** 翻译器 converts between them.

■ Assembler

An **assembler** 汇编器 translates **assembly language** 汇编语言 into machine code: each mnemonic instruction (`LDD`, `ADD`, `JMP`) becomes exactly one machine-code instruction, and symbolic addresses and labels are replaced by real addresses. It is needed because the processor executes only machine code, and assembly is used where the programmer needs direct control of the hardware (embedded systems, device drivers).

■ Compiler

A **compiler** 编译器 translates a high-level program into machine code **once, before it runs**.

- it reports **all** errors at compile time; once clean, it produces a stand-alone **executable** 可执行文件 that runs **without the compiler installed** and can be run many times.
- generally **faster at run time** (no translation while running), but tied to one CPU/OS – recompile for each platform.

The two-mark description: *a compiler translates the **whole** high-level program into machine code (**object code** 目标代码) **before** it is run, produces an executable file, and reports **all** the syntax errors together as a list at the end of translation.* It does not run the program.

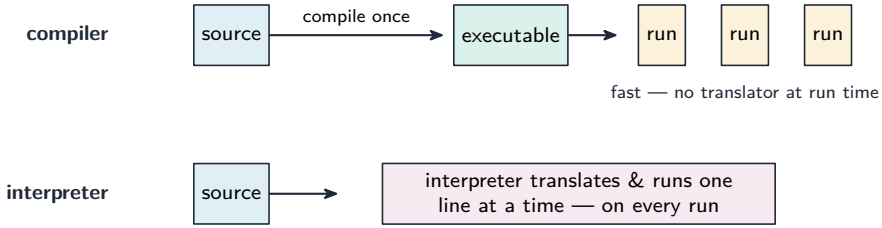
■ Interpreter

An **interpreter** 解释器 translates and runs a high-level program **one line at a time**, producing no executable.

- it reports an error **when it reaches that line**, then stops; you can fix it and continue – good for development.
- the **interpreter must be installed** to run the program; generally **slower** (each run re-translates), but easy to **port** across platforms.

The two-mark description: *an interpreter translates **one statement** of the high-level program at a time and **executes** it immediately before moving to the next; no executable file is produced; it stops at the **first** error it meets and reports it.* Both the source code and the interpreter must be present every time the program runs.

exercises.lol exercises.com.cn



A compiler translates once into a standalone program; an interpreter translates line by line, every run

■ Choosing between them

Use a compiler when:	Use an interpreter when:
run-time speed matters	you want fast edit–run cycles
distributing to users without dev tools	writing cross-platform scripts
the program runs many times	the program is small or run once
	teaching beginners

Benefits and drawbacks, as the mark scheme lists them.

	compiler	interpreter
execution speed	fast – already machine code	slower – translated on every run
what the user needs	only the executable; no translator, and the source code stays private	the source code and the interpreter
finding errors	all errors listed at once, after the whole program is translated	each error reported at the line where it occurs, as you develop
changing the code	recompile the whole program after every change	edit and run again immediately
portability	machine code runs on one platform only; recompile for each	the same source runs wherever an interpreter exists

> **Worked example.** A developer uses an interpreter while writing a program and a compiler when it is finished. Explain how each is used (4 marks).

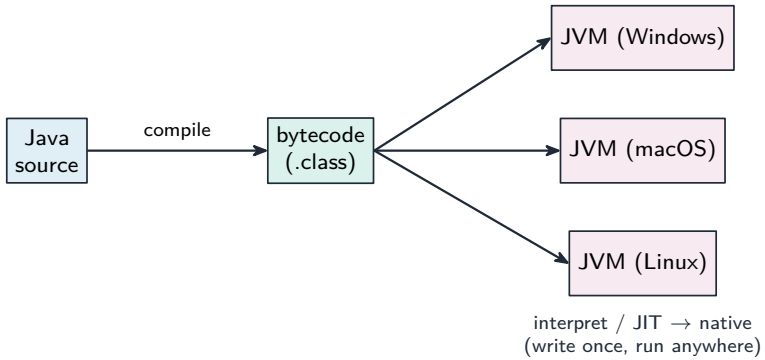
During development the interpreter runs the partly written program at once, without

waiting for a complete translation; when it meets an error it reports the line, so the developer fixes it and runs again immediately – a fast edit–run cycle that is easier for debugging. When the program is finished, the compiler translates the whole program into an executable that runs faster, needs no translator on the user's computer, and does not reveal the source code, so it can be sold to the public.

■ Hybrid: Java

Java is compiled into **bytecode** 字节码 (a platform-independent intermediate form), which a **virtual machine** 虚拟机 (the JVM) then interprets – or uses **just-in-time compilation** 即时编译 to turn hot parts into native code. So errors are caught early, the bytecode runs anywhere with a JVM ("write once, run anywhere"), and long-running programs reach near-native speed. C# and Python use similar designs.

The syllabus phrase is "**partially compiled and partially interpreted**": the compiler stage catches syntax errors and produces compact, **portable** 可移植的 bytecode; the interpreting stage lets that one bytecode file run on any machine that has a virtual machine, at the cost of some speed. Java in **console mode** (a text program run from the command line) is the syllabus's example.



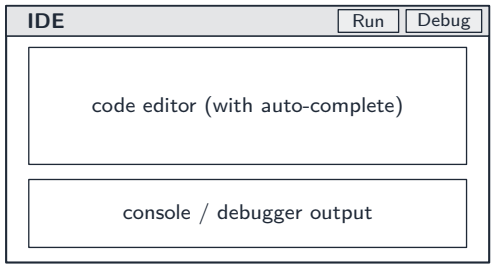
Java compiles to portable bytecode that any JVM runs – write once, run anywhere

exercises.lol exercises.com.cn

> **Worked example.** Java source is compiled to **bytecode**, which a JVM then interprets. Why use both, instead of compiling straight to machine code? A compiler produces machine code for **one** processor and operating system, so a program compiled on one machine will not run on another. Java's compiler instead targets a **virtual** machine, so the bytecode it produces is identical everywhere; each platform then supplies its own **JVM** to interpret that bytecode into its own native instructions. One compiled file therefore runs anywhere a JVM exists - "write once, run anywhere". The price is speed: interpreting bytecode is slower than running native code, which is why a real JVM also uses **JIT** compilation to turn frequently-run bytecode into native code while the program runs. Name **both** sides - the marks are for portability **bought at the cost of speed**.

Integrated Development Environment (IDE)

An **integrated development environment** 集成开发环境 (IDE) brings the tools to write, test and debug code into one application:



An IDE bundles the editor, a Run button and a debugger

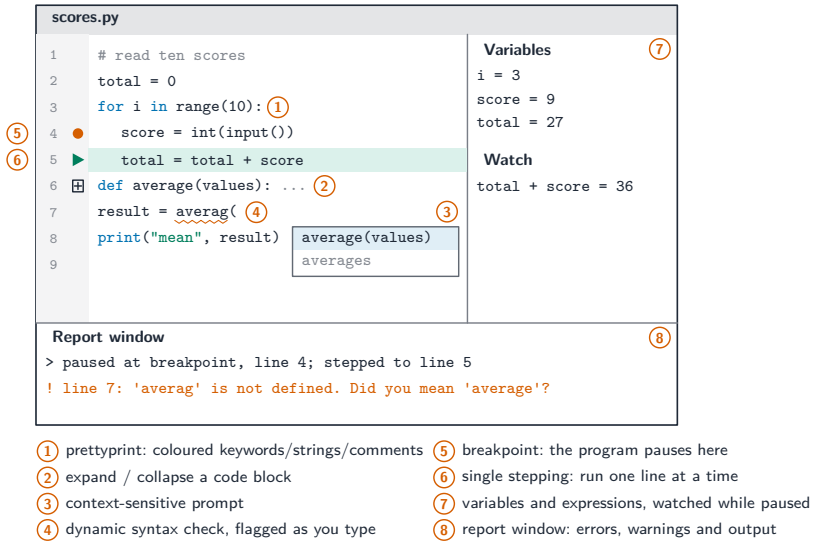
The syllabus groups the features into four kinds. Learn which feature belongs to which, because questions ask you to sort them and to describe one from each group.

- **For coding: context-sensitive prompts** 上下文相关提示 – as you type, the IDE pops up the identifiers, keywords or parameters that fit at that point in the code; **auto-complete** 自动补全 finishes the name for you; automatic indentation and bracket matching keep the layout right as you type.
- **For initial error detection: dynamic syntax checks** 动态语法检查 – the editor checks the syntax as you type and underlines or highlights a mistake immediately, before the program is translated; after translation, error messages with line numbers.
- **For presentation: prettyprint** 代码美化 – keywords, identifiers, strings and comments shown in different colours or fonts (**syntax highlighting** 语法高亮) with consistent indentation, so the structure is visible at a glance; **expand and collapse code blocks** – hide the body of a loop, an IF or a subroutine so you see

the outline.

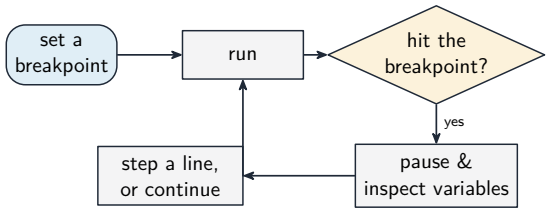
- **For debugging: breakpoints** 断点 – the program pauses when it reaches a marked line; **single stepping** 单步执行 – from the pause, run one line at a time; a window that shows the current values of **variables and expressions** as they change; and a **report window** 报告窗口 that lists errors, warnings and output.

Other features: translator integration (compile or run with one key, errors shown inline), a **debugger** 调试器 that drives the debugging features above, **version control** 版本控制 integration (git), project management, a help system, **refactoring** 重构 tools (safe renaming) and **unit test** 单元测试 integration.



The IDE features the syllabus names, in the places you would see them on screen

An IDE speeds development by putting writing → running → debugging → fixing behind one interface. Common IDEs: Visual Studio, PyCharm, Eclipse, VS Code.



A debugger: set a breakpoint, run, then pause to inspect variables and step through the code

exercises.lol exercises.com.cn

> **Worked example.** A function `Calculate()` returns an unexpected value when the program runs. Describe how the debugging features of a typical IDE help find the cause (4 marks).

Set a **breakpoint** on the first line of `Calculate()`, so the program pauses there instead of running through. Then **single-step** through the function one line at a time. After each step read the values of the variables and of any expression you have asked the IDE to **watch**, and compare them with the values you expected; the first line after which a value is wrong is where the logic error is. The **report window** shows any run-time error message and the output produced so far.

> **Worked example.** Put each feature in its syllabus group: prettyprint, context-sensitive prompt, dynamic syntax check, breakpoint, expand/collapse code blocks, report window.

Coding: context-sensitive prompt. Initial error detection: dynamic syntax check. Presentation: prettyprint, expand/collapse code blocks. Debugging: breakpoint, report window.

■ Software licences used when translating or distributing code

A software licence specifies the permitted use, copying, modification and redistribution of a copyright work. **Free Software** concerns the freedoms to run, study, change and share software; it can be sold. **Open-source** licences permit access to source and licensed modification/redistribution. Freeware can cost nothing while withholding source and modification rights. For a redistribution question, identify the actual licence's conditions; see topic 7 for comparisons and justified choices.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
operating system	software that manages the computer's hardware and resources and provides an interface between the user, the application programs and the hardware
utility software	system software that performs a specific task to maintain, optimise or protect the computer, such as a virus checker or a defragmenter
program library	a collection of pre-written, tested routines (subroutines, classes, modules) that a program can use instead of writing its own

Term	Definition
Dynamic Link Library (DLL)	a program library whose routines are loaded into memory only when the program calls them, at run time, and are shared between programs
assembler	a translator that converts an assembly language program into machine code, one instruction for each instruction
compiler	a translator that converts the whole of a high-level language program into machine code before it is run, producing an executable file
interpreter	a translator that translates and executes a high-level language program one statement at a time
integrated development environment	a single application that provides the tools for writing, translating, running and debugging a program
context-sensitive prompt	a pop-up that suggests identifiers, keywords or parameters that fit at the current point in the code
dynamic syntax check	checking the syntax of the code as it is typed and flagging an error before the program is translated
prettyprint	displaying code with keywords, identifiers and comments in different colours or fonts and with consistent indentation
breakpoint	a marked line at which the running program pauses so that variables can be inspected
single stepping	running a paused program one statement at a time under the programmer's control

Exam tips

- List the OS's jobs by their syllabus names (memory, process, hardware, file and security management) and say what each *does* – "manages resources" alone is too vague.
- Compare **compiler vs interpreter vs assembler**: what each translates, when it translates it, and how errors are reported.
- Explain what an **IDE** provides using the syllabus groups: coding, initial error detection, presentation, debugging.

exercises.lol exercises.com.cn

- A "benefit to the developer" answer names the developer's saving: time, cost, expertise, reliability or maintenance. A "drawback" names a dependence: availability, version, fit, security.
- For "describe the operation of" a translator, give three things: what is translated (whole program or one statement), when (before running or while running), and how errors are reported (all at once or at the first error).

Common mistakes

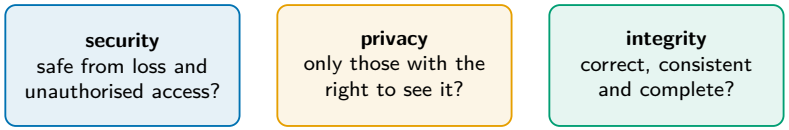
- Writing "the OS controls the computer" or "manages resources" with no example task. Each mark is one named task with what it does.
- Saying an interpreter "compiles line by line". An interpreter translates **and executes** each statement; it never produces an executable.
- Saying a compiler runs the program. It only translates; the executable runs later, without the compiler.
- Putting a DLL "inside" the executable. That is a static library; a DLL stays a separate file loaded at run time.
- Saying defragmentation "deletes" or "compresses" files, or is needed on an SSD. It only moves blocks so each file is stored contiguously.
- Filing prettyprint or collapsing blocks under "debugging". They are presentation features; debugging is breakpoints, single stepping, watching variables and the report window.

Security, privacy and data integrity

A-Level Computer Science

Security, privacy and integrity – three different ideas

These sound alike but mean different things:



a locked file with a typo is secure but lacks integrity

Three different questions about the same data.

- **security** 安全 – protecting data from **unauthorised** 未授权 access, change or destruction.
- **privacy** 隐私 – an individual's right to **control who sees their personal data**, with consent and a clear purpose.
- **integrity** 完整性 – the data being **accurate and complete** – not corrupted or accidentally changed.

A file can be secure (only the right people can open it) but lack integrity (a typo corrupted it); or accurate but not private (anyone can read it). All three are needed.

The differences the scheme wants, one sentence each: **security** is keeping the data safe from loss and from unauthorised access; **privacy** is keeping the data confidential, so that only those with the right to see it can; **integrity** is the data being correct, consistent and complete. So "the difference between security and privacy": security is about protecting the data from being accessed, changed or lost by people who should not; privacy is about the individual's right to decide who may see their personal data. "The difference between security and integrity": security protects the data from unauthorised access; integrity is about the data being accurate and up to date, which validation and verification protect.

Why security matters

Two things to protect: the **data** itself (keep it confidential, intact and available) and the **computer system** (a compromised system can attack others, steal credentials, or be held to ransom).

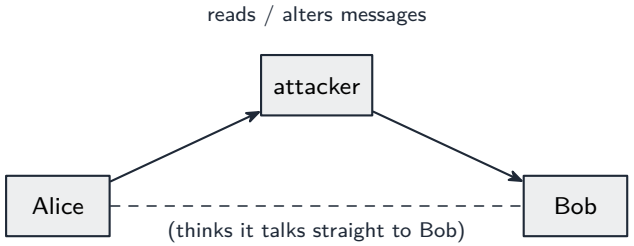
"Why does the school need to keep both secure?" Data: it is personal and confidential, so it must not be read, changed or deleted by an unauthorised person, and its loss would stop the school working. System: an intruder who reaches the computer system

exercises.lol exercises.com.cn

can install malware, use it to attack other systems, damage the hardware or software, or lock it with ransomware; a secure system is the first line of defence for the data on it.

Threats from networks and the internet

Threats fall into three groups.



A man-in-the-middle attacker sits between the two parties

1. **Malware** 恶意软件 (malicious software) – harmful programs:

- **virus** 病毒 – self-copying code that attaches to other programs and spreads when they run.
- **worm** 蠕虫 – self-copying code that spreads over **networks** 网络 with no user action.
- **Trojan horse** 木马 – looks useful but hides malicious code.
- **spyware** 间谍软件 – secretly collects information (keystrokes, passwords).
- **ransomware** 勒索软件 – encrypts your files and demands payment.
- **adware** 广告软件 – pushes unwanted adverts.

2. **Tricking people** (social attacks):

- **phishing** 网络钓鱼 – fake emails/sites that trick users into giving credentials.
- **pharming** 域名欺骗 – redirects a user to a fake site even when they type the correct address.
- **social engineering** 社会工程 – tricking people into giving up information.

The scheme's descriptions of the four named threats: a **virus** is malicious software that replicates (copies itself), attaches itself to other files and deletes or corrupts data; **spyware** is malicious software that records the user's key presses and actions and sends them to a third party, to obtain passwords and personal data; a **phishing** email pretends to come from a legitimate organisation and contains a link to a fake website where the user is asked for personal or bank details; **pharming** is malicious code installed on the user's computer or on a web server that redirects the user to a fake

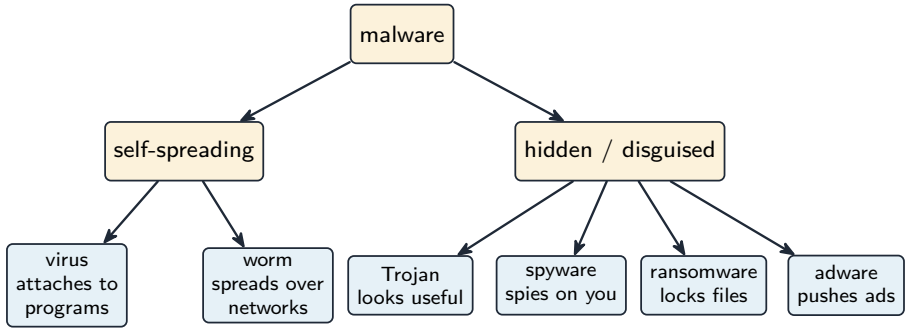
website even though they typed the correct address. Similarities of spyware and a virus: both are malware, both are installed without the user's knowledge, both can send data to a third party or damage the system; the difference is that a virus replicates itself while spyware records and transmits information. Phishing and pharming both lead the user to a fake website that collects their data; phishing needs the user to click a link in an email, pharming works through code on the computer or the DNS server and needs no email.

3. Attacks on the network:

- **hacking** 黑客入侵 by **hackers** 黑客 – unauthorised access, often via weak passwords or software flaws.
- **denial of service** 拒绝服务 (DoS/DDoS) – floods a server so real users cannot reach it.
- **eavesdropping** 窃听 – capturing data in transit (a risk on open Wi-Fi).
- **man-in-the-middle** 中间人攻击 – an attacker secretly relays or alters messages between two parties.

> **Worked example.** Identify and describe two threats to the data on a school network, and give a different prevention method for each.

Threat 1, malware: a virus copied onto a computer from an email attachment or a download replicates itself and corrupts or deletes files; prevention: anti-virus software that scans files and is kept up to date. Threat 2, hacking: an unauthorised person gains access to the network, for example by guessing a weak password, and reads or changes the data; prevention: a firewall that blocks unauthorised connections, or strong passwords with two-factor authentication. A third pair, phishing: an email leads a user to a fake site that collects their login; prevention: training users to check the sender and the URL, and filtering email. The measure must match the threat: encryption does not stop a virus, and anti-virus software does not stop phishing.



Malware by behaviour: self-spreading (virus, worm) versus hidden/disguised (Trojan, spyware, ransomware, adware)

Security measures

Measures protect both the **security of data** (against loss, theft or corruption) and the **security of the computer system** (its hardware, software and network).

A standalone PC

- a strong password; **antivirus** kept up to date; prompt **software updates**; **backup** 备份 to separate media; full-disk **encryption** 加密; a locked screen.

A networked PC

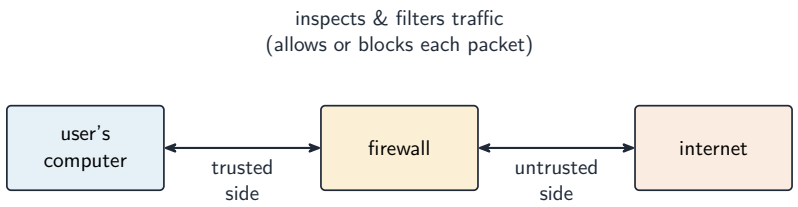
All the above, plus a **firewall** 防火墙, per-user **permissions** (admin rights only for admins), central management of **user accounts** 用户账户, and **audit logs** 审计日志 (who logged in, what they touched).

How the measures work, in the wording the scheme awards:

- **firewall**: examines every incoming and outgoing transmission and compares it with set criteria (a whitelist or blacklist of addresses, ports and protocols); blocks any that do not meet the criteria; can prevent access to certain sites and warn of unauthorised access attempts.
- **encryption**: the data is scrambled (encoded) with a key into ciphertext, so an intercepted copy cannot be understood without the key; the receiver uses a key to decrypt it. It protects data in transmission and in storage, but it does not stop the data being intercepted or deleted.
- **passwords and user accounts**: only a user who knows the password can log in; a strong password (long, mixed characters, changed regularly) cannot be guessed; accounts lock after repeated failures; each account carries its own access rights.
- **anti-virus and anti-spyware software**: scans files and programs against a database of known malware signatures, checks behaviour, quarantines or deletes what it finds, and must be updated so that new malware is recognised.
- **access rights**: each user (or group) is given permissions for each file or table, such as read-only or read and write, so a user cannot see or change data that is not theirs; a database can also present each user with a view containing only the fields they need.
- **biometrics**: the device captures an image of the face, fingerprint or iris, converts it to digital data, compares it with the stored data for that user and allows access only on a match; it cannot be forgotten, lent or guessed like a password.
- **backups**: a copy of the data on separate media, kept off-site, so that lost or corrupted data can be restored.

To restrict the risks of malware, in three marks: install anti-malware software and keep it updated; use a firewall; do not open attachments or download files from unknown sources; keep the operating system and applications patched; and train users.

exercises.lol exercises.com.cn

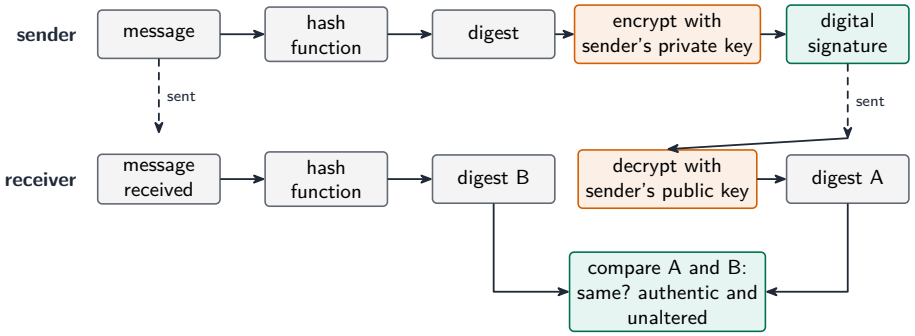


A firewall sits between the user's computer and the internet

Across the internet

- **VPN** 虚拟专用网 – encrypts traffic between the user and the corporate gateway.
- **HTTPS / TLS** – encrypt web traffic.
- **digital signatures** 数字签名 – prove who sent a message and that it was not altered in transit.
- intrusion detection – watches traffic for known attack patterns.

How a digital signature authenticates a document (five marks): the sender puts the message through a hash function to produce a digest; the sender encrypts the digest with their **private key**, and that encrypted digest is the digital signature; the message and the signature are sent together; the receiver decrypts the signature with the sender's **public key** to recover the digest; the receiver hashes the received message and compares the two digests; if they match, the message came from the sender (only they hold the private key) and was not altered in transmission. A signature proves who sent the message and that it is intact; it does not hide the contents, which is what encryption of the message is for.



A digital signature: a hash of the message, encrypted with the sender's private key, checked by the receiver against a fresh hash

exercises.lol exercises.com.cn

Matching measures to threats

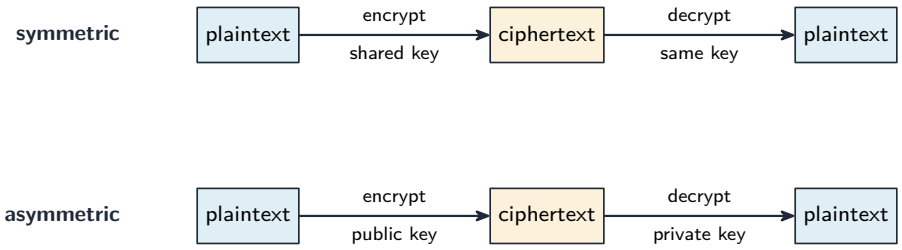
- **interception in transit** → **encrypt** the data (HTTPS, VPN). Intercepted ciphertext is useless without the key.
- **unauthorised access** → strong **authentication** 身份验证 (long passwords; **two-factor authentication** 双因素认证 with a phone code or key); user **authorisation** 授权; lock-out after failed logins.
- **malware** → **anti-virus software** and **anti-spyware** 反间谍软件 with real-time scanning; patching; avoid untrusted downloads.
- **phishing** → user training; email filtering; check the URL before entering credentials.
- **internal threats** → the **least-privilege** 最小权限 principle (give each user only what they need); auditing.
- **DDoS** → rate limiting and traffic filtering.

For confidential data crossing the internet, the scheme's method is **encryption**: the data is encoded with a key into ciphertext, so that an unauthorised person who intercepts it cannot read it, and only the intended receiver, who has the key, can decode it. For a program file sent by email for testing, the same answer applies (encrypt the file, or send it over an encrypted connection), together with a password on the file itself.

Protecting the data itself

- **encryption** – turn **plaintext** 明文 into **ciphertext** 密文 with a key. **Symmetric encryption** 对称加密 (AES) uses one shared key; **asymmetric encryption** 非对称加密 (RSA) uses a **public key** 公钥 and a **private key** 私钥. Protects data at rest and in transit.
- **access control** 访问控制 – file permissions (read/write/execute) and **access rights** 访问权限, enforced by the OS.
- **authentication** – **authentication techniques** verify the user: something you know (password), have (token, phone), or are (**biometrics** 生物识别 – fingerprint, face, iris); strongest combined.
- **backups** – keep copies (some **off-site**) so loss or corruption is recoverable.
- **physical security** – locked server rooms, cable locks.

Access rights in a database, described for three marks: each user is given an account with a username and password; the database administrator assigns each account permissions for each table, such as read-only, read and write, or no access; users see only the tables and fields they are allowed to, so a customer cannot open the staff table and a clerk can read but not change the prices. The **DBMS** enforces this with its access rights and with **views**, and it can encrypt the stored data as well.



Symmetric uses one shared key; asymmetric uses a public key to encrypt and a private key to decrypt



A security token shows a changing code for two-factor authentication ("something you have")



A fingerprint reader checks "something you are" – a feature of the person, not a password

Data integrity

Data has integrity when it is accurate and complete. Two techniques: **data validation** (catch bad data before storing) and **data verification** (confirm data was entered or transferred correctly).

exercises.lol exercises.com.cn

Validation – does the data make sense?

Validation 验证 checks data against sensible rules, automatically:

age field: range check 0–120

enter 16 → accepted

enter 200 → rejected

A range check accepts sensible values and rejects the rest

- **range check** – within limits (a month is 1–12).
- **limit check** – on the correct side of a single limit (e.g. age $\geq$ 18).
- **existence check** – the referenced item exists (e.g. a product code is in the table).
- **length check** – the right number of characters.
- **type / character check** – the right kind of data (a phone field allows only digits).
- **format check** – matches a pattern (an email must contain `@`).
- **presence check** – required fields are not empty.
- **check digit** 校验位 – an extra digit computed from the others (ISBN, card numbers) that spots transcription errors.

> **Worked example.** In a simple check-digit scheme the check digit is the remainder when the sum of the digits is divided by 10, appended to the number. The number 4162 has digit sum 13, so it is stored as 41623. A user types 14623: the first two digits are swapped, but the sum is still 13, so the check digit still matches and the error is **not** caught. A user who types 41523 is caught, because $4+1+5+2 = 12$ gives check digit 2. Weighted schemes can detect some transpositions that this unweighted rule misses. Detection depends on the weights, the positions and the modulus: an ISBN-13 rule, for example, does not detect every possible transposition. A check digit is validation: it tests the number against a rule at the moment it is entered.

- **lookup check** and **consistency check** (e.g. delivery date $\geq$ order date).

Validation catches data that is wrongly **formatted**, but not data that is the right format yet factually wrong ("Bob" for "Bib").

> **Worked example.** Identify the validation check each piece of pseudocode performs.

Pseudocode	Check
`IF x < 0 OR x > 10 THEN OUTPUT "Invalid"`	range check: the value must lie between two limits
`IF x = "" THEN OUTPUT "Invalid"`	presence check: the field must not be empty
`IF NOT(x = "Red" OR x = "Yellow" OR x = "Blue") THEN OUTPUT "Invalid"`	lookup (existence) check: the value must be one of a list
`IF LENGTH(x) <> 6 THEN OUTPUT "Invalid"`	length check: the right number of characters
`IF MID(x, 1, 1) < "A" OR MID(x, 1, 1) > "Z" THEN OUTPUT "Invalid"`	format check: a particular character must be a letter

To validate a car registration number that must be one letter, three digits and two letters: a **format check** tests each position against its pattern, and a **length check** confirms six characters. To validate a date of birth: a format check (`DD/MM/YYYY`), a range check (the month is 1 to 12, the year is not in the future) and a presence check (it is not left blank). A mark between 0 and the maximum for the test needs a type check (an integer) and a range check, with the upper limit read from the test's own record: that is how validation protects integrity, by refusing data that could not be correct.

■ Verification – was the data entered or transferred correctly?

Verification 核对 checks the data was not changed in moving from one place to another.

During entry: **double entry** (type it twice and compare, as for a new password) or **visual** check.

In the scheme's words, **double entry** is entering the data twice, by the same person or by two people, and having the computer compare the two versions and report any difference; a **visual check** is the person comparing what is on the screen with the original source document and correcting any difference before saving. Both protect integrity by making sure the stored data matches the source. Even after validation and verification the data can still be wrong: it can be sensible and match the source, yet the source itself was wrong, or the user typed a different but valid value from the one intended.

During transfer (bits can flip):

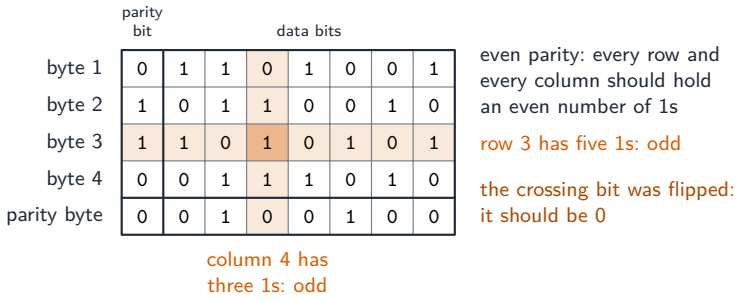
- **parity check** 奇偶校验 – an extra bit makes the number of 1s even (even parity) or odd. The receiver re-counts. Catches single-bit errors.
- **checksum** 校验和 – the sender sends a summary value of the data; the receiver recomputes it and compares.

exercises.lol exercises.com.cn

- **cyclic redundancy check** 循环冗余校验 (CRC) – a stronger checksum using polynomial division, catching many more error types.

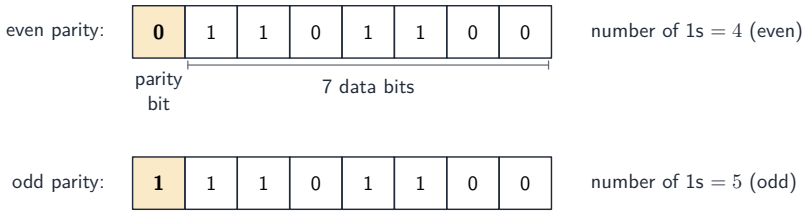
A **parity block check** 奇偶块校验 goes further and *locates* the error. Arrange the bytes in a grid: give each byte a **row** parity bit, then compute one extra **parity byte** whose bits are the **column** parity of the bytes above. A single flipped bit now fails **one row and one column** – their intersection pinpoints exactly which bit changed, so it can even be corrected.

> **Worked example.** Four bytes are sent with even parity, followed by a parity byte. Find the bit that was corrupted.

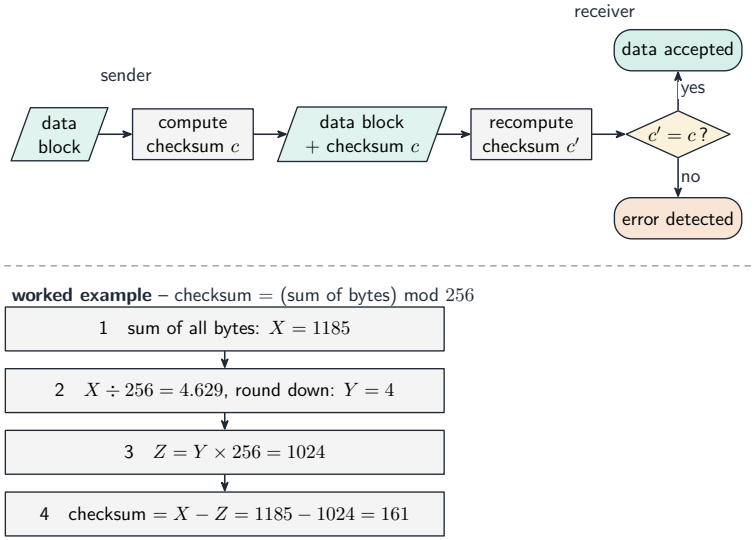


A parity block check: the row that fails and the column that fails cross at the flipped bit

Count the 1s in each row and each column. Every row and column should have an even number; byte 3 has five and column 4 has three. The bit where that row and that column cross is the one that changed, so it is reset from 1 to 0. A parity check on its own detects an error in a byte but cannot say which bit; two errors in the same byte cancel and pass unnoticed. A **checksum**, explained for three marks: the sender puts the block of data through an algorithm that produces a checksum value; the data and the checksum are sent together; the receiver runs the same algorithm on the data it received; if the two checksums match, the data is accepted, and if not, it is rejected and sent again.



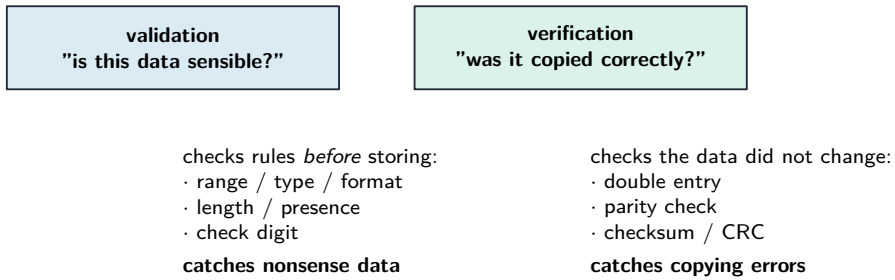
The parity bit is set to make the number of 1s even or odd



Working out a checksum for a block of data

Verification only proves what arrived matches what was sent – not that the data is correct, and not against deliberate tampering. **Validation** asks "is this sensible?"; **verification** asks "was this copied correctly?" – use both.

The table questions sort the methods by when they are used: **during data entry**, double entry and a visual check; **during data transfer**, a parity check (byte or block) and a checksum. Transferring video files from a camera to a server uses a checksum: the camera computes it, the server recomputes it, a mismatch means retransmit.



Validation checks the data makes sense; verification checks it was copied without change

exercises.lol exercises.com.cn

> **Worked example.** A user types their date of birth as `31/02/2009`, and types their email address twice. Which check catches which error, and what is the difference? **Validation** asks "is this data **sensible**?" - the computer tests it against a rule, and a format or range check rejects `31/02/2009` because February never has 31 days. **Verification** asks "was this data **entered correctly**?" - typing the email twice is **double entry**, and comparing the two copies catches a typing slip. The limit is what makes this a favourite question: validation can never tell you the data is **right**, only that it is **possible** - `01/02/2009` passes every validation rule even if the user was actually born on a different day. Say what each check **can** and **cannot** catch.

A proxy server in a security answer

A **proxy server** handles requests on a client's behalf. An organisation may configure it to filter access to websites, log requests or cache responses. Explain the configured protection: merely naming a proxy does not show how it prevents the stated threat. A router forwards packets between networks; these roles are different.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly.

Term	Definition
data security	keeping data safe from loss and from unauthorised access, change or deletion
data privacy	keeping data confidential, so that it is seen only by those who have the right to see it
data integrity	the data being accurate, consistent and complete
malware	malicious software that is installed without the user's knowledge to damage a system or steal data
virus	malware that replicates itself, attaches to other files and corrupts or deletes data
spyware	malware that records the user's key presses or actions and sends them to a third party
phishing	an email pretending to be from a legitimate organisation that leads the user to a fake website to collect personal data
pharming	malicious code that redirects the user to a fake website even when the correct address is entered

Term	Definition
firewall	hardware or software that examines all traffic entering or leaving a system against set criteria and blocks what does not meet them
encryption	scrambling data with a key into ciphertext, so that it cannot be understood without the key to decrypt it
digital signature	a hash of a message encrypted with the sender's private key, used to prove who sent it and that it was not altered
data validation	an automatic check that entered data is reasonable and follows set rules
data verification	a check that data has been entered or transferred correctly, by comparing it with the source or with a recomputed value
check digit	an extra digit calculated from the other digits of a number and appended to it, so that an error in the number can be detected
parity check	an extra bit added to a byte so that the number of 1s is even (or odd), which the receiver recounts
checksum	a value calculated from a block of data by an algorithm and sent with it, recalculated by the receiver and compared

Exam tips

- Keep the three ideas separate: **security** (keeping data safe), **privacy** (who may see it), **integrity** (keeping it correct).
- Match each **threat** (malware, hacking, phishing, interception) to a **measure** (firewall, encryption, authentication, access rights).
- Encryption protects **confidentiality, not integrity** – use a checksum, parity or check digit for integrity.
- Distinguish a **virus, worm and Trojan** and how each spreads.

Common mistakes

- Giving the same measure for two threats, or a measure that does not fit the threat. Each threat in the table needs a different prevention that actually stops it.
- Naming a measure without saying how it works. "Firewall" scores when it is followed by "compares traffic with set criteria and blocks what fails".
- Calling validation a check that the data is correct. Validation checks that data is reasonable; verification checks that it matches the source. Neither proves it is true.

exercises.lol exercises.com.cn

- Saying a digital signature encrypts the message. It encrypts a message hash with the private key; the receiver decrypts with the public key and compares hashes.
- Describing a check digit as verification, or a parity check as validation. The check digit is a validation rule on entry; parity and checksums verify a transfer.
- A virus **replicates**; spyware **records** – never swap the two in a definition.

Ethics and Ownership

A-Level Computer Science

Ethics for computing professionals

A **computing professional** is someone whose work – software, systems, networks, data – affects other people. Because the work is technical, others often cannot judge whether it was done well or honestly. So the profession follows shared **ethics** 伦理 (principles for good behaviour).

Why ethics matters

- **trust** – users and employers trust professionals to act in their interest. Without that trust, software loses credibility.
- **impact** – software runs medical devices, banking, vehicles. Careless or dishonest work can hurt people.

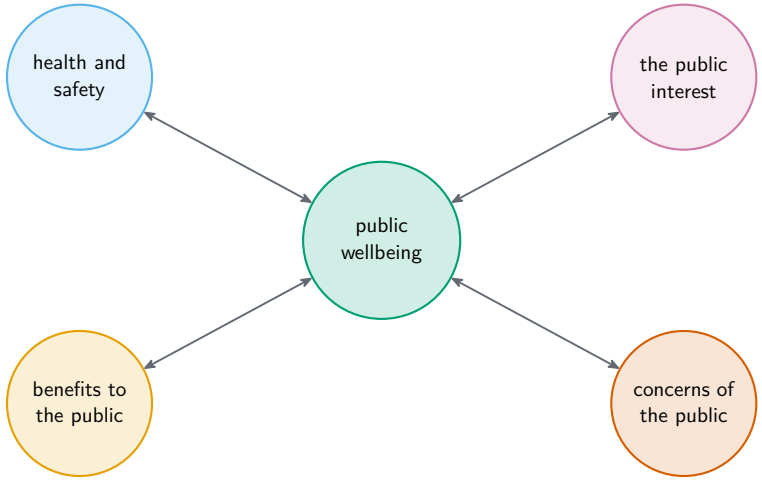
Professional bodies (BCS, ACM, IEEE) publish codes of ethics for members.



CCTV raises privacy concerns – one of the ethical issues a computing professional must weigh



Discarded electronics (e-waste) are a growing environmental cost of computing



Software development affects the public's wellbeing in several ways

Typical principles

- **public interest first** – protect the safety and welfare of those affected.
- **honesty and competence** – be honest about your skills; don't claim expertise you lack.
- **confidentiality** 保密性 – protect clients' and employers' private information.
- avoid **conflicts of interest** 利益冲突 – don't take work where your interest clashes with the client's.
- keep your skills current; respect **intellectual property** 知识产权 and **privacy** 隐私; treat colleagues fairly.

Joining a professional body

The syllabus names two: the **BCS** (British Computer Society) and the **IEEE** (Institute of Electrical and Electronics Engineers). Both publish a **code of conduct** 行为准则 that members agree to follow. The benefits of joining, in the scheme's words: a set of ethical guidelines to follow, so decisions are not left to personal judgement; training, conferences and publications that keep the member up to date; advice and support, including legal help, when a problem arises; and recognised professional status, so employers and clients trust the member's work. The consequences of **not** joining: no guidance on ethical decisions, so the programmer may act unethically without realising; less credibility with employers and customers, so it is harder to win work; no support in a dispute; and being out of date with developments and law. The purpose of a code of conduct (two marks): to create a safe, respectful and professional working environment, and to make sure every employee understands what is expected and the

exercises.lol exercises.com.cn

consequences of their actions.

	joins the BCS or IEEE	does not join
ethics	a code of conduct to follow, so decisions are not left to personal judgement	no guidance, so may act unethically without realising
keeping up	training, conferences and publications	out of date with developments and the law
support	advice and legal help when a problem arises	no support in a dispute
status	recognised professional status, so employers and clients trust the work	less credibility, so harder to win work

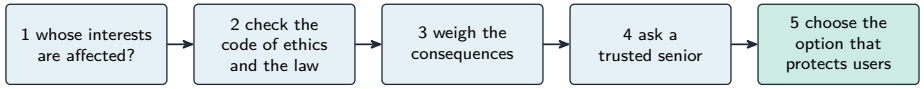
Joining a professional body: each benefit sets up the matching consequence of not joining

> **Worked example.** Explain why a programmer needs to act ethically towards colleagues and towards the public.

Colleagues: treat them fairly and without discrimination; respect their work, their ideas and their confidential information; be honest about mistakes and give credit where it is due; support their development rather than undermine it. The public: protect their personal data and privacy; produce software that is safe, reliable and properly tested, because faults can cause harm; be honest about what the software can do; take on only work within your competence; obey the law and consider the wider effects on society and the environment. Each side earns marks for a reason and its consequence, not for the word "fair" alone.

Acting ethically vs unethically

Acting **ethically** protects users, strengthens reputation, reduces legal risk, and builds trust. Acting **unethically** (skipping testing, hiding bugs, misusing data) can harm real users, lead to dismissal or legal action, damage reputation, and erode trust in technology generally.



AI hiring tool that rejects more older applicants:
steps 1 to 4 say **hold it back until the bias is fixed**

A borderline decision in five steps: the AI hiring tool is held back until its bias is fixed

When you face a borderline decision: identify whose interests are affected, check the code of ethics and the law, weigh the consequences, ask a trusted senior, and choose the option that protects users above short-term convenience.

> **Worked example.** Your team's new AI hiring tool sorts CVs ten times faster, but you notice it rejects more older applicants. Shipping it pleases your manager, but it treats one group unfairly. The ethical choice is to **hold it back until the bias is fixed** – public interest and fairness come before short-term convenience.

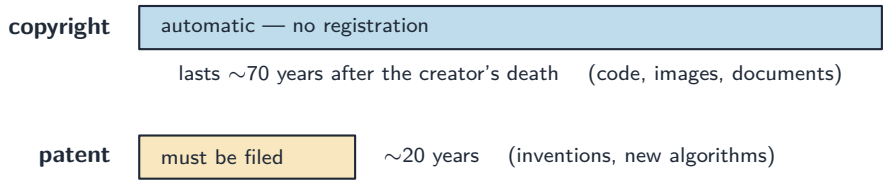
Ethics also applies to users. A student who connects a personal computer to the school network should respect other people's privacy and data, not use social media inappropriately or bully others, not download or share copyrighted material, not introduce malware or try to access systems they are not allowed to, and use the network for the purpose it was provided. A "give three ethical considerations" answer lists three of these.

Copyright

Copyright 版权 is the legal right of the creator of an original work to control how it is **copied, distributed, modified and performed**. It applies automatically (no registration) to source code, software, documents, images, audio and video.

Without copyright, anyone could copy software freely, the developer would not be paid, and plagiarism would be legal. With copyright, developers can earn from their work (encouraging more software), users know who made it, and re-use happens on the developer's terms through licensing. Copyright lasts a long time (often 70 years after the creator's death). General ideas and algorithms are not covered by copyright but may be covered by a **patent** 专利.

Why a programmer should copyright a program, in the scheme's words: to be identified as the owner and author (formal recognition of ownership); so that there are legal consequences if anyone copies or steals it; to restrict competitors from selling the same work; and to be able to earn money by licensing it. Copyright applies to the program as written; a different program that does the same job does not infringe it.



Copyright is automatic and long-lasting; a patent must be filed and lasts about 20 years

exercises.lol exercises.com.cn

Software licences

A **software licence** 软件许可证 is a contract granting permission to use software on the owner's terms; choosing and applying one is called **software licencing**.

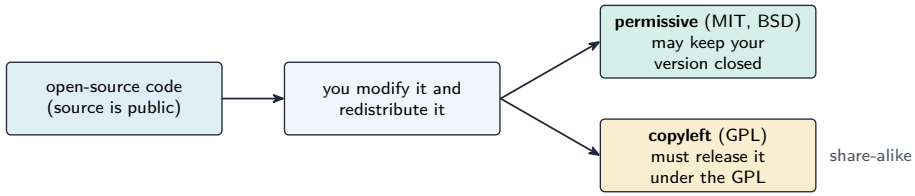
Commercial (proprietary)

- **commercial software** is sold: you **buy a licence**; the software is used only within its terms.
- the source code is not given (a **proprietary** 专有 product); you cannot modify or redistribute it.
- examples: Microsoft Office, Adobe Photoshop, most games.

Used when the developer wants **revenue** per user and to **keep control** of the code.

Open-source

- the source code is **public**; users can read, modify and redistribute it (**open-source** 开源).
- **permissive** licences (MIT, BSD) allow almost any use; **copyleft** 著佐权 licences (GPL) require that modified versions are released under the same licence ("share-alike").
- the **Free Software Foundation** (FSF) and the **Open Source Initiative** (OSI) promote and approve open-source licences.



Permissive or copyleft: what happens to a modified copy

The syllabus names both, and they are marked as distinct answers. **Free Software** (the FSF's term) means free as in freedom, not price: the user may run the program for any purpose, study and change it (so the source code must be available), redistribute copies, and distribute modified versions; a fee may still be charged for a copy. **Open Source** (the OSI's definition) requires that the source code is available, that the program may be modified and redistributed, and that the licence does not discriminate against any person or field of use. "Identify two licence types that let other people edit and redistribute the program" is answered with these two.

- examples: Linux, Python, Apache.

Used when the developer wants the software **widely used and improved by the community**.

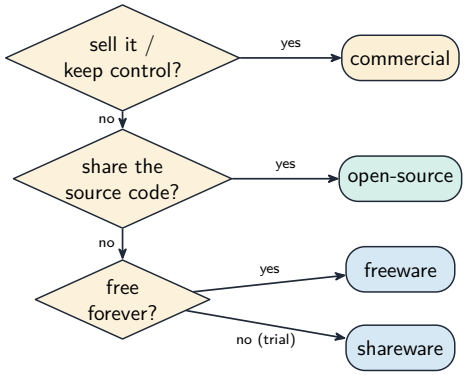
exercises.lol exercises.com.cn

Freeware and shareware

- **freeware** 免费软件 – free of charge, no source code, may be redistributed but not modified (Acrobat Reader, WhatsApp).
- **shareware** 共享软件 – free for a trial period, then you pay to keep using it; no source code.

The scheme's descriptions: **shareware** is distributed free for a trial (a limited time or limited features) and the user pays to continue using the full version; **commercial software** is sold for a fee, the source code is not supplied, the licence protects the developer's intellectual property, and the fee usually buys support and updates. Benefits of shareware to the programmer: users can try the program before buying, so they are more likely to purchase; it spreads widely at almost no advertising cost; and those who keep it pay. Benefits of a commercial licence: the developer earns a fee for every copy; the code and its rights stay protected; and the income funds support, updates and further development.

Type	Cost	Source	Redistribute	Modify
Commercial	Paid	No	No	No
Open-source	Usually free (a fee is allowed)	Yes	Yes	Yes, under the licence conditions
Freeware	Free	No	Yes	No
Shareware	Free trial, then paid	No	Sometimes	No



Choosing a licence from the developer's goal

To justify a licence choice, link it to the developer's goal (revenue, reach, community), the user's needs (cost, customising), and the use case.

> **Worked example.** A programmer has written a game to sell to the public. Identify the most appropriate licence and justify it.

A commercial licence: the game is sold for a fee, so the programmer earns money from every copy; the source code is not released, so nobody can copy the game or change it and sell it as their own; the licence protects the intellectual property; and buyers receive updates and support. Open source would not fit, because the source code would be available, so the game could be copied, changed and redistributed without payment.

> **Worked example.** A program helps shoppers by reading product labels aloud. Explain why an open source licence might not be appropriate.

The source code would be accessible, so it could be changed; a changed version might output the wrong product information, so shoppers could buy the wrong item; and the programmer would lose control over the quality and safety of what is distributed under the program's name. Going the other way, programs are released as open source so that other developers can improve and extend them, so that they are adopted widely at no cost, and so that users can adapt them to their own needs.

Artificial Intelligence (AI)

Artificial intelligence 人工智能 builds systems that do tasks once thought to need **human intelligence** – recognising speech and images, translating, playing games, driving.

Most modern AI uses **machine learning** 机器学习 – algorithms that improve at a task by learning patterns from large amounts of data, instead of being programmed step by step. **Deep learning** 深度学习, using **neural networks** 神经网络 with many layers, is the leading approach today.

Everyday examples

AI tasks split into two kinds – **understanding input**, and **producing output or decisions**.

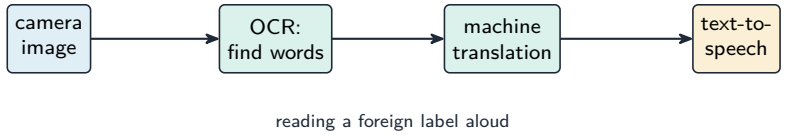
Understanding input:

- **speech recognition** 语音识别 – spoken words to text (voice assistants).
- **image recognition** 图像识别 – finding objects, faces or text in images.

Producing output or decisions:

- **machine translation** 机器翻译 – automatic translation between languages.
- **recommendation systems** 推荐系统 – suggesting products, videos or music.
- **autonomous vehicles** 自动驾驶汽车 and robots.

A common exam scenario: a program reads a label with a camera, translates it, and reads it aloud – using **optical character recognition** 光学字符识别 to find the words, machine translation to convert them, and **text-to-speech** 文本转语音 for the audio.



A common scenario: OCR → machine translation → text-to-speech reads a foreign label aloud

A four-mark "explain how AI is used" answer follows the pipeline step by step: image recognition (OCR) analyses the pixels of the photograph to locate the characters; the patterns of pixels are converted into individual characters and words; machine translation converts the words into the user's language; and text-to-speech produces the spoken output. Each step is a mark.

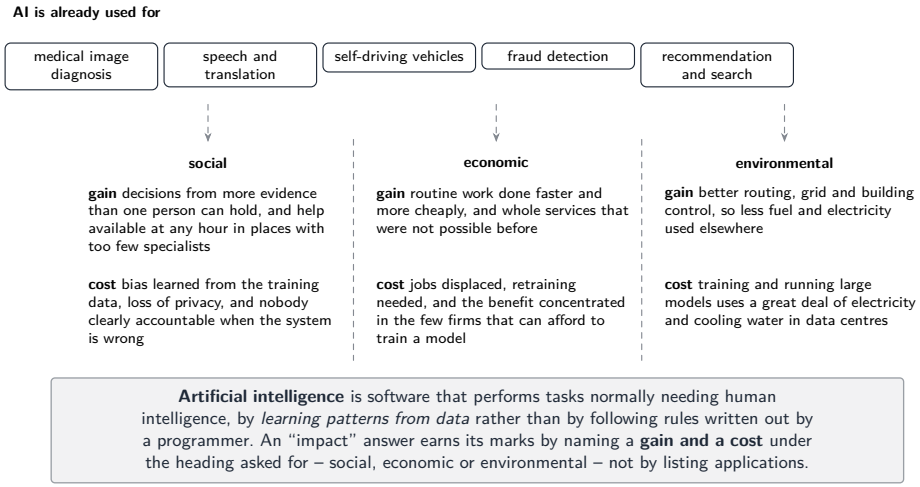
Benefits

- **accessibility** – speech/image AI helps users with impairments; translation helps non-native speakers.
- **productivity** – automating repetitive tasks frees people for creative work.
- **decision support** – AI spots patterns in huge datasets (medical diagnosis, fraud detection).
- **always available**, and **personalised** to each user.

Impacts: social, economic, environmental

The syllabus asks for the impact of AI under three headings, and a question names one of them. Give the impact **and** its consequence.

exercises.lol exercises.com.cn



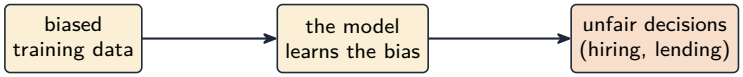
Each impact heading needs a gain and a cost

- **Social**: benefits – a label-reading program helps people with a visual impairment, people who cannot read the language, and people with reading difficulties; facial recognition at an airport speeds up identity checks and can stop wanted people entering. Harms – facial recognition can misidentify people and tracks everyone without consent, so privacy is lost; students who use AI to do their homework may not develop reasoning and problem-solving skills, may rely on it instead of learning, and may lose the collaboration and face-to-face communication that working together brings.
- **Economic**: an AI fault-diagnosis module in a repair garage diagnoses faults faster and more accurately, so more vehicles are repaired per day and costs fall; but fewer skilled mechanics may be needed, so jobs are lost, and the module must be bought and maintained. More generally, AI raises productivity and creates new jobs in some fields while removing routine jobs in others.
- **Environmental**: training and running large models uses a great deal of electricity and water for cooling in data centres, and the hardware becomes e-waste; on the other side, AI is used to cut energy use in buildings, optimise transport and monitor the environment.
- **Ethical** (the classroom question): an AI that marks work or watches students must be fair to every student, must not leak their data, must be explainable when it makes a decision about them, and must not replace the judgement of a teacher where that matters.

exercises.lol exercises.com.cn

Concerns

- **bias** 偏见 – unfair patterns in the training data become unfair AI decisions (hiring, lending).
- **job displacement** – AI may replace some roles.
- **privacy** – training often uses large amounts of personal data.
- **transparency** – large models are "black boxes", hard to explain.
- **accountability** – when AI is wrong, who is responsible: developer, user, or operator?
- **misuse** – deepfakes, misinformation, surveillance.



How bias gets into AI: biased data → a biased model → unfair decisions

Professionals must **understand the limits** of the AI they build, **inform users**, and **reduce harm**.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly.

Term	Definition
ethics	the moral principles that govern how a person behaves, in a profession the standards set out in its code of conduct
code of conduct	the rules an organisation or professional body sets for how its members must behave
copyright	the legal right of the creator of an original work to control how it is copied, distributed and modified
software licence	the legal agreement that states how a piece of software may be used, copied and distributed
commercial software	software sold for a fee, without source code, under a licence that protects the developer's intellectual property
free software (FSF)	software whose users are free to run, study, change and redistribute it, so its source code is available
open source (OSI)	software whose source code is available and which may be modified and redistributed under its licence

Term	Definition
shareware	software distributed free for a trial period or with limited features, after which the user pays to continue
freeware	software that is free of charge to use and copy, but whose source code is not released and may not be modified
artificial intelligence	computer systems that perform tasks which normally need human intelligence, such as recognising images and speech
machine learning	a form of AI in which a system improves at a task by learning patterns from data rather than by explicit programming

Exam tips

- Answer ethics questions against a **professional code of conduct** (public interest, competence, honesty), not personal opinion.
- Distinguish **copyright** (protects the expression) from a **patent** (protects an invention).
- Compare software licences: proprietary, open-source, freeware, shareware and FOSS.

Common mistakes

- Giving a personal opinion ("it is wrong") instead of a reason with a consequence ("faulty software could harm users, so it must be tested").
- Treating free software and freeware as the same thing. Free software is about the freedom to study and change the code; freeware is merely free of charge.
- Saying open source means free of charge. It means the source code is available and may be modified and redistributed; a fee may still be charged.
- Writing that copyright must be registered. It applies automatically to the work as written.
- Naming an impact without its consequence. "Job losses" scores when it is tied to why: the AI does the diagnosis, so fewer mechanics are needed.
- Describing what AI is instead of how it is used. The marks are for the steps: recognise, convert, translate, speak.

exercises.lol exercises.com.cn

Facial recognition in context

A camera captures an image; software detects a face and extracts features. An AI model compares these features with stored examples or a template and returns a match or confidence score. The system applies a threshold to decide whether to accept the match. Training uses many labelled examples; recognition uses the trained model on a new image. Poor lighting, biased training data and similar faces can cause false matches. Access control therefore needs a fallback check and protection of biometric data.

A light sensor can measure ambient light; an ADC converts its analogue output to digital data so a controller can adjust illumination against a threshold. Distinguish this sensor input from the camera image used to identify a person.

Free Software and Open Source definitions overlap: GPL and MIT are examples accepted under both. Free Software describes users' freedoms to run, study, modify and redistribute; it does not require a zero selling price. GPL is copyleft; MIT is permissive. Check the actual licence conditions rather than treating FSF and OSI as opposite licence types.

Databases

A-Level Computer Science

File-based storage and its limits

Before databases, programs stored data in **flat files** 平面文件 – usually one file per program. This is fine for small data but breaks down at scale.

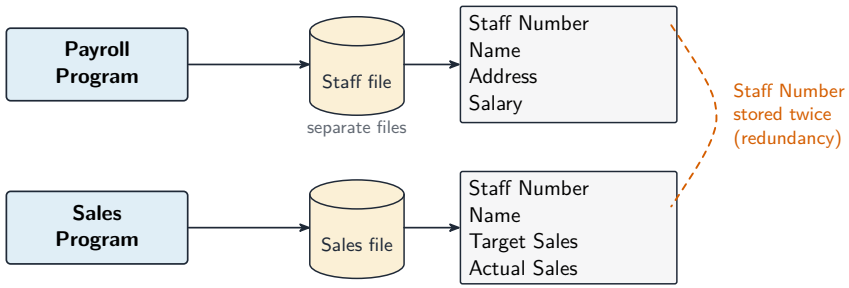


File-based storage keeps data in separate files, like papers in a filing cabinet – hard to search and easy to duplicate

Limitations

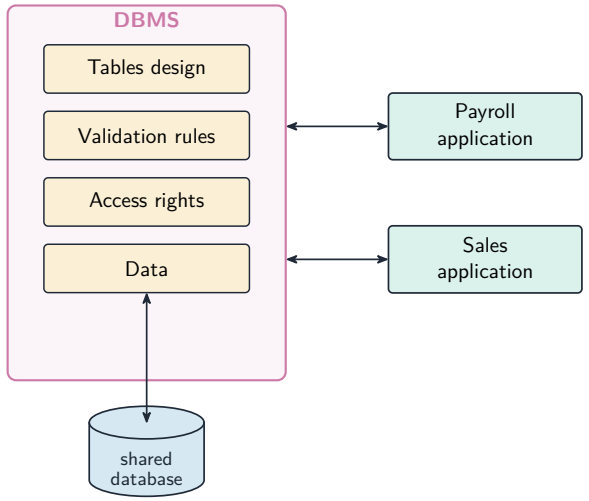
- **data redundancy** 数据冗余 – the same data (a customer's address) is held in several files, one per program, so storage is wasted and every copy must be updated.
- **data inconsistency** 数据不一致 – when one copy is updated and another is not, the files disagree and nobody knows which is right.
- **data dependence** – each program is written for the exact layout of its files; change a field's length or add a field and every program that reads the file must be rewritten.
- **no shared access** – a file is locked while one program uses it, so users cannot work on the data at the same time.
- weak **integrity** 完整性 – no central rules stop an invalid value or a link to a customer who does not exist; weak **security** – access is per file, not per field; and **queries** across files need a new program each time.

exercises.lol exercises.com.cn



The file-based approach: each program keeps its own files

A **relational database** 关系数据库 fixes these by storing data in **tables** managed by one piece of software (the DBMS) that all programs use.



The database approach: one DBMS serves all the programs

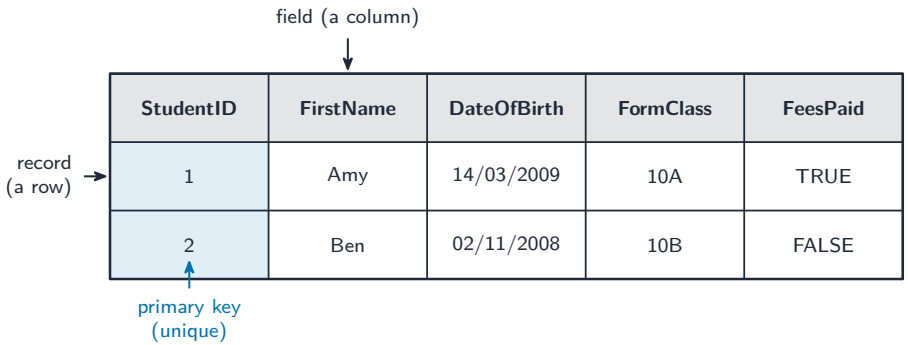
Why a relational database is better – the three-mark answer. Each item of data is stored **once**, in one table, and tables are linked by keys, so there is no redundancy and no inconsistency; the data is **independent** of the programs, which ask the DBMS for what they need and are unaffected when the structure changes; and the DBMS enforces **integrity** rules, controls **access** per user and per field, allows **many users at once**, and answers any **query** without a new program being written.

> **Worked example.** A repair shop stores its customers, devices and repair jobs using a file-based approach, one file per program. Give three problems this causes, and describe how a relational database would remove them.

The customer's name and phone number are stored in the repairs file *and* the invoices file (redundancy); when a customer changes number, one file is updated and the other is not (inconsistency); and when the shop wants a new report – repairs per technician – a new program has to be written to read the files (no ad-hoc queries). In a relational database the customer is stored once in a CUSTOMER table and referred to by CustomerID from the REPAIR table, so a change is made once and is seen everywhere; the report is a single SQL query.

Relational model – terms

- **table** 表 (relation) – a grid of rows and columns; one table per type of **entity** 实体 (e.g. `CUSTOMER`).
- **record** 记录 (row, also called a **tuple** 元组) – one row; one instance of the entity.
- **field** 字段 (column, also called an **attribute** 属性) – one column; one piece of information about each record.
- **primary key** 主键 – a field (or fields) that **uniquely identifies** each record; never null or duplicated.
- **foreign key** 外键 – a field whose value matches the primary key of another table, linking the two.
- **composite key** 复合键 – a primary key made of two or more fields together.
- **candidate key** 候选键 – any field(s) that could be the primary key.
- **secondary key** 次键 – a non-primary field that is indexed for fast searching.
- **indexing** 索引 – building an index on a field so look-ups and joins run faster.
- **referential integrity** 参照完整性 – every foreign-key value must match an existing primary key (no orphan records).

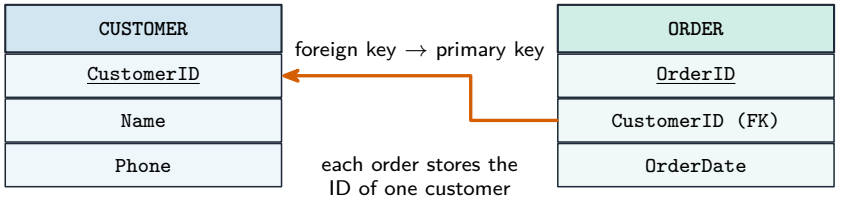


Each row is a record and each column is a field; the primary key (StudentID) is unique for every record

exercises.lol exercises.com.cn

A table is written in shorthand with the primary key underlined and foreign keys noted:

```
CUSTOMER(CustomerID, Name, Phone)
ORDER(OrderID, CustomerID, OrderDate)   -- CustomerID is FK → CUSTOMER
```



A foreign key links two tables: ORDER.CustomerID matches the primary key CUSTOMER.CustomerID

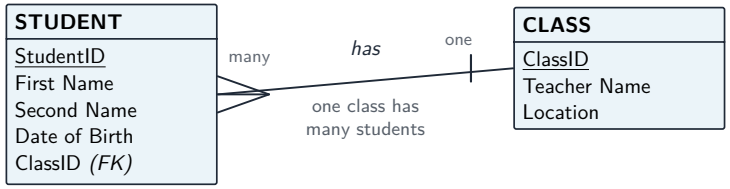
> **Worked example.** State what is meant by *entity*, *primary key* and *referential integrity* in a relational database, and complete the term ↔ description table for *tuple* and *attribute*.

An **entity** is something about which data is stored – a person, object or event – which becomes one table. A **primary key** is the attribute (or combination of attributes) that uniquely identifies each record in a table. **Referential integrity** means that every foreign-key value must match the value of a primary key in the table it refers to, so a record cannot refer to one that does not exist. A **tuple** is one row of a table (one record); an **attribute** is one column (one field). Learn the pairs: table/relation, record/tuple, field/attribute.

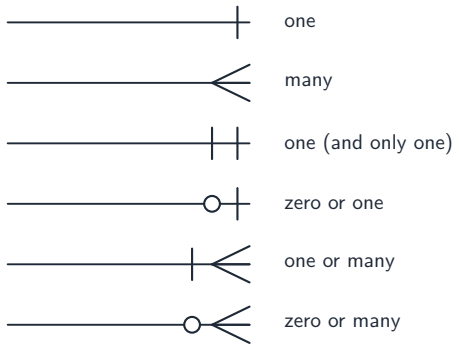
Entity-relationship (E-R) diagrams

An **entity-relationship diagram** 实体关系图 shows the structure: each entity is a rectangle, each relationship a line, with the **cardinality** 基数 marked at each end:

- **one-to-one** (1:1).
- **one-to-many** 一对多 (1:M) – each Customer has many Orders; each Order has one Customer.
- **many-to-many** (M:N) – Students take many Courses, and Courses have many Students.



An E-R diagram: one class has many students



Crow's-foot symbols for the cardinality of a relationship

A many-to-many relationship cannot be stored directly. Break it into two one-to-many relationships through a **link table** 连接表 holding the two foreign keys:

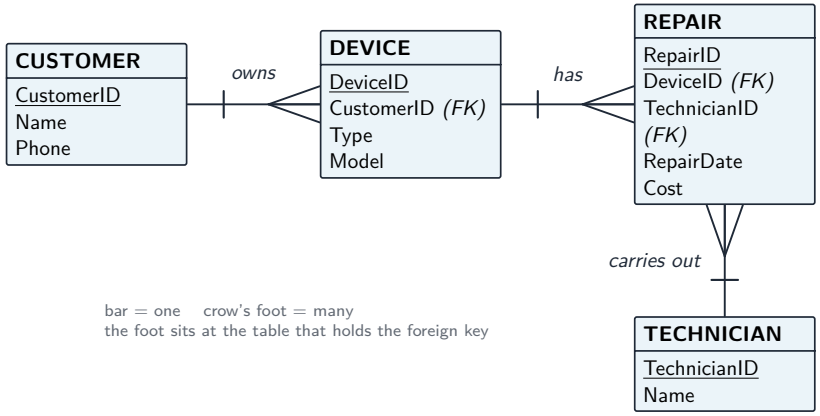
```
ENROLMENT(StudentID, CourseID, EnrolmentDate)
```



a link table turns many-to-many into two one-to-many relationships

A link table resolves a many-to-many relationship into two one-to-many relationships

Drawing the E-R diagram for a given set of tables. Each table becomes an entity. A relationship exists wherever one table holds a **foreign key** to another; it runs from the table holding the foreign key (the *many* end) to the table whose primary key it is (the *one* end). A table with two foreign keys and no other identity is usually a **link table** resolving a many-to-many relationship. Label each line with the relationship type.



Drawing the diagram from the tables: every foreign key is a one-to-many relationship, with the "many" at the table that holds it

> **Worked example.** A repair shop has the tables `CUSTOMER(CustomerID, Name, Phone)`, `DEVICE(DeviceID, CustomerID, Type, Model)`, `TECHNICIAN(TechnicianID, Name)` and `REPAIR(RepairID, DeviceID, TechnicianID, RepairDate, Cost)`. Identify the relationships and their types.

`DEVICE` holds `CustomerID`, so CUSTOMER–DEVICE is **one-to-many** (one customer, many devices). `REPAIR` holds `DeviceID`, so DEVICE–REPAIR is **one-to-many**; it also holds `TechnicianID`, so TECHNICIAN–REPAIR is **one-to-many**. There is no direct CUSTOMER–REPAIR line: the link runs through DEVICE. Three lines, three crow's feet, all at the REPAIR or DEVICE ends.

Normalisation

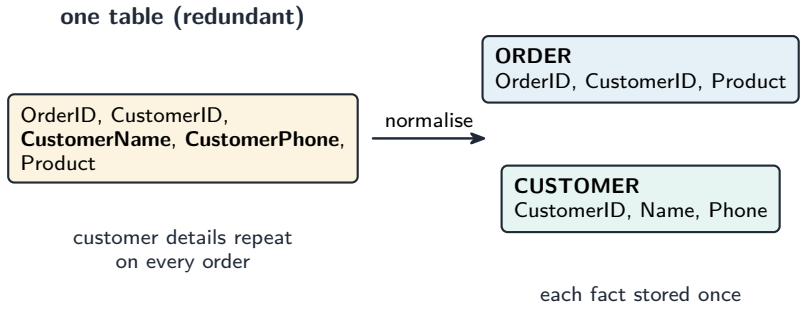
Normalisation 规范化 organises tables to **cut redundancy and inconsistency**, going through **normal forms** 范式 in order.

- **First normal form (1NF)** – every field holds a single (**atomic** 原子) value, with no repeating groups, and a primary key.
- **Second normal form (2NF)** – in 1NF, and every non-key field depends on the **whole** primary key (only matters for a composite key).
- **Third normal form (3NF)** – in 2NF, and every non-key field depends **only** on the primary key, not on another non-key field (no **transitive dependency** 传递依赖).

A 3NF design stores each fact once, so insert/update/delete anomalies disappear. The trade-off is more tables and more joins. Aim for 3NF.

exercises.lol exercises.com.cn

To produce a 3NF design: find the entities and their attributes; choose a primary key for each; split repeating/non-atomic fields (1NF); split fields depending on part of a composite key (2NF); split fields depending transitively on the key (3NF); add foreign keys for the relationships.

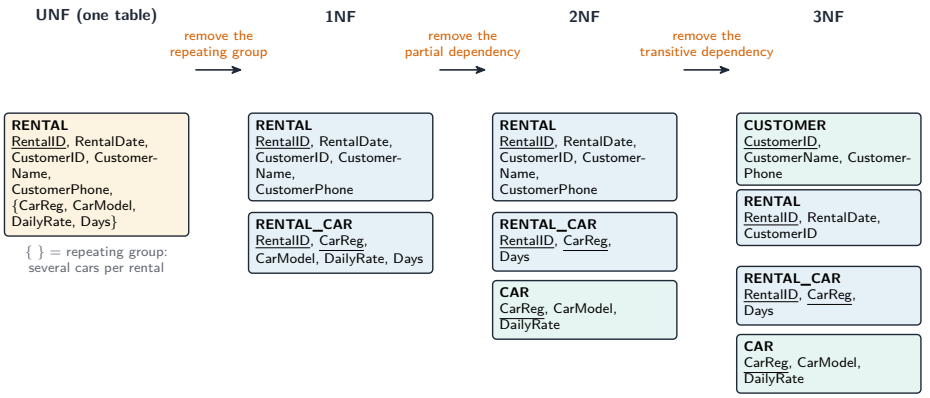


Normalisation removes redundancy by splitting repeated data into its own table

> **Worked example.** The table `ORDER(OrderID, CustomerID, CustomerName, ProductID, Quantity)` has the composite primary key `(OrderID, ProductID)`. Normalise it to 3NF. Test each non-key field against the key. `Quantity` depends on **both** `OrderID` and `ProductID`, which is fine. But `CustomerID` depends on `OrderID` **alone** - only *part* of the composite key. That is a **partial dependency**, so the table is not in **2NF**. Split it into `ORDER_LINE(OrderID, ProductID, Quantity)` and `ORDER(OrderID, CustomerID, CustomerName)`. Now test 3NF: in that new `ORDER` table, `CustomerName` depends on `CustomerID`, which is **not** the key - a **transitive dependency**. Split again: `ORDER(OrderID, CustomerID)` and `CUSTOMER(CustomerID, CustomerName)`. Name the **dependency** that breaks each form (partial breaks 2NF, transitive breaks 3NF); "it has repeated data" describes the symptom and earns nothing.

The three questions to ask of any table. *Is every cell a single value, with no repeating group?* If not, it is not in 1NF. *If the key is composite, does every non-key field depend on the whole key?* If some field depends on part of it, there is a **partial dependency** 部分依赖 and the table is not in 2NF. *Does every non-key field depend on the key alone?* If a field depends on another non-key field, there is a transitive dependency and the table is not in 3NF. An "explain why the table is not in 3NF" answer names the dependency and the fields involved.

exercises.lol exercises.com.cn



1NF removes the repeating group, 2NF the partial dependency, 3NF the transitive dependency

> **Worked example.** A car-rental shop records each rental as `RENTAL(RentalID, RentalDate, CustomerID, CustomerName, CustomerPhone, CarReg, CarModel, DailyRate, Days)`, where one rental can include several cars. Explain why the table is not normalised and produce a 3NF design.

Not in 1NF: the car fields `CarReg, CarModel, DailyRate, Days` form a **repeating group** – one rental has several cars. Move them to `RENTAL_CAR(RentalID, CarReg, CarModel, DailyRate, Days)` with the composite key `(RentalID, CarReg)`. *Not in 2NF:* in `RENTAL_CAR`, `CarModel` and `DailyRate` depend on `CarReg` alone – a **partial dependency**. Move them to `CAR(CarReg, CarModel, DailyRate)`, leaving `RENTAL_CAR(RentalID, CarReg, Days)`. *Not in 3NF:* in `RENTAL`, `CustomerName` and `CustomerPhone` depend on `CustomerID`, a non-key field – a **transitive dependency**. Move them to `CUSTOMER(CustomerID, CustomerName, CustomerPhone)`, leaving `RENTAL(RentalID, RentalDate, CustomerID)`. The 3NF design is four tables – `CUSTOMER`, `RENTAL`, `RENTAL_CAR`, `CAR` – with `CustomerID`, `RentalID` and `CarReg` as foreign keys; underline every primary key.

Database Management System (DBMS)

A **DBMS** 数据库管理系统 manages the database centrally. Features that fix the file-based limits:

- **data dictionary** 数据字典 – a description of every table, field, type and key; programs query it instead of hard-coding the structure.
- redundancy/consistency control – each fact stored once.
- **concurrent access** 并发访问 control – locks and transactions let many users work at once.

- **backup** 备份 and recovery; security and per-user permissions.
- **integrity rules** – keys, unique and range constraints, enforced centrally.
- **transactions** 事务 – a group of operations that all succeed or all fail.
- **views** 视图 – virtual tables that show each user "their" slice of the data.
- **data management** 数据管理 and **data modelling** 数据建模 – control how data is stored and define its structure as a **logical schema** 逻辑模式 (the logical design, independent of physical storage).
- **data integrity** 数据完整性 and **data security** 数据安全 – enforce correctness and control access centrally.
- a **query processor** 查询处理器 runs queries; a **developer interface** 开发者接口 gives tools and APIs for building applications.

Its tools include a data-dictionary editor, a query builder, a forms builder, a report generator, user management, and an SQL editor.

What the data dictionary holds (a "give three items" question): the names of the tables; the names of the fields in each table; each field's data type and length; the primary and foreign keys and the relationships between tables; validation rules; indexes; and who may access each table. It is metadata – data *about* the data – and the DBMS uses it to check every query and every change.

How the DBMS keeps the data secure (a "describe two methods" question): **authentication** 身份验证 – a username and password, or a biometric, before any access; **access rights** – each user or group is allowed to read, write or delete only certain tables or fields, often through a **view**; **encryption** of the stored data and of data sent to it, so a copied file is unreadable; **backups** taken regularly, so the data can be restored after loss; and a transaction log that records who changed what.

The two software tools. The **developer interface** is what a programmer uses to build the database and the applications on it: create tables and set keys and validation, write queries and SQL, and design forms and reports, without knowing how the data is physically stored. The **query processor** takes a query (SQL from a program, or a query built in the interface), checks it against the data dictionary, works out the most efficient way to run it, retrieves the data and returns the results.

Logical schema. The DBMS keeps the **logical** design (which tables and fields exist and how they relate) separate from the **physical** storage (files, indexes, disk blocks). Programs work with the logical schema, so the physical storage can be reorganised without changing a single program – this is the data independence the file-based approach lacked.

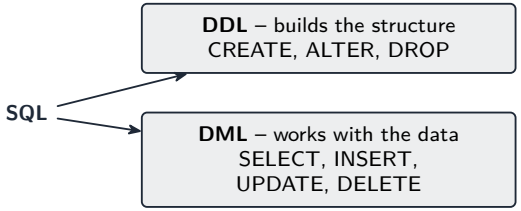
exercises.lol exercises.com.cn



The physical storage the logical schema hides: a hard disk's spinning platters and read/write head

DDL and DML

SQL 结构化查询语言 (Structured Query Language) has two halves:



DDL builds the database structure; DML works with the data

- **Data Definition Language** 数据定义语言 (DDL) – creates or changes the **structure** (tables, keys, constraints).
- **Data Manipulation Language** 数据操纵语言 (DML) – works with the **data** (insert, update, delete, **query** 查询).

DDL basics

```
CREATE TABLE CUSTOMER (
  CustomerID INTEGER PRIMARY KEY,
  Name VARCHAR(50) NOT NULL,
  Phone VARCHAR(20)
);
```

Add a foreign key:

```
CREATE TABLE ORDER (
  OrderID INTEGER PRIMARY KEY,
  CustomerID INTEGER,
  OrderDate DATE,
  FOREIGN KEY (CustomerID) REFERENCES CUSTOMER(CustomerID)
);
```

Modify and drop:

```
ALTER TABLE CUSTOMER ADD Email VARCHAR(100);
DROP TABLE CUSTOMER;
```

Common types: `INTEGER`, `REAL`, `VARCHAR(n)`, `CHAR(n)` (also `CHARACTER(n)`), `DATE`, `TIME`, `BOOLEAN`, `DECIMAL(p, s)`.

DML basics

Query with `SELECT`:



A SELECT query returns only the rows that match its condition

```
SELECT Name, Phone
FROM CUSTOMER
WHERE City = 'London'
ORDER BY Name ASC;
```

`SELECT` lists fields, `FROM` names the table, `WHERE` filters rows, `ORDER BY` sorts.

A **join** 连接 combines two tables using a foreign-key relationship:

```
SELECT C.Name, O.OrderDate
FROM CUSTOMER C INNER JOIN ORDER O
  ON C.CustomerID = O.CustomerID
WHERE O.OrderDate >= '2024-01-01';
```

```
SELECT C.Name, COUNT(D.DeviceID) AS Devices ..... the fields to output; a calculated column, given a name
FROM CUSTOMER C ..... the first table, with a short alias
INNER JOIN DEVICE D ON C.CustomerID = D.CustomerID ..... the second table, linked through the foreign key
WHERE D.Type = 'phone' ..... keep only the rows that match
GROUP BY C.Name ..... one output row per customer
ORDER BY Devices DESC; ..... sort the result, largest first
```

The parts of a query, in the order they must be written

Aggregate functions 聚合函数 (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) are often used with `GROUP BY`:

```
SELECT CustomerID, COUNT(*) AS NumOrders
FROM ORDER
GROUP BY CustomerID;
```

Insert, update, delete:

```
INSERT INTO CUSTOMER (CustomerID, Name, Phone)
VALUES (101, 'Ada Lovelace', '020-1234-5678');

UPDATE CUSTOMER SET Phone = '020-9999-0000' WHERE CustomerID = 101;

DELETE FROM CUSTOMER WHERE CustomerID = 101;
```

Always put a `WHERE` clause on `UPDATE` and `DELETE`, or the change hits **every** row.

Tips for exam SQL

- use the exact table and field names from the question.
- quote strings with single quotes (`'Smith'`); don't quote numbers.
- comparisons: `=`, `<`, `>`, `<=`, `>=`, `<>`.
- `LIKE 'A%'` matches anything starting with A (`%` = any string, `_` = one character); `IN (1,2,3)`; `BETWEEN 10 AND 20`.
- combine conditions with `AND` / `OR` / `NOT`, and end each statement with a semicolon.

exercises.lol exercises.com.cn

The DDL pattern the exam wants. Every `CREATE TABLE` names each field with its type, marks the primary key, and declares each foreign key with the table it references; a composite key is declared on its own line:

```
CREATE TABLE RENTAL_CAR (
  RentalID INTEGER,
  CarReg VARCHAR(8),
  Days INTEGER,
  PRIMARY KEY (RentalID, CarReg),
  FOREIGN KEY (RentalID) REFERENCES RENTAL(RentalID),
  FOREIGN KEY (CarReg) REFERENCES CAR(CarReg)
);
```

Worked example. Using `CUSTOMER(CustomerID, Name, Phone)` and `DEVICE(DeviceID, CustomerID, Type, Model)`, write SQL scripts to: (a) list the name and phone number of every customer who owns a device of type `'tablet'`, in alphabetical order of name; (b) count the devices of each type; (c) record that customer 17 now has the phone number `'0771 234 5678'`; (d) add a new device, ID 305, a `'laptop'` of model `'X1'` belonging to customer 17.

(a)

```
SELECT CUSTOMER.Name, CUSTOMER.Phone
FROM CUSTOMER INNER JOIN DEVICE
  ON CUSTOMER.CustomerID = DEVICE.CustomerID
WHERE DEVICE.Type = 'tablet'
ORDER BY CUSTOMER.Name ASC;
```

(b)

```
SELECT Type, COUNT(DeviceID) AS NumberOfDevices
FROM DEVICE
GROUP BY Type;
```

(c) `UPDATE CUSTOMER SET Phone = '0771 234 5678' WHERE CustomerID = 17;`
(d) `INSERT INTO DEVICE (DeviceID, CustomerID, Type, Model) VALUES (305, 17, 'laptop', 'X1');`

Marks are given per clause – the fields, the tables, the join condition, the `WHERE`, the `ORDER BY` – so a script with one wrong clause still scores the rest. Write `Table.Field` whenever two tables are involved.

Worked example. Explain what this script does: `SELECT T.Name, SUM(R.Cost) AS Total FROM TECHNICIAN T INNER JOIN REPAIR R ON T.TechnicianID = R.TechnicianID GROUP BY T.Name;`

It outputs each technician's name with the total cost of the repairs that technician has carried out, one row per technician: the two tables are joined on TechnicianID, the rows are grouped by name, and the costs in each group are added. When asked what a script does, describe the result, not the syntax.

■ SQL conditions and keys

`LIKE 'Li%'` matches text starting with `Li`; `%` represents any number of characters and `_` exactly one. `AND` requires both conditions; `OR` requires at least one. Use brackets when combining them.

```
CREATE DATABASE Club;
CREATE TABLE Booking (
  MemberID INTEGER,
  EventID INTEGER,
  PRIMARY KEY (MemberID, EventID)
);
ALTER TABLE Booking ADD FOREIGN KEY (MemberID)
  REFERENCES Member(MemberID);
SELECT Name FROM Member
WHERE Name LIKE 'Li%' AND (Year = 12 OR Year = 13)
ORDER BY Name DESC;
```

The composite primary key makes the **pair** unique, not each column separately. A foreign key prevents a non-null reference to a missing parent record. `ON UPDATE CASCADE` carries a parent-key change to referencing rows; `ON DELETE CASCADE` deletes those rows when the parent is deleted.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
entity	something about which data is stored – a person, object or event – which becomes a table in a relational database
attribute	one item of data about an entity (a column of the table)
tuple	one row of a table: one instance of the entity
primary key	an attribute, or combination of attributes, that uniquely identifies each record in a table
foreign key	an attribute in one table whose value matches a primary key in another table, used to link the two
candidate key	any attribute (or combination) that could be chosen as the primary key

exercises.lol exercises.com.cn

Term	Definition
secondary key	a non-primary attribute that is indexed so the table can be searched or sorted on it quickly
composite key	a primary key made of two or more attributes together
referential integrity	every foreign-key value must match an existing primary-key value in the table it refers to
first normal form	a table in which every attribute is atomic, there are no repeating groups, and there is a primary key
second normal form	in 1NF, and every non-key attribute depends on the whole of the primary key (no partial dependency)
third normal form	in 2NF, and no non-key attribute depends on another non-key attribute (no transitive dependency)
data dictionary	the metadata a DBMS keeps about the structure of the database: tables, fields, types, keys, relationships, validation
DDL / DML	the language used to define or change the structure of a database / the language used to query and maintain the data in it

Exam tips

- Define the terms exactly: entity, attribute, **primary key, foreign key**, and the relationship types (1:1, 1:many, many:many).
- Give a reason at each normal form: **1NF** (no repeating groups), **2NF** (no partial dependency), **3NF** (no non-key dependency) – and name the fields involved.
- Explain what a **DBMS** provides (data independence, security, integrity, concurrent access, a data dictionary, a developer interface, a query processor).
- Distinguish **DDL** (define the structure) from **DML** (query and change the data), and write SQL clause by clause: `SELECT`, `FROM`, `INNER JOIN … ON`, `WHERE`, `GROUP BY`, `ORDER BY`.
- To draw an E-R diagram from tables, find each foreign key first: every foreign key is one one-to-many relationship, with the "many" at the table that holds it.

exercises.lol exercises.com.cn

Common mistakes

- Drawing a many-to-many relationship directly. It must be split into two one-to-many relationships through a link table holding both foreign keys.
- Explaining "not in 3NF" by "the data is repeated". Name the dependency (partial or transitive) and the fields involved.
- Double quotes round strings in SQL, or quotes round numbers. Strings take `'single quotes'`; numbers take none.
- Leaving out the `ON` condition after `INNER JOIN`. Without it the two tables are not linked.
- Putting an ordinary field next to `COUNT` or `SUM` in a `SELECT` without a `GROUP BY`.
- `UPDATE` or `DELETE` without a `WHERE`. It changes or removes every row in the table.

Algorithm Design and Problem-solving

A-Level Computer Science

Computational thinking

Computational thinking 计算思维 is the set of mental tools for **analysing a problem and designing a solution a computer can run**. Two key ones are abstraction and decomposition.



Computational thinking breaks a big problem into smaller, easier parts – like solving a jigsaw

Abstraction

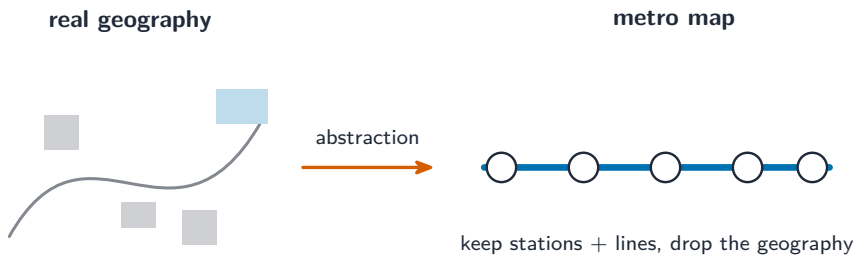
Abstraction 抽象 means **keeping the essential features** of a problem and **ignoring the irrelevant detail**, giving a simpler model.

Examples:

- a **train-network map** keeps the stations and lines but drops the geography.
- a **class** in object-oriented programming keeps only the attributes and methods the system needs.
- a **function** hides a piece of work behind a name.

A full model of any real problem would be too big to reason about, so abstraction is essential.

The examiner asks for the **purpose** of abstraction and for its **benefits**. Purpose: to produce a simpler model of a problem that contains only the details needed to solve it. Benefits: the problem is easier to understand and to program; the program is smaller and faster to write and test; the same model can be reused for similar problems. When you are asked to **produce an abstract model** of a system, list only the data and actions the task needs. For a school timetable that means the classes, rooms, teachers and periods; it does not mean the colour of the rooms or the age of the teachers.



Abstraction keeps the essentials (stations and lines) and drops irrelevant detail (the geography)

Decomposition

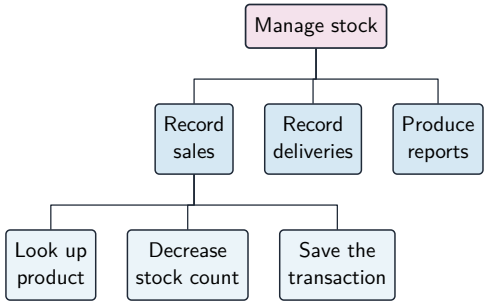
Decomposition 分解 means **breaking a large problem into smaller sub-problems**, each easier to solve and tackled one at a time.

1. find the main parts of the task.
2. break each into smaller sub-tasks.
3. continue until each is small enough to design directly.
4. solve the small tasks and combine them.

For stock control: "manage stock" → "record sales", "record deliveries", "produce reports" → ("record sales") "look up product", "decrease stock count", "save the transaction". Decomposition makes big problems manageable, lets a team divide the work, and gives modular code – each module becomes a **procedure** 过程 or function.

"Explain why decomposition is used" is a three-mark question with a fixed shape. Give three separate benefits: each **sub-problem** 子问题 is small enough to design, code and test on its own; different programmers can work on different **modules** 模块 at the same time; a module that already exists (or a library routine) can be reused, and a fault is easier to find because it lies inside one module. A structure chart (topic 12) is the diagram of a decomposition: the program at the top, its modules beneath, and the data passed between them.

exercises.lol exercises.com.cn



Decomposing a program into modules and sub-modules

Algorithms

An **algorithm** 算法 is a **solution expressed as a sequence of defined steps**. Each step is **unambiguous** 无歧义 (one meaning), **deterministic** 确定性 (same input → same output), **finite** (the steps end), and **effective** (each can be done). An algorithm says *what to do*, independent of the programming language used to implement it.

Identifier table

When you start an algorithm, list every piece of data in an **identifier table** 标识符表 – its **identifier** 标识符 (the **variable** 变量 name), **data type** 数据类型, and description. The exam's table has exactly these three columns:

Identifier	Data type	Description
`Category`	`STRING`	the product category
`SaleDate`	`DATE`	when the item was sold
`ItemCost`	`REAL`	cost of the item
`InStock`	`BOOLEAN`	`TRUE` if in stock
`Sales`	`ARRAY[1:30] OF REAL`	the last 30 daily sales totals

Use **descriptive** names (`ItemCost`, not `x`): an identifier starts with a letter, contains no spaces, and is written the same way every time it appears. Common types are `INTEGER`, `REAL`, `STRING`, `CHAR`, `BOOLEAN`, `DATE`, plus arrays. The table forces you to name every piece of data before writing code, and a "complete the identifier table" question gives one mark for each correct data type or description, so write the type exactly as the pseudocode guide does.

list every piece of data first - name, type, description

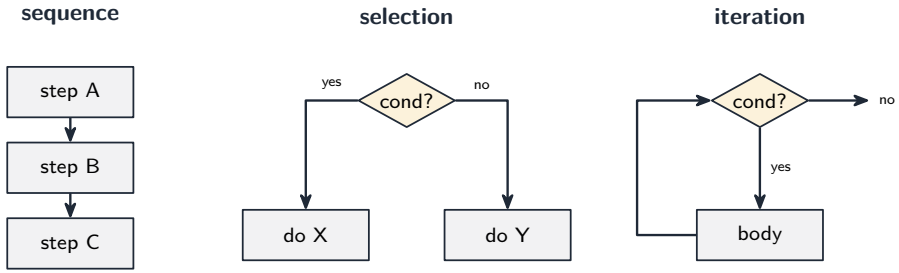
name	data type	description
`ItemCost`	`REAL`	cost of the item
`Category`	`STRING`	the product category
`SaleDate`	`DATE`	when the item was sold
`InStock`	`BOOLEAN`	TRUE if in stock

use **descriptive** names (`ItemCost`, not `x`) - the table forces you to plan the data

An identifier table names every piece of data before you write code

Pseudocode – the three basic constructs

Pseudocode 伪代码 is a structured, language-neutral way to describe algorithms.



The three building blocks of any algorithm: sequence, selection and iteration

1. Sequence

Steps run one after another (**sequence** 顺序):

```
INPUT Name
INPUT Age
OUTPUT "Hello", Name
```


exercises.lol exercises.com.cn

2. Selection

A choice of which steps run, based on a condition (**selection** 选择):

```
IF Age >= 18 THEN
    OUTPUT "Adult"
ELSE
    OUTPUT "Minor"
ENDIF
```

For more options, use `CASE OF ... ENDCASE`.

3. Iteration

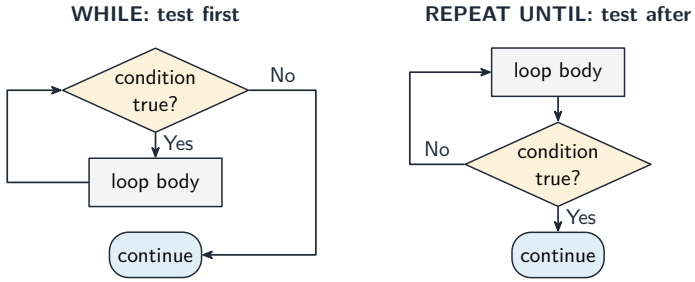
Repeating a block (**iteration** 迭代, a **loop** 循环):

```
FOR i ← 1 TO 10
    OUTPUT i
NEXT i
```

A **WHILE** loop tests the condition **before** each pass (may run zero times); a **REPEAT...UNTIL** loop tests **after** each pass (always runs at least once).

```
WHILE Total < 100 DO
    INPUT Value
    Total ← Total + Value
ENDWHILE

REPEAT
    INPUT Mark
UNTIL Mark >= 0 AND Mark <= 100
```



A WHILE loop tests before the body runs; a REPEAT ... UNTIL loop tests after it, so its body always runs at least once

Choosing the loop is itself a mark: `FOR` when you know how many times (a **count-controlled loop** 计数循环); `WHILE` when the loop might not run at all (a **pre-condition loop** 前测循环); `REPEAT ... UNTIL` when it must run at least once, as in validating an input (a **post-condition loop** 后测循环). A "describe the iteration construct" answer names the construct, says where the condition is tested, and gives the consequence (zero times or at least once).

Common operations

- **assignment** 赋值: `x ← 5` (an arrow; `=` is for comparison).
- input/output: `INPUT variable`, `OUTPUT expression`.
- comparisons `=`, `<>`, `<`, `>`, `<=`, `>=`; logic `AND`, `OR`, `NOT`.
- arithmetic `+ - * /`, plus `DIV` (integer division) and `MOD` (remainder).
- strings: `LENGTH`, `LEFT`, `RIGHT`, `MID`, and `&` for **concatenation** 拼接 (joining).

The pseudocode the exam expects

Every pseudocode answer is marked against Cambridge's published pseudocode guide. Write these forms exactly:

Construct	Pseudocode
Variable	`DECLARE Total : INTEGER`
Array	`DECLARE Marks : ARRAY[1:30] OF REAL`
Constant	`CONSTANT MaxTries = 3`
Assignment	`Total ← Total + Value`
Input / output	`INPUT Name` `OUTPUT "Hello ", Name`

Selection	`CASE OF Choice`
	`1 : OUTPUT "Add"`
	`OTHERWISE OUTPUT "Error"`
	`ENDCASE`

FOR loop	`FOR i ← 1 TO 10 STEP 2 ... NEXT i`
WHILE loop	`WHILE Total < 100 DO ... ENDWHILE`
REPEAT loop	`REPEAT ... UNTIL Mark >= 0`
Integer arithmetic	`17 DIV 5 = 3` `17 MOD 5 = 2`
Strings	`LENGTH(S)`, `LEFT(S, 3)`, `RIGHT(S, 2)` `MID(S, 2, 4)`, `TO_UPPER(S)`, `TO_LOWER(S)`
Conversions	`INT(3.7) = 3`, `NUM_TO_STR(12)` `STR_TO_NUM("4.5")`, `ASC('A') = 65`, `CHR(66) = 'B'`

exercises.lol exercises.com.cn

Construct	Pseudocode
Random	`RAND(100)` `INT(RAND(100)) + 1`

`RAND(100)` gives a real number from 0 up to (but not including) 100. `INT(RAND(100)) + 1` gives an integer from 1 to 100.

Two habits earn marks on every question: **declare** every variable you use, with the type from your identifier table, and **initialise** 初始化 every **counter** 计数器 and total (`Count ← 0`, `Total ← 0`) before the loop that changes it.

■ Input → Process → Output

Every program follows this shape:

```
INPUT Length
INPUT Width
Area ← Length * Width
OUTPUT "Area = ", Area
```

Listing the inputs and outputs first makes the algorithm cleaner.

> **Worked example.** Write pseudocode that inputs 100 integers and outputs how many of them, and the total of those, that lie between 10 and 20 inclusive.

Identifier table: `Count : INTEGER` (loop counter), `Value : INTEGER` (the integer just input), `InRange : INTEGER` (how many were in range), `Total : INTEGER` (their sum).

```
DECLARE Count, Value, InRange, Total : INTEGER
InRange ← 0
Total ← 0
FOR Count ← 1 TO 100
    INPUT Value
    IF Value >= 10 AND Value <= 20 THEN
        InRange ← InRange + 1
        Total ← Total + Value
    ENDIF
NEXT Count
OUTPUT InRange, Total
```

If the question then asks you to "identify two constructs and state how each is used", answer in the same shape: iteration, the `FOR` loop, repeats the input 100 times; selection, the `IF` statement, adds a value only when it is in range.

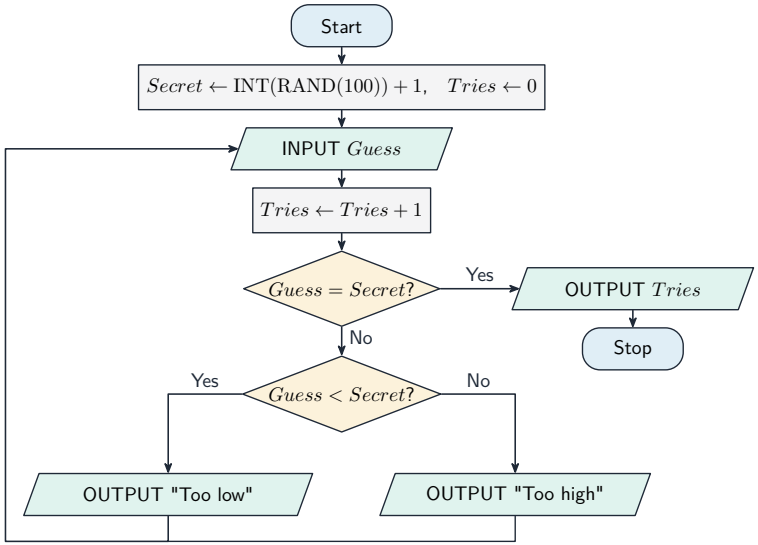
exercises.lol exercises.com.cn

> **Worked example.** A program picks a secret integer from 1 to 100. The user guesses until they are right; after each wrong guess the program says "Too low" or "Too high", and at the end it outputs how many guesses were made.

Identifier table: `Secret : INTEGER` (the number to guess), `Guess : INTEGER` (the user's input), `Tries : INTEGER` (how many guesses so far).

```
DECLARE Secret, Guess, Tries : INTEGER
Secret ← INT(RAND(100)) + 1
Tries ← 0
REPEAT
    INPUT Guess
    Tries ← Tries + 1
    IF Guess < Secret THEN
        OUTPUT "Too low"
    ELSE
        IF Guess > Secret THEN
            OUTPUT "Too high"
        ENDIF
    ENDIF
UNTIL Guess = Secret
OUTPUT "You took ", Tries, " guesses"
```

A `REPEAT ... UNTIL` loop is the right choice because the user must guess at least once. The marks are for: the random number in the right range, a loop that ends on a correct guess, the counter that starts at zero and increases inside the loop, the two messages under the right conditions, and the final output.

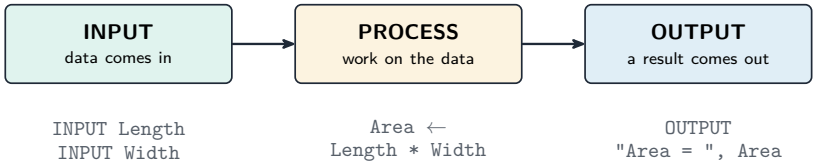


The same guessing game as a flowchart: the two decision diamonds are the two IF statements, and the return arrow is the REPEAT ... UNTIL loop

Worked example. Output two different random integers, each between −10 and 10 inclusive.

There are 21 possible values, so `INT(RAND(21))` gives 0 to 20 and subtracting 10 shifts it to the range −10 to 10. The second number must be generated again until it differs from the first:

```
DECLARE First, Second : INTEGER
First ← INT(RAND(21)) - 10
REPEAT
    Second ← INT(RAND(21)) - 10
UNTIL Second <> First
OUTPUT First, Second
```



Every program follows the Input, Process, Output shape

Three notations

The same algorithm can be written three ways.

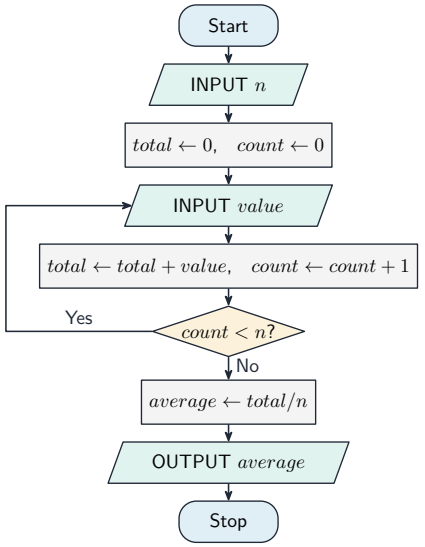
- **structured English** 结构化英语 – natural language with indentation and fixed keywords; good for a high-level description.
- **flowchart** 流程图 – a diagram with standard shapes:

Shape	Meaning
Rounded rectangle	Start / Stop
Parallelogram	Input / Output
Rectangle	Process
Diamond	Decision
Arrow	Flow of control

- **pseudocode** – the keyword notation above; closest to code.

You should be able to convert between any pair: each `IF` is a decision diamond, each loop is a back-arrow, and a sequence is stacked rectangles.

IF ... THEN ... ELSE ... ENDIF



A flowchart for averaging a list of numbers, using the standard shapes

exercises.lol exercises.com.cn

Stepwise refinement

Stepwise refinement 逐步求精 starts with a high-level outline and **expands each step** until it is small enough to code. For an average of n numbers:

Level 1:

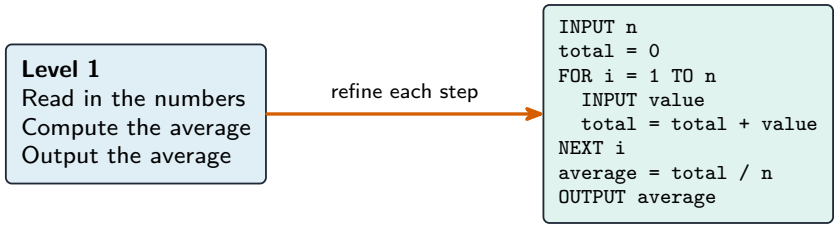
```
Read in the numbers
Compute the average
Output the average
```

Level 2:

```
INPUT n
total ← 0
FOR i ← 1 TO n
    INPUT value
    total ← total + value
NEXT i
average ← total / n
OUTPUT average
```

Each refinement keeps the previous structure and adds detail.

A six-mark "apply stepwise refinement" question gives you a high-level outline and wants each step expanded into the concrete statements a programmer could code. Keep the steps in the same order, name the data each step reads or produces, and stop when every line is a single input, assignment, output, loop or condition. For example, "validate the password" becomes: input the password; check its length is at least 8; check it contains at least one digit; output "accepted" if both checks pass, otherwise output "rejected".



Stepwise refinement: expand each high-level step into detailed pseudocode

Logic statements

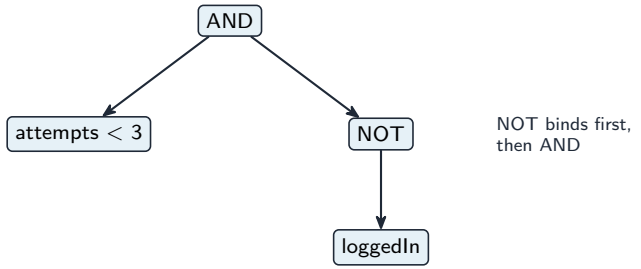
A **logic statement** 逻辑语句 is a **Boolean** 布尔 condition that controls branching, built from comparisons (`x > 10`), connectives (`AND`, `OR`, `NOT`) and brackets. Use it as the condition of `IF`, `WHILE` or `REPEAT...UNTIL`:

```
WHILE attempts < 3 AND NOT loggedIn DO
    INPUT password
    IF password = correctPassword THEN
        loggedIn ← TRUE
    ELSE
        attempts ← attempts + 1
    ENDIF
ENDWHILE
```

Precedence 优先级 (highest to lowest): `NOT`, then `AND`, then `OR`. Use brackets when unsure. Common mistakes:

- `a = 1 OR 2` is wrong – write `a = 1 OR a = 2`.
- `NOT a > 5` means `NOT (a > 5)`, i.e. `a <= 5`.
- `NOT (A AND B)` is the same as `(NOT A) OR (NOT B)` (**De Morgan's law** 德摩根定律) – handy for simplifying conditions.

Turning a sentence into a logic statement is a skill the papers test directly. "A ticket is free for anyone under 5 or over 65" becomes `Age < 5 OR Age > 65`. "A mark is valid if it is a whole number from 0 to 100" becomes `Mark >= 0 AND Mark <= 100`. "The loop stops when the file is finished or ten records have been read" becomes `UNTIL EOF(File) OR Count = 10`. Write each comparison in full: `Age > 65` and `Age < 5`, never `Age > 65 OR < 5`.



Precedence: NOT binds to loggedIn first, then AND combines the two sides

exercises.lol exercises.com.cn

Worked example. Write an identifier table and pseudocode to read 10 numbers and output the largest. The **identifier table** names each variable with its data type and purpose: `Count : INTEGER` (loop counter), `Num : REAL` (the number just read), `Max : REAL` (largest so far).

```
Max ← -999999
FOR Count ← 1 TO 10
    INPUT Num
    IF Num > Max THEN
        Max ← Num
    ENDIF
NEXT Count
OUTPUT Max
```

The design decision carrying the marks is **initialising `Max`**: it must start **lower than any possible input** - or, safer still, be set to the **first** number read. Initialise it to `0` and the algorithm wrongly returns 0 for a list of negative numbers, a bug your trace only exposes if the test data include a negative.

Programming prerequisites for algorithm questions

Before writing or tracing algorithms, use the expression and built-in-function methods on sheets 11.1.1–11.1.2. `TO_UPPER` and `TO_LOWER` accept whole strings; `UCASE` and `LCASE` in the guide act on a single character. Follow the question's insert.

Join strings with `&`; convert a number with `NUM_TO_STR` before joining it. Call a procedure with `CALL Name(arguments)`; use a function as an expression. `SETDATE(day, month, year)` constructs a DATE value. Check types before evaluation: an invalid expression is `ERROR`, not a guessed conversion.

For a flowchart with several totals, initialise **each** accumulator before the loop. Update them on the correct branches, advance the count, then test termination. Put averages after the loop and use the number of values actually included.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
abstraction	keeping the essential details of a problem and leaving out the details that are not needed
decomposition	breaking a problem down into smaller sub-problems, each of which can be solved separately
algorithm	a solution to a problem expressed as a sequence of defined steps

exercises.lol exercises.com.cn

Term	Definition
identifier table	a table listing each identifier used in an algorithm with its data type and a description of its purpose
pseudocode	a structured, language-independent way of writing the steps of an algorithm
flowchart	a diagram that shows the steps and decisions of an algorithm using standard symbols joined by arrows
sequence	statements executed one after another in the order written
selection	choosing which statements to execute according to a condition
iteration	repeating a group of statements while, or until, a condition holds
stepwise refinement	breaking each step of an outline into smaller steps, repeatedly, until each step can be coded directly
logic statement	a condition built from comparisons and the operators AND, OR and NOT that evaluates to TRUE or FALSE

Exam tips

- Define an **algorithm** as an unambiguous, finite, deterministic sequence of steps, independent of language.
- Use the three constructs correctly – **sequence, selection, iteration** – and keep an identifier table with data types.
- Break a problem down by **decomposition and abstraction**, then stepwise refinement.
- Write pseudocode that would actually run: declare variables and follow the exam's pseudocode style.

Common mistakes

- Using `=` to assign a value. Assignment is `←`; `=` is a comparison.
- Forgetting `ENDIF`, `ENDWHILE`, `ENDCASE` or `NEXT`. Every construct closes, and the closing word is where the mark for the construct is checked.
- Not initialising a total or counter before the loop, so the algorithm adds to a value that never existed.
- Using a `FOR` loop when the number of repetitions is unknown. Reading until a sentinel value or a correct guess needs `WHILE` or `REPEAT ... UNTIL`.

- Writing `Age > 65 OR < 5`. Each side of `OR` and `AND` must be a complete comparison.
- Answering "explain why decomposition is used" with one benefit written three ways. Three marks need three different benefits.

exercises.lol exercises.com.cn

Data Types and Structures

A-Level Computer Science

Choosing data types

Every variable needs a **data type** 数据类型 – the kind of value it holds and the operations allowed:

integer	whole number: 7
real	decimal: 3.5
char	one letter: 'A'
string	text: "hello"
boolean	true or false

The basic data types: integer, real, char, string and boolean

- `INTEGER` – a whole number (`42`, `-7`). For counts, indexes, IDs.
- `REAL` – a number with a fractional part (`3.14`). For money, measurements.
- `STRING` – characters in quotes (`"Hello"`). For text.
- `CHAR` – a single character (`'A'`).
- `BOOLEAN` – `TRUE` or `FALSE`. For flags.
- `DATE` – a calendar date.

Pick the smallest precise type that fits: `INTEGER` for whole counts, `BOOLEAN` for flags (not the strings `"yes"`/`"no"`).

The "give the appropriate data type" tables are decided by how the value is used: the average mark of a class is `REAL` (it has a fractional part); an email address is `STRING`; the number of students is `INTEGER`; whether a student has paid is `BOOLEAN`; a date of birth is `DATE`; an array index is always `INTEGER`; a single grade letter is `CHAR`; a phone number is a `STRING`, because it starts with `0` and is never used in arithmetic. A `BOOLEAN` is used for a flag with only two states: whether a search has found its target, whether a member has paid, whether a seat is booked. For the identifier table, the variable name must be meaningful too: `NumberOfPeople`, not `n`.

Records

A **record** 记录 (a **record structure** 记录结构) holds **several fields of different types** under one name – useful when several values describe one thing.

field	Name	Initial	Member	Age	Height	Joined
record	Amy	'M'	TRUE	42	1.62	14/03/2009
data type	*text/ alphanumeric*	*character*	*Boolean*	*integer*	*real*	*date/time*

Each field in a record holds one data type

```
TYPE TStockItem
    DECLARE ItemID : INTEGER
    DECLARE Category : STRING
    DECLARE ItemCost : REAL
    DECLARE InStock : BOOLEAN
ENDTYPE
```

This defines the **type** `TStockItem`; declare variables of it:

```
DECLARE Item1 : TStockItem
DECLARE Items : ARRAY[1:100] OF TStockItem
```

Use dot notation to reach each **field** 字段:

```
Item1.Category ← "Fruit"
OUTPUT Item1.Category, " costs ", Item1.ItemCost
```

Use a record when values **always belong together** (a customer, a stock item); use separate variables for unrelated values.

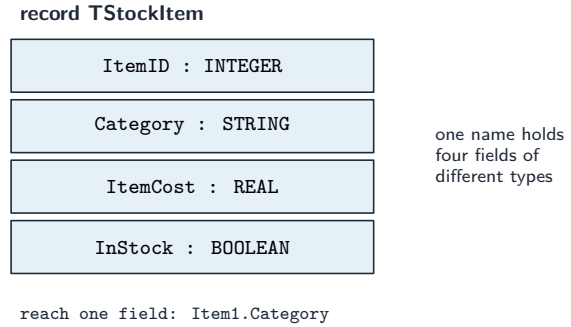
exercises.lol exercises.com.cn

Worked example. A club stores, for each student, a student ID (a string), a name, a date of birth and up to three club numbers (integers). Write pseudocode to declare the record type, an array to hold 3000 students, and a statement that stores a name in the first element.

```
TYPE Student
    DECLARE StudentID : STRING
    DECLARE Name : STRING
    DECLARE DateOfBirth : DATE
    DECLARE Club : ARRAY[1:3] OF INTEGER
ENDTYPE

DECLARE Membership : ARRAY[1:3000] OF Student
Membership[1].Name ← "Li Wei"
```

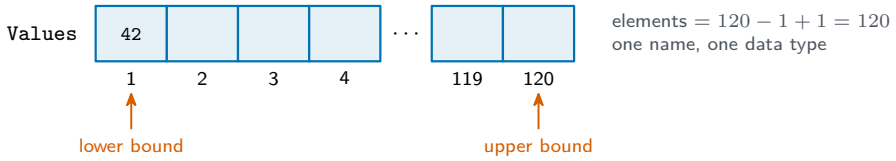
The marks: `TYPE` with the identifier and `ENDTYPE`; each field declared with a suitable type; the array declared with its bounds and `OF Student`; the field reached with the index and a dot. A "state the error in the record declaration" question usually points at a missing `ENDTYPE`, a field with no type, or a field declared as a `STRING` that must hold arithmetic. Two conventions score marks on their own: an **unused element** is marked with a value that cannot be real data (an empty string, `-1`, an ID of `0`), and it is good practice to use the same marker everywhere so that every module can recognise an unused slot; an unused club field is `0`. The benefits of an array of records, for a "state three benefits": all the data for one entity is held under one identifier; the fields can have different data types; one array replaces several parallel arrays that would have to be kept in step; the whole set can be processed by one loop or passed as one parameter; and adding a field changes the type definition only. For one customer the suitable structure is a **record** (fields of different types under one name); for all customers it is an **array of records**.



A record holds several fields of different types under one name

Arrays

An **array** 数组 is an ordered collection of items of the **same type**, under one name, reached by an **index** 索引.



Elements = upper bound - lower bound + 1.

- **element** 元素 – one item in the array.
- **bounds** 边界 – the lowest and highest valid indices.
- **dimension** 维度 – 1-D (a list), 2-D (a table), etc.
- **lower bound** 下界 and **upper bound** 上界 – the first and last valid index; the number of elements is upper bound minus lower bound plus one, and for a 2-D array the product of the two counts.

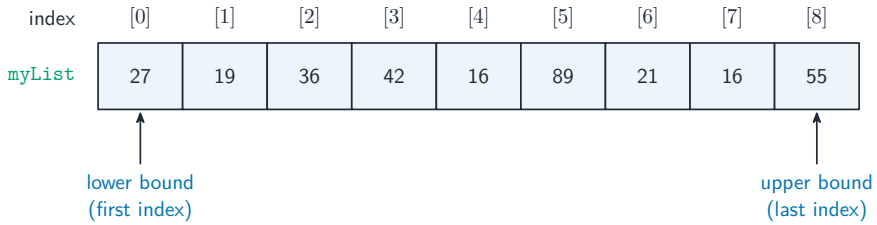
So in `ThisArray[n] ← 42` the array has one dimension, the index is the variable `n` (an `INTEGER`), and the element at that index receives `42`. Before an array can be declared you need its **data type** as well as its bounds. To declare 120 values that may include a decimal place: `DECLARE Data : ARRAY[1:120] OF REAL`; a 150-row, two-column table of strings: `DECLARE Data : ARRAY[1:150, 1:2] OF STRING`, which has 300 elements. The benefits of an array over separate variables, for a two-mark explain: one identifier instead of thirty; the elements can be processed by a loop with the index as the counter; the size is easy to change; and the whole set can be passed to a module as one parameter. An array can also replace a chain of selection statements: `Days InMonth[Month]` looks up the answer directly instead of twelve `IF` clauses, which is shorter, faster to write and easier to maintain.

1-D arrays

```
DECLARE Names : ARRAY[1:5] OF STRING
Names[3] ← "Cara"
OUTPUT Names[3]
```

Process every element with a `FOR` loop:

```
FOR i ← 1 TO 5
    OUTPUT Names[i]
NEXT i
```

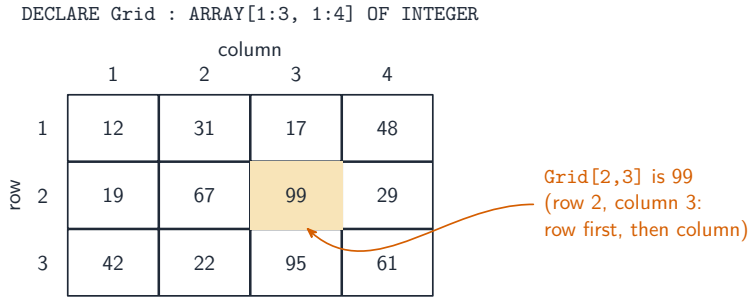



A 1-D array (a list) with indices and bounds

2-D arrays (2D array)

```
DECLARE Grid : ARRAY[1:3, 1:4] OF INTEGER
Grid[2, 3] ← 99
```

The first index is the row, the second the column. Use nested loops to visit every cell. Use **1-D** for a single sequence, **2-D** for two natural dimensions (a grid, rows × columns).



A 2-D array (a table) with row and column indices

exercises.lol exercises.com.cn

Worked example. `Marks : ARRAY[1:30, 1:5] OF INTEGER` holds the marks of 30 students (rows) in 5 tests (columns). Write pseudocode that outputs each student's total, and then the number of the student with the highest total.

```
Best ← 0
BestTotal ← -1
FOR Student ← 1 TO 30
    Total ← 0
    FOR Test ← 1 TO 5
        Total ← Total + Marks[Student, Test]
    NEXT Test
    OUTPUT "Student ", Student, ": ", Total
    IF Total > BestTotal THEN
        BestTotal ← Total
        Best ← Student
    ENDIF
NEXT Student
OUTPUT "Highest total: student ", Best
```

The outer loop is over rows (students) and the inner loop over columns (tests). `Total` must be reset to 0 **inside** the outer loop, before each student: set once before both loops, it would keep adding every student's marks together. `BestTotal` starts at -1, below any possible total, so the first student always replaces it.

Common operations

A **linear search** 线性查找 checks each element until found:

```
FOR i ← 1 TO n
    IF A[i] = Target THEN
        OUTPUT "Found at ", i
    ENDIF
NEXT i
```

To find a sum, count, maximum or minimum, set a running variable then sweep through:

```
Max ← A[1]
FOR i ← 2 TO n
    IF A[i] > Max THEN
        Max ← A[i]
    ENDIF
NEXT i
```

A **bubble sort** 冒泡排序 puts an array in order: pass through it comparing each adjacent pair and **swapping** any that are out of order; repeat the passes until one pass makes no swaps.

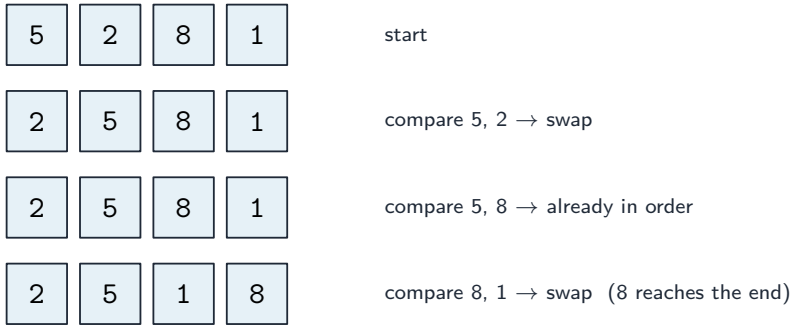
exercises.lol exercises.com.cn

Paper 2 asks for these algorithms both as pseudocode and as **steps in words**, and sometimes in their "efficient" form:

- **Largest value**: set `Largest` to the first element; for each remaining element, if it is bigger than `Largest`, store it in `Largest`; after the loop output `Largest`. For the *position* of the largest, keep a second variable that stores the index each time `Largest` changes.
- **Linear search returning a position**: set `FoundAt ← -1` before the loop (a value that can never be a valid index, so it means "not found"); loop through the array; when the element matches, store the index and leave the loop; after the loop test `FoundAt`.
- **Count or output the non-blank elements**: compare each element with the marker for an unused element (`""` or `-1`) and count or output only those that differ.
- **Remove an item**: find its index by a linear search; move every later element one place towards the start, so the gap closes; mark the last element as unused (or decrease the count).
- **Insert into a sorted array**: find the first index whose element is larger; move that element and every later one one place towards the end; store the new value in the gap.
- **Efficient bubble sort**: a `Swapped` flag so that the passes stop as soon as a pass makes no swap, and an upper limit that falls by one each pass because the largest value has already reached the end.

```
REPEAT
    Swapped ← FALSE
    FOR Index ← 1 TO Limit - 1
        IF Data[Index] > Data[Index + 1] THEN
            Temp ← Data[Index]
            Data[Index] ← Data[Index + 1]
            Data[Index + 1] ← Temp
            Swapped ← TRUE
        ENDIF
    NEXT Index
    Limit ← Limit - 1
UNTIL Swapped = FALSE
```

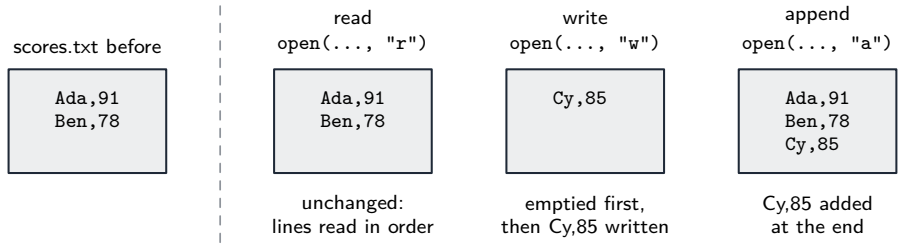
The marks are for the outer loop that repeats until no swaps, the flag set inside the `IF`, the three-line swap with a temporary variable, and the shrinking limit. A sort in "steps" (stepwise refinement) is: repeat until sorted; on each pass compare adjacent pairs; swap a pair that is out of order; after each pass the largest unsorted value is at the end. Two 1-D arrays of records or of parallel data are processed with one loop and one index; a 2-D array needs a nested loop, the outer over rows and the inner over columns, and a search in one row fixes the row index and loops over the column.



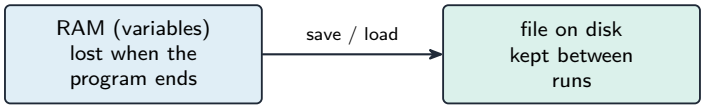
One pass of a bubble sort: adjacent pairs are compared and swapped, bubbling the largest value to the end

Files

A **file** 文件 is data stored on **secondary storage** 辅助存储器, kept between program runs. Variables in RAM disappear when the program ends, so to save data permanently (high scores, records, settings) the program writes to a file. Files also let programs share data and restart from a saved state.



Mode "w" destroys what was there; mode "a" keeps it



Variables in RAM vanish when the program ends; a file on disk persists between runs

A **text file** 文本文件 holds one or more lines of readable characters; programs read and write **text files** line by line. Open a file before use and **close** it after:

```
OPENFILE "data.txt" FOR READ      // or FOR WRITE, FOR APPEND
WHILE NOT EOF("data.txt") DO
    READFILE "data.txt", LineString
    OUTPUT LineString
ENDWHILE
CLOSEFILE "data.txt"
```

`EOF` tests the **end of file** 文件结束 before reading. To write:

```
OPENFILE "log.txt" FOR WRITE
FOR i ← 1 TO 100
    WRITEFILE "log.txt", "Event " & i
NEXT i
CLOSEFILE "log.txt"
```

Always **close** every file – otherwise buffered writes may be lost and other programs may be locked out.

Why files (two marks): the data is kept after the program ends, so it is available the next time the program runs; it can be shared with other programs; and it can hold more than fits in memory. The characteristic of a text file that lets a program work through it is that it is a sequence of lines, read one after another from the start. The three **modes**: `READ` to read from the start; `WRITE` to create a new file, which **deletes any existing contents**, so it cannot be used to add to a file; `APPEND` to add lines at the end of an existing file. Test `EOF` before every read, and open the file only once, even when several modules use it.

exercises.lol exercises.com.cn

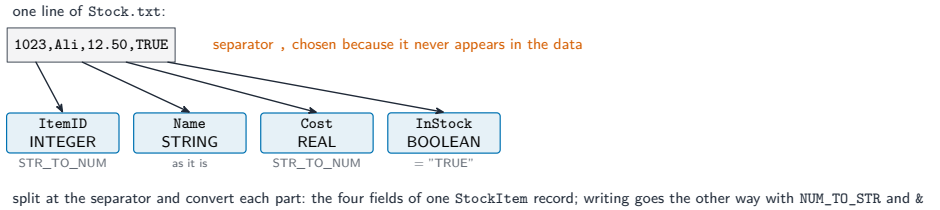
Worked example. Write pseudocode for a procedure `LastLines(FileName : STRING)` that outputs the last three lines of a text file, in order.

```
PROCEDURE LastLines(BYVAL FileName : STRING)
    DECLARE LineX, LineY, LineZ : STRING
    LineX ← ""
    LineY ← ""
    LineZ ← ""
    OPENFILE FileName FOR READ
    WHILE NOT EOF(FileName) DO
        LineX ← LineY
        LineY ← LineZ
        READFILE FileName, LineZ
    ENDWHILE
    CLOSEFILE FileName
    OUTPUT LineX
    OUTPUT LineY
    OUTPUT LineZ
ENDPROCEDURE
```

Each new line pushes the previous three along, so when the file ends the three variables hold its last three lines; a file with fewer lines outputs empty strings. To output the *first* five lines, count the lines read and stop the loop at five or at `EOF`, whichever comes first; a file that is empty is detected by `EOF` being `TRUE` immediately after opening.

Fields in a line. A text file holds strings, so a record is written as one line with its fields joined by a **separator** 分隔符 character, and each number or Boolean converted with `NUM_TO_STR` (and read back with `STR_TO_NUM`, or by comparing with `"TRUE"`). Choose a separator that can never appear in the data: a comma or `|` for names and numbers, never a space when a name may contain one. If a field may contain any character, the separator can be confused with data; the fix is to put each field on its own line, or to write the field's length before it. One item per line is simple to read back but uses more lines and makes a record harder to see as a unit. Reading a file whose lines are in a known order (ascending by an ID) allows the search to stop as soon as a larger ID is read, instead of reading to the end. A save file that is created each time the game is saved needs a **meaningful filename**, for instance the player's name and the date and time, so that any earlier save can be restored.

exercises.lol exercises.com.cn



One line of a text file is one record: fields joined by a separator, converted to their types when read back

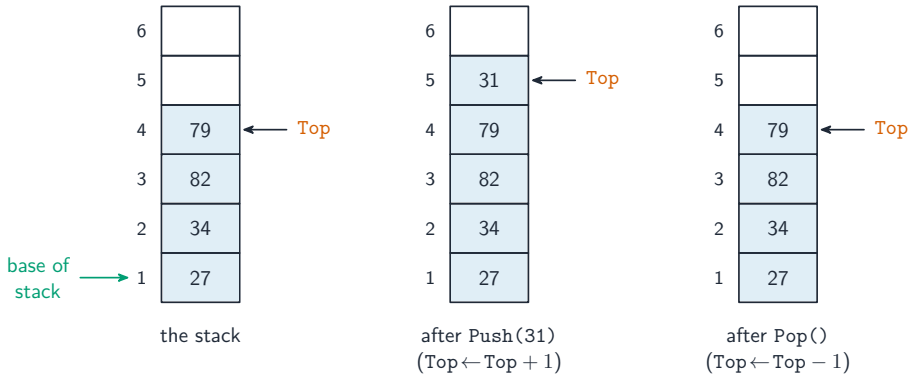
Abstract Data Types (ADTs)

An **Abstract Data Type** 抽象数据类型 (ADT) is a **collection of data plus operations on it**, defined by **what** it does, not how it is stored. The user works only through the operations; the implementation is hidden, so it can change without affecting code that uses the ADT. Know three: stack, queue, linked list.

The one-mark definition: an ADT is a collection of data together with a set of operations on that data. A stack, a queue, a linked list, a binary tree and an array are all ADTs. To **justify** a choice: a queue when items must be handled in the order they arrived (print jobs, key presses, customers in a shop), because it is first in, first out; a stack when the most recent item must be handled first (undo, going back through web pages, reversing an order, the return addresses of nested calls), because it is last in, first out; a linked list when items are inserted and deleted in the middle of an ordered sequence often, because only pointers change and nothing has to be shifted. To **compare a stack and a queue**: both are linear structures of items with an order, both are implemented with an array and pointers, and both need a check for full before adding and for empty before removing; a stack has one pointer and adds and removes at the same end, a queue has two pointers and adds at one end and removes at the other.

Stack

A **stack** 栈 works in **LIFO** 后进先出 order (Last In, First Out). Operations: **push** 入栈 (add to the top), **pop** 出栈 (remove from the top), peek (look at the top), and tests for empty/full. Uses: undo history, function-call return addresses, expression parsing, backtracking.



Push and pop change the top pointer; the base pointer stays put

Worked example. A stack of characters holds, from the bottom, `'P'`, `'N'`, `'Z'`, `'X'`, `'Y'`, `'W'`, with the top-of-stack pointer at `'W'` (memory location 202 of 200–207). The operations `POP`, `POP`, `PUSH 'A'`, `PUSH 'B'`, `POP` are performed. What is on the stack, and where does the pointer point?

The two pops remove `'W'` then `'Y'`; the pushes add `'A'` then `'B'` in their places; the last pop removes `'B'`. The stack now holds `'P'`, `'N'`, `'Z'`, `'X'`, `'A'` and the pointer is at `'A'`, location 203. The value that has been on the stack longest is the bottom item, `'P'`; at most five further pops are possible before the stack is empty, and a pop on an empty stack is an error, which is why `Pop()` tests for empty first. A `Push()` function that returns `TRUE` on success first tests whether the pointer is at the top of the array (full) and returns `FALSE` if so. The array elements need no initialising before use, because the pointer alone says which elements are in use.

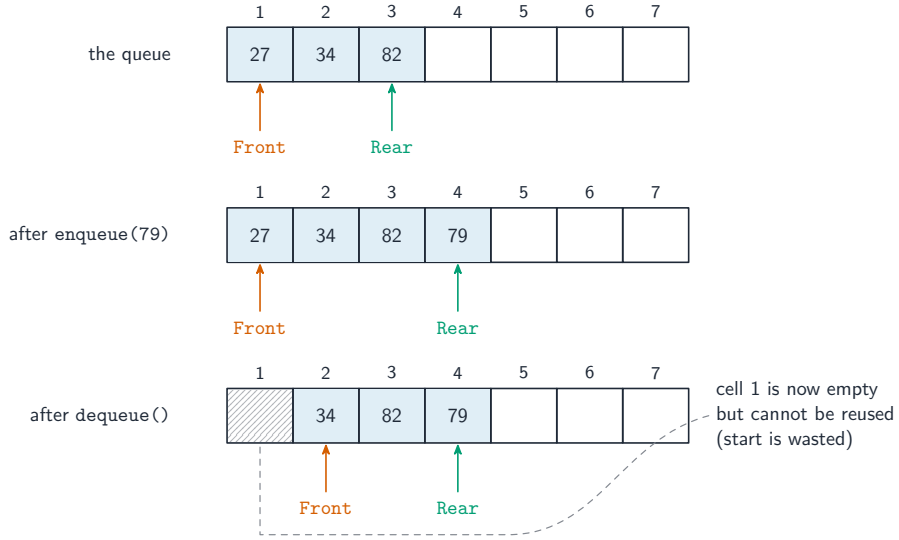


*A pile of books is a stack you can see. You can only add or take a book from the **top**, so the last one you put on is the first one you take off – that is exactly LIFO*

exercises.lol exercises.com.cn

Queue

A **queue** 队列 works in **FIFO** 先进先出 order (First In, First Out). Operations: **enqueue** 入队 (add to the rear), **dequeue** 出队 (remove from the front), and tests for empty/full. Uses: print spooling, scheduling, breadth-first search, buffering.



Enqueue adds at the rear; dequeue removes from the front

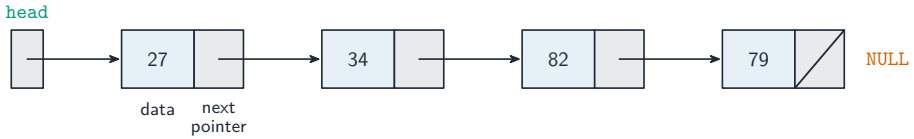
To **describe adding** an item: check that the queue is not full; store the item at the position given by the end-of-queue pointer; increment the end pointer (and the count). To **describe removing**: check that the queue is not empty; read the item at the front pointer; increment the front pointer (and decrement the count). State the convention you use: if the end pointer marks the next free space, front and end pointers being equal means the queue is **empty**; if it marks the last item, equal pointers mean one item. In a linear queue the front pointer only ever moves forward, so cells behind it are wasted; that is what the circular queue below fixes. The two features of a queue to state: items are added at the rear and removed from the front, so the first item added is the first removed.



*A line of people is a queue you can see. You join at the **back** and are served from the **front**, so whoever waited longest is served first – that is exactly FIFO*

Linked list

A **linked list** 链表 stores data as a sequence of **nodes** 节点. Each node holds a value and a **pointer** 指针 to the next node; a head pointer marks the start, and the last node's pointer is a sentinel (e.g. `NULL`). Operations: insert, delete, search, and **traverse** 遍历 (visit each node in order). Its advantage over an array is cheap insertion/deletion (just adjust pointers); its disadvantage is slow random access (you must follow pointers from the head).



A linked list: each node points to the next

Adding a node in order (four marks): traverse the list from the head, following the pointers, until the node before the position is found (the last node whose value is smaller); take a free node and store the new value in it; set the new node's pointer to the address the previous node pointed to; set the previous node's pointer to the new node. If the new value belongs at the front, the head pointer is changed instead. **Deleting** a node: find the node before it, and set that node's pointer to the address the deleted node pointed to, so the list bypasses it; the freed node returns to the free list. Compared with a 1-D array, inserting or deleting in a linked list needs no shifting of the other items, and the list can grow until memory runs out; the cost is the extra pointer stored with every item, and that reaching the nth item means following n pointers, since there is no direct index.

Implementing ADTs using arrays

Stack using an array

Hold items in `Stack[1:MaxSize]` with an integer `Top` (0 when empty).

- `Push(x)`: if `Top = MaxSize` the stack is full (**overflow** 溢出); else `Top ← Top + 1`; `Stack[Top] ← x`.
- `Pop()`: if `Top = 0` the stack is empty (**underflow** 下溢); else return `Stack[Top]` and `Top ← Top - 1`.

Queue using a circular array

A simple queue lets `Front` and `Rear` march off the end, wasting the start. The fix is a **circular array** 循环数组 – when a pointer reaches `MaxSize` it wraps back to 1:

- `Enqueue(x)`: check full; else `Rear ← (Rear MOD MaxSize) + 1`; `Queue[Rear] ← x`.

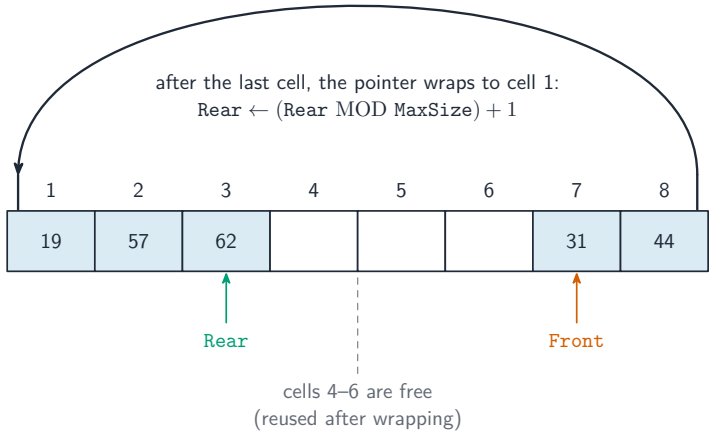
exercises.lol exercises.com.cn

- `Dequeue()`: check empty; else return `Queue[Front]` and `Front ← (Front MOD MaxSize) + 1`.

Track a separate count to tell empty from full.

The algorithm for the end pointer, in words: if the count equals the size, report that the queue is full and stop; otherwise add one to the end pointer; if it is now past the last index, set it to the first index; store the item there and add one to the count. The declarations that a five-mark "describe the declaration and initialisation" answer lists: the array with its size and element type; a front pointer and an end pointer, both initialised to the first index (or the front to the first index and the end to the next free space); and a count of items, initialised to 0.

For example, with `MaxSize = 6`: if `Rear = 5`, then `(5 MOD 6) + 1 = 6`, so the next item goes in cell 6; if `Rear = 6`, then `(6 MOD 6) + 1 = 1`, so the pointer **wraps back** to cell 1.



A circular queue wraps the pointers back to the start of the array

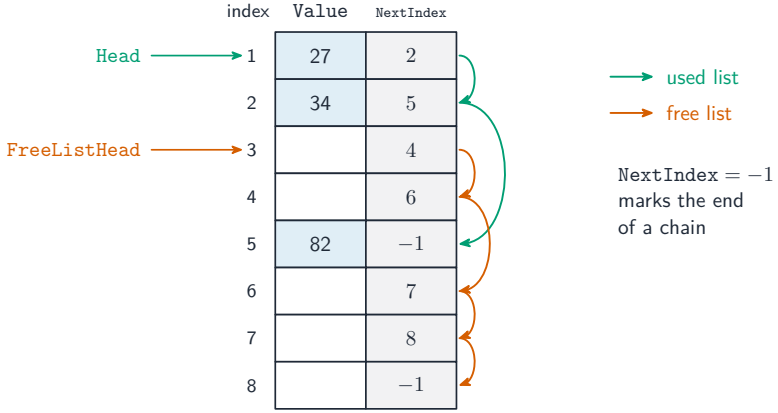
Linked list using an array

Use an array of records, each with a `NextIndex` index:

```
TYPE TNode
    DECLARE Value : INTEGER
    DECLARE NextIndex : INTEGER     // index of the next node, or -1 for
    ↪  end
ENDTYPE

DECLARE Nodes : ARRAY[1:MaxSize] OF TNode
DECLARE Head : INTEGER         // index of first node, -1 if empty
DECLARE FreeListHead : INTEGER // first available free node
```

A **free list** 空闲列表 chains the unused slots, just as the data list chains its used ones. To insert: take a slot from `FreeListHead`, set the new node's value and `NextIndex`, and update the previous node's `NextIndex` (or `Head`). To delete: unlink the node and return its slot to the free list. This gives the flexibility of a linked structure with the static allocation of an array.



A linked list stored in an array: a data array and a pointer array

> **Worked example.** A linked list is held in a `Data` array and a `Pointer` array, with `Start` pointing to index 1. The list is $1 \to 3 \to 4$ (index 1 holds `D40`, index 3 holds `D32`, index 4 holds `D11`, whose pointer is $\emptyset$); the free list starts at index 2 and continues $2 \to 5$. Insert `D6` between `D32` and `D11`.

Take the first free node, index 2, and set `FreeStart` to its pointer, 5; store `D6` in `Data[2]`; set `Pointer[2]` to the value `Pointer[3]` held, which is 4; set `Pointer[3]` to 2. The list reads $1 \to 3 \to 2 \to 4$ and the free list is $5 \to \emptyset$. The answer to "how can the linked list be implemented" is exactly these parts: an array (or array of records) for the data, a parallel array for the pointers holding indices, a start pointer, a free-list pointer and a null value such as -1 for the end.

> **Worked example.** A circular queue is held in an array of size 5 (indices 0 to 4) with `Front = 3`, `Rear = 3` and one item stored. Two items are added, then two are removed. Where are the pointers, and why use a *circular* queue at all? Every move uses `(pointer + 1) MOD size`, so the pointers **wrap**. Adding twice moves `Rear`: $3 \to 4$, then $4 \to 0$ (because $(4+1) \bmod 5 = 0$), so `Rear = 0` and three items are stored. Removing twice moves `Front` the same way: $3 \to 4$, then $4 \to 0$, leaving `Front = 0` and one item. The wrap is the whole point: in a **linear** array queue the pointers march to the end and the freed space at the front is wasted even when the queue is empty. Remember a queue removes at the **Front** and adds at the **Rear** - a stack uses one pointer for both.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly.

Term	Definition
record	a data structure that holds a set of data items (fields) of different data types under one identifier
array	a data structure that holds a fixed number of elements of the same data type under one identifier, each accessed by an index
index	the number that identifies one element of an array
upper bound, lower bound	the largest and smallest valid index of an array
text file	a file that stores data as lines of characters, which a program reads and writes one line at a time
abstract data type	a collection of data together with a set of operations on that data
stack	a list in which items are added to and removed from the same end, the top, so the last item added is the first removed (LIFO)
queue	a list in which items are added at the rear and removed from the front, so the first item added is the first removed (FIFO)
linked list	a list in which each node holds a data item and a pointer to the next node, with a start pointer to the first node

exercises.lol exercises.com.cn

Term	Definition
pointer	a variable that holds the address (or index) of a node or of a position in a structure
linear search	checking each element in turn from the first until the target is found or the end is reached
bubble sort	repeated passes through the array comparing adjacent pairs and swapping those out of order, until a pass makes no swaps

Exam tips

- Choose the right **data structure** and justify it (a record for mixed fields, a 2-D array for a grid).
- Know how to implement a **stack, queue and linked list** with an array and pointers (top; front/rear; next).
- Distinguish an **ADT** (its behaviour) from its implementation (array plus pointers).

Common mistakes

- A record declaration without `ENDTYPE`, or fields without types. Every field is a `DECLARE` line with a type.
- Reading past the end of a file, or writing with `WRITE` when the file must keep its contents. Test `EOF` before each read; use `APPEND` to add.
- Writing a number to a text file without converting it. A file holds strings: `NUM_TO_STR` out, `STR_TO_NUM` back.
- Forgetting the checks. `Push` and enqueue test for full first; `Pop` and dequeue test for empty first, and the answer says so.
- Losing the rest of the list when inserting a node. Set the new node's pointer to the old next node before changing the previous node's pointer.
- A linear search that never says "not found". Initialise the position to -1 and test it after the loop.

exercises.lol exercises.com.cn

Programming

A-Level Computer Science

Programming basics

```
1 usage
public Hotel (Integer capacity) { this.capacity = capacity; }
1 usage
public synchronized boolean checkIn(AccommodationRequest request) {
    if (guests.size() < capacity) {
        guests.add(request);
        return true;
    }
    else {
        return false;
    }
}
```

Programming turns a design into instructions written as code

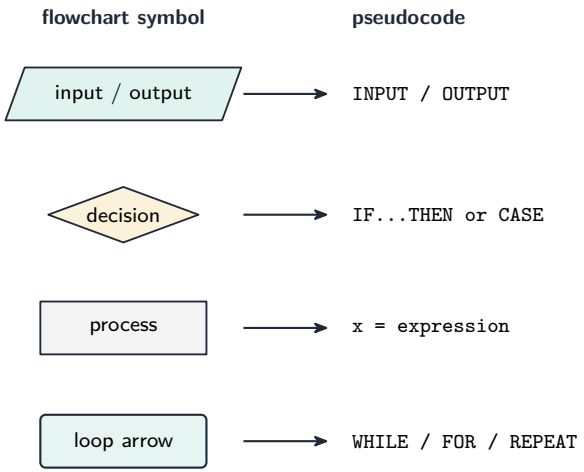


A programmer writes the code and tests it as they go

From design to code

You should be able to turn a design – a **flowchart** 流程图 (**program flowchart**) or **structured English** 结构化英语 – into **pseudocode** 伪代码, and then into a real language:

1. find the **variables** 变量 and their **data types** 数据类型.
2. turn input/output boxes into `INPUT` / `OUTPUT`.
3. turn decision diamonds into `IF...ELSE...ENDIF` (or `CASE`).
4. turn loop arrows into `WHILE`, `REPEAT...UNTIL`, or `FOR`.
5. turn process boxes into assignments or calculations.
6. check by tracing a small input.



Each flowchart symbol becomes a pseudocode keyword

Constants and variables

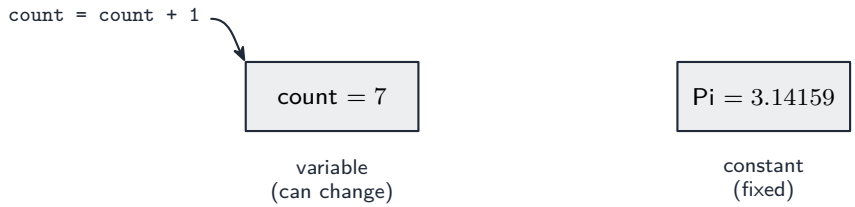
A **constant** 常量 holds a value that **never changes**; a variable holds one that may change. Declare them with a type:



A variable is a named store whose value can change

```
CONSTANT Pi = 3.14159
DECLARE Radius : REAL
DECLARE Area : REAL

Radius ← 5
Area ← Pi * Radius * Radius
```

A variable's value can change; a constant stays fixed

Use constants for fixed values that recur (`Pi`, `MaxScore`); they make code clearer and easy to change in one place.

In the exam, a constant is the answer to "identify a more appropriate way of representing" a fixed value, such as a tax rate or a maximum score, that appears at several places in the pseudocode. The benefits the scheme lists: the value is set once and **cannot be changed accidentally** by the program; a change is made **in one place** and reaches every statement that uses it; the identifier gives the value a **meaning** (`MaxScore` rather than `100`), so the code is easier to read and to check; and there is less risk of a typing error in a long value such as `3.14159`. A "state a value that could be replaced by a constant" question wants the literal from the pseudocode (`0.2`, `40`), not a new name.

Every variable is declared once, with an **identifier** 标识符 (its name) and a data type, before it is used. The six types in the 9618 pseudocode guide:

Type	Holds	Written in the code as	Typical use
`INTEGER`	whole numbers	`42`, `-3`	a count, an array index, a loop counter
`REAL`	numbers with a fractional part	`3.75`	a price, an average
`CHAR`	one character	`'A'` (single quotes)	a grade letter, a menu key
`STRING`	a sequence of characters	`"Hello"` (double quotes)	a name, a postcode
`BOOLEAN`	`TRUE` or `FALSE`	`TRUE`	a flag such as `Found`
`DATE`	a calendar date	`12/05/2026`	a date of birth

A "give the appropriate data type" question is answered from how the variable is **used** in the pseudocode: a value with a decimal point is `REAL`; something set to `TRUE` or `FALSE` is `BOOLEAN`; a value in single quotes is `CHAR`; a value used as an array index, or with `DIV` and `MOD`, is `INTEGER`. Write the type in capitals, spelled as the guide spells

exercises.lol exercises.com.cn

it.

Worked example. State the appropriate data type for each variable.

```
Found ← FALSE
Initial ← 'K'
Price ← 12.99
Count ← Count + 1
Name ← "Li Wei"
```

`Found` is `BOOLEAN` (it holds `FALSE`); `Initial` is `CHAR` (one character in single quotes); `Price` is `REAL` (a decimal value); `Count` is `INTEGER` (a counter that goes up by one); `Name` is `STRING` (text in double quotes).

Assignment and expressions

Use `←` for **assignment** 赋值:

```
Total ← Total + 1
Average ← Sum / Count
```

Expressions use **operators** 运算符:

- arithmetic `+ - * /`, plus `DIV` (integer division) and `MOD` (remainder): `7 DIV 2 = 3`; `7 MOD 2 = 1`.
- comparisons `=`, `<>`, `<`, `>`, `<=`, `>=`.
- logic `AND`, `OR`, `NOT`.

Precedence 优先级 (highest to lowest): `NOT` → `* / DIV MOD` → `+ -` → comparisons → `AND` → `OR`. Use brackets when unsure.

Input and output

```
OUTPUT "Enter your name:"
INPUT Name
OUTPUT "Hello, ", Name
```

Built-in functions and library routines

Many tasks have ready-made **library routines** 库例程, so you need not write them. The Paper 2 **insert** 附页 lists the ones you may use, with their exact names, parameters and return types; any other function a question needs is given in the question. The names below are the insert's names. `VAL` and `STR` are IGCSE names and appear in neither 9618 document, so they earn nothing. `UCASE` and `LCASE` are a different case: they **are** 9618, defined in the Pseudocode Guide, but they take a single `CHAR`, and the insert does not list them at all – for a whole string on Paper 2 the routine is `TO_UPPER`.

A **program library** 程序库 holds routines that have already been written, compiled and tested; a program calls them instead of writing its own. The benefits the scheme accepts, for a "state three benefits" question: the routines are **already tested**, so they are less likely to contain errors; they **save development time**; they may do things the programmer could not write (complex statistics, graphics); they are written by experts and **reused** across many programs; and a routine with a fixed interface can be called from anywhere in the program.

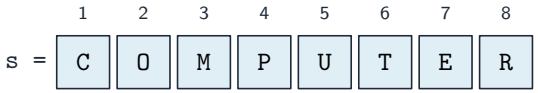
Routine	Returns	Example
`LENGTH(s)`	the number of characters in `s`	`LENGTH("Hello") = 5`
`LEFT(s, n)` / `RIGHT(s, n)`	the first / last `n` characters	`RIGHT("Hello", 2) = "lo"`
`MID(s, start, n)`	`n` characters from position `start` (positions count from 1)	`MID("Hello", 2, 3) = "ell"`
`TO_UPPER(s)` / `TO_LOWER(s)`	`s` in capitals / in small letters	`TO_UPPER("ab1") = "AB1"`
`NUM_TO_STR(x)` / `STR_TO_NUM(s)`	a number as a string / a string as a number	`STR_TO_NUM("3.5") = 3.5`
`IS_NUM(s)`	`TRUE` if `s` is a valid number	`IS_NUM("12a") = FALSE`
`ASC(c)` / `CHR(n)`	the character code of `c` / the character with code `n`	`ASC('A') = 65`, `CHR(66) = 'B'`
`INT(x)`	the whole-number part of `x`	`INT(7.9) = 7`
`RAND(n)`	a random real number from 0 up to, but not including, `n`	`INT(RAND(6)) + 1` is a dice roll
`DAY(d)`, `MONTH(d)`, `YEAR(d)`	the parts of a `DATE`	`YEAR(TODAY())`
`DAYINDEX(d)`, `SETDATE(d, m, y)`, `TODAY()`	the day of the week (1 = Sunday); a date built from three integers; today's date	
`EOF(f)`	`TRUE` when the file `f` has no more lines to read	`WHILE NOT EOF("data.txt")`

Strings are joined with `&` (**concatenation** 连接): `"A" & "BC"` is `"ABC"`. Use the exact names from the insert, with the parameters in its order.

Dates and random numbers come up as one-line statements. `SETDATE(17, 11, 2007)` builds 17 November 2007; `12 - MONTH(MyDOB)` is the number of months from the month of birth to the end of the year; `IF DAYINDEX(MyDOB) = 5 THEN` tests for a

exercises.lol exercises.com.cn

Thursday, because Sunday is day 1. `RAND(n)` returns a real number from `0` up to, but not including, `n`, so a random **integer** from `Low` to `High` inclusive is `INT(RAND(High - Low + 1)) + Low`: `INT(RAND(21)) - 10` gives a value from `-10` to `10`.




```
LENGTH(s) = 8              RIGHT(s, 2) = "ER"
LEFT(s, 3) = "COM"         TO_UPPER(s) = "COMPUTER"
MID(s, 4, 3) = "PUT"       TO_LOWER(s) = "computer"
```

The common string routines acting on `s = "COMPUTER"` (positions 1–8)

Worked example. Evaluate each expression, given `Word ← "Program"`, `Code ← 'Q'` and `N ← 7`.

Expression	Value	Why
`LENGTH(Word)`	`7`	seven characters
`MID(Word, 4, 2)`	`"gr"`	two characters, starting at position 4
`LEFT(Word, 3) & "!"`	`"Pro!"`	joined with `&`
`TO_UPPER(RIGHT(Word, 2))`	`"AM"`	the inner function runs first
`ASC(Code) - ASC('A')`	`16`	`'Q'` is 81 and `'A'` is 65
`N DIV 2 + N MOD 2`	`4`	`3 + 1`
`NUM_TO_STR(N) & "th"`	`"7th"`	the number becomes a string first
`INT(N / 2)`	`3`	`3.5` cut to its whole part

Work from the inside out, and keep the quotes: `"7"` is a string and `7` is a number.

Worked example. Each statement may contain an error in its use of a function or operator. Describe the error, or write NO ERROR. (Assume every variable has the correct type.)

exercises.lol exercises.com.cn

Statement	Error
`Result ← 2 & 4`	`&` joins strings; `2` and `4` are integers, so `+` is needed
`SubString ← MID("pseudocode", 4, 1)`	NO ERROR: one character from position 4, `"u"`
`IF x = 3 OR 4 THEN`	`OR` needs a Boolean on each side: `IF x = 3 OR x = 4 THEN`
`Result ← Status AND INT(x / 2)`	`AND` needs two Booleans; `INT(x / 2)` is an integer
`Message ← "Done" + LENGTH(MyString)`	`+` cannot add a string to an integer: `"Done" & NUM_TO_STR(LENGTH(MyString))`

Every operator works on particular types: `&` on strings, `+ - * / DIV MOD` on numbers, `AND OR NOT` on Booleans, and `= <>` on two values of the **same** type. An "evaluate each expression, or write ERROR" table is marked the same way: `LENGTH(42)` and `"A" + 1` are ERROR, because the type does not match the function or the operator.

Worked example. With `Points ← 100`, `Active ← TRUE` and `Exempt ← FALSE`, evaluate each expression.

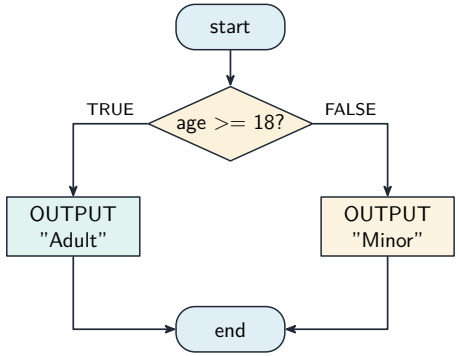
Expression	Value	Why
`(Points > 99) OR Active`	`TRUE`	both sides are true; one would do
`(Points MOD 2 = 0) OR Exempt`	`TRUE`	`100 MOD 2` is `0`
`(Points <= 75) AND (Active OR Exempt)`	`FALSE`	the first side is false, and `AND` needs both
`(Active OR NOT Active) AND NOT Exempt`	`TRUE`	`Active OR NOT Active` is always true

The last expression simplifies: `X OR NOT X` is `TRUE` whatever `X` is, so the whole expression is just `NOT Exempt`. Evaluate the brackets first, then `NOT`, then `AND`, then `OR`.

Selection

Selection 选择 chooses which steps run.

```
IF age >= 18 THEN
    OUTPUT "Adult"
ELSE
    OUTPUT "Minor"
ENDIF
```



An IF…ELSE tests the condition once, then runs exactly one branch

For more than two cases you can use a **nested** 嵌套 IF, but deep nesting is hard to read – a `CASE` is cleaner when testing one value against several options:

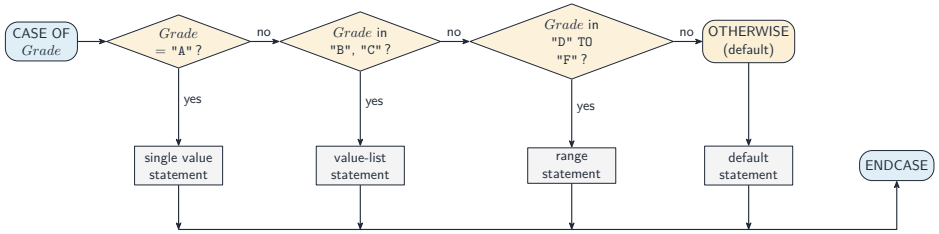
```
CASE OF Grade
    "A": OUTPUT "Excellent"
    "B": OUTPUT "Good"
    OTHERWISE: OUTPUT "Try again"
ENDCASE
```

Cambridge `CASE` allows single values, value lists (`1, 2, 3:`), and ranges (`1 TO 5:`).

A nested IF is an IF inside a branch of another IF. Each `IF` needs its own `ENDIF`, and the examiner checks that every construct is closed:

```
IF Mark >= 50 THEN
    IF Mark >= 80 THEN
        OUTPUT "Distinction"
    ELSE
        OUTPUT "Pass"
    ENDIF
ELSE
    OUTPUT "Fail"
ENDIF
```

Boundaries are where marks are lost. "A mark of 50 or more passes" is `Mark >= 50`, not `Mark > 50`; the last `CASE` branch, for "anything else", is written `OTHERWISE`, not a condition such as `> 200`. A wrong comparison here is a **logic error** 逻辑错误: the program runs, but gives the wrong output for some inputs – and a trace table with a boundary value such as `50` is how you find it.



A CASE statement runs the branch that matches the value

Worked example. Rewrite this with the same functionality, without using a CASE structure.

```
CASE OF MySwitch
    1: ThisChar ← 'a'
    2: ThisChar ← 'y'
    3: ThisChar ← '7'
    OTHERWISE: ThisChar ← '*'
ENDCASE
```

Each value becomes a branch of a chain of IFs, and `OTHERWISE` becomes the last `ELSE`:

```
IF MySwitch = 1 THEN
    ThisChar ← 'a'
ELSE
    IF MySwitch = 2 THEN
        ThisChar ← 'y'
    ELSE
        IF MySwitch = 3 THEN
            ThisChar ← '7'
        ELSE
            ThisChar ← '*'
        ENDIF
    ENDIF
ENDIF
```

Two clauses that assign the same value are merged into one clause with a value list: `1, 2: ThisChar ← 'a'`. The guards are tested **in order**: with ranges such as `1 TO 50:` followed by `40 TO 60:`, a value of `45` takes the first branch that matches, so an assignment in a later branch may **never be performed** – and when the earlier branches already cover every possible value, the `OTHERWISE` branch is never reached either.

Going the other way, nested IFs that test several Booleans are clearer as one condition per outcome: `IF A AND B AND C THEN CALL Sub1()`, then `IF A AND B AND NOT C THEN CALL Sub2()`, and so on. Joining tests with `AND` and `OR` removes the nesting, and `IF A THEN` is accepted in place of `IF A = TRUE THEN`.

exercises.lol exercises.com.cn

Iteration

Iteration 迭代 repeats a block. Three loops differ in how many times the body runs.

Count-controlled (FOR) loop

A **count-controlled loop** 计数循环 – use it when you know how many times to repeat:

```
FOR i ← 1 TO 10
    OUTPUT i
NEXT i
```

A `STEP` can change the count (e.g. `FOR i ← 10 TO 1 STEP -1`). Best for a fixed number of repeats or processing each element of an **array** 数组.

Pre-condition (WHILE) loop

A **pre-condition loop** 前测循环 tests the condition **before** each pass, so it may run **zero** times:

```
WHILE total < 100 DO
    INPUT n
    total ← total + n
ENDWHILE
```

Post-condition (REPEAT...UNTIL) loop

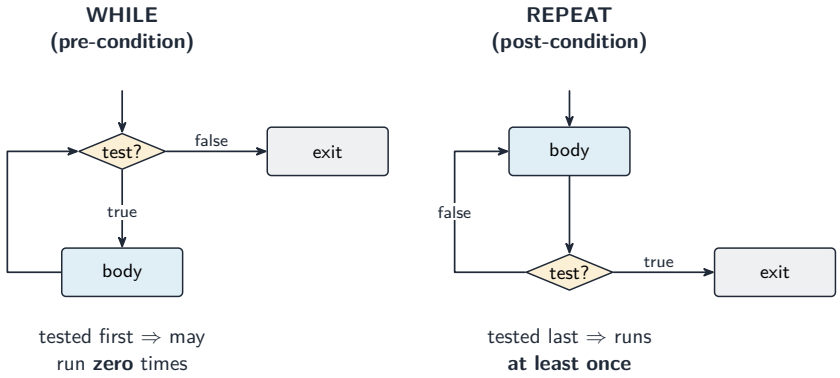
A **post-condition loop** 后测循环 tests the condition **after** each pass, so it always runs **at least once**:

```
REPEAT
    INPUT password
UNTIL password = correctPassword
```

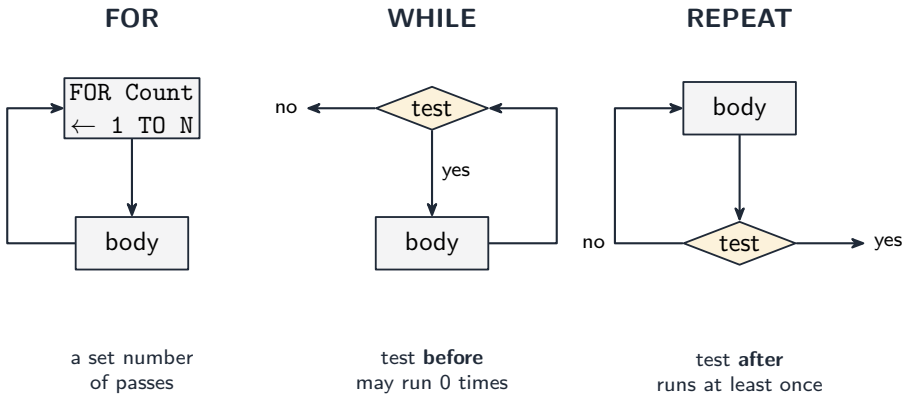
Choosing the right loop

- count known up front → **FOR**.
- may need zero passes → **WHILE**.
- always at least one pass → **REPEAT...UNTIL**.

exercises.lol exercises.com.cn



A pre-condition (WHILE) loop tests before the body, so it may run zero times; a post-condition (REPEAT) loop tests after, so it runs at least once



The three loops differ in where the condition is tested – before the body (WHILE), after it (REPEAT), or a set number of times (FOR)

Justify your choice by whether the count is known and whether the body must run at least once. A typical question gives a scenario ("ask for a password until correct, but always ask at least once") and asks which loop fits.

The two marks are for the **name** of the loop and the **reason**, in the scheme's words: *count-controlled, because the number of iterations is known before the loop starts*; *post-condition, because the loop body must be executed at least once*; *pre-condition, because the loop may not need to execute at all.* A loop over the four elements of an array that has been written as a `WHILE` with a counter is "not the most appropriate": the count, four, is known, so a `FOR` loop fits.

Worked example. Which loop suits each task? (a) print the 12 times table; (b) keep reading numbers until the user enters 0; (c) ask for a password until it is correct. Choose by asking **how many times** the body runs and **when the test happens**. (a) The count is known in advance (12), so use a **FOR** loop. (b) The count is unknown, and the very first input might already be 0 - so the test must come **before** the body: a **WHILE** loop, which runs **zero** or more times. (c) The count is unknown, but you must always ask **at least once** before there is anything to test - so the test comes **after** the body: a **REPEAT...UNTIL**, which runs **one** or more times. The deciding question is whether the body must run at least once: WHILE may run zero times, REPEAT always runs once.

Dry running with a trace table

A **trace table** 跟踪表 records the value of each variable as you **dry run** 手工跟踪 (work through by hand) an algorithm. It is how you test a loop on paper, and a six-mark question on most Paper 2s.

count	total	output
1	5	
2	12	
3	20	20

A trace table records each variable's value as the program runs

```
total <- 0
count <- 0
FOR each value (4, 7, 5)
  total <- total + value
  count <- count + 1
```

total keeps a running sum;
count adds 1 each pass -
a **trace table** follows both

value	total	count
start	0	0
4	4	1
7	11	2
5	16	3

A trace table follows a totalling and counting loop pass by pass

exercises.lol exercises.com.cn

```
DECLARE Count, Total : INTEGER
Count ← 1
Total ← 0
WHILE Total < 10
    Total ← Total + Count * 2
    Count ← Count + 1
ENDWHILE
OUTPUT Count, Total
```

Count	Total	Total < 10	OUTPUT
1	0	TRUE	
2	2	TRUE	
3	6	TRUE	
4	12	FALSE	4, 12

Rules that earn the marks: one column per variable, in the order the question gives; write a value only when it **changes**; start a new row each time the loop repeats; evaluate the condition with the current values, and stop the moment it is `FALSE`; put the output in its own column, exactly as it would appear. Trace the algorithm as written, not the one you think was intended – if it never stops, say so.

Worked example. Which constructs does each line use – selection, iteration or a subroutine call?

Pseudocode	Selection	Iteration	Subroutine
IF Ready = TRUE THEN			

```
CALL Start()
```

ENDIF | `FOR I ← 1 TO 20 ... NEXT I` | | yes | | | `WHILE NOT IsFull() ... ENDWHILE` | | yes | yes | | `CASE OF Key ... OTHERWISE ... ENDCASE` | yes | | |

`IF` and `CASE` are selection; `FOR`, `WHILE` and `REPEAT` are iteration; a name followed by brackets – `Start()`, `IsFull()` – is a call to a procedure or a function, wherever it appears, including inside a condition.

Procedures and functions

Structured programming 结构化编程 builds a program from small named **subroutines** 子程序, each with one job.

Procedure

A **procedure** 过程 is a named block that **does an action**; it may take **parameters** 参数 but does **not return a value**.

```
PROCEDURE Greet(name : STRING)
    OUTPUT "Hello, ", name
ENDPROCEDURE

CALL Greet("Ada")
```

Function

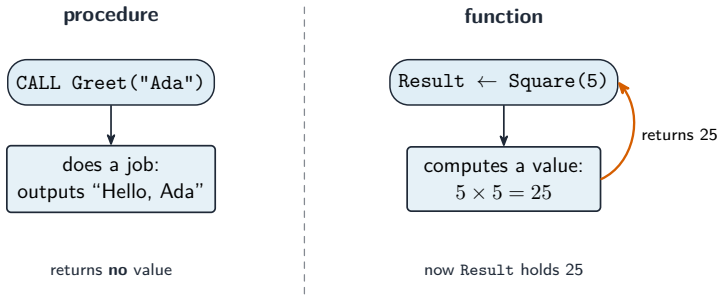
A **function** 函数 is like a procedure but it **returns a value** that becomes part of an expression.

```
FUNCTION Square(x : INTEGER) RETURNS INTEGER
    RETURN x * x
ENDFUNCTION

result ← Square(5) + 1      // result = 26
```

Use a **procedure** when the subroutine performs an action; use a **function** when it computes a value for the caller.

The syllabus asks *where in the construction of an algorithm* each is appropriate. A procedure is appropriate where the same group of steps is needed at several points (validate an input, print a menu, swap two values): the steps are written once and `CALL`ed by name. A function is appropriate where a single value must be calculated and then used in an expression – a total, a `TRUE`/`FALSE` result, the larger of two numbers – because the **return value** 返回值 replaces the call: `IF IsValid(Code) THEN`.



A procedure does an action and returns nothing; a function returns a value you use in an expression

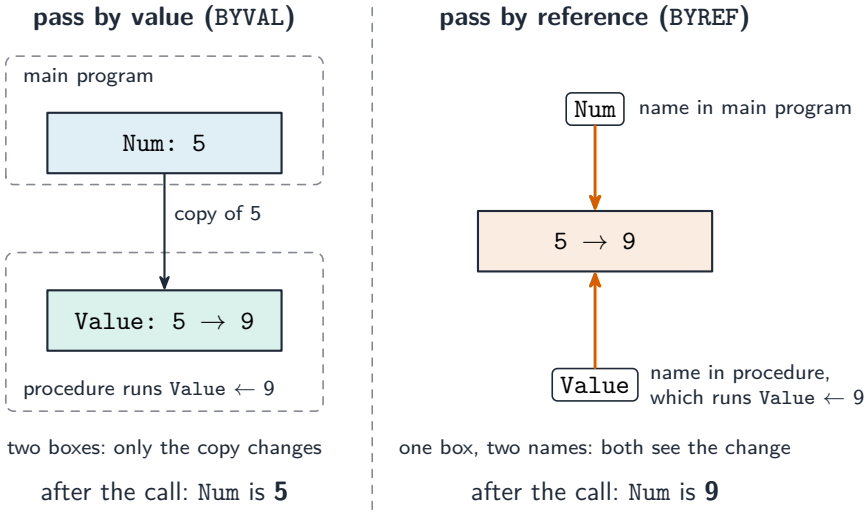
exercises.lol exercises.com.cn

Parameters

A **parameter** is a variable a subroutine declares to receive input; the values the caller supplies are **arguments** 实参. Two ways to pass them:

- **pass by value** 传值 – the routine gets a **copy**; changes inside it do not affect the caller. Use for inputs it only reads.
- **pass by reference** 传引用 – the routine gets a **reference** to the caller's variable; changes **do** affect the caller. Use when it must update a parameter.

The figure shows one call where the procedure runs `Value ← 9` while the caller's `Num` holds 5.



Pass by value changes only a copy; pass by reference changes the caller's own variable

```
PROCEDURE Swap(BYREF a : INTEGER, BYREF b : INTEGER)
    DECLARE temp : INTEGER
    temp ← a
    a ← b
    b ← temp
ENDPROCEDURE
```

Cambridge pseudocode writes the mode in the header, `BYVAL` or `BYREF`, before each parameter. **If neither is written, `BYVAL` is assumed!** A mode also carries on to the following parameters until the other one is written: in `Swap(BYREF a : INTEGER, b : INTEGER)`, `b` is `BYREF` too. So a routine that must change the caller's variable – `Swap`, or a procedure that updates a running total – needs `BYREF` in its header.

Worked example. What is output?

```
PROCEDURE Adjust(BYREF X : INTEGER, BYVAL Y : INTEGER)
    X ← X + Y
    Y ← Y * 2
ENDPROCEDURE

A ← 5
B ← 3
CALL Adjust(A, B)
OUTPUT A, B
```

`X` is a reference to `A`, so `A` becomes `8`. `Y` is a copy of `B`, so doubling `Y` leaves `B` at `3`. The output is `8, 3`. Had the header said `BYVAL X`, `A` would still be `5`.

Local vs global variables

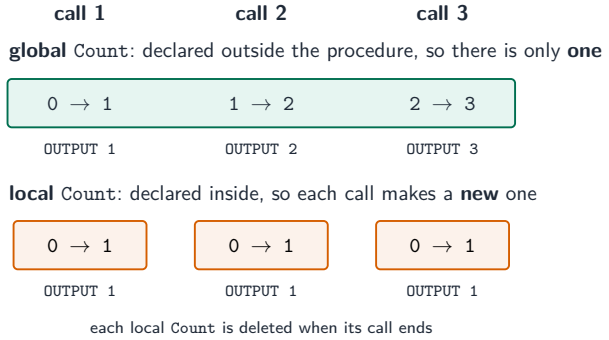
A **local variable** 局部变量 is declared inside a subroutine and exists only while it runs. A **global variable** 全局变量 is declared outside and is visible everywhere. Prefer locals and parameters – heavy use of globals makes code hard to follow and test. (The region where a name is visible is its **scope** 作用域.)

The one-line difference: a global variable can be accessed from anywhere in the program, a local variable only inside the subroutine that declares it. Benefits of local variables the scheme accepts: the same identifier can be used in another subroutine without a clash; the value cannot be changed accidentally by other parts of the program; the memory is released when the subroutine ends; and the subroutine is **self-contained**, so it can be tested on its own and reused in another program.

A local variable is created **each time** the subroutine is called and deleted when the call ends, so it cannot carry a value from one call to the next. A procedure that builds up a string over repeated calls therefore needs that string to be global (or passed `BYREF`). If `MyString` is changed from a global to a local declared inside `MyOutput()`, every call starts with a new, empty `MyString`, the text added by earlier calls is lost, and the procedure "does not work as expected".

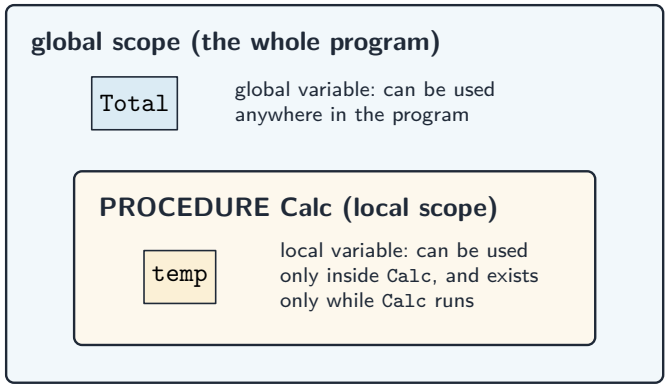
The figure shows the same effect with a counter. A procedure runs `Count ← Count + 1`, then `OUTPUT Count`, and is called three times. A global `Count` outputs 1, 2, 3. A local `Count` is made new and set to 0 in every call, so it outputs 1, 1, 1.

exercises.lol exercises.com.cn



A local variable is made new in every call; only a global variable (or a BYREF parameter) keeps a value between calls

The last figure shows scope: where each variable can be used.



A global variable can be used anywhere; a local variable only inside its own procedure

When to use a subroutine

Use a subroutine when:

- the same logic appears in **more than one place** – write it once, call it many times.
- a block has a **clear named purpose** – the name documents what it does.
- the program is **complex** – break it into parts (**decomposition** 分解).
- you want to **test a piece in isolation**.

Don't make them so tiny that the call costs more than the work inside.

Terminology

- **definition** – the `PROCEDURE ... ENDPROCEDURE` (or function) block.
- **call** – where it is invoked. **argument** – a value passed in. **parameter** – the variable that receives it.
- **return value** – what a function passes back.
- **procedure/function header** – the first line giving the name and parameters (`PROCEDURE Name(params)` or `FUNCTION Name(params) RETURNS type`).
- **procedure/function interface / signature** 签名 – name + parameters + return type: what a caller must know to use it.

Worked example. Describe each term used in the header `FUNCTION Pass2(Count : INTEGER) RETURNS BOOLEAN`.

Term	Meaning
`FUNCTION`	a subroutine that returns a value
`Pass2`	the identifier used to call it
`Count`	the parameter: the identifier that receives the argument passed in
`INTEGER`	the data type of the parameter
`RETURNS BOOLEAN`	the data type of the value the function returns

The two identifiers in `PROCEDURE MyProc(Count : INTEGER, Message : STRING)` are **parameters**: they receive the values passed in when the procedure is called, and are used inside it like local variables.

To convert a procedure into a function: change `PROCEDURE` to `FUNCTION` and add `RETURNS <type>`; replace the `OUTPUT` (or the `BYREF` parameter that carried the result out) with a `RETURN` statement; and change every call so that the returned value is used, `Result ← Unpack(Text)` instead of `CALL Unpack(Text, Result)`. For a "write the header" question, write the whole line: `FUNCTION Calculate(Expression : STRING) RETURNS INTEGER`. An array parameter is passed **by reference**, so a procedure that writes into an array changes the caller's array.

When a program gains a new module, the interface is what is agreed first: the name, the parameters (how many, in what order, of what type) and the return type, plus any global data the module reads or writes. A module that sends a reminder before a due date needs the record (or its index) as a parameter and returns nothing, so it is a procedure; the main program calls it once per record.

exercises.lol exercises.com.cn

Writing a module for Paper 2

Half of Paper 2 is "write pseudocode for module X". The scheme awards a mark per feature, so a module that is not finished still scores for every correct part. The parts the examiner looks for:

```
FUNCTION CountAbove(BYVAL Limit : INTEGER) RETURNS INTEGER
    DECLARE Index, Count : INTEGER
    Count ← 0
    FOR Index ← 1 TO 50
        IF Score[Index] > Limit THEN
            Count ← Count + 1
        ENDIF
    NEXT Index
    RETURN Count
ENDFUNCTION
```

header: name, parameters, return type
local variables declared with a type
total initialised before the loop
loop over every element
the condition, with the right boundary
the update, inside the IF
every construct closed
the return value, after the loop

Each part of a module answer carries its own mark, so write all of them even when one is uncertain

1. **The header**, as the question describes it: `PROCEDURE Name(Param : TYPE)` or `FUNCTION Name(Param : TYPE) RETURNS TYPE`, with `BYREF` where the routine must change the argument.
2. **Local declarations**: `DECLARE` every local variable with its type, and initialise counters and totals (`Count ← 0`).
3. **The loop** that visits every element: `FOR Index ← 1 TO 50` for an array whose size is given; `WHILE NOT EOF(...)` for a file.
4. **The condition**, with the right comparison and boundary, on the right item: `IF Score[Index] > Limit THEN`.
5. **The update** inside the branch: the count increased, the value stored, or the message output.
6. **The end**: `RETURN` once, after the loop, in a function; `ENDFUNCTION` or `ENDPROCEDURE`; and every `IF`, `FOR` and `WHILE` closed.

Worked example. A global array `Score : ARRAY[1:50] OF INTEGER` holds test scores. Write a function `CountAbove(Limit : INTEGER)` that returns how many scores are greater than `Limit`.

```
FUNCTION CountAbove(BYVAL Limit : INTEGER) RETURNS INTEGER
    DECLARE Index, Count : INTEGER
    Count ← 0
    FOR Index ← 1 TO 50
        IF Score[Index] > Limit THEN
            Count ← Count + 1
        ENDIF
    NEXT Index
    RETURN Count
ENDFUNCTION
```

Marks: the header with its parameter and `RETURNS INTEGER`; `Count` declared and set to `0`; a loop over all 50 elements; the comparison `> Limit` (not `>=`); the count updated inside the `IF`; `RETURN Count` after the loop. The main program uses the return value in an expression or an output: `OUTPUT "Above 70: ", CountAbove(70)`.

Worked example. Write a function `IsValid(Code : STRING)` that returns `TRUE` when `Code` is two capital letters followed by four digits – the **format** 格式 `AB1234` – and `FALSE` otherwise.

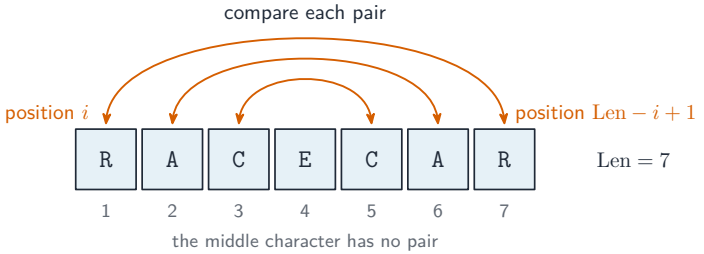
```
FUNCTION IsValid(BYVAL Code : STRING) RETURNS BOOLEAN
    DECLARE Index : INTEGER
    DECLARE Ch : STRING
    IF LENGTH(Code) <> 6 THEN
        RETURN FALSE
    ENDIF
    FOR Index ← 1 TO 6
        Ch ← MID(Code, Index, 1)
        IF Index <= 2 THEN
            IF Ch < "A" OR Ch > "Z" THEN
                RETURN FALSE
            ENDIF
        ELSE
            IF Ch < "0" OR Ch > "9" THEN
                RETURN FALSE
            ENDIF
        ENDIF
    NEXT Index
    RETURN TRUE
ENDFUNCTION
```

The length check comes first, so `MID` is never asked for a position that does not exist. **Validation** 验证 like this returns a `BOOLEAN` so the caller can write `IF IsValid(Entry) THEN ... ELSE OUTPUT "Invalid code" ENDIF`: a message to the user is output by the caller, not by the function – a function calculates, a procedure acts.

exercises.lol exercises.com.cn

Worked example. Write a function `IsPalindrome(Word : STRING)` that returns `TRUE` when `Word` reads the same backwards, such as `"RACECAR"`.

Compare the characters from the two ends, moving inwards: position `Index` is paired with position `Len - Index + 1`, and only the first half needs testing.



A palindrome check pairs position `i` with position `Len - i + 1` and stops at the middle

```
FUNCTION IsPalindrome(BYVAL Word : STRING) RETURNS BOOLEAN
    DECLARE Len, Index : INTEGER
    Len ← LENGTH(Word)
    FOR Index ← 1 TO Len DIV 2
        IF MID(Word, Index, 1) <> MID(Word, Len - Index + 1, 1) THEN
            RETURN FALSE
        ENDIF
    NEXT Index
    RETURN TRUE
ENDFUNCTION
```

The same three tools – a `FOR` over the positions, `MID(s, i, 1)` to read one character, and `&` to build a new string – answer most string modules on Paper 2: counting how often a character occurs (`IF MID(s, i, 1) = Ch THEN Count ← Count + 1`), replacing every instance of a character (add either `NewChar` or the original character to `NewString` at each position), hiding all but the last four digits of a card number (add `'*'` for every position up to `Len - 4`), or writing your own `MID()` by joining the characters from `Start` to `Start + Length - 1`. Asking `MID` for a position past the end of the string is a **run-time error**, so check `LENGTH` first.

Files. Values in variables disappear when the program ends, so a module that must keep data for the next run writes it to a file: `OPENFILE "scores.txt" FOR WRITE`, one `WRITEFILE "scores.txt", NUM_TO_STR(Score[Index])` per line inside the loop, and `CLOSEFILE "scores.txt"` once, after the loop; reading back uses `FOR READ`, `READFILE`

and `WHILE NOT EOF("scores.txt")`. Topic 10 has the full file section; here the marks are for opening in the right mode, the read or write inside the loop, and closing once after it.

Writing efficient pseudocode

Three features that make pseudocode easier to understand – the answer to a "state three features" question – are **meaningful identifiers** (`Total`, not `t`), **indentation** of the statements inside each construct, and **comments** (`// ...`) that explain the purpose; keywords in capitals, one statement per line and blank lines between sections are also accepted. Efficient pseudocode goes further:

- **move invariants out of loops** – if a value (an **invariant** 不变量) does not change with the loop counter, compute it once before the loop.
- **exit a loop early** when the answer is found (stop a **linear search** 线性查找 as soon as the target appears).
- **avoid redundant work** – store a result and reuse it instead of recomputing.
- **choose the right data structure** – an array beats many separate variables when the items belong together.
- **replace deep nested IFs with CASE** when testing one value against many.
- **comment the intent**, not the mechanics (`// validate the postcode`, not `// loop 6 times`).
- **use meaningful names** (`numberOfPupils`, not `n`) and **initialise variables** before use.

slower: Rate is found 10 times

```
FOR Index ← 1 TO 10
  Rate ← GetRate()
  Pay[Index] ← Hours[Index] * Rate
NEXT Index
```

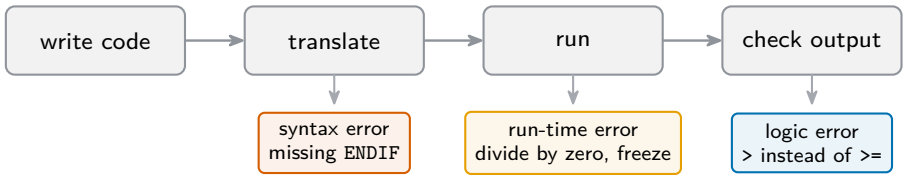
faster: Rate is found once

```
Rate ← GetRate()
FOR Index ← 1 TO 10
  Pay[Index] ← Hours[Index] * Rate
NEXT Index
```

Move unchanging work out of the loop so it runs once

Testing and errors

Three kinds of error, each found in a different way:



Each kind of error shows itself at a different stage.

Error	What it is	Example	Found by
syntax error 语法错误	a statement that breaks the rules of the language	a missing `ENDIF`; `OUTPT "Hi"`	the translator, before the program runs
run-time error 运行时错误	the program runs, but a statement cannot be carried out	division by zero; an array index of 0 or 51; a function called with an invalid parameter; a loop that never ends, so the program "freezes"	while running: the program stops or hangs
logic error	the program runs to the end, but the output is wrong	`>` where `>=` was needed; a total never set to `0`	testing with a trace table and chosen test data

An **IDE** 集成开发环境 helps find the last two: a **breakpoint** 断点 stops the program at a chosen line; **single stepping** 单步执行 then runs one statement at a time; and the report (or watch) window shows the value of each variable at that moment, so the line where a value goes wrong is seen directly. Test methods and test data are in topic 12.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly.

Term	Definition
procedure	a subroutine that carries out a task (a sequence of steps) and does not return a value; it is called with `CALL`
function	a subroutine that returns a single value to the point where it was called, so it can be used in an expression

exercises.lol exercises.com.cn

Term	Definition
parameter	the identifier in a subroutine header that receives a value or a reference when the subroutine is called
argument	the value (or variable) supplied in the call, matched to a parameter
passing by value	a copy of the argument's value is given to the subroutine, so changes inside it do not affect the original variable
passing by reference	the address of the variable is given to the subroutine, so changes inside it change the original variable
header	the first line of a subroutine definition: its name, its parameters and, for a function, its return type
interface	what a calling program must know to use a subroutine: its name, its parameters (number, order, type) and its return type
return value	the value a function passes back to the expression that called it
local variable	declared inside a subroutine; it exists only while the subroutine runs and can be used only inside it
global variable	declared outside every subroutine; it can be used anywhere in the program
count-controlled loop	repeats a fixed number of times, controlled by a counter (`FOR ... NEXT`)
pre-condition loop	tests its condition before each iteration, so the body may never run (`WHILE ... ENDWHILE`)
post-condition loop	tests its condition after each iteration, so the body runs at least once (`REPEAT ... UNTIL`)
constant	a named value that cannot change while the program runs
subroutine	a self-contained block of code that performs a task and is called by name: a procedure or a function
library routine	a subroutine that has already been written and tested, and is available to be called from a program

exercises.lol exercises.com.cn

Exam tips

- Distinguish a **procedure** (no return value) from a **function** (returns a value); know **pass by value vs by reference**.
- Choose the right loop: **count-controlled (FOR)** when the number of repeats is known, **condition-controlled (WHILE/REPEAT)** otherwise.
- Distinguish **local vs global** variables and scope; prefer local variables in reusable modules.
- Use the insert's exact routine names and parameter order. `VAL` and `STR` are IGCSE names and score nothing; `UCASE` and `LCASE` are real 9618 routines from the Pseudocode Guide but act on **one character**, so on Paper 2 a whole string takes `TO_UPPER` or `TO_LOWER`.
- In a "write pseudocode" answer the header, the declarations, the loop, the condition, the update and the `RETURN` each carry a mark: write all six parts, even if one is uncertain.

Common mistakes

- Calling a function and not using what it returns. Assign the result, or use it in the expression or output: `Sorted ← BubbleSort(MyArray, 7)`.
- Passing a length one out: `6` for a seven-element array, or the last index where the length was wanted. Decide whether the parameter is a length or an index, and check that the last element is visited.
- Closing a file inside the loop that reads it. Open once, close once, after the loop.
- Using the input as a filename directly. Add the extension the question gave: `File Name ← Choice & ".txt"`.
- Leaving constructs open. Every `IF` needs its `ENDIF`, every `FOR` its `NEXT`, every `WHILE` its `ENDWHILE`, and every function its `RETURN`; the scheme has a mark for it.
- Wrong boundaries: `>` for "at least" (which is `>=`), or a `FOR` that starts at `0` for an array declared `[1:50]`.
- A counter or total that is never set to `0` before the loop.
- In a trace table, rewriting every variable on every row, or changing a value before the statement that changes it has run.
- Half a condition: `IF x = 3 OR 4` – each side of `OR` and `AND` must be a complete comparison. And `+` does not join strings; `&` does.
- Declaring as local a value that must survive between calls. A running total or a string built up over several calls is global or `BYREF`.

Software Development

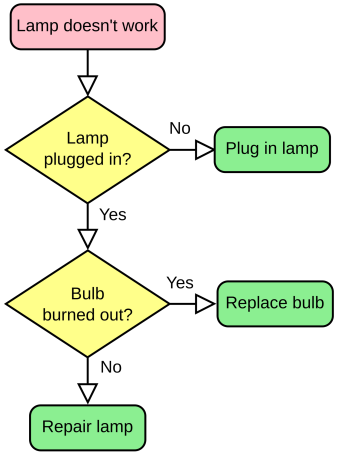
A-Level Computer Science

Program development life cycle

A **development life cycle** 开发生命周期 is the set of stages from idea to finished, maintained software. It exists to **plan, manage and control** a project – to build the right product, on time, with good quality.



Software is built by teams who follow a development life cycle to stay coordinated



A flowchart plans a program's logic during the design stage of the cycle

Why a life cycle is needed

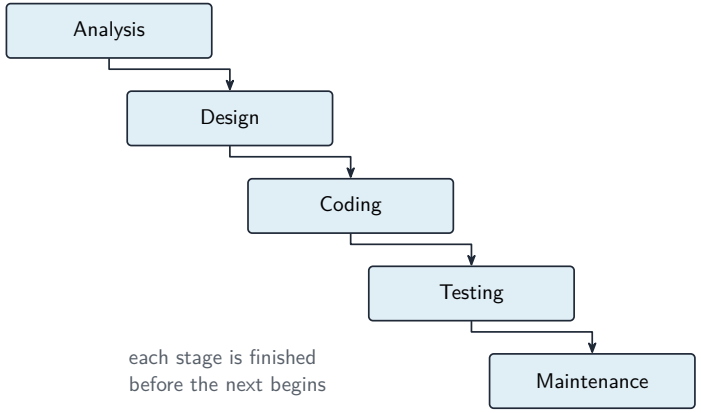
The examiner's list for "the purpose of a development life cycle": it breaks a large project into **stages** that can be planned and managed; it makes sure the **requirements** are found and agreed before design and coding begin; it builds in **testing** and **documentation** rather than leaving them to the end; it lets the team track progress against **milestones** and manage risk; and it gives the customer defined points at which to review the work. Without one, a team codes first and discovers late that it built the wrong thing.

Why there are different ones

No single life cycle fits every project, so several **development life cycles** exist. The choice depends on the size and complexity, how clear the **requirements** 需求 are at the start, how much change is expected, the risk level, the team, and the deadline.

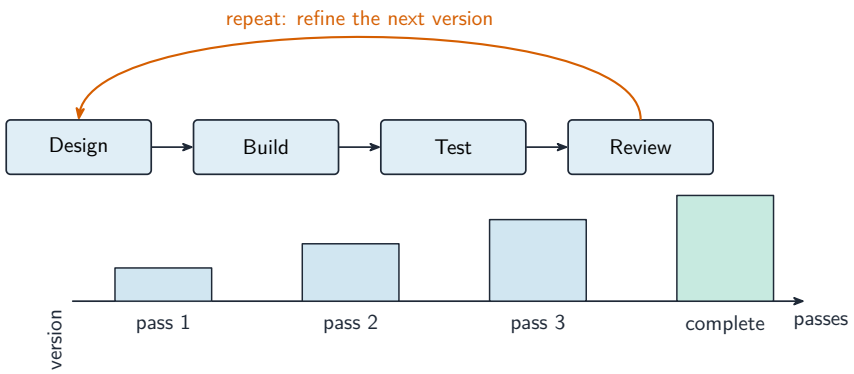
Common models

- **Waterfall** 瀑布模型 – a linear sequence (Analysis → Design → Coding → Testing → Maintenance), each stage finished before the next. Clear and well-documented; good for **stable requirements**, but poor at coping with mid-project change, and the customer sees nothing working until the end.
- **Iterative model** 迭代模型 – repeated passes, each producing a partial version that is reviewed and refined. Catches problems earlier; good when requirements are **discovered over time**, but harder to estimate.
- **Rapid Application Development** 快速应用开发 (RAD) – heavy use of a **prototype** 原型 and user feedback. Very fast first delivery; good for **changing requirements**, but depends on user availability and suits smaller systems.
- **Agile** 敏捷 – short iterations ("sprints"), constant collaboration and testing. Flexible and adaptive, but needs a committed customer and a skilled team.

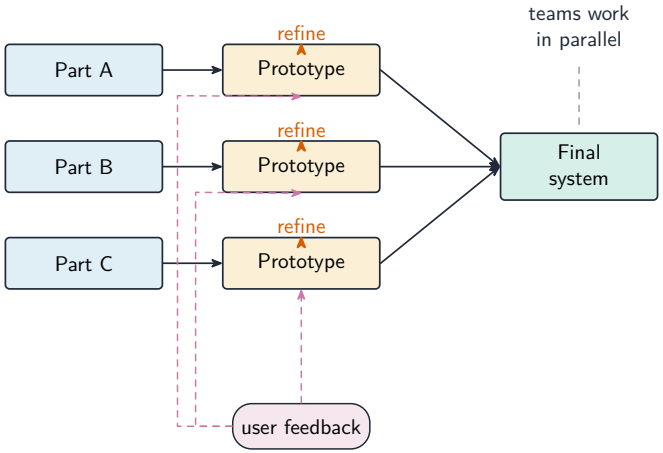


The waterfall model: each stage is finished before the next begins

exercises.lol exercises.com.cn



The iterative model: repeated passes refine the program



Rapid application development: teams work on parts in parallel

Principles, benefits and drawbacks – as the mark scheme lists them.

exercises.lol exercises.com.cn

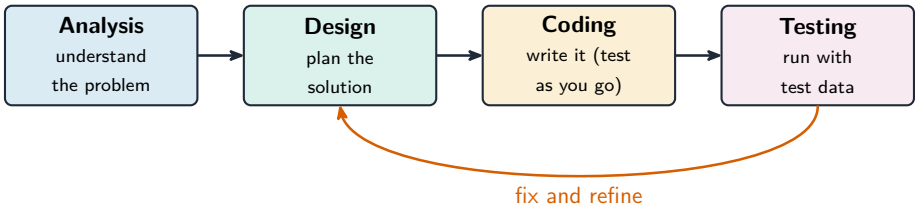
Model	Principle	Benefits	Drawbacks
waterfall	the stages run in a fixed order, each completed and signed off before the next starts; going back means restarting the sequence	simple to manage; every stage is fully documented; requirements are fixed early, so costs and dates can be estimated	inflexible once a stage is finished; no working software until late; a mistake in analysis is expensive to fix later; the customer cannot see progress
iterative	a small working version is built first, then repeatedly improved through further versions until complete	working software early and often; problems found in early versions; the customer's feedback shapes each version; requirements can change	hard to estimate the total time and cost; repeated testing costs effort; needs the customer to be available; can drift if versions are not planned
RAD	prototypes of parts of the system are built quickly and refined with the user until accepted, often in parallel by several teams	very fast delivery of a first version; the user is involved throughout, so the product fits their needs; changes are easy to absorb	needs skilled developers and committed users; documentation is weak; less suited to large or safety-critical systems

> **Worked example.** A company must be the first to launch a website for a new games console, and the design will change as the console's features are announced. Name the most suitable life cycle and justify it.

RAD. A prototype of the site can be built and shown to the users within days, and refined as the requirements change; the site is small enough for a prototype-driven approach, and speed of delivery is the main requirement. Waterfall would fix the requirements before any page was built and deliver nothing until the end.

The standard stages

Each stage has a purpose, an output and typical activities – a "describe the … stage" question wants two or three of these.



The four stages of program development; testing feeds back to fix and refine the design

- **analysis** – find out **what** the program must do. Activities: interviews, questionnaires and observation of the current system; a feasibility study; agreeing the **requirements** specification, which every later stage is checked against.
- **design** – decide **how** it will do it. Outputs: the **structure chart** (modules and parameters), **flowcharts** or **pseudocode** for each module, **identifier tables** and data structures, screen and file layouts, and the **test plan** written *now*, from the specification, before any code exists.
- **coding** (**implementation** 实现) – write the program in a high-level language, module by module, following the design; each module is tested as it is written.
- **testing** – run the program against the test plan (normal, abnormal, extreme and boundary data) and correct the errors found; integration, alpha, beta and acceptance testing follow.
- **maintenance** 维护 – after release, correct faults, adapt the program to new hardware, software or law, and improve it (see below).

> **Worked example.** Complete the waterfall diagram *Analysis* → *?* → *?* → *?* → *Maintenance* and describe what happens at the design stage.

The missing stages are **Design, Coding, Testing**. At the design stage the requirements are turned into a plan for the program: the problem is decomposed into modules (a structure chart), the algorithm for each module is written as pseudocode or a flowchart, the data structures and identifiers are chosen, the screens and files are laid out, and the test plan is written from the specification.

Program design tools

Structure chart

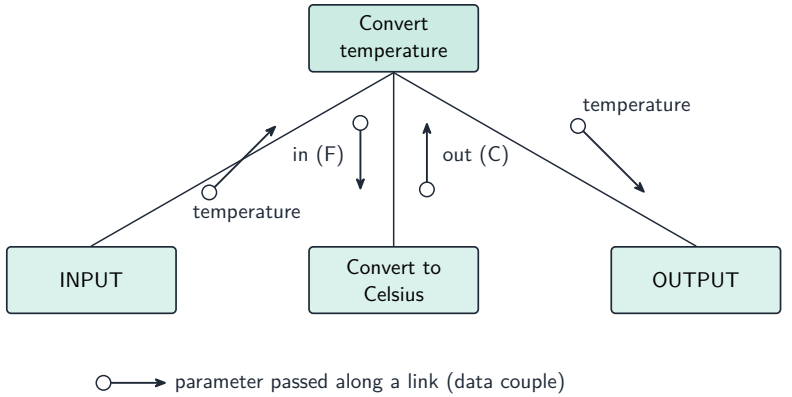
A **structure chart** 结构图 shows the **hierarchical decomposition** 分解 of a program into modules (**subroutines** 子程序) and the **parameters** 参数 passed between them. Each module is a rectangle; lines link caller (above) to callee (below); small arrows show data going down and results coming back up. The design can then be

exercises.lol exercises.com.cn

turned into equivalent **pseudocode** 伪代码.

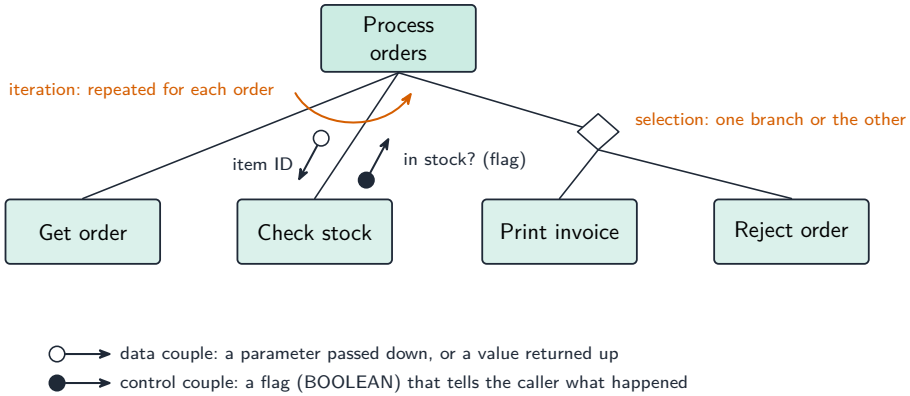
```
              CalculatePay
            /       |        \
GetEmployee  CalculateBonus  CalculateTax
Returns:     Takes: sales    Takes: gross
employeeID   Returns: bonus  Returns: tax
```

It is a design-stage tool, and you can read the procedure signatures off it.



A structure chart: modules with the parameters passed between them

The symbols the examiner asks about. A box is a module; a line links a caller (above) to the modules it calls (below), read left to right in the order they are called. A small arrow with an **open circle** at its tail is a **data couple** – a parameter passed *down* into a module or a value returned *up*; an arrow with a **filled circle** is a **control couple**, a flag (usually BOOLEAN) that tells the caller what happened. A **diamond** at a branch means **selection**: only one of the modules below it is called, depending on a condition. A **curved arrow** sweeping across the links means **iteration**: the modules under it are called repeatedly in a loop.



The structure-chart symbols: data and control couples, a selection diamond and an iteration arrow

> **Worked example.** Four modules are defined as `PROCEDURE Main()`, `PROCEDURE ReadData(BYREF Count : INTEGER)`, `FUNCTION IsValid(Value : INTEGER) RETURNS BOOLEAN` and `PROCEDURE Report(Total : INTEGER, Count : INTEGER)`. Main calls ReadData, then calls IsValid once for each value read, then calls Report. Describe the structure chart.

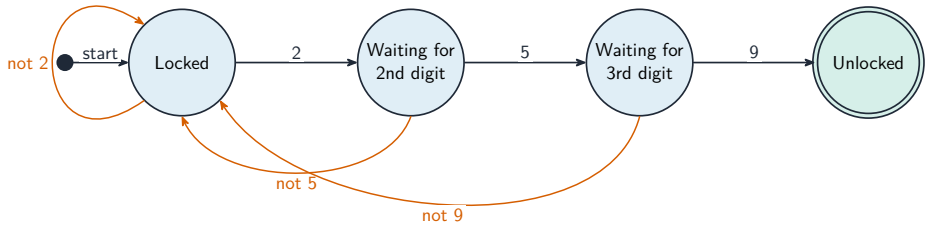
Main at the top; ReadData, IsValid and Report in a row beneath it, left to right in calling order. On the ReadData link an upward data couple `Count` (a BYREF parameter comes back). On the IsValid link a downward data couple `Value` and an upward control couple (the BOOLEAN result), with a curved iteration arrow across that link because it is called for each value. On the Report link two downward data couples, `Total` and `Count`. Reading the other way, a function is any module that returns a value – its header needs `RETURNS` and the returned type.

State-transition diagram

A **state-transition diagram** 状态转换图 shows the **states** 状态 a system can be in and the **events** that move it between them – good for vending machines, traffic lights, user interfaces. **State-transition diagrams** are used to document the behaviour of an algorithm or system. Each state is a circle; each transition is an arrow labelled with the event.

```
   coin inserted                item selected
[Idle] ---------------→ [Awaiting selection] -----------→ [Dispensing]
```

It makes missing transitions easy to spot ("what if a second coin is inserted while awaiting selection?").



A state-transition diagram for a door lock with code 259

Reading and drawing one. Each transition is labelled *input* / *output* (or *condition* / *action*): what happened, then what the system does as it changes state. A question gives a table of *current state, input, output, next state* and asks for the diagram, or the reverse – every row of the table is exactly one arrow. Check that every state has an arrow leaving it for every input that can occur, including the ones that leave the state unchanged (an arrow that loops back to the same state).

> **Worked example.** A pump controller has states *pump off* and *pump on*. In *pump off*, the input *low level detected* produces the output *activate pump* and moves to *pump on*; in *pump on*, *normal level detected* produces *deactivate pump* and moves to *pump off*. Any other input leaves the state unchanged. Draw the table.

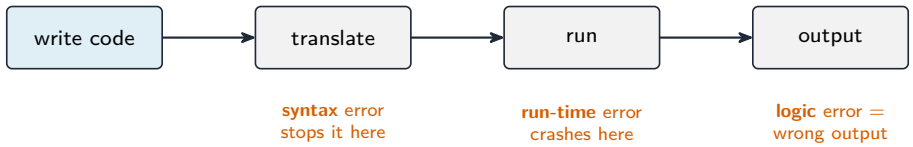
Current state	Input	Output	Next state
pump off	low level detected	activate pump	pump on
pump off	normal level detected	–	pump off
pump on	normal level detected	deactivate pump	pump off
pump on	low level detected	–	pump on

The two "no change" rows become loop arrows on the diagram; leaving them out loses the mark for completeness.

Errors

- **syntax error** 语法错误 – breaks the language's grammar (missing bracket, misspelled keyword). Caught at translation time; the program won't run until fixed.
- **run-time error** 运行时错误 – happens while running (divide by zero, file not found, array index out of range). The program crashes or raises an exception; fix by adding checks.
- **logic error** 逻辑错误 – the program runs but gives **wrong results** (using `+` for `-`, an off-by-one loop, conditions in the wrong order). The hardest to find; the only sign is wrong output, so use careful testing and tracing.

exercises.lol exercises.com.cn



When each error shows up: syntax at translation, run-time during the run, logic in the output

Exposing and avoiding faults. Faults are *exposed* by testing against a test plan, by a dry run or trace table, by a walkthrough with colleagues, and by the IDE's debugger (breakpoints, single stepping, watching variables). They are *avoided* by designing before coding (structure chart, pseudocode), by modular code with meaningful identifiers and comments, by **validation** of every input, by handling exceptions rather than letting a run-time error crash the program, and by the IDE's dynamic syntax checks as you type.

> **Worked example.** State the type of error in each case and how it shows itself. (a) `Result <- STR_TO_NUM(x) / STR_TO_NUM(y)` is run with `y = "0"`. (b) The same line is run with `x = "12a"`. (c) A loop written as `FOR i <- 1 TO 9` processes a ten-element array. (d) `OUTPUT "Total: " Total` is missing a comma.

(a) **Run-time error** – division by zero; the program crashes when this line is executed with that data. (b) **Run-time error** – the string cannot be converted to a number. (c) **Logic error** – the program runs but the tenth element is never processed, so the output is wrong. (d) **Syntax error** – the statement breaks the language's rules and is reported by the translator before the program runs.

Worked example. Correct the errors in this pseudocode, which should output the average of ten marks.

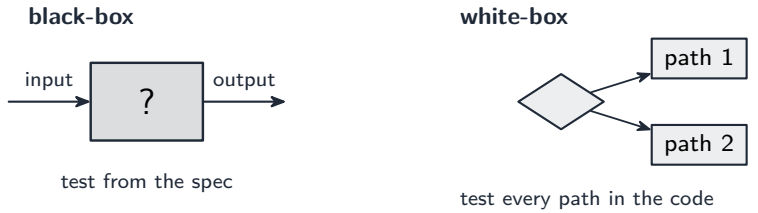
```
Total <- 0
FOR i <- 1 TO 10
    INPUT Mark
    Total <- Total + Mark
NEXT i
Average <- Total / 9
OUTPUT "Average" Average
```

The division should be by 10, not 9 (a logic error); the output line needs a comma or an `&` between the string and the value (a syntax error); and `Average` is never declared as REAL (a syntax or run-time error, depending on the language). Say which line and what the corrected line is: `Average <- Total / 10`.

exercises.lol exercises.com.cn

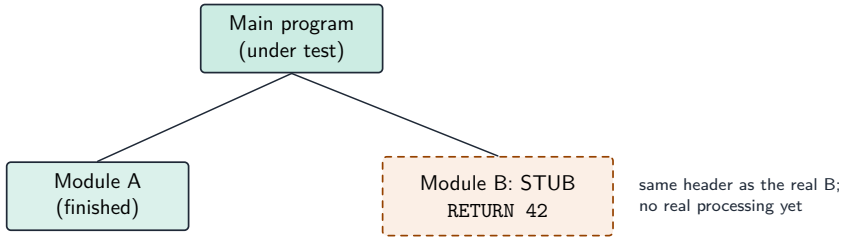
Testing methods

- **dry run** 手工跟踪 – trace the code on paper, writing each variable's value in a table.
- **walkthrough** 走查 – a team review of the code.
- **white-box testing** 白盒测试 – designed from the code's internal structure, covering every statement, branch and loop.
- **black-box testing** 黑盒测试 – designed from the specification only: feed inputs, check outputs.
- **integration testing** 集成测试 – combine modules and test the interfaces between them.
- **alpha testing** α 测试 – by the developers/in-house before release; **beta testing** β 测试 – by a limited group of real users in their own environment.
- **acceptance testing** 验收测试 – by the customer, to decide if the product is fit for purpose.
- **stub** 桩 – a placeholder for a module that does not exist yet, so the structure can be tested top-down.



Black-box tests the specification; white-box tests the code paths

Which method, when. A **dry run** and a **walkthrough** need no computer – the dry run is you, tracing the algorithm with a **trace table** 跟踪表; the walkthrough is a meeting in which the author explains the code line by line and colleagues look for faults, so it also spreads knowledge of the code through the team and checks it against the design. **White-box** tests are written by someone who can see the code and aims to exercise every path; **black-box** tests are written from the specification and check only inputs against expected outputs, so a user or a separate tester can do them. **Integration** testing follows module testing: modules that pass alone can still fail when the data passed between them is the wrong type or in the wrong order. **Alpha** testing is in-house; **beta** testing gives a release candidate to a sample of real users, who report faults from real use; **acceptance** testing is the customer checking the finished product against the requirements before paying for it. A **stub** lets top-down testing start before every module exists.



Main can be run and tested before B is written; the stub is replaced by the real module later

A stub stands in for a module that is not written yet, so the modules above it can be tested now

> **Worked example.** After the program passed its in-house tests it was given to a group of users to try before release. Name this type of testing, and state what happens next.

Beta testing – real users in their own environment, reporting faults the developers did not find. The faults are corrected, then the customer carries out **acceptance testing** against the requirements and the program is released; faults found in live use are then handled by corrective maintenance.

> **Worked example.** Give three benefits of testing a program by walkthrough.

Errors are found by people who did not write the code and so read it without assumptions; the logic is checked against the design and specification, not only against test data; several people learn how the code works, which helps later maintenance; and no test data or working computer is needed, so it can be done early.

Test strategy and test plan

A **test strategy** 测试策略 is the **high-level approach** – which kinds of testing, who does them, when, and the criteria to move on. A **test plan** 测试计划 is the **detailed list of tests** – each with input data, expected output, and a column for the actual output.

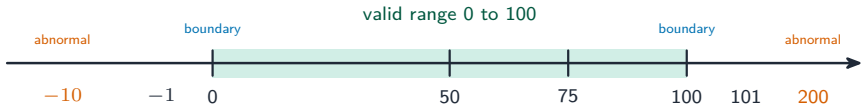
What each contains. A *test strategy* states which testing methods will be used at which stage (module testing by the programmer, then integration, alpha, beta, acceptance), who is responsible for each, what test data is required, and the criteria for passing to the next stage. A *test plan* lists the individual tests: for each, the module or feature under test, the input data, the **reason** the data was chosen (normal, abnormal, extreme, boundary), the **expected** result, a space for the **actual** result, and what to do if they differ. The plan is written at the **design** stage, from the specification, so that it tests what the program *should* do rather than what it happens to do.

exercises.lol exercises.com.cn

Choosing test data

For each field or condition, include three kinds:

- **normal data** 正常数据 – typical values inside the valid range (for marks 0–100: `50`, `75`).
- **abnormal data** 异常数据 – values that should be rejected (`-10`, `200`, `"abc"`).
- **extreme data** 极端数据 – the largest and smallest values still **accepted** (`0` and `100`).
- **boundary data** 边界数据 – values at the edges, where off-by-one errors hide (each accepted extreme and the rejected value just outside it: `0`/`-1`, `100`/`101`).



normal (50, 75) inside; extremes (0, 100) the accepted edges; abnormal (−1, 101, −10, 200) rejected

Test data for a 0–100 field: normal inside, extremes at the boundaries, abnormal outside

> **Worked example.** A field accepts an exam mark from 0 to 100. Give test data of each kind with its expected result. **Normal**: `50` - accepted, a typical value inside the range. **Abnormal**: `-10`, `200`, `"abc"` - all rejected, being out of range or the wrong data type. **Extreme**: `0` and `100` - the largest and smallest values that are still **accepted**. **Boundary**: the pairs straddling each edge - `-1` rejected alongside `0` accepted, and `100` accepted alongside `101` rejected. Every value must carry its **expected result**, or the test plan proves nothing. Extreme and boundary are the pair most often confused: an extreme value sits **inside** and is accepted, while a boundary test is always a **pair** either side of the edge - which is exactly where off-by-one errors hide.

> **Worked example.** A component passes if its weight, measured to the nearest gram, is within 3 g of the target of 50 g, i.e. from 47 g to 53 g inclusive. Draw up the test-plan rows for the check.

Test data	Type	Reason	Expected result
50	normal	a typical value well inside the range	accepted
47, 53	extreme (boundary)	the smallest and largest values that must still be accepted	accepted
46, 54	boundary	the values just outside the range, where an off-by-one error would accept them	rejected
20, 90	abnormal	values far outside the range	rejected
“abc”, −5	abnormal	the wrong type, a negative weight	rejected

Each row must say **why** the value was chosen and **what should happen**; a bare list of numbers earns nothing.

Maintenance

Most of a program’s lifetime cost is in maintenance. Three kinds:



Three kinds of maintenance: perfective, adaptive and corrective

- **perfective maintenance** 完善性维护 – improving performance or features even though it works (a faster query, a new option).
- **adaptive maintenance** 适应性维护 – keeping it working in a changing environment (a new OS, a new API, a legal change).
- **corrective maintenance** 纠正性维护 – fixing bugs found in use.

A program may need all three throughout its life.

Why each is needed – the reasons the mark scheme lists. *Corrective*: a fault is reported by a user after release, or an incorrect output is noticed in particular circumstances that testing did not cover. *Adaptive*: the operating system, hardware or browser is upgraded; a law or company rule changes (tax rates, data-protection requirements); the program must work with a new external system or file format. *Perfective*: users ask for extra features or a better interface; the program is made faster or made to use less memory; the code is tidied to make future changes easier.

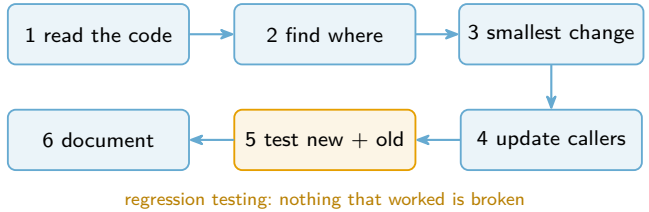
exercises.lol exercises.com.cn

> **Worked example.** (a) A released program outputs a wrong value under certain circumstances. (b) The hardware that runs a program is replaced. (c) Customers ask for the coffee-shop loyalty program to send a message on a customer’s birthday. Name the maintenance type in each case.

(a) **Corrective** – a fault in the delivered program is being fixed. (b) **Adaptive** – the program is changed to run in its new environment. (c) **Perfective** – a feature is added to a program that already works.

Amending an existing program

When asked to add a feature or fix a bug:



Test the old behaviour as well as the new.

1. **read the existing code** until you understand the algorithm and data flow.
2. **find where the change goes** – which subroutine, which lines.
3. **make the change as small as possible** – don’t rewrite working code.
4. **update related parts** – every caller of a changed parameter list, every routine using a changed data structure.
5. **test** the new behaviour **and** the old (**regression testing** 回归测试 – check you broke nothing).
6. **document** the change.

Clear comments, meaningful names, decomposed subroutines and a structure chart make a program much easier to amend – which is why the design tools matter even after the first release.

Analysing a program you did not write. Start from the identifier table and the module headers: they tell you what each module receives and returns before you read a line of its body. Then trace the algorithm with a trace table for one small input, noting where each output value comes from. Only then decide where the enhancement goes – usually a new module called from the existing one, so the working code is disturbed as little as possible – and write the pseudocode for the change and the test data that proves it.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
development life cycle	the sequence of stages, from analysis to maintenance, followed to produce and support a program
waterfall model	a life cycle in which the stages are carried out in a fixed order, each completed before the next begins
iterative model	a life cycle in which a working version is produced and then repeatedly refined until it is complete
rapid application development	a life cycle that builds prototypes quickly, refining them with user feedback until they are accepted
structure chart	a diagram that shows how a program is decomposed into modules, the order in which they are called and the parameters passed between them
state-transition diagram	a diagram that shows the states a system can be in and the inputs that cause it to move between them
syntax error	an error in the way a statement is written, so it breaks the rules of the language and cannot be translated
logic error	an error in the algorithm, so the program runs but produces the wrong result
run-time error	an error that occurs while the program is running, such as division by zero, and stops it
dry run	working through the algorithm by hand, recording the values of the variables in a trace table
walkthrough	a review in which the author steps through the code with colleagues who look for errors
stub	a placeholder module with the correct header that returns a fixed value, used so the modules that call it can be tested
test plan	a list of the tests to be carried out, each with its test data, the reason for the data and the expected result
boundary data	values at each edge of the valid range, both the last value accepted and the first value rejected

exercises.lol exercises.com.cn

Term	Definition
corrective / adaptive / perfective maintenance	fixing faults found in use / changing the program to suit a changed environment / improving a program that already works

Exam tips

- Compare development models (waterfall, iterative, RAD) by **principle, benefit, drawback**, and know the five stages of the **program development life cycle** and what each produces.
- Distinguish **syntax, logic and run-time** errors by *when* each shows itself: at translation, in the output, during the run.
- Choose **test data** of every kind – normal, abnormal, extreme and boundary – and give each value with its reason and expected result.
- Distinguish the types of **maintenance** (corrective, adaptive, perfective) by *why* the change is being made.
- On a structure chart, name every symbol: box, calling line, data couple, control couple, selection diamond, iteration arrow. Reading module headers off a chart, remember a function has `RETURNS`.

Common mistakes

- Describing a life cycle stage by its name only ("in the design stage the program is designed"). Say what is produced: structure chart, pseudocode, test plan.
- Calling a wrong output a "run-time error". If the program runs to the end, it is a logic error.
- Giving boundary data as just the extremes. The mark needs the values on **both** sides of the edge.
- Treating alpha and beta testing as the same. Alpha is in-house by the developers; beta is by real users outside.
- Confusing adaptive and perfective maintenance. Adaptive responds to a change *outside* the program; perfective improves a program nobody had to change.
- Drawing a structure chart with the modules in any order. They read left to right in the order they are called, and each parameter needs its arrow.

Data Representation

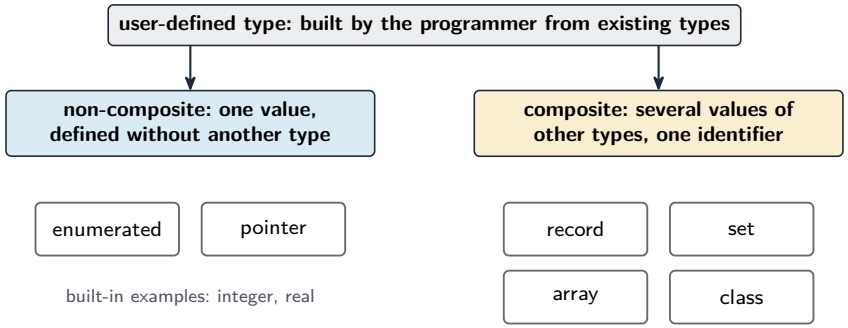
A-Level Computer Science

User-defined data types

The built-in types (`INTEGER`, `REAL`, `STRING`, `CHAR`, `BOOLEAN`) cover the simplest cases. For richer problems you can define **user-defined types** 用户定义类型, making the code clearer and the compiler stricter.

Why they are needed

A built-in `STRING` lets you store nonsense in a field that should hold one of a few legal values; a user-defined type can restrict it. Real entities are usually a **collection** of values of different types. And `DECLARE Taxi : Vehicle` is clearer (self-documenting) than `DECLARE Taxi : STRING`.



Non-composite types hold one value; composite types collect other types

"Describe the purpose of a user-defined data type" (two marks). *A data type defined by the programmer, built from existing (built-in) types, so that data specific to the problem can be represented when no built-in type fits.* Both halves score: **defined by the programmer** and **based on existing types**. The examiner also accepts "to make the program easier to read and maintain" as a supporting point, never on its own.

"Explain what is meant by non-composite and composite data types" (four marks). A **non-composite** type is defined **without reference to another type**: it holds a **single** value, for example an integer, a real, or an enumerated value. A **composite** type is a collection of **other types** (which may themselves be composite): it holds **several** values under one identifier, for example a record, a set, an array or a class. Give an example with each definition; the exam asks for one.

exercises.lol exercises.com.cn

Non-composite types

Enumerated type

An **enumerated type** 枚举类型 has values that are a **fixed list of named constants**:

```
TYPE Vehicle = (M100, M230, T101, T102, T120, T150)
DECLARE MyTaxi : Vehicle
MyTaxi ← T102
```

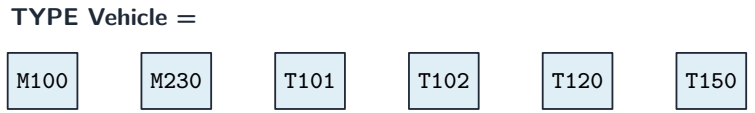
The names are values of the new type (stored internally as small integers); you cannot assign anything outside the list. Uses: days of the week, colours, status codes.

"State what is meant by an enumerated data type." *A non-composite user-defined type defined by listing all its possible values (in order).* Because the values are **ordered**, they can be compared and stepped through: with `TYPE Month = (January, February, ..., December)`, the test `IF ThisMonth > June` is legal, and the values are stored internally as integers. The pseudocode has three parts and the exam marks each: the keyword `TYPE`, the identifier with `=`, and the list in brackets separated by commas.

Worked example. Write pseudocode to define an enumerated type for the days on which a school is open (Monday to Friday), and declare a variable of that type set to Wednesday.

```
TYPE SchoolDay = (Monday, Tuesday, Wednesday, Thursday, Friday)
DECLARE Today : SchoolDay
Today ← Wednesday
```

A variable of an enumerated type cannot be given a value outside the list, which is the whole point: `Today ← Saturday` is a compile-time error, whereas a `STRING` would have accepted `"Saturdy"`.



An enumerated type is a fixed list of named values

Pointer type

A **pointer** 指针 holds the **memory address of another variable** (or `NULL` for "no target"). Pointers build dynamic structures (linked lists, trees) and pass references without copying.

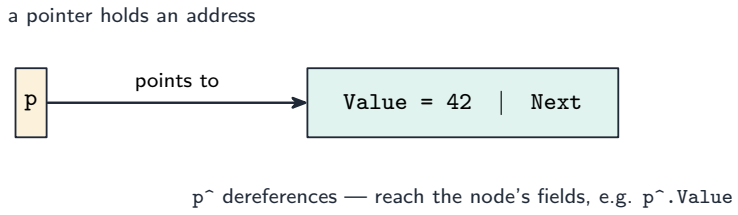
```
TYPE PNode = ^TNode    // pointer to a TNode
DECLARE p : PNode
p ← NEW TNode
p^.Value ← 42          // dereference to reach the fields
```

To **dereference** 解引用 (`p^`) means to reach the variable it points to.

"State what is meant by a pointer data type." *A non-composite type whose value is the memory address of (a reference to) a variable of a given type.* The pseudocode declares the type with a caret before the type it points to, and the exam asks for exactly that line:

```
TYPE SelectParts = ^Parts        // a pointer to a value of type Parts
DECLARE Chosen : SelectParts
Chosen ← ^Keyboard               // Chosen now holds the address of
↪  Keyboard
OUTPUT Chosen^                   // dereference: the value stored at that
↪  address
```

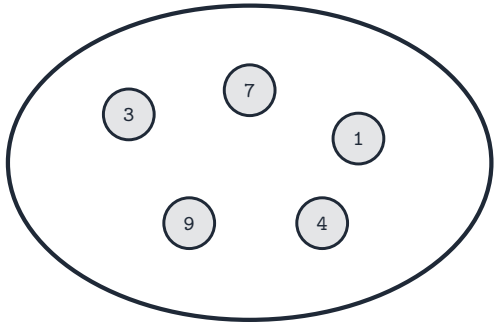
Pointers are what a dynamic **linked list** or **binary tree** (Topic 19) is built from: each node holds a pointer to the next. Two marks are commonly lost here: writing the pointer type as if it held the value itself, and forgetting the caret when reading through the pointer.



A pointer holds an address; `p^` dereferences it to reach the node's fields

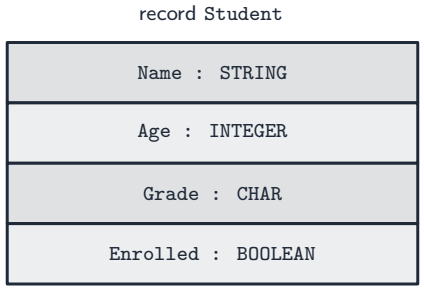
Composite types

A **composite type** 复合类型 (one of the **composite data types**) groups several values under one name.



unordered, every value unique

A set is an unordered collection of unique values



one name, several fields of different types

A record groups fields of different types under one name

- **record** 记录 (Topic 10) – fields of different types in a `TYPE ... ENDTYPE` block.
- **set** 集合 – an unordered collection of unique values, with operations add, remove, membership test, union, intersection:

exercises.lol exercises.com.cn

```
DECLARE Available : SET OF Colour
Available ← {Red, Blue}
IF Green IN Available THEN
    ...
ENDIF
```

- **class** 类 / **object** 对象 – the OOP composite type, combining data fields (**attributes** 属性) with operations on them (**methods** 方法). An object is an instance of a class:

```
CLASS Taxi
    PRIVATE Capacity : INTEGER
    PUBLIC FUNCTION GetCapacity() RETURNS INTEGER
        RETURN Capacity
    ENDFUNCTION
ENDCLASS
```

Choosing a type

Use **enumerated** for a value from a fixed list, **pointer** for indirection, **record** for a group of fields, **set** for an unordered unique collection, and **class** when you need state **and** behaviour together.

field	Name	Initial	Member	Age	Height	Joined
record	Amy	'M'	TRUE	42	1.62	14/03/2009
data type	*text/ alphanumeric*	*character*	*Boolean*	*integer*	*real*	*date/time*

Each field in a record holds one data type

"Describe the user-defined data type set" (three marks). *A composite type that holds a collection of values of the same type, in no particular order and with no duplicates; values can be added and removed, and a value can be tested for membership.* Declare the type with `SET OF`, then define a set constant with its values in brackets:

```
TYPE EvenNumbers = SET OF INTEGER
DEFINE Evens (2, 4, 6, 8, 10, 12) : EvenNumbers
TYPE SymbolSet = SET OF CHAR
DEFINE Operators ('+', '-', '*', '/') : SymbolSet
```

"Describe the user-defined data type record" (three marks). *A composite type made up of a fixed number of fields (items), each with its own identifier and its own type, referred to under a single identifier; the fields are accessed with dot notation.*

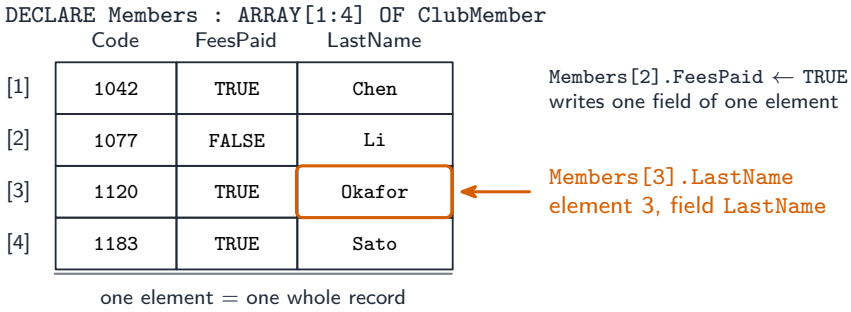
exercises.lol exercises.com.cn

Worked example. Write pseudocode to declare a record type `ClubMember` for a club member's first name, last name, membership code (an integer), date of joining and whether fees have been paid; then declare a variable and set two of its fields.

```
TYPE ClubMember
    DECLARE FirstName : STRING
    DECLARE LastName : STRING
    DECLARE Code : INTEGER
    DECLARE DateJoined : DATE
    DECLARE FeesPaid : BOOLEAN
ENDTYPE

DECLARE NewMember : ClubMember
NewMember.LastName ← "Chen"
NewMember.FeesPaid ← TRUE
```

Every field needs its own `DECLARE` line with an appropriate type, the block ends with `ENDTYPE`, and a **field** 字段 is reached as `variable.field`. Asked to choose a type for each field, match it to the data: a code that is only ever compared is a `STRING` if it can contain letters, an `INTEGER` if arithmetic or ordering is needed; a yes/no is `BOOLEAN`; a date is `DATE`. A field that can take one of a few named values (a pet's species, a colour) is the one to make an **enumerated** type.



An array of records: each element is a whole record, an index chooses the element, and a dot chooses the field

Records in arrays and files. A table of many members is `DECLARE Members : ARRAY[1:100] OF ClubMember`; then `Members[3].LastName` is one field of one element, and a loop over the index processes every record. A record is also the natural unit written to and read from a file (below), one record per `PUTRECORD` or `WRITEFILE`.

Worked example. A composite type `Pet` stores each pet's name (string), species (one of dog, cat, rabbit or hamster) and weight in kilograms (real). Define the types and declare a variable.

```
TYPE Species = (Dog, Cat, Rabbit, Hamster)
TYPE Pet
    DECLARE Name : STRING
    DECLARE Kind : Species
    DECLARE Weight : REAL
ENDTYPE
DECLARE MyPet : Pet
MyPet.Kind ← Rabbit
```

The enumerated type is defined **first**, because the record uses it: order matters in pseudocode as it does in a compiler.

Classes in pseudocode. A **class** is the composite type that also carries behaviour. The exam asks for the declaration with its attributes marked `PRIVATE`, a **constructor** 构造函数 named `NEW` that sets them, and `PUBLIC` methods to get or change them:

```
CLASS Appointment
    PRIVATE PatientName : STRING
    PRIVATE Treatment : STRING
    PRIVATE Medication : STRING
    PUBLIC PROCEDURE NEW(Name : STRING, Treat : STRING, Med : STRING)
        PatientName ← Name
        Treatment ← Treat
        Medication ← Med
    ENDPROCEDURE
    PUBLIC FUNCTION GetTreatment() RETURNS STRING
        RETURN Treatment
    ENDFUNCTION
ENDCLASS

DECLARE Visit : Appointment
Visit ← NEW Appointment("A. Chen", "filling", "none")
OUTPUT Visit.GetTreatment()
```

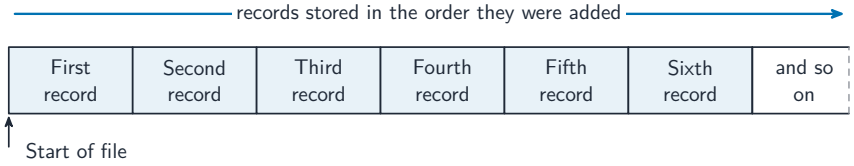
Attributes are private so that they can only be changed through methods (encapsulation, Topic 20); the constructor is a procedure called `NEW` with one parameter per attribute; a getter is a function that returns the attribute. Each of these is a separate mark.

exercises.lol exercises.com.cn

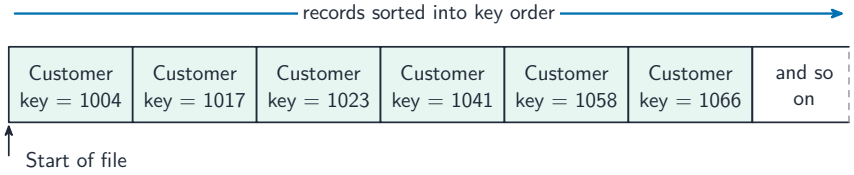
File organisation and access

File organisation 文件组织 is how the data is laid out; **file access** is how the program reaches a record.

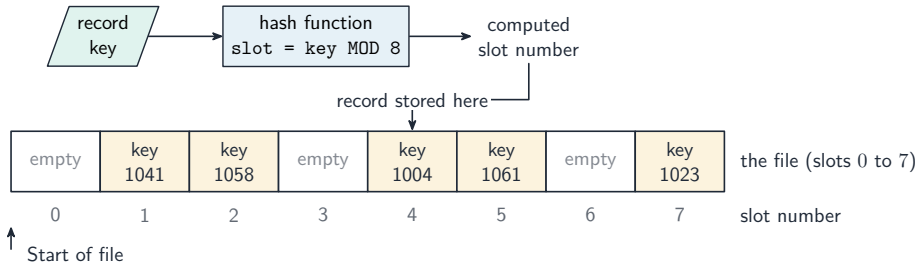
- **serial file** 串行文件 – records in the **order added**, no sorting. Access is sequential only; appending is fast; searching is slow. Used for logs and audit trails.
- **sequential file** 顺序文件 – records **sorted by a key**. Searching is faster (you can stop early or binary-search); inserting is slow (records must shift). Used for master files updated in batch.
- **random file** 随机文件 (direct-access file) – records at positions computed from the key (often by a hash). Direct access by key is very fast; reading in key order is harder. Used for large lookup tables and customer accounts.



Serial file: records are kept in the order they were added



Sequential file: records are sorted by a key field

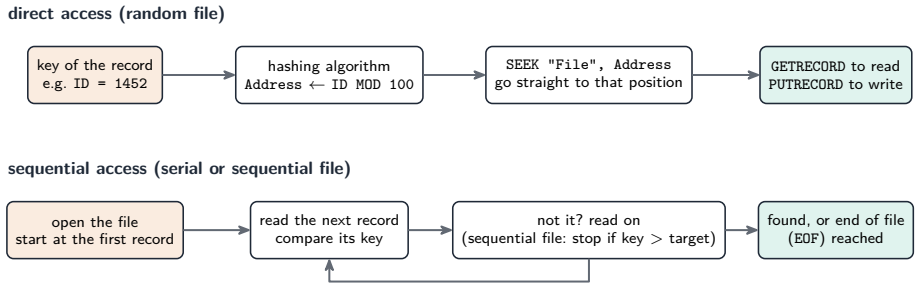


Random file: records sit at positions computed from the key

The two access methods are **sequential access** 顺序存取 (read from start to end) and

direct access 直接存取 (jump straight to a known position). Match the structure to the dominant operation: single-key lookups favour random; in-order reports favour sequential.

Describing each organisation (the wording that scores). *Serial:* records are stored **one after another in the order in which they were added**, with no ordering by key. *Sequential:* records are stored **in order of a key field** (sorted). *Random:* each record is stored at an **address calculated from its key** by a hashing algorithm, so the records are not in any order. **Comparing serial and sequential:** both store records one after another and both are read sequentially, but a sequential file is ordered by key, so a search can stop as soon as a key larger than the target is read, and a new record must be inserted in its correct position (usually by rewriting the file), whereas a serial file is simply appended to.



The two access methods as procedures: direct access computes where to look; sequential access looks everywhere in turn

Describing each access method. *Sequential access:* start at the **beginning** of the file and read the records **one after another** (in the order stored) until the required record is found or the end of the file is reached. Applied to a **serial** file this means reading every record up to the match, and reading the whole file to establish that a record is absent; applied to a **sequential** file the search can **stop early**, as soon as a key greater than the target is read. *Direct access:* the **address** of the record is calculated from its key (by a hashing algorithm, or from an index), and the program goes **straight to that position** without reading the records before it; this is the access method for random files, and for a record referenced by a unique address on a disk.

Choosing. A payroll or utility-billing master file processed in batch, every record in turn, suits a **sequential** file; a log of transactions in the order they happened suits a **serial** file; a stock or customer file where single records are looked up and updated by key while the program runs suits a **random** file with direct access.

File handling in pseudocode. The exam expects the standard statements, and Paper 3 sets algorithms that use them:

exercises.lol exercises.com.cn

Task	Statements
open a text file	`OPENFILE "Scores.txt" FOR READ` (or `FOR WRITE`, which creates or overwrites, or `FOR APPEND`)
read or write a line	`READFILE "Scores.txt", Line` and `WRITEFILE "Scores.txt", Line`
test for the end	`WHILE NOT EOF("Scores.txt")`
close	`CLOSEFILE "Scores.txt"`
open a random file	`OPENFILE "Stock.dat" FOR RANDOM`
move to a record position	`SEEK "Stock.dat", Address`
read or write a whole record	`GETRECORD "Stock.dat", Item` and `PUTRECORD "Stock.dat", Item`

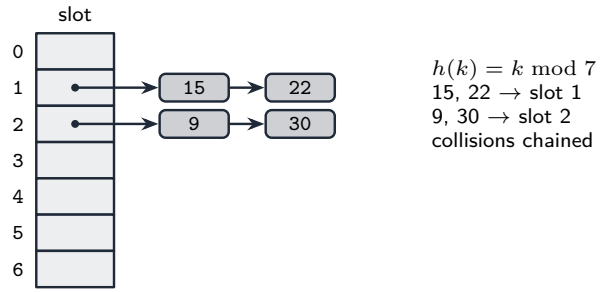
Worked example. A random file `Stock.dat` holds records of type `StockItem`, stored at the address given by `ItemID MOD 100`. Write pseudocode that stores a new item at its hashed address if that position is empty, reporting the position if it is already in use.

```
DECLARE Item, Existing : StockItem
DECLARE Address : INTEGER
INPUT Item.ItemID, Item.Description, Item.Quantity
Address ← Item.ItemID MOD 100
OPENFILE "Stock.dat" FOR RANDOM
SEEK "Stock.dat", Address
GETRECORD "Stock.dat", Existing
IF Existing.ItemID = 0 THEN    // 0 marks an empty position
    SEEK "Stock.dat", Address
    PUTRECORD "Stock.dat", Item
    OUTPUT "Stored at ", Address
ELSE
    OUTPUT "Position ", Address, " is in use"
ENDIF
CLOSEFILE "Stock.dat"
```

Two details the mark scheme checks: `SEEK` **before** each `GETRECORD` or `PUTRECORD` (reading moves the position on, so seek again before writing), and the file opened `FOR RANDOM` and closed at the end. To copy every record of a random file to another, loop over the addresses with `SEEK`, `GETRECORD` from one file and `PUTRECORD` to the other, skipping empty positions.

Hashing

A **hash function** 散列函数 (a **hashing algorithm**) takes a record key and produces an **address** where the record is stored. A good one is fast, **deterministic** 确定性, and spreads keys evenly.



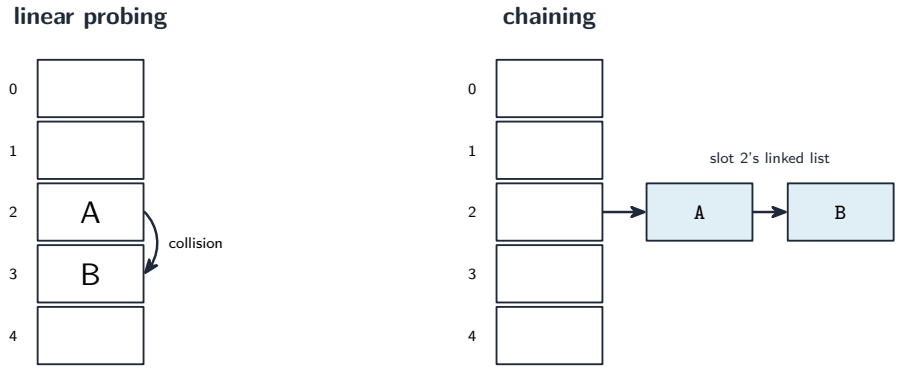
Two keys in one slot is a collision; chaining keeps a list in the slot

Common **hashing algorithms** for N slots: modulo hash `address ← key MOD N`; folding (split the key, add the pieces, MOD N); a string hash (sum the character codes, MOD N).

A **collision** 冲突 is when two keys hash to the same address. Three ways to resolve it:

Strategy	How it works	Trade-off
linear probing 线性探测	use the next free slot (wrapping around)	simple, but keys **cluster**
chaining 链接法	each slot points to a **linked list** 链表 of records	no clustering, but uses more memory
rehashing	apply a second hash function	spreads keys, but more work

exercises.lol exercises.com.cn



Resolving a hash collision: linear probing uses the next free slot; chaining keeps a linked list per slot

To search: hash the key, read that slot; if the keys match you are done, else follow the resolution strategy until a match or an empty slot. To insert: hash the key, write to that slot or the next free one. Keep the **load factor** 装填因子 (records ÷ slots) below about 70% for near-O(1) lookups.

"Explain what is meant by a hashing algorithm in the context of file access" (three marks). *A calculation (function) performed on the key field of a record that produces a value, which is used as the address (location) at which the record is stored in the file and from which it is retrieved.* The same calculation on the same key always gives the same address, which is why the record can be found again without searching.

"Outline two methods of overcoming a collision." (1) **Linear probing** (open addressing): store the record in the **next free** location after the calculated address, wrapping round to the start if necessary; to retrieve, start at the hashed address and read forward until the key matches. (2) An **overflow area** 溢出区 or **chaining**: store the colliding record in a separate overflow area (or a linked list attached to the address), which is searched sequentially after the main address fails to match. Either scores; describe the retrieval as well as the storage.

> **Worked example.** A random file has 11 record positions, numbered 0 to 10, and the hashing algorithm is `Address ← Key MOD 11`. Records with keys 1250, 1381, 1452, 1613 and 1470 are stored in that order, using linear probing. Show where each record goes, and describe how key 1470 is retrieved.

1250 mod 11 = 7; 1381 mod 11 = 6; 1452 mod 11 = 0; 1613 mod 11 = 7, a **collision** with 1250, so 1613 takes the next free position, 8; 1470 mod 11 = 7 again, and positions 7 and 8 are full, so 1470 goes to 9. To retrieve 1470: calculate 7, read position 7 (key 1250, no match), read 8 (1613, no), read 9 (1470, found). If an **empty** position is reached before a match, the record is not in the file. Collisions are the price of a small

file: a good hashing algorithm spreads the keys evenly, and the file is kept well below full so that probes stay short.

Floating-point numbers

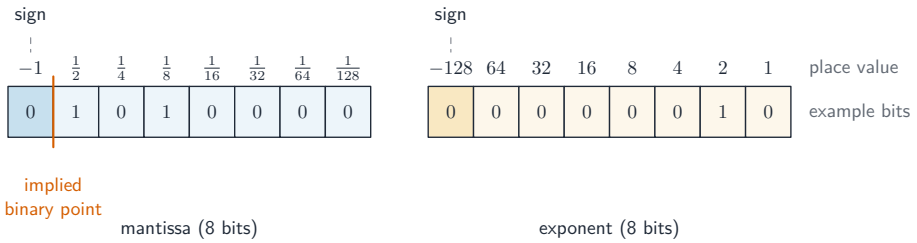
To store **real numbers** of very different sizes, computers use a **floating-point** 浮点 format – a binary form of scientific notation, with two fields:

- a **mantissa** 尾数 – the significant digits.
- an **exponent** 指数 – the power of 2 to multiply by.

Both are stored as **two's complement** 补码 integers. The value is

$$\text{number} = \text{mantissa} \times 2^{\text{exponent}}.$$

Read the mantissa as a binary fraction – the first bit after the point is worth 1/2, the next 1/4, then 1/8, and so on. So `0.1010000` is $1/2 + 1/8 = 0.625$; with exponent `00000010` ($= 2$) the value is $0.625 \times 2^2 = 2.5$.



The place values of an 8-bit mantissa and an 8-bit exponent

Converting

- **binary → denary**: read the mantissa (use two's-complement rules if negative) as a fraction, read the exponent as a signed integer, then multiply mantissa by 2^{exponent}.
- **denary → binary**: write the number as a binary fraction × a power of 2, then store the mantissa and exponent in the agreed formats.

> **Worked example.** A number has mantissa `10110000` and exponent `00000011`. Find its denary value.

The exponent `00000011` is +3. The mantissa begins with a 1, so it is negative. Read as `1.0110000` in two's complement, the sign bit is worth −1 and the fraction bits add $\frac{1}{4} + \frac{1}{8} = 0.375$, so the mantissa is $-1 + 0.375 = -0.625$. Then

$$\text{number} = -0.625 \times 2^3 = -5.0.$$

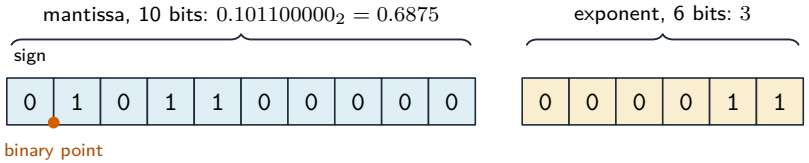
exercises.lol exercises.com.cn

> **Worked example.** Store +2.5 in this format.

In binary $2.5 = 10.1$. Written as a normalised fraction, $2.5 = 0.101 \times 2^2$. So the mantissa is `01010000` (sign bit 0, then `.101`) and the exponent is `00000010` ($= 2$).

The exam's format: two's complement, a mantissa and an exponent

The exam states a format such as **10 bits for the mantissa and 6 bits for the exponent, both in two's complement**. The mantissa's binary point sits after its first (sign) bit, so a positive mantissa is `0.xxxxxxxxx` and a negative one `1.xxxxxxxxx`; the exponent is an ordinary signed integer. Every conversion uses the same three moves: read the mantissa as a fraction (two's-complement rules if it starts with 1), read the exponent as an integer, multiply by 2^{exponent}.



$$\text{value} = 0.6875 \times 2^3 = 5.5$$

The 10-bit mantissa and 6-bit exponent of the first worked example

> **Worked example (binary to denary).** Mantissa `0101100000`, exponent `000011`.

Mantissa: $0.1011000002 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = 0.6875$. Exponent: $000011_2 = 3$. Value: $0.6875 \times 2^3 = 5.5$.

> **Worked example (negative mantissa).** Mantissa `1011000000`, exponent `000010`.

The mantissa starts with 1, so it is negative. Its value is $-1 + 0.011000000_2 = -1 + (\frac{1}{4} + \frac{1}{8}) = -0.625$; exponent = 2; value $-0.625 \times 4 = -2.5$. (Alternatively, take the two's complement of the mantissa, `0101000000` = 0.625, and attach the minus sign.) A **negative exponent** such as `111110` $= -2$ divides instead: a mantissa of 0.5 with that exponent is $0.5 \times 2^{-2} = 0.125$.

> **Worked example (denary to binary).** Store +6.5 and −6.5 in the 10-bit and 6-bit format, normalised.

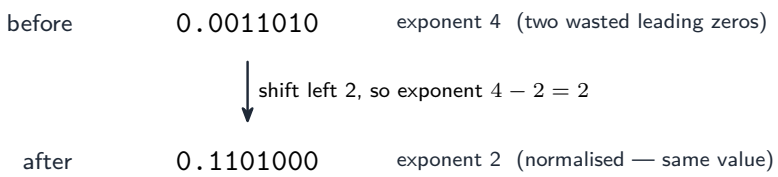
$6.5 = 110.1_2 = 0.1101_2 \times 2^3$, so the mantissa is `0110100000` and the exponent `000011`. For −6.5, take the two's complement of the mantissa: `1001100000` (check: $-1 + 0.0011_2 = -1 + 0.1875 = -0.8125$, and $-0.8125 \times 8 = -6.5$), exponent `000011` unchanged. The sign never goes into the exponent; a negative number has a negative **mantissa**.

Normalisation

A number is **normalised** 规格化 when the first significant bit is **immediately after the binary point** (no wasted leading zeros). This **maximises precision**, because every mantissa bit carries information. To normalise, shift the mantissa left and decrease the exponent (or shift right and increase it) until the first significant bit is in place; the value is unchanged. For negative (two's-complement) mantissas, the sign bit (1) is followed immediately by a 0.

Recognising and producing normalised form. A positive normalised mantissa begins `01`; a negative one begins `10`. So `0011000000` is not normalised (shift left one place and subtract one from the exponent: `0110000000`, exponent one less) and `1100000000` is not either (shift left until the pattern is `10...`). Each shift left of the mantissa must be matched by subtracting one from the exponent, or the value changes.

"Explain why numbers are stored in normalised form" (two marks). (1) It gives the **maximum precision** (accuracy) for the number of bits available, because no bits are wasted on leading zeros (or leading ones for a negative number); (2) each number then has a **unique** representation, so numbers can be compared; and (3) it makes the best use of the available **range**. Any two of these score.



Normalising: shift the mantissa left to remove leading zeros, lowering the exponent by the same amount

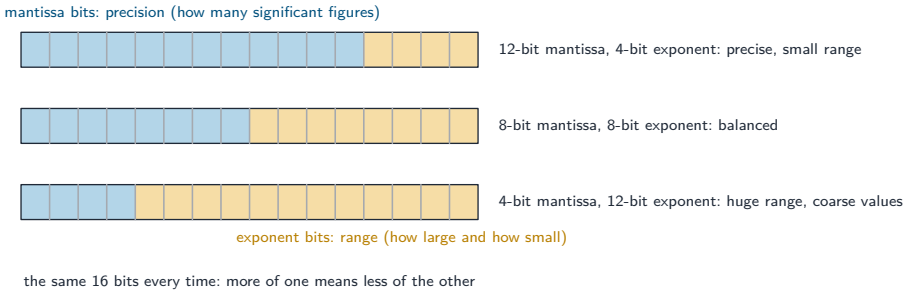
Approximation and rounding errors

Many denary reals **cannot be stored exactly** in binary – e.g. 0.1_{10} is the repeating binary fraction $0.000110011\ldots_2$, which must be truncated. Consequences:

- **rounding errors** 舍入误差 build up over many operations (`0.1 + 0.2` is not exactly `0.3`).
- **comparisons fail** – never test a real for equality. Test that the **difference** is smaller than a small tolerance, `IF Difference < 0.000001`, where the difference is taken the right way round or through a modulus function that the question would define. `ABS` is not on the 9618 insert or in the Pseudocode Guide, so do not assume it: the guide says any function a question needs will be given.
- subtracting two nearly-equal values loses precision.
- **overflow** 溢出 (a result too large for the exponent's range) and **underflow** 下溢 (a result too small, rounding to zero) occur when the exponent runs out of range.

exercises.lol exercises.com.cn

For exact needs (currency), use **fixed-point** 定点 or **BCD** 二进码十进数 instead of floating-point.



The same total of bits shared two ways: mantissa bits buy precision, exponent bits buy range, and one can only grow at the other's expense

"Describe the effect of changing the allocation of bits" (three marks). With a fixed total number of bits, **increasing the mantissa** and reducing the exponent gives greater **precision** 精度 (more significant figures, smaller rounding errors) but a smaller **range** 范围 (the largest and smallest magnitudes that can be stored shrink); **increasing the exponent** does the opposite: a larger range at the cost of precision. Name both effects and both directions.

Largest and smallest. In the 10-bit mantissa, 6-bit exponent format the largest positive number has mantissa `0111111111` ($= 1 - 2^{-9}$) and exponent `011111` ($= 31$): about 2^{31}. The smallest positive **normalised** number has mantissa `0100000000` ($= 0.5$) and exponent `100000` ($= -32$): $0.5 \times 2^{-32} = 2^{-33}$. The most negative number has mantissa `1000000000` ($= -1$) and exponent 31: -2^{31}.

"Explain what is meant by overflow and underflow." Overflow occurs when the result of a calculation is **larger than the largest** number that can be represented, so the exponent would need more bits than it has; **underflow** occurs when a result is **smaller than the smallest** (non-zero) number that can be represented, too close to zero for the exponent to express, so it is stored as zero. Both come from the **exponent's** range, not the mantissa's.

Why a binary representation is only an approximation. A binary fraction can only represent sums of $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$ exactly; a value such as 0.1 or $\frac{1}{3}$ has an infinite binary expansion, and the mantissa has a fixed number of bits, so the stored value is the nearest one that fits. The difference is a **rounding error**; it is small for one number but **accumulates** over repeated calculations (adding 0.1 ten times may not give exactly 1), which is why real numbers should never be tested for exact equality.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
user-defined data type	a data type defined by the programmer, based on existing types, to represent data specific to the problem
non-composite type	a type defined without reference to another type; it holds a single value (integer, real, enumerated, pointer)
composite type	a type made up of other types; it holds several values under one identifier (record, set, array, class)
enumerated type	a non-composite type defined by listing all its possible values, in order
pointer type	a non-composite type whose value is the memory address of a variable of a given type
set	a composite type holding a collection of values of one type, unordered and without duplicates
record	a composite type with a fixed number of fields, each with its own identifier and type, accessed by dot notation
class	a composite type combining attributes (data) with the methods (procedures and functions) that act on them; an object is an instance of a class
serial file	records stored one after another in the order in which they were added
sequential file	records stored one after another in order of a key field
random file	records stored at addresses calculated from their keys by a hashing algorithm
sequential access	reading the records in turn from the start of the file until the one required is found
direct access	calculating the address of a record from its key and going straight to that position
hashing algorithm	a calculation on the key of a record that gives the address at which the record is stored and found
collision	two different keys producing the same address
mantissa	the part of a floating-point number that holds its significant bits, as a two's-complement fraction

exercises.lol exercises.com.cn

Term	Definition
exponent	the two's-complement integer giving the power of two by which the mantissa is multiplied
normalised	a floating-point number whose mantissa begins 01 (positive) or 10 (negative), so no bits are wasted on leading zeros or ones
overflow	a result too large to be represented in the number of bits available
underflow	a non-zero result too small to be represented, so it is stored as zero
rounding error	the difference between a real number and the nearest value that the binary representation can hold

Exam tips

- Pseudocode declarations are marked line by line: `TYPE ... = (...)` for enumerated, `TYPE ... = ^...` for pointer, `TYPE ... = SET OF ...` then `DEFINE ... (...) : ...` for a set, `TYPE ... DECLARE ... ENDTYPE` for a record, `CLASS ... PRIVATE ... PUBLIC PROCEDURE NEW ... ENDCLASS` for a class.
- Match the type to the data: fixed named values, enumerated; a group of different fields, record; a collection of unique values, set; data plus behaviour, class; an address, pointer.
- File organisation is how records are **stored**; file access is how they are **found**. Serial and sequential are read sequentially; random files use direct access via a hash of the key. Sequential search of a **sequential** file can stop early; of a serial file it cannot.
- Random-file pseudocode: `OPENFILE ... FOR RANDOM`, `SEEK` before every `GETRECORD` or `PUTRECORD`, `CLOSEFILE` at the end. Say how a collision is resolved when you describe hashing.
- Floating point: mantissa as a two's-complement fraction (point after the sign bit), exponent as an integer, multiply by 2^{exponent}; shift left and subtract one from the exponent to normalise; the mantissa buys precision, the exponent buys range.
- The three "explain" stock answers: why normalise (precision, unique form, range), the effect of re-allocating bits (precision against range), and why 0.1 cannot be stored exactly (an infinite binary fraction in a finite mantissa).

Common mistakes

- Writing `DECLARE` instead of `TYPE` for a new type, or leaving out `ENDTYPE`; declaring a set without `SET OF`, or an enumerated type with quotation marks round its values.
- Putting the sign of a floating-point number in the exponent; the sign is the first bit of the mantissa.
- Reading a negative mantissa as if it were sign-and-magnitude; it is two's complement, so `1011000000` is -0.625, not -0.375.
- Shifting the mantissa to normalise without changing the exponent, or changing it the wrong way (shift left, exponent down).
- Describing a random file as "in random order"; the records are at addresses computed from their keys.
- Saying sequential access reads "the whole file" for a sequential file; it stops when a larger key is met.
- Explaining hashing without saying what the calculated value is used for (the address to store and retrieve the record), or without a way of handling collisions.
- Defining overflow as "too many digits" instead of a result beyond the largest representable value, or blaming the mantissa for it.

exercises.lol exercises.com.cn

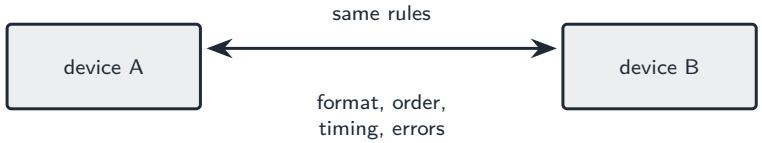
Communication and internet technologies

A-Level Computer Science

Why protocols are needed

A **protocol** 协议 is a **set of rules** for how devices **communicate**. Both ends must follow the same rules, or one side's signals are meaningless to the other. Protocols define the **format** of the data (where addresses and payload sit), the **order** of messages (who speaks first, when to acknowledge), the **meaning** of each message, the **timing** (timeouts, retransmits), and **what to do on error**. Without an agreed protocol, communication fails – like two people speaking different languages with no translator.

"Explain why protocols are essential for communication between computers" (three marks). (1) A protocol is a **set of rules** agreed by both the sender and the receiver; (2) without it the two computers would **interpret the data differently** (format, order, meaning of each part), so the message could not be understood; (3) it allows computers of **different types and manufacturers** to communicate, because everyone implements the same standard. Mention what the rules cover: the format of the data, the order of messages, error detection and recovery, and speed or timing.



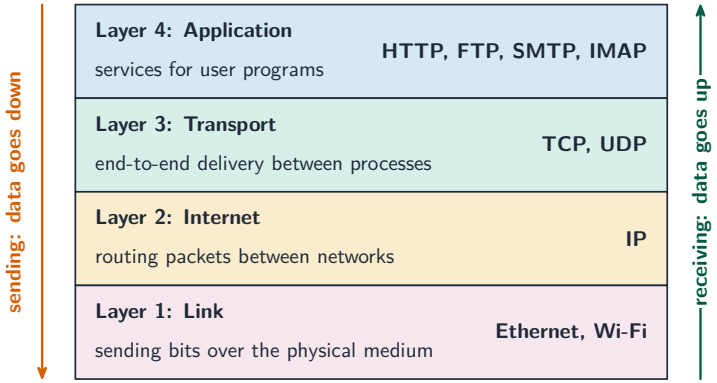
A protocol is the shared rules: format, order, timing and errors

Layered protocols

Networking is complex, so it is split into **layers** 层, each with one focused job, talking only to the layer above and below. Benefits: **modularity** 模块化 (replace one layer – say Ethernet with Wi-Fi – without touching the others), standardisation (vendors interoperate), and **abstraction** 抽象 (you ignore details handled elsewhere). The internet uses the **TCP/IP protocol suite** 协议栈 (4 layers).

TCP/IP protocol suite

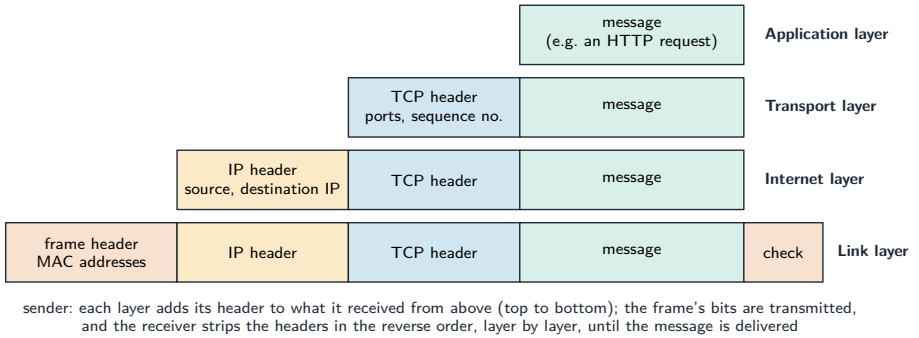
Layer	Purpose	Examples
Application	what the user program does	HTTP, FTP, SMTP, IMAP
Transport	end-to-end delivery between processes	TCP, UDP
Internet	routing packets between networks	IP
Link	sending bits over the physical medium	Ethernet, Wi-Fi



The four layers of the TCP/IP protocol suite

The purpose of each layer, as the mark scheme words it. *Application layer:* provides the **protocols that user applications use** (HTTP for the web, SMTP for email) and the interface between the application and the network; it produces the data to be sent and passes it to the transport layer. *Transport layer:* establishes the **end-to-end connection**, **splits** the data into packets (segments) and adds **port numbers** and **sequence numbers**; on receipt it **reassembles** the packets in order and requests any that are missing (TCP), or sends without those guarantees (UDP). *Internet layer:* adds the **source and destination IP addresses** to form IP packets (datagrams) and **routes** them across networks via routers; it does not guarantee delivery. *Link layer:* adds the **MAC addresses** and error-check bits to form a **frame** and transmits the bits over the **physical** local network (Ethernet or Wi-Fi) through the network interface card. "Complete the stack" means these four, in this order, from the top: Application, Transport, Internet, Link.

exercises.lol exercises.com.cn



Encapsulation: each layer adds its own header 首部 to what it receives from the layer above, so the bits on the wire carry four sets of information; the receiver removes them one layer at a time

"Describe how the TCP/IP suite is applied when a message is sent from one host to another" (five marks). At the **sender** the message passes **down** the stack: (1) the application layer produces the data using a protocol such as HTTP or SMTP; (2) the transport layer splits it into packets and adds a header with the port numbers and a sequence number; (3) the internet layer adds a header with the source and destination IP addresses and chooses the route; (4) the link layer adds the MAC addresses of the next device and sends the frame over the physical link. Routers along the way read the internet-layer header and forward each packet. At the **receiver** the frame passes **up** the stack: each layer removes and acts on its own header, the transport layer reassembles the packets in sequence-number order and asks for any that are missing, and the application layer presents the message. The same protocol at each layer at both ends is what makes the exchange work.

■ Application layer

The **application layer** 应用层 gives services to user programs and defines the protocols they speak (HTTP for web, SMTP for email). This is where a programmer most often works.

■ Transport layer

The **transport layer** 传输层 delivers data **end-to-end** between processes, identified by **port numbers** 端口号. Two protocols:



TCP connects and delivers in order; UDP sends and forgets

- **TCP** 传输控制协议 – **connection-oriented** 面向连接: sets up a connection, ensures **all data arrives in order**, retransmits lost **packets** 数据包, controls flow. Reliable but with overhead. Used by HTTP, HTTPS, SMTP, FTP.
- **UDP** 用户数据报协议 – **connectionless** 无连接: sends and forgets, with no acknowledgements or ordering. Low overhead, no guarantees. Used for streaming, DNS and gaming, where speed beats reliability.

Internet layer

The **internet layer** 网络层 carries packets between hosts using **IP**. Each packet has a source and destination **IP address** IP 地址, and **routers** 路由器 forward it onward. It does not guarantee delivery – that is TCP's job.

A home router does this job for your house: it reads each packet's destination address and sends it on towards the internet, and back to the right device.



A home Wi-Fi router: it forwards packets between your devices and the internet

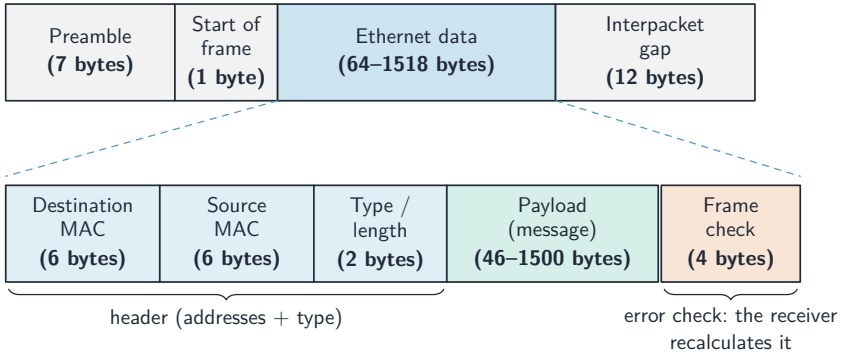
Before the router reaches the wider internet, a **modem** 调制解调器 connects the home to the internet provider over the provider's cable or phone line. Its lights show the link is up and online.



A cable modem connects a home network to the internet provider

Link layer

The **link layer** 链路层 sends bits over one physical link (Ethernet, Wi-Fi). It adds a frame header with **MAC addresses** MAC 地址 and handles medium access (e.g. **CSMA/CD** 载波侦听多路访问/冲突检测 on Ethernet).



The parts of a typical Ethernet frame

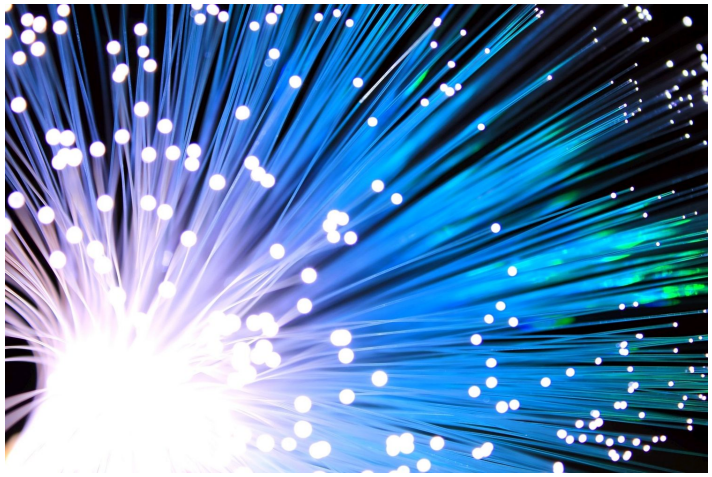
On a wired local network, a **switch** 交换机 joins many devices together. Each device plugs into a port with an Ethernet cable (an RJ45 plug), and the switch uses the MAC addresses in each frame to send it only to the correct port.

exercises.lol exercises.com.cn



A network switch connects many wired devices on a local network

The physical link can be a copper wire, a radio signal (Wi-Fi), or a **fibre-optic cable** 光纤. In a fibre-optic cable, the bits travel as flashes of light through very thin strands of glass, which is fast and carries data a long way.



A fibre-optic cable: data travels as light through thin glass strands

A radio link can reach much further. A **satellite dish** 卫星天线 sends and receives radio signals to and from a satellite, carrying data to places that wired links cannot easily reach.

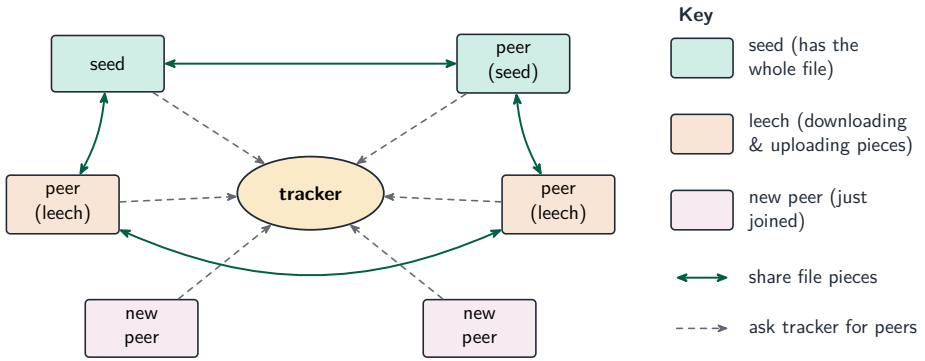
exercises.lol exercises.com.cn



A satellite dish sends and receives data by radio over a long distance

Common application-layer protocols

- **HTTP** 超文本传输协议 – browsers fetch web pages from servers (over TCP, port 80). **HTTPS** is HTTP over TLS – encrypted, port 443.
- **FTP** 文件传输协议 – transfer files between client and server.
- **SMTP** 简单邮件传输协议 – **send** email between client and server, and between servers. Receiving uses POP3 or IMAP.
- **POP3** – downloads email and usually deletes it from the server. **IMAP** – leaves email on the server and syncs across devices, so the same inbox appears everywhere.
- **BitTorrent** – a **peer-to-peer** 对等网络 protocol; a file is split into pieces downloaded from many peers in parallel, so no single server carries all the load.



BitTorrent: a tracker helps peers find each other, then they share file pieces directly

The purpose of each protocol, in the words that score.

Protocol	Purpose (state this)
HTTP	transfers web pages (hypertext) between a web server and a browser; HTTPS is the encrypted version
FTP	transfers files between a client and a server (uploading to and downloading from a file server)
SMTP	sends email from a client to a mail server, and between mail servers (a "push" protocol)
POP3	downloads email from the server to the client, usually deleting it from the server, so it is read on one device
IMAP	lets the client read and manage email that stays on the server, so the same mailbox is seen on every device
BitTorrent	shares files peer-to-peer: pieces of a file are downloaded from, and uploaded to, many other users at once

Asked for the two email protocols, give **SMTP** for sending and **POP3 or IMAP** for receiving; asked to describe IMAP, say that the messages **remain on the server** and are **synchronised** across devices, which is the difference from POP3.

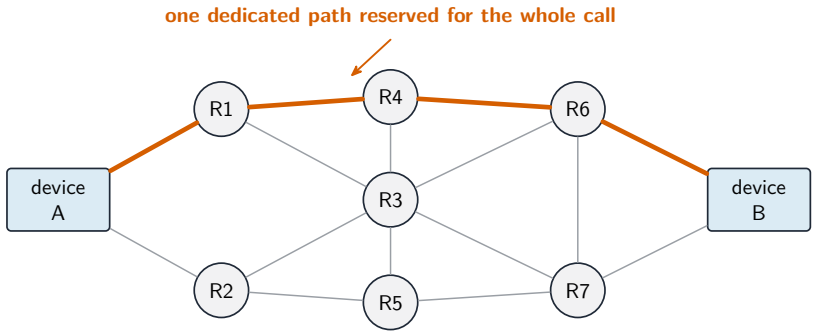
"Describe how files are shared using the BitTorrent protocol" (four marks). (1) The file is split into **pieces** (typically 256 KB each), and a small **torrent** file describes them (their hashes) and names a **tracker** 追踪器. (2) A peer wanting the file contacts the tracker, which keeps a list of the peers in the **swarm** 群 currently sharing that file. (3) The peer downloads **different pieces from many peers at the same time**, and as soon as it holds a piece it **uploads** it to others; a peer with the whole file is a **seed** 种子, one still downloading a leech. (4) When all pieces are in, they are reassembled and checked against the hashes. **"Explain what peer-to-peer file sharing means":** there is **no central server** holding the file; every computer is both a **client and a server**, downloading from and uploading to the others, so the load and the bandwidth are spread across the swarm and the more peers there are, the faster it gets.

Circuit switching vs packet switching

Circuit switching

A **dedicated path** is set up between the two ends before any data is sent (**circuit switching** 电路交换), reserved for the whole conversation, then released. It gives **reserved bandwidth** 带宽 and in-order delivery, but is **inefficient** during silences and slow to set up. Classic example: the traditional telephone network.

exercises.lol exercises.com.cn



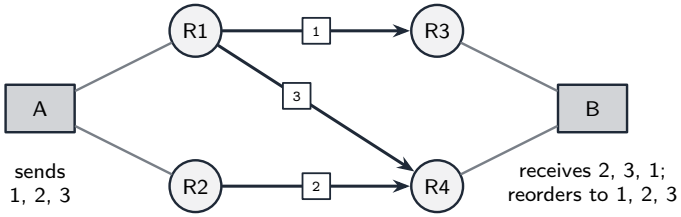
Circuit switching: one dedicated path is reserved end to end

"Describe circuit switching as a method of data transmission" (three marks). (1) A **dedicated** path (circuit) is **set up** between the sender and the receiver **before** any data is sent; (2) the **whole** message is sent along that path, in order, as one continuous stream; (3) the circuit is reserved for the **duration** of the communication and released afterwards.

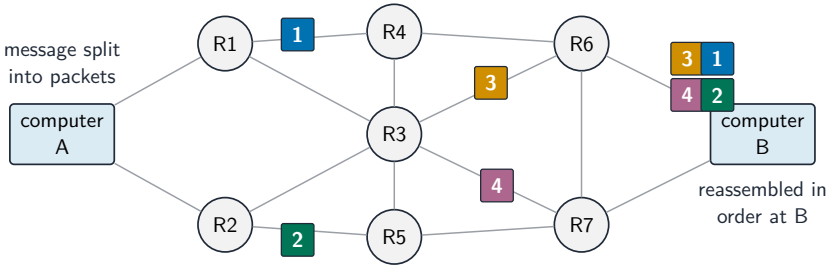
Benefits and drawbacks. *Benefits:* the full **bandwidth** of the circuit is available and guaranteed; data arrives **in order** with no reassembly and no delay once the circuit is up; the route does not change, so timing is predictable (good for real-time voice and video). *Drawbacks:* time is spent **setting up** the circuit before anything is sent; the circuit is reserved even while **no data** is flowing, so bandwidth is wasted and other users cannot share it; both ends must be free at the same time; a **failure anywhere** on the path breaks the whole call, and there is no automatic alternative route. *Where it is appropriate:* a telephone call or a live video link, where a steady, uninterrupted stream matters more than efficiency.

Packet switching

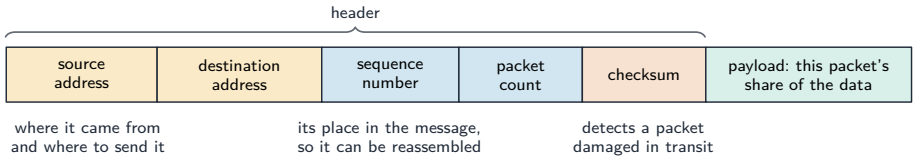
The data is split into **packets**, each sent **independently** (**packet switching** 分组交换). Each packet carries the destination address; routers make per-packet decisions, so packets may take different routes and arrive out of order, and the destination reassembles them. It is **efficient** (one link is **multiplexed** 多路复用 across many conversations), **robust** (reroute around a failure), but has **variable latency** 延迟 and possible loss (TCP handles reliability). Used by the internet.



Packets can take different routes and arrive out of order



Packet switching: packets travel independently and may take different routes



What lets a packet travel on its own: the addresses say where, the sequence number says which part of the message it is, and the checksum shows whether it arrived undamaged

"Describe how packet switching is used to pass messages across a network" (four marks). (1) The message is **split into packets** of a fixed maximum size; (2) each packet is given a **header** containing the **source and destination addresses**, a **sequence number** 序号 and an error check; (3) each packet is sent **independently** and may take a **different route**, chosen by the routers it meets; (4) at the destination the packets are **reassembled in order** using the sequence numbers, and any missing packet is requested again. If the question excludes checking and resending, leave out the last clause.

"Describe the function of a router in packet switching" (three marks). A router **receives** a packet, **reads the destination IP address** in its header, and

exercises.lol exercises.com.cn

consults its **routing table** 路由表 to decide the **best next hop** towards that destination, taking account of the traffic (congestion) and failed links; it then **forwards** the packet onto that link. Packets of the same message may leave by different routes; the router holds packets in a queue when a link is busy.

"Describe two ways packet switching ensures the complete message is received." (1) Each packet carries a **sequence number**, so the receiver can put the packets in order and can tell that one is **missing**, and (2) the receiver sends an **acknowledgement** 确认 for packets that arrive; a packet not acknowledged within a time limit is **retransmitted** by the sender. A **checksum** 校验和 in each packet lets the receiver detect a corrupted packet and discard it, which then triggers the resend.

Benefits and drawbacks. *Benefits:* no circuit to set up; the network's links are **shared** by many messages, so bandwidth is used efficiently; packets can be **rerouted** around a failed or congested link, so transmission is robust; a lost or damaged packet means resending only **that packet**, not the whole message. *Drawbacks:* packets may arrive **out of order** and must be reassembled, and some may be **lost** or delayed; the **headers** add overhead; the variable delay makes it less suitable for real-time voice and video without extra measures; a heavily loaded network drops packets. *Where it is appropriate:* email, web pages, file downloads and any "bursty" traffic, and the internet in general.

Aspect	Circuit switching	Packet switching
Path	dedicated, reserved	shared, per-packet
Setup time	slow	none
Bandwidth use	inefficient	efficient
Order	in order	may be out of order
Robustness	one failure cuts the circuit	reroute around failures
Suits	constant-rate flows (voice)	bursty flows (web, email)

Modern networks use packet switching for its efficiency and resilience.

Four differences, stated as pairs. (1) Circuit switching sets up a **dedicated path** before sending; packet switching sends **without setting up** a path. (2) In circuit switching the whole message follows **one route**; in packet switching the packets may take **different routes**. (3) Circuit switching delivers the data **in order** without reassembly; packet switching needs **sequence numbers** to reassemble it. (4) Circuit switching **reserves** bandwidth for one conversation even when idle; packet switching **shares** the links between many messages. (Also acceptable: a failed link breaks a circuit but packets are rerouted; circuit switching suits real-time streams, packet switching suits bursty data.) Write each difference as **both** halves; one side alone earns nothing.

Describing packet switching in a few sentences

A good exam answer: "The message is broken into small packets. Each packet carries the destination and source addresses and a sequence number. Each packet travels through the network independently, with routers choosing the next hop per packet. Packets may take different paths and arrive out of order. The destination uses the sequence numbers to reassemble the message, and missing packets can be requested again."

> **Worked example.** A phone call and a large file download share a network. Which switching method suits each, and why? A **phone call** needs a steady stream with low delay, and it would suffer badly if pieces arrived late or out of order - so **circuit switching** suits it: a dedicated path is set up for the whole call and its capacity is reserved for the duration. A **file download** does not care about timing or arrival order, because the receiver reassembles it, and it benefits from using whatever capacity happens to be spare - so **packet switching** suits it: the file is split into packets that travel **independently**, each carrying source and destination addresses and a **sequence number**, with routers choosing a next hop per packet. Name the **property of the traffic** that decides it: reserved capacity and low delay for the call, efficiency and resilience for the download.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
protocol	a set of rules governing how data is transmitted, agreed by sender and receiver so that both interpret it the same way
protocol stack	the layers of protocols, each with its own function, that together carry out communication; each layer communicates only with the layers above and below
application layer	provides the protocols used by applications to exchange data (HTTP, SMTP, FTP, IMAP, POP3)
transport layer	establishes end-to-end communication, splits data into packets with port and sequence numbers, reassembles them and requests missing ones (TCP), or sends without guarantees (UDP)
internet layer	adds IP addresses to form packets and routes them between networks via routers
link layer	adds MAC addresses to form frames and transmits the bits over the physical local network

exercises.lol exercises.com.cn

Term	Definition
router	a device that reads a packet's destination address and forwards it along the best available route towards that destination
circuit switching	a dedicated communication path is established between the two ends before data is sent and held for the whole transmission
packet switching	the message is split into packets, each with a header, sent independently over possibly different routes and reassembled at the destination
packet	a unit of data carrying a header (addresses, sequence number, error check) and a payload
peer-to-peer	file sharing without a central server, each computer acting as both client and server

Exam tips

- Why protocols: shared rules, same interpretation, any make of computer. Why layers: each layer has one job and can be changed independently.
- The four layers in order, top to bottom: Application, Transport, Internet, Link. Give each layer's job in one sentence, and the "message from host to host" answer as a walk down the stack and back up.
- Protocol purposes are one-liners: HTTP web pages, FTP files, SMTP sending mail, POP3 downloading mail, IMAP mail kept on the server, BitTorrent peer-to-peer pieces from a swarm.
- Circuit switching: dedicated path first, whole message, held for the duration. Packet switching: split, header with addresses and sequence number, independent routes, reassemble. Benefits and drawbacks come in pairs of opposites.
- A router reads the destination address, consults its routing table, forwards along the best route; it is the packet-switching question the exam asks most.
- "Where appropriate": circuit switching for a phone or live video call; packet switching for email, the web and downloads.

Common mistakes

- Defining a protocol as "a language" or "software"; it is a set of rules.
- Putting the layers in the wrong order, or giving the OSI seven layers instead of the four of TCP/IP.
- Describing the transport layer as "routing" or the internet layer as "splitting into packets"; ports and splitting are transport, IP addresses and routing are internet.

- Confusing POP3 with IMAP, or saying SMTP receives email.
- Describing packet switching without the header (addresses and sequence number) or without reassembly.
- Saying a router "sends the packet everywhere"; it chooses one next hop from its routing table.
- Giving a benefit of packet switching as a drawback of circuit switching without stating the circuit-switching side; each difference needs both halves.
- Claiming packet switching guarantees delivery by itself; the transport layer's sequence numbers and acknowledgements do that.

exercises.lol exercises.com.cn

Hardware and Virtual Machines

A-Level Computer Science

RISC vs CISC processors

Two styles of CPU design. The CPU itself plugs into the **motherboard** 主板, the main board that links the processor, the memory and every other part of the computer together.

CISC
many, complex instructions
variable length
does more per instruction

RISC
few, simple instructions
fixed length
each instruction is fast

CISC has many complex instructions; RISC has few simple ones



A motherboard links the CPU, memory and other parts together

CISC

A **CISC** 复杂指令集 (Complex Instruction Set Computers) has **many, often complex** instructions (one may do several memory accesses and operations), of **variable length**, so decoding is intricate. It does more per instruction in hardware. Examples: Intel x86.

RISC

A **RISC** 精简指令集 (Reduced Instruction Set Computers) has a **small set of simple** instructions, each doing one basic operation, all of **fixed length** (fast to decode). Only **load** and **store** touch memory; everything else is **register** 寄存器 to register. Programs are longer but each instruction is quick and predictable, which suits pipelining. Examples: ARM, RISC-V.

Feature	CISC	RISC
Instruction set	many	few
Instruction length	variable	fixed
Memory access	many instructions	only load/store
Pipeline-friendly	harder	naturally
Per-instruction cycles	varies	usually 1

The trade-off is doing **more per instruction** (CISC) vs doing each instruction **faster and more predictably** (RISC). Modern Intel chips translate CISC instructions into simpler RISC-like micro-ops internally.

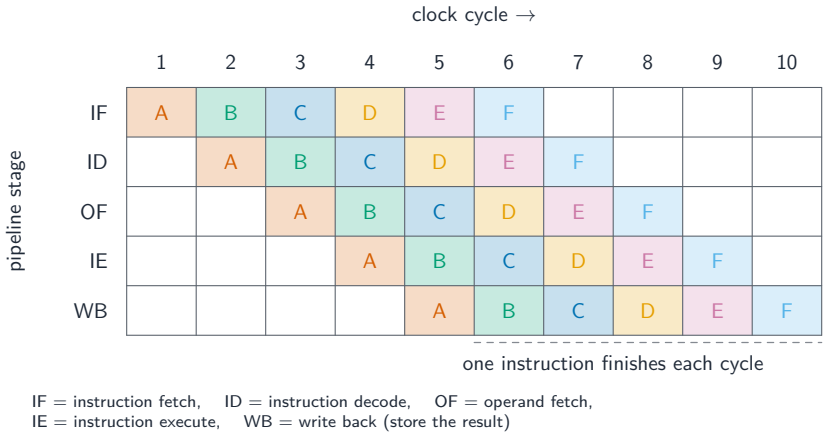
"Identify four features of a RISC processor." Any four of: a **small** set of **simple** instructions; instructions of **fixed length** (one word); most instructions complete in **one clock cycle**; **many** general-purpose registers; only **load and store** instructions access memory (all arithmetic is register to register); **hard-wired** control (no microcode); designed for **pipelining**; the **compiler** does more of the work, so programs contain more instructions and need more memory. **"Identify four features of a CISC processor."** Any four of: a **large** set of instructions, many of them **complex** (one instruction may do several operations); instructions of **variable length**; instructions that take **several clock cycles**; **fewer** registers; instructions that can **access memory directly**; **microprogrammed** control; less suited to pipelining; **shorter** programs, so a simpler compiler and less memory. **"Describe what is meant by RISC and CISC" (two marks each):** name the expansion and give the defining idea (few simple single-cycle instructions; many complex multi-cycle instructions).

Interrupt handling on the two designs. On a **CISC** processor the current instruction, however complex, is **completed** before the interrupt is serviced; the processor then saves the contents of its registers (including the program counter) on the stack, jumps to the interrupt service routine, and restores the registers afterwards. On a **RISC** processor with a **pipeline**, several instructions are part-way through at the moment the **interrupt** 中断 arrives, so the processor must either let every instruction in the pipeline finish, or **discard** (flush) the partly executed instructions and restart them after the interrupt; either way the pipeline is emptied, the registers are saved, and the service routine runs. The exam phrasing: "pipelining makes interrupt handling more complex, because the contents of the pipeline must be dealt with before the interrupt can be serviced".

exercises.lol exercises.com.cn

Pipelining

A **pipeline** 流水线 processes instructions in **overlapping stages**, like an assembly line. Cambridge uses five stages: **instruction fetch (IF)** → **instruction decode (ID)** → **operand fetch (OF)** → **instruction execute (IE)**, often in the **ALU** 算术逻辑单元 → **write back (WB)**, which stores the result. Each stage works on a different instruction at once, so once the pipeline is full, **one instruction completes per cycle**. RISC's fixed-length, simple instructions make every stage take the same time. A pipeline can stall on a **hazard** 冒险 – a data hazard (an instruction needs a result not ready yet) or a control hazard (a branch makes the next address unknown).



Pipelining overlaps the stages of six instructions, so one finishes each cycle

RISC chips keep data in **many registers** because memory is slow and registers are fast; the compiler allocates values to registers wisely.

"Describe the use of pipelining in RISC processors" (three marks). (1) The **fetch–execute cycle is divided into stages** (instruction fetch, decode, operand fetch, execute, write back); (2) **several instructions** are in the pipeline at once, each at a **different stage**, so while one is being executed the next is being decoded and the one after fetched; (3) a new instruction is started, and one completed, in **every clock cycle** once the pipeline is full, which increases **throughput** 吞吐量 (the number of instructions completed per second), although each instruction still takes the same time on its own. Fixed-length single-cycle RISC instructions are what make the stages equal and the pipeline possible.

> **Worked example.** A processor uses five pipeline stages (IF, ID, OF, IE, WB). Four instructions enter the pipeline one after another. In which cycle does the last instruction complete, and how many cycles would the four take without pipelining?

Instruction 1 occupies IF in cycle 1, ID in 2, OF in 3, IE in 4 and WB in 5; instruction

2 starts one cycle later and finishes in cycle 6; instruction 3 in cycle 7; instruction 4 in cycle **8**. In general n instructions through k stages take $n + k - 1$ cycles, here $4 + 5 - 1 = 8$. Without pipelining each instruction takes all five cycles before the next starts: $4 \times 5 = 20$ cycles. The exam's table is filled by writing each instruction's stages diagonally, one column to the right of the previous instruction.

A processor running this fast gives off a lot of heat, so a **heat-sink** 散热器 and fan sit on top of it. The metal fins spread the heat and the fan blows it away, keeping the CPU cool enough to work.



A CPU heat-sink and fan carry heat away from the processor

Flynn's taxonomy

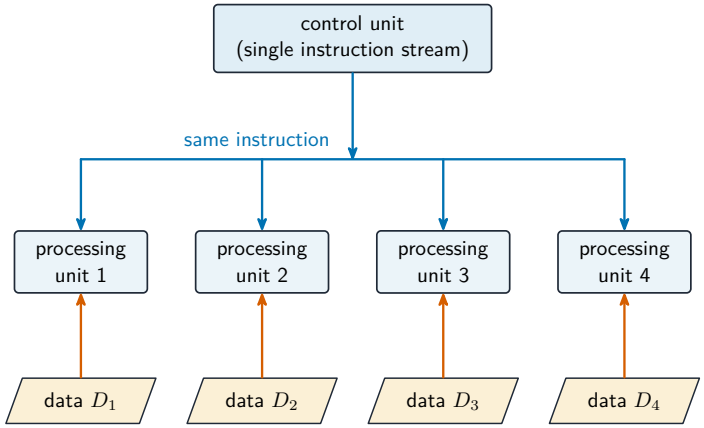
Flynn's taxonomy 弗林分类 sorts computers by the number of instruction and data streams:

- **SISD** – one instruction, one data stream (a traditional single core).
- **SIMD** 单指令多数据 – one instruction works on **many data items at once** (GPUs, CPU vector extensions). Great for images, video, scientific arrays.
- **MISD** – several operations on the same data; rare, mostly theoretical.
- **MIMD** 多指令多数据 – many processors run **different instructions on different data** (multi-core CPUs, clusters). The most general.

Describing the four architectures (two marks each). *SISD:* a **single processor** executes **one instruction at a time** on **one item of data**; no parallelism, the traditional von Neumann machine. *SIMD:* **one instruction** is applied simultaneously to **many data items**, by many processing elements acting in step; used for array and graphics processing. *MISD:* **several processors** apply **different instructions** to the **same data**; rarely used, for example a fault-tolerant system where several processors

exercises.lol exercises.com.cn

check one stream. *MIMD:* **many processors**, each executing **its own instructions** on **its own data**, independently; the multi-core computer and the cluster.

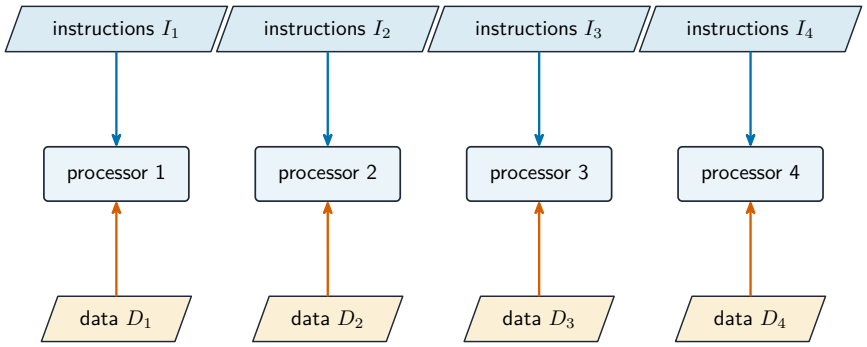


SIMD: many processors run the same instruction on different data

A **graphics card** 显卡 (with its GPU) is a real example of SIMD hardware: it has thousands of small cores that run the same instruction on many pixels or numbers at once, which is why GPUs are so fast for images, video and machine learning.



A graphics card: its GPU runs the same instruction on many data items at once (SIMD)



MIMD: each processor runs its own instructions on its own data

Massively parallel computers

A **massively parallel** 大规模并行 system uses **thousands of processors** on a fast network, each with its own memory (**distributed memory** 分布式内存), exchanging data by messages. It is MIMD, needs specially-written software (MPI, CUDA), and suits climate simulation, large **machine learning** 机器学习 training, and astrophysics. The largest **supercomputers** 超级计算机 are massively parallel.

"Outline the characteristics of massively parallel computers" (three marks). A **very large number of processors** (thousands), **each with its own memory**, connected by a **network** (a high-speed interconnect or bus) so that they can pass **messages** to one another; they work **simultaneously** on parts of the **same problem**, so the problem must be written as a program that can be split into parts that run in parallel and combine their results. It is an MIMD arrangement.

The processors live in tall **server** 服务器 racks, often filling a whole room (a **data centre** 数据中心), wired together so they can work on one big problem at the same time.



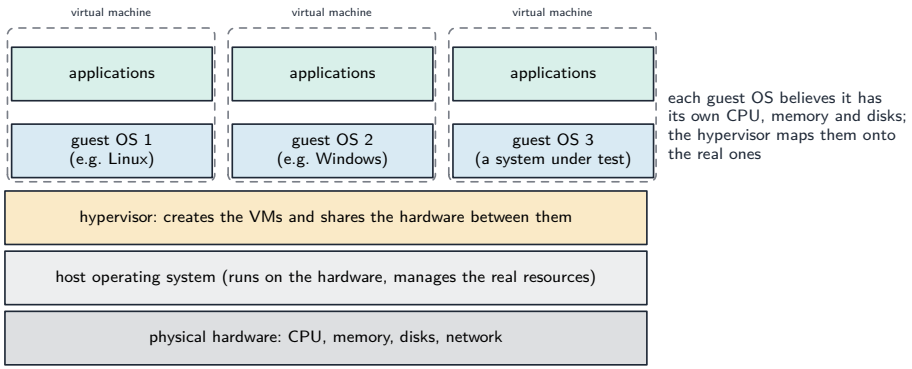
Rows of servers in a data centre, like those used for massively parallel computing

Virtual machines

A **virtual machine** 虚拟机 (VM) is a **software emulation of a whole computer** – the software inside sees a CPU, memory and disks that look real but are managed by host software.

- a **system VM** runs a complete OS. A **hypervisor** 虚拟机监控器 creates and manages VMs, each booting its own guest OS. Uses: run different OSes on one machine; server consolidation; **sandboxing** 沙箱 (risky software runs isolated); snapshots.
- a **process (language) VM** runs one program in portable **bytecode** 字节码 – the JVM (Java), the CLR (.NET), CPython. Benefits: portability ("write once, run anywhere"), runtime safety checks, and **just-in-time compilation** 即时编译 for near-native speed. The cost is an extra layer and needing the VM installed.

exercises.lol exercises.com.cn



One real computer, several apparent ones: the host operating system and hypervisor share the hardware, and each guest operating system runs as if it had a machine of its own

"Describe what is meant by a virtual machine" (two marks). *A software emulation (implementation) of a computer system that runs on a host computer and behaves, to the programs running inside it, like a separate physical computer with its own processor, memory and storage.* The **host operating system** 宿主操作系统 runs on the actual hardware, manages the real resources and (through the hypervisor) creates and controls the virtual machines; each **guest operating system** 客户操作系统 runs inside a virtual machine, manages the applications in it, and is unaware that its hardware is virtual.

Benefits (give two). Several **different operating systems** can run on one machine at the same time; software can be **tested** on many systems without buying the hardware; a **new** computer system can be emulated and tried before it is built; each VM is **isolated**, so a crash or malware in one does not affect the host or the others; VMs can be **copied, moved and backed up** as files, and a server can be shared between many users, **reducing** hardware cost. **Limitations (give two).** A VM runs **more slowly** than the real hardware because every instruction passes through the emulation layer; it consumes the host's **memory and processing power**, so the host must be powerful; some hardware features or devices are **not emulated** exactly, so the tested software may behave differently on the real machine; **licences** are needed for each guest OS, and setting the system up needs expertise.

Boolean algebra

Boolean algebra 布尔代数 simplifies **Boolean** 布尔 expressions, which can equally be described by **truth tables** 真值表. Symbols: + for OR, $\cdot$ for AND (often omitted), an overbar for NOT.

Key laws include commutative, associative and distributive (as in ordinary algebra), plus:

- identity $A+0=A$, $A\cdot 1=A$; null $A+1=1$, $A\cdot 0=0$.
- idempotent $A+A=A$; inverse $A+\overline{A}=1$, $A\cdot\overline{A}=0$.
- **De Morgan's laws** 德摩根定律: $(A+B)'=A'\cdot B'$; $(A\cdot B)'=A'+B'$ – negate the whole, swap AND/OR, negate each operand.
- **absorption** 吸收律: $A+AB=A$.

Simplifying reduces the number of terms, so the resulting logic circuit has fewer gates. Example: $Z=AB+A\overline{B}=A(B+\overline{B})=A$.

The laws with their names (quote the name at each step when "show all working" is asked).

Law	OR form	AND form
identity	$A+0=A$	$A\cdot 1=A$
null (annulment)	$A+1=1$	$A\cdot 0=0$
idempotent	$A+A=A$	$A\cdot A=A$
complement (inverse)	$A+\overline{A}=1$	$A\cdot\overline{A}=0$
commutative	$A+B=B+A$	$A\cdot B=B\cdot A$
associative	$A+(B+C)=(A+B)+C$	$A(BC)=(AB)C$
distributive	$A+BC$ $=(A+B)(A+C)$	$A(B+C)$ $=AB+AC$
absorption	$A+AB=A$	$A(A+B)=A$
De Morgan	$\overline{A+B}=\overline{A}\cdot\overline{B}$	$\overline{A\cdot B}=\overline{A}+\overline{B}$
double negation	$\overline{\overline{A}}=A$	

Worked example. Simplify $X=\overline{\overline{(A\cdot B)}\cdot\overline{(A+B)}}$, showing all working.

$X=\overline{\overline{(A\cdot B)}}+\overline{\overline{(A+B)}}$ (De Morgan on the outer bar) $=A\cdot B+A+B$ (double negation) $=A+B$ (absorption, $A+AB=A$, applied with $A+B$ absorbing AB).

Worked example. Simplify $(\overline{A+B})\cdot(\overline{A}+B)$.

$=\overline{A}\cdot\overline{B}\cdot(\overline{A}+B)$ (De Morgan) $=\overline{A}\,\overline{B}\,\overline{A}+\overline{A}\,\overline{B}\,B$ (distributive) $=\overline{A}\,\overline{B}+0$ (idempotent, complement) $=\overline{A}\,\overline{B}$.

Worked example. Simplify $Y=\overline{A}\,\overline{B}\,\overline{C}+\overline{A}\,\overline{B}\,C+A\,\overline{B}\,C$.

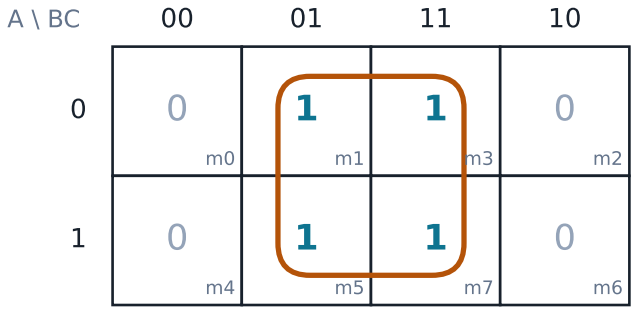
$=\overline{A}\,\overline{B}(\overline{C}+C)+A\,\overline{B}\,C$ (distributive) $=\overline{A}\,\overline{B}+A\,\overline{B}\,C$ (complement, identity) $=\overline{B}(\overline{A}+AC)$ (distributive) $=\overline{B}(\overline{A}+C)$, using $\overline{A}+AC=(\overline{A}+A)(\overline{A}+C)=\overline{A}+C$. Applying De Morgan to a three-input term works the same way: $\overline{A+B+C}=\overline{A}\cdot\overline{B}\cdot\overline{C}$.

Sum-of-products from a truth table. Take every row whose output is 1, write the AND of its inputs (a variable barred where it is 0), and OR the terms: a row with $A=1,B=0,C=1$ gives $A\,\overline{B}\,C$. This is the **sum-of-products** 积之和 form the

exam asks for, and it is the starting point for both algebraic simplification and the Karnaugh map.

Karnaugh maps

A **Karnaugh map** 卡诺图 (K-map) simplifies a Boolean expression by **grouping adjacent 1s** from a truth table. Columns and rows use **Gray code** 格雷码 order (`00`, `01`, `11`, `10`) so adjacent cells differ in one variable.



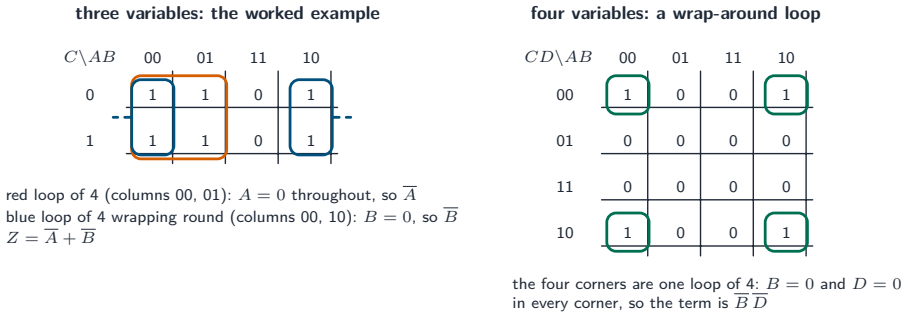
group of four: only C stays 1, so F = C

$F = \Sigma m(1, 3, 5, 7)$: the group of four keeps only C, so $F = C$

Place a 1 in each cell where the output is 1. Find rectangular groups of 1s whose sides are powers of 2 (1, 2, 4, 8), wrapping around edges if it makes a bigger group. **The larger the group, the simpler the term**: a group of 2 drops one variable, a group of 4 drops two, and so on – variables that change within the group disappear. OR the group terms together for the simplified expression. Cover every 1 using as few, as large, groups as possible.

> **Worked example.** A Karnaugh map for A and B has 1s in the cells $\overline{A}B$ and AB. Simplify. The two 1s are **adjacent** - they share the $B = 1$ column - so group them as a rectangle of 2. Inside that group B stays 1 throughout while A **changes** from 0 to 1, and any variable that changes within a group **disappears**. So the group leaves simply $X = B$. Compare that with the sum of products read straight off the table, $\overline{A}B + AB$: the same circuit, two gates fewer. Two rules do most of the work - make each group **as large as possible** (a group of 2 drops one variable, 4 drops two, 8 drops three), and remember the map **wraps around** its edges, so the leftmost and rightmost columns are adjacent. That wrap is the grouping most candidates miss.

exercises.lol exercises.com.cn



Loops of 1, 2, 4 or 8 ones; the term for a loop keeps only the variables that do not change inside it. Edges join, so a loop may wrap round, and the four corners count as adjacent

Building and reading a K-map. Label the columns AB and the rows C (or CD) in Gray-code order `00 01 11 10`, so that neighbouring cells differ in one variable only. Put a 1 in every cell whose minterm appears in the expression (or whose truth-table row outputs 1). Then draw the **fewest, largest** loops that cover every 1: each loop must be a rectangle of $1, 2, 4$ or 8 cells, loops may **overlap**, may **wrap** across the left–right and top–bottom edges, and the four corners together make a loop. For each loop write the variables that are **constant** inside it (barred if 0), and OR the loop terms: that is the optimal sum-of-products. **Why use one?** It gives the simplest expression **without algebra**, in a few steps, with less chance of error, and the same map suits three or four variables.

> **Worked example.** $Z = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + \overline{A}\,B\,C + A\,\overline{B}\,\overline{C} + A\,\overline{B}\,C$.

On the three-variable map the 1s fill columns 00, 01 and 10 in both rows. The loop of four over columns 00 and 01 has $A = 0$ throughout and B, C both varying: term $\overline{A}$. The loop of four over columns 00 and 10 (wrapping round) has $B = 0$ throughout: term $\overline{B}$. So $Z = \overline{A} + \overline{B}$, which Boolean algebra confirms: $\overline{A}(\overline{B} + B) + \ldots = \overline{A} + \overline{B}$. Two loops of two would also be correct but not **optimal**; a loop is as large as the 1s allow.

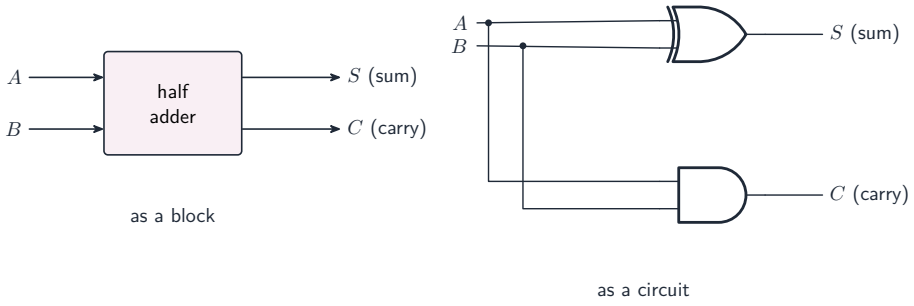
> **Worked example (four variables).** A map has 1s only in its four corners: $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$, $A\,\overline{B}\,\overline{C}\,\overline{D}$, $\overline{A}\,\overline{B}\,C\,\overline{D}$ and $A\,\overline{B}\,C\,\overline{D}$. Because the top and bottom rows are adjacent and so are the outer columns, the corners are one loop of four; $B = 0$ and $D = 0$ in all of them while A and C vary, so $Z = \overline{B}\,\overline{D}$.

Half adder and full adder

A **half adder** 半加器 adds two single bits A and B, giving a sum S and a **carry** 进位 C:

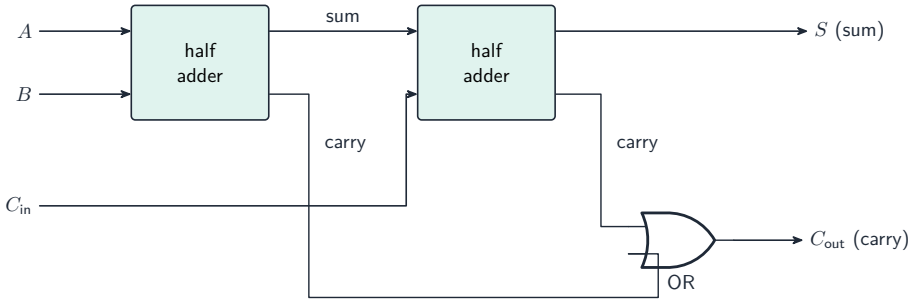
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

So $S = A$ XOR B and $C = A$ AND B. It ignores any carry-in – hence "half".



A half adder, as a block and as a circuit of an XOR and an AND gate

A **full adder** 全加器 adds three bits (A, B, carry-in), giving a sum and a carry-out: $S = A$ XOR B XOR C_{in}. It can be built from two half adders plus an OR gate. Chaining full adders (each carry-out feeding the next carry-in) makes a multi-bit "ripple-carry" adder.



A full adder is built from two half adders and an OR gate

The full-adder truth table. With inputs A, B and the carry-in C_{in}: the sum S is 1 when an **odd** number of inputs is 1, and the carry-out is 1 when **two or more** inputs are 1.

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The circuit questions the exam sets. Given a circuit of an XOR and an AND gate sharing two inputs, or two half adders and an OR gate, "complete the truth table (show your working)" means adding a **column for every intermediate gate output** and filling the rows in order; "state the name of the circuit" is **half adder** or **full adder**; "state the purpose of each output" is the **sum** of the bits and the **carry** to the next column. Sum-of-products for the half adder: $S = \overline{A}B + A\overline{B}$, $C = AB$. A chain of full adders, each passing its carry-out to the next carry-in, adds two multi-bit numbers.

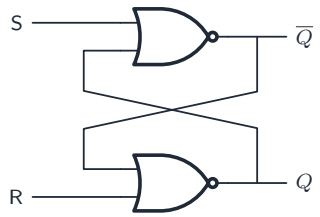
Flip-flops

A **flip-flop** 触发器 is a **bistable** 双稳态 circuit – two stable states (0 and 1) – that **remembers** its state. It stores one bit and is the basic element of registers and SRAM.

■ SR flip-flop

An **SR flip-flop** SR 触发器 has inputs **S** (set) and **R** (reset) and outputs Q and $\overline{Q}$. `S=1,R=0` sets Q to 1; `S=0,R=1` resets it to 0; `S=0,R=0` holds; `S=1,R=1` is **invalid**. Built from two cross-coupled NOR gates.

exercises.lol exercises.com.cn



two NOR gates, each output fed back to the other's input

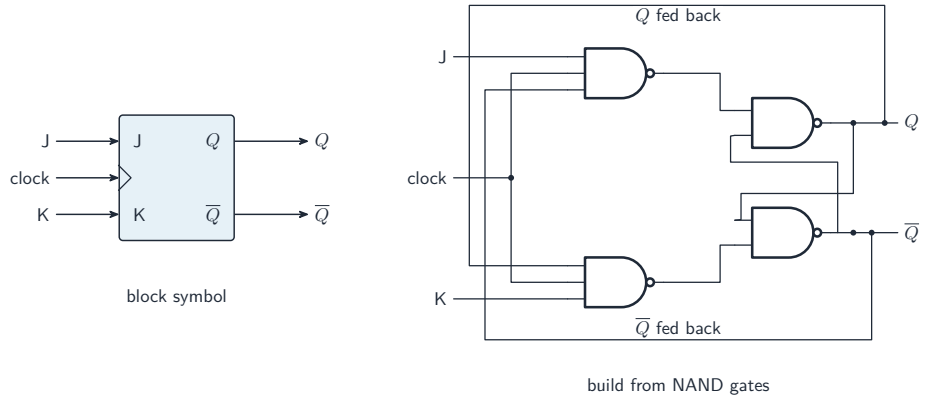
S	R	Q	$\overline{Q}$	effect
0	0	Q	$\overline{Q}$	hold: no change (memory)
1	0	1	0	set Q to 1
0	1	0	1	reset Q to 0
1	1	0	0	invalid: both outputs 0

The SR flip-flop: two NOR gates feeding each other. With both inputs 0 the outputs hold whatever they were, which is the memory; S sets Q to 1, R resets it, and S = R = 1 is not allowed

"Draw a logic circuit for an SR flip-flop and label the inputs." Two **NOR** gates (or two NAND gates), the output of each connected back to one input of the other; the free input of one gate is **S**, of the other **R**; the outputs are Q and $\overline{Q}$. The **feedback** is what the marks are for: without it there is no memory. **"State the purpose of a flip-flop."** *To store one bit of data*; it is the basic memory element from which registers and static RAM are built, and it holds its value until it is deliberately changed. The invalid input $S = R = 1$ makes both outputs 0, so that $\overline{Q}$ is no longer the complement of Q, and the state after both inputs return to 0 is unpredictable, which is the SR flip-flop's weakness.

■ JK flip-flop

A **JK flip-flop** JK 触发器 improves on it by using the previously-invalid `1,1` input as a **toggle** 翻转 (the output flips). This makes it ideal for building **counters** 计数器 (a chain of toggling flip-flops). It is usually **clocked** – inputs act only on a clock edge, keeping flip-flops synchronised.



A JK flip-flop: its symbol and a build from NAND gates

Flip-flops are the building blocks of registers (n bits = n flip-flops), counters, and **SRAM** 静态 RAM cells.

JK flip-flop truth table. The **clock** 时钟 input decides when the J and K inputs are read, so the output changes only on a clock pulse: with $J = K = 0$ the output is **held**; $J = 1, K = 0$ **sets** Q to 1; $J = 0, K = 1$ **resets** it to 0; $J = K = 1$ **toggles** it (Q becomes $\overline{Q}$). The last row is exactly the SR flip-flop's forbidden input turned into a useful one, which is why the JK is preferred: every input combination is valid, and the clocked operation makes it the building block of counters and shift registers.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
RISC	a processor with a small set of simple, fixed-length instructions, most executed in one clock cycle, using many registers and pipelining
CISC	a processor with a large set of complex, variable-length instructions, many taking several clock cycles and accessing memory directly
pipelining	dividing the fetch–execute cycle into stages so that several instructions are processed at once, each at a different stage
SISD / SIMD / MISD / MIMD	one instruction on one data item; one instruction on many data items; many instructions on one data item; many instructions on many data items

exercises.lol exercises.com.cn

Term	Definition
massively parallel computer	thousands of processors, each with its own memory, connected by a network and working simultaneously on one problem
virtual machine	a software emulation of a computer system running on a host computer and behaving like a separate physical computer
hypervisor	the software that creates virtual machines and shares the host's hardware between them
truth table	a table listing every combination of inputs to a logic circuit with the resulting output(s)
sum-of-products	a Boolean expression written as the OR of AND terms, one term for each input combination giving 1
Karnaugh map	a grid of the truth-table outputs, arranged in Gray-code order, in which loops of adjacent 1s give the simplified expression
half adder	a circuit that adds two bits, producing a sum and a carry
full adder	a circuit that adds two bits and a carry-in, producing a sum and a carry-out
flip-flop	a bistable circuit that stores one bit, holding its output until its inputs change it

Exam tips

- RISC and CISC are answered as lists of features: simple, fixed, one cycle, many registers, load/store, pipelined against complex, variable, multi-cycle, fewer registers, direct memory access, microcode. Four of each.
- Pipelining: stages, several instructions at once, one completed per cycle, higher throughput; $n + k - 1$ cycles for n instructions through k stages; interrupts must empty the pipeline.
- Flynn's four categories are "how many instruction streams" by "how many data streams"; say what runs on what. Massively parallel: many processors, own memory, network, same problem.
- Virtual machine: emulation of a computer on a host; host OS on the hardware, hypervisor sharing it, guest OS inside. Two benefits and two limitations, each a full sentence.
- Boolean algebra: name each law as you use it; De Morgan swaps the operator and negates each term; check with a truth table if in doubt.

exercises.lol exercises.com.cn

- K-map: Gray-code order, largest loops of 1/2/4/8, wrapping allowed, one term per loop with the unchanging variables. State why: simplest expression with no algebra.
- Half adder gives sum and carry; full adder also takes a carry-in; SR flip-flop is two cross-coupled NOR/NAND gates and stores one bit; JK's 1,1 input toggles.

Common mistakes

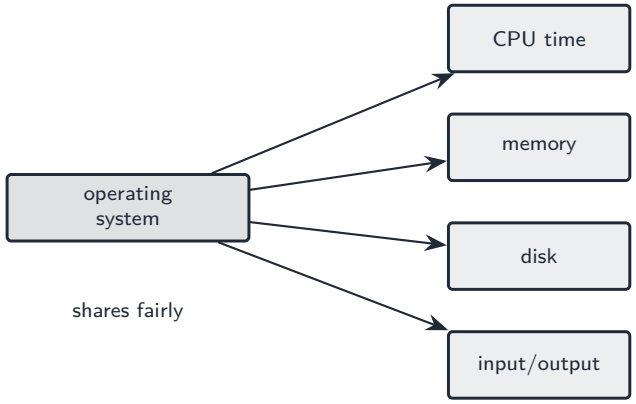
- Swapping the RISC and CISC feature lists, or offering "faster" as a feature; give the design features, not a verdict.
- Describing pipelining as "running instructions in parallel on several cores"; it is stages of one processor overlapping.
- Confusing SIMD (one instruction, many data) with MIMD (many of both), or describing MISD as the common case.
- Defining a virtual machine as "a copy of a computer" without the word emulation or the host and guest.
- Applying De Morgan to only part of an expression under a long bar, or dropping the bar without swapping AND for OR.
- Looping a group of three, or a non-rectangular group, in a K-map; ordering the columns 00, 01, 10, 11 instead of Gray code.
- Writing the carry of a half adder as XOR and the sum as AND.
- Drawing an SR flip-flop as two gates with no feedback, or leaving out the invalid state from its truth table.

System Software

A-Level Computer Science

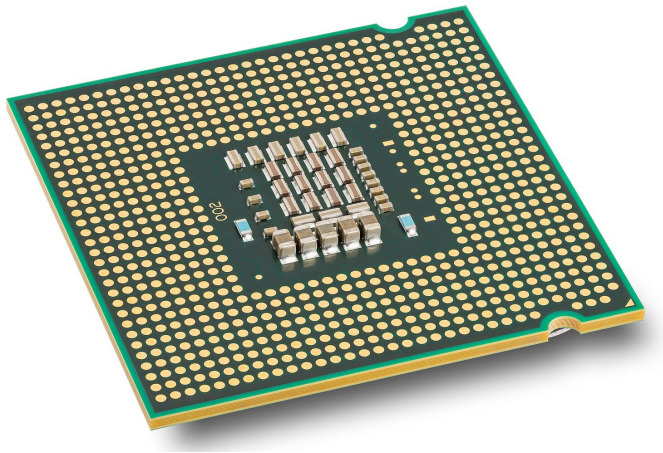
How an OS maximises use of resources

A computer has many resources (CPU time, memory, disk, I/O) and many programs competing for them. The OS **shares them fairly and efficiently** so each is well used and the system stays responsive:



The OS shares the CPU, memory, disk and I/O between programs

- **multi-tasking** 多任务 – switch the CPU quickly between processes so several seem to run at once.
- **memory management** – give each process the memory it needs; use disk **paging** 分页 when **RAM** runs out.
- **spooling** 假脱机 and buffering – print jobs queue on disk so the CPU never waits for the printer.
- **caching** – keep recently-used disk data in **cache** 高速缓存 / RAM.



The processor is a key resource the OS shares between competing tasks

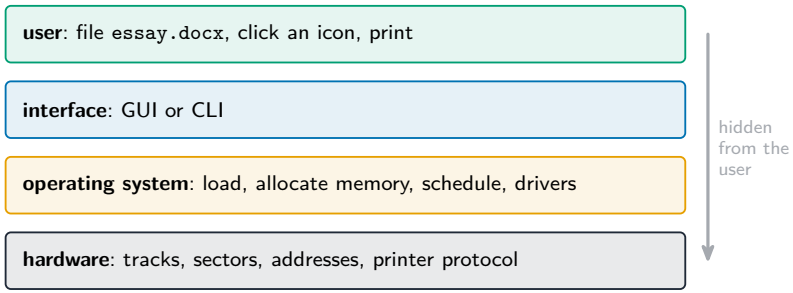


The OS also manages memory (RAM), deciding what to keep in it and what to page out to disk

The user interface

The user interface hides the hardware behind friendly abstractions: the user sees windows, menus and folders, not addresses or sectors. One click on an icon makes the OS find the program on disk, allocate memory, load it and start it. A **CLI** (command line) is powerful and scriptable for experts; a **GUI** (graphical) is easier to learn. Most systems offer both.

exercises.lol exercises.com.cn



The OS hides the hardware behind abstractions.

"Describe two ways in which the complexities of the hardware are hidden from the user." (1) The user works with **files and folders by name**, and the OS translates them into the tracks, sectors and blocks of the disk; (2) the user runs a program with a **click or a command**, and the OS loads it, allocates memory and schedules it without the user knowing any addresses; (3) **device drivers** let the user print or save without knowing how the printer or disk is controlled; (4) a **graphical interface** replaces machine-level commands with icons, windows and menus. The benefit to a student, with an example: the OS makes the hardware **usable without technical knowledge**, for instance saving a document to a USB drive by dragging its icon.

"Show how an OS maximises the use of resources." It **schedules the processor** so that it is never idle while a process is ready; it **manages memory**, allocating it to processes, reclaiming it and extending it with virtual memory; it **manages input and output**, using buffers and spooling so that fast and slow devices overlap their work; and it **manages storage**, keeping track of free space and files. Each point names a resource and what the OS does with it.

Process management

A **process** 进程 is a program in execution – its code, current state, memory and open files.

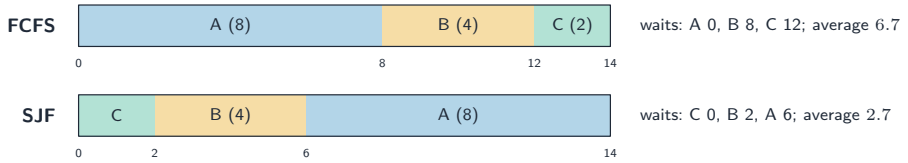
■ Scheduling

The **scheduler** 调度器 chooses which **ready** process runs next, and for how long:

- **round robin** 轮转 – each process gets a fixed **time slice** 时间片, then goes to the back of the queue.
- first-come-first-served; shortest job first; **shortest remaining time** (run the job with the least work left); priority; multilevel feedback queues.

The trade-off is responsiveness vs throughput vs fairness.

"Describe what is meant by multi-tasking and how it benefits process management." *Several processes are held in memory at the same time and the processor switches between them so quickly that they appear to run simultaneously*, each given a share of processor time in turn. The benefit: the processor is **never left idle** while one process waits for input or output, so **throughput** is higher and the user can work on several programs at once. **"Explain the need for scheduling."** There are **more processes than processors**, so a decision must be made about **which** process runs next and **for how long**; scheduling makes sure every process **makes progress**, that the processor is **fully used**, that **response times** are acceptable, and that **priorities** can be respected.



three jobs arriving together with CPU times 8, 4 and 2: the same total work, but running the short jobs first cuts the average waiting time from 6.7 to 2.7 units

The same work in a different order: shortest-job-first gets the short jobs out of the way, so most jobs wait less, at the risk of a long job waiting for ever

The scheduling routines, as the exam wants them described.

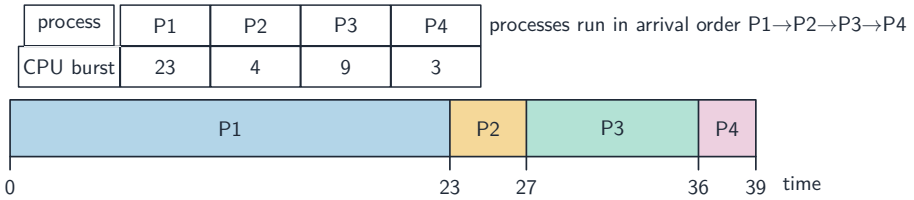
exercises.lol exercises.com.cn

Routine	Function	Benefit	Drawback
first come first served (FCFS)	processes run in the order in which they arrive in the ready queue, each to completion	simple; every process is dealt with in turn, none is starved	a long process holds up all the short ones behind it; poor response
shortest job first (SJF)	the ready process with the **shortest estimated run time** runs next, to completion	minimises the average waiting time; many short jobs finish quickly	run times must be known in advance; a long job may never run (starvation)
shortest remaining time (SRT)	**pre-emptive** 抢占式 version of SJF: if a new process arrives with less time left than the running one, it takes over	short processes are served even faster; good throughput	more context switches; a long job can be interrupted repeatedly and starve
round robin (RR)	each ready process gets a fixed **time slice** in turn; when it expires the process goes to the back of the queue	fair; every process responds within a bounded time, good for interactive use	context-switch overhead; a very short slice wastes time, a long one delays others
priority	the ready process with the highest priority runs first	important or time-critical work is done first	low-priority processes may starve unless priorities age

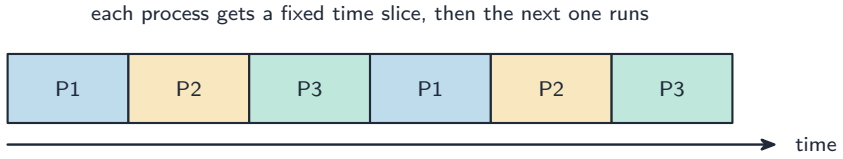
Worked example. Three processes arrive together with CPU times of 8, 4 and 2 ms. Compare the average waiting time under FCFS (in arrival order A, B, C) and under shortest job first.

FCFS: A waits 0, B waits 8, C waits 12; average $(0 + 8 + 12)/3 = 6.7$ ms. SJF runs C, B, A: C waits 0, B waits 2, A waits 6; average 2.7 ms. The total work is the same 14 ms either way; the order decides who waits. Round robin with a 2 ms slice would give A, B and C each a turn in the first 6 ms, so C finishes at 6 ms, B at 10 ms and A at 14 ms: the most responsive, not the fastest on average.

exercises.lol exercises.com.cn



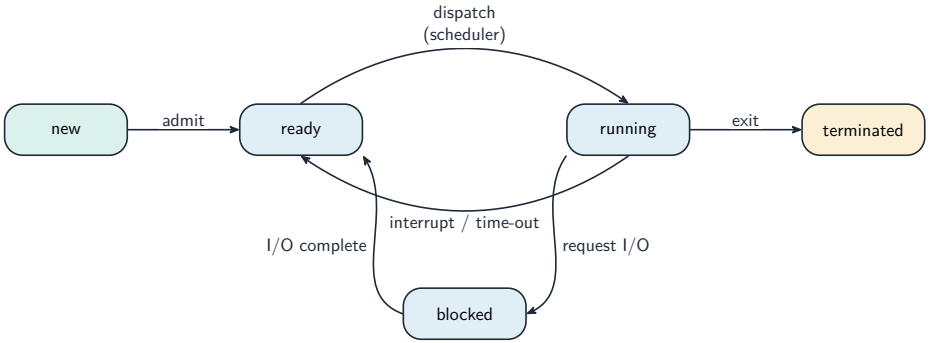
First-come-first-served scheduling of four processes



Round-robin: each process gets a fixed time slice in turn, then the next runs (unlike first-come-first-served)

Process states

A process is **new**, **ready** (waiting for the CPU), **running**, **blocked** 阻塞 (waiting for I/O or a lock), or **terminated**. When its time slice ends it goes running → ready; when it requests I/O it goes running → blocked; when the I/O finishes it goes blocked → ready.



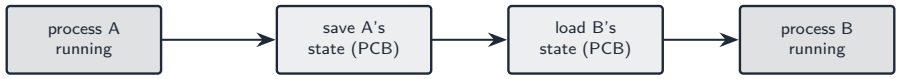
A process moves between the new, ready, running, blocked and terminated states

The three states and why a process moves. *Running:* the process has the processor. *Ready:* it could run but is waiting for the processor. *Blocked:* it cannot run until something else happens. Reasons for each transition, which the exam asks for one

at a time: **running to ready** when its **time slice** ends, or when a **higher-priority** process becomes ready and pre-empts it (an interrupt); **running to blocked** when it requests **input or output** or waits for a resource or another process; **blocked to ready** when the I/O it was waiting for **completes** (signalled by an interrupt); **ready to running** when the **scheduler** dispatches it. A blocked process can never go straight to running: it must become ready first.

■ Process control block and context switch

For each process the OS keeps a **process control block** 进程控制块 (PCB) – the saved program counter, registers, state and memory info.



A context switch saves one process's state and loads another's

- a **context switch** 上下文切换 suspends one process and starts another: it saves the state into one PCB and restores it from another. This small cost is paid on every switch.
- the **kernel** 内核 (the core of the OS) acts as an **interrupt handler** 中断处理程序. When a device or the timer raises an interrupt, **interrupt handling** 中断处理 saves the running process and runs the right routine – this is what drives low-level scheduling.

"Outline how the kernel acts as an interrupt handler" (two marks). When an interrupt is raised, the kernel **saves the state** of the running process (its registers and program counter, in its process control block), **identifies the source and priority** of the interrupt, **runs the appropriate interrupt service routine**, and then **restores** the interrupted process (or a higher-priority one) so that execution continues. This is how the timer ends a time slice and how a completed I/O operation unblocks a process.

■ Inter-process communication

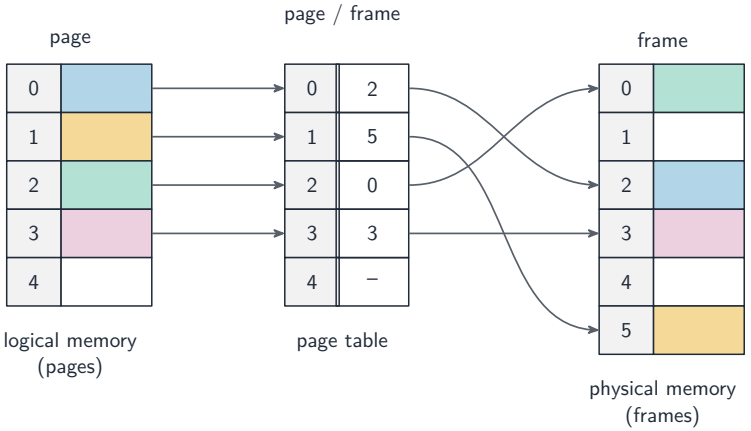
Processes are isolated, so the OS provides **inter-process communication** 进程间通信: **pipes** 管道 (one program's output feeds another's input), **shared memory** 共享内存 (a region several processes can use), and message passing.

Virtual memory, paging, segmentation

Each process gets its own **virtual address space** 虚拟地址空间 – a clean, contiguous range of addresses the OS maps to physical memory. This gives each process a simple space, **protects processes from each other**, and lets the total memory **exceed physical RAM**.

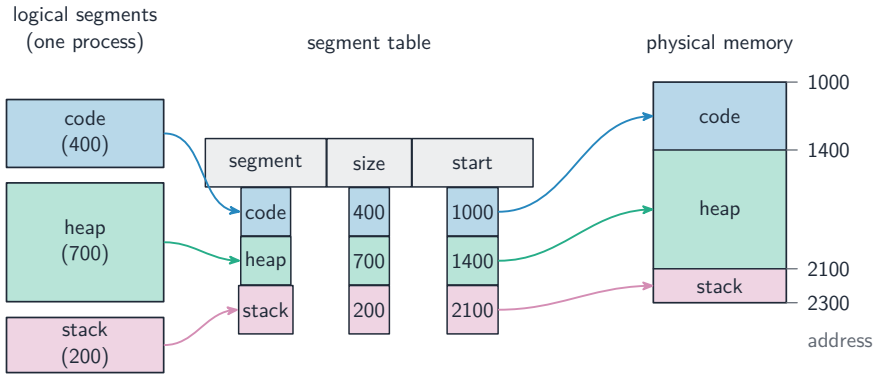
exercises.lol exercises.com.cn

In **paging**, the virtual space is split into fixed-size **pages** 页 and physical memory into same-sized **frames** 页框. A page table maps each page to a frame. If an accessed page is not in RAM – a **page fault** 缺页 – the OS reads it from the **swap file** 交换文件 into a frame, evicting another page if RAM is full. Frequent faults cause **thrashing** 抖动 (**disk thrashing**), where the OS spends most of its time swapping pages instead of doing useful work.



Paging maps each page of logical memory to a frame of physical memory

In **segmentation** 分段, memory is split into variable-sized logical segments (code, stack, heap), each with its own permissions. Many systems use paging within segments.



Segmentation maps variable-sized segments using a segment map table

"Explain what is meant by virtual memory" (three marks). *Secondary storage (disk) is used to extend the RAM*, so that the available memory appears larger than

the physical memory; the address space of a process is divided into **pages**, and only the pages currently needed are held in RAM while the rest wait on disk; pages are **swapped** between RAM and disk as required, and the OS translates each virtual address into a physical one. **Why an OS needs it:** the programs running may need **more memory than the RAM installed**; it lets more (or larger) programs run at once; a program can be larger than the physical memory; memory is used efficiently because only the active parts of programs occupy RAM.

Paging against segmentation: the difference the exam wants. *Paging* divides memory into blocks of **fixed size** (pages and frames) chosen by the hardware, with no regard to the program's structure, and the mapping is invisible to the programmer; *segmentation* divides a program into **variable-sized logical units** (a procedure, an array, the stack) whose sizes and boundaries follow the program, so a segment can be protected or shared as a unit. **"Describe the process of segmentation":** the program is split into segments of different sizes, each given a **segment number**; a **segment table** records where each segment starts in memory and how long it is; a logical address is a segment number plus an offset, and the OS adds the offset to the segment's base address to find the physical location.

"Explain what is meant by disk thrashing" and when it occurs. Disk thrashing 磁盘抖动 is the state in which pages are **swapped in and out of RAM so frequently** that the processor spends more time moving pages than executing instructions, and the system slows almost to a halt. It occurs when the RAM is **too small** for the pages the running processes need (their working sets): a page just moved out is needed again almost at once, so it is fetched back, which pushes out another page that is soon needed, and so on. Too many processes, or a program that accesses memory unpredictably, brings it on; more RAM or fewer processes cure it.

How an interpreter runs a program

An **interpreter** 解释器 translates and runs the source **at the same time**. For each statement it reads the line, does lexical and syntax analysis, checks types, then **executes** the action, and moves on. Errors are reported immediately and it usually stops; no executable is produced. The translation is redone every run (slower), but it gives fast development feedback and is portable.

"Explain how an interpreter executes a program without producing a translated version" (three marks). The interpreter takes **one statement (line) at a time**, **translates** (analyses) it, and **executes** it **immediately**, before moving to the next; **no translated version** of the whole program is created or stored, so every statement is translated **every time** it is executed, including each pass through a loop; if a statement contains an error, execution **stops there** and the error is reported. This is what makes an interpreter good for developing and testing (errors are found as they are reached, and a change can be tried at once) but slower for running finished programs.

exercises.lol exercises.com.cn

Stages of compilation

A **compiler** 编译器 turns source into **machine code** 机器码 in phases:

1. **lexical analysis** 词法分析 – the lexer groups characters into **tokens** 词法单元 (keywords, identifiers, operators, literals), discarding whitespace and comments.
2. **syntax analysis (parsing)** 语法分析 – check the tokens fit the grammar and build an **abstract syntax tree** 抽象语法树. A missing bracket gives a **syntax error** 语法错误.
3. **semantic analysis** 语义分析 – check the program makes sense (variables declared, types match).
4. **code generation** 代码生成 – walk the tree and emit target code, choosing registers and layouts.
5. **code optimisation** 代码优化 – remove redundant work, fold constants, reorder for the pipeline.

The output is an executable.



The phases of compilation, from source code to an optimised executable

The purpose of each stage, in the words that score. *Lexical analysis:* removes **white space and comments**; converts the characters of the source code into **tokens** (keywords, identifiers, operators, constants), checking that each is **valid** in the language; enters identifiers into the **symbol table** 符号表. *Syntax analysis:* checks that the sequence of tokens obeys the **grammar** (syntax rules) of the language; builds a **parse tree** (abstract syntax tree); reports **syntax errors**; type checking and the checking of variable declarations are sometimes counted here as semantic analysis. *Code generation:* converts the checked tree into **object code** or machine code (possibly via an intermediate code), allocating memory and registers. *Optimisation:* makes the code **run faster** or use **less memory**, by removing **redundant** instructions, combining or simplifying calculations, and reorganising loops, without changing what the program does. The matching question pairs each stage with one of these descriptions.

Grammar: BNF and syntax diagrams

A **grammar** 文法 says which token sequences are valid programs.

Backus-Naur Form 巴科斯-诺尔范式 (BNF) is textual. A **production rule** 产生式 has the form:

```
<symbol> ::= alternative1 | alternative2 | ...
```

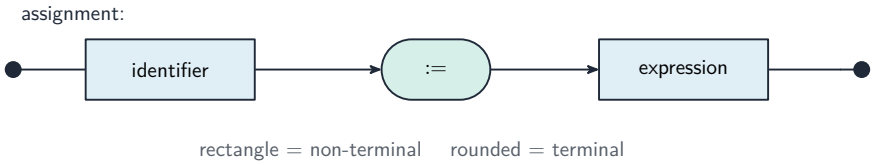
Each alternative is a sequence of **terminal** 终结符 symbols (literal text) and **non-terminal** 非终结符 symbols (other rule names):

```
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
```

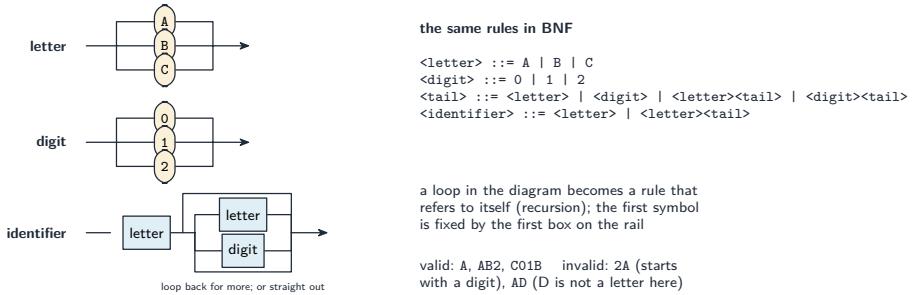
The recursive third rule expresses "a letter followed by any number of letters or digits". An `IF` statement:

```
<if-statement> ::= IF <condition> THEN <statement> ENDIF
                 | IF <condition> THEN <statement> ELSE <statement> ENDIF
```

A **syntax diagram** 语法图 (railroad diagram) shows the same thing graphically: boxes for non-terminals, rounded boxes for terminals, arrows for valid paths, loops for repetition. The two notations are equivalent. The parser uses the grammar to decide whether a program is valid.



A syntax (railroad) diagram for an assignment statement



A syntax diagram and a BNF rule say the same thing: a choice becomes alternatives separated by bars, and a loop becomes a rule that refers to itself

Reading the exam's diagrams. Each diagram defines one non-terminal; follow the arrows from the entry to the exit, and every path you can trace is a valid string. A **choice** of boxes side by side is a set of alternatives; a **loop** back is "repeat as many times as you like"; a box for another non-terminal means "insert anything that rule allows". **"State why the string is invalid"** wants the rule it breaks, in words: `9K` is invalid as a variable because the first character must be a **letter**, not a digit; `JJ90` is an invalid passcode if the rule allows only **one** letter before the digits, or if `J` is not in the set of letters listed. Always check the string against the set of characters the diagram actually allows, not against what a real language would accept.

Writing BNF from a diagram. Each diagram becomes one rule `<name> ::= ...`; alternatives are separated by `|`; a sequence is written one symbol after another; and **repetition is written with recursion**, because BNF has no loop symbol: "one or more letters" is `<word> ::= <letter> | <letter><word>`, and "zero or more digits after a letter" is `<variable> ::= <letter> | <letter><digits>` with `<digits> ::= <digit> | <digit><digits>`.

Worked example. Complete the BNF for a vehicle registration that must begin with two letters (from `A B C`) followed by one, two or three digits (from `0 1 2`).

```
<letter>       ::= A | B | C
<digit>        ::= 0 | 1 | 2
<digits>       ::= <digit> | <digit><digit> | <digit><digit><digit>
<registration> ::= <letter><letter><digits>
```

`AB12` is valid; `A12` is not (only one letter); `AB1234` is not (four digits); `AD1` is not (`D` is not a listed letter). Asked to add a constraint such as "the third character may also be a symbol", add the extra alternative to the rule for that position only, and define `<symbol>` with its own rule.

Worked example. Write BNF for an expression that is a variable, followed by an operator, followed by either a variable or a number, where a variable is a single lower-case letter from `a b c` and an operator is `+` or `-`.

```
<variable>   ::= a | b | c
<operator>   ::= + | -
<number>     ::= <digit> | <digit><number>
<expression> ::= <variable><operator><variable> |
↪  <variable><operator><number>
```

The recursive `<number>` rule allows any number of digits; the two alternatives of `<expression>` cover both cases named in the definition. Keep every non-terminal in angle brackets and every terminal without them.

Reverse Polish Notation (RPN)

In **infix** 中缀 notation the operator sits between its operands (`3 + 4 * 2`), needing brackets and precedence rules. In **Reverse Polish Notation** 逆波兰表示法 (RPN, **postfix** 后缀) the operator follows its operands (`3 4 2 * +`), needing no brackets.

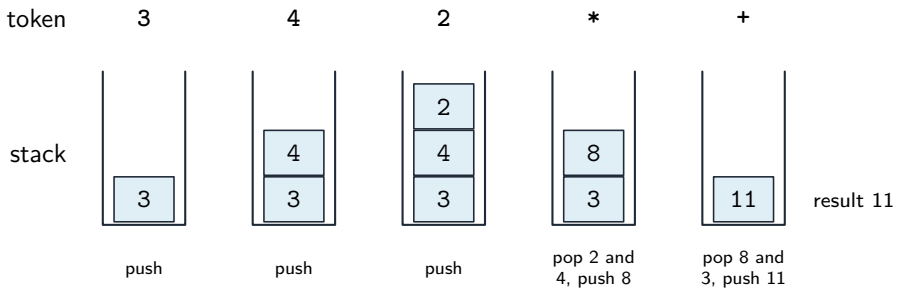
exercises.lol exercises.com.cn

■ Converting infix to RPN

Use an operator **stack** 栈. Scan left to right: output an operand; for an operator, first pop any stacked operators of **higher or equal precedence** 优先级 to the output, then push it; push `(` on `)` pop to output until the matching `(`. At the end, pop all operators. Example: `(3 + 4) * 2` → `3 4 + 2 *`.

■ Evaluating RPN

Use a stack of operands. Scan left to right: push each operand; on an operator, pop the top two, apply it, and push the result. Evaluating `3 4 2 * +`:



*Evaluating 3 4 2 * + on a stack*

Token	Stack
3	3
4	3, 4
2	3, 4, 2
*	3, 8
+	11

Result: 11. RPN needs no brackets at evaluation time and suits a stack machine – which is how the JVM and many **bytecode** 字节码 interpreters work.

"Explain why RPN is used to evaluate expressions" (two marks). In RPN the operators appear **in the order in which they are applied**, so an expression can be evaluated in a **single left-to-right pass** with **no brackets** and **no precedence rules**; it is therefore simpler and faster for the compiler or interpreter to process. **"Identify, with reasons, a suitable data structure":** a **stack**, because evaluation needs the **most recently** pushed operands first (last in, first out): each operand is pushed, and each operator pops the top two, applies itself, and pushes the result. Show the stack contents after every token when asked.

Converting infix to RPN by hand. (1) Fully bracket the expression using the precedence rules; (2) move each operator to just after the closing bracket of its own pair; (3) remove the brackets. So $(a-b)*(a+c)/7$ becomes $((a-b)*(a+c))/7$, then `a b - a c + * 7 /`. Note that `*` and `/` are applied left to right, so the division is the last operator, not the multiplication. More conversions: $((7+3)-(2*8))/6$ is `7 3 + 2 8 * - 6 /`; $(7-2+8)/(9-5)$ is `7 2 - 8 + 9 5 - /`; $a*b+b-d+15$ is `a b * b + d - 15 +`; $(2-6)*(13+7)/5$ is `2 6 - 13 7 + * 5 /`.

Converting RPN back to infix. Work through the RPN with a stack of **expressions**: push each operand; for each operator pop two, write them either side of it **in brackets**, and push the result. So `a b / 4 * a b + -` is $((a/b)*4)-(a+b)$; `5 2 + 9 3 - / 3 *` is $((5+2)/(9-3))*3$; `b a c - + d b + * c /` is $((b+(a-c))*(d+b))/c$; `a b - c + c a - * d /` is $(((a-b)+c)*(c-a))/d$. Keep the brackets: dropping them can change the meaning.

> **Worked example.** Evaluate `a b - c d + * e /` when $a=17$, $b=5$, $c=7$, $d=3$ and $e=10$, showing the stack.

token	action	stack (top on the right)
a	push 17	17
b	push 5	17, 5
-	pop 5 and 17, push $17-5$	12
c	push 7	12, 7
d	push 3	12, 7, 3
+	pop 3 and 7, push $7+3$	12, 10
*	pop 10 and 12, push 12×10	120
e	push 10	120, 10
/	pop 10 and 120, push $120/10$	12

Result **12**. The order of the pops matters for `-` and `/`: the value popped **second** is the left operand, so `a b -` is $a-b$, not $b-a$. Two more, in the same way: `d a b + * c a - /` with $a=6, b=12, c=15, d=5$ gives $5\times(6+12)/(15-6)=90/9=10$; `c a - b d + * b c + /` with $a=4, b=12, c=24, d=6$ gives $(24-4)\times(12+6)/(12+24)=360/36=10$.

> **Worked example.** Convert $(A+B)\times(C-D)$ to RPN, then evaluate $(3+4)\times(5-2)$. Scan left to right using an operator **stack**. Push `(`; output `A`; push `+`; output `B`; on `)` pop back to the matching `(`, giving `A B +` so far. Push `×`, and the second bracket behaves the same way, giving `C D -`. At the end pop the `×`. Result: **`A B + C D - ×`**. To evaluate the numbers, use a stack of **operands**: push 3, push 4; `+` pops both and pushes 7; push 5, push 2; `-` pops both and pushes 3; `×` pops 7 and 3 and pushes **21**. Two things make these reliable: the operands keep their **original order** through the conversion (only the operators move), and every operator acts on the **two values immediately below it** on the stack.

exercises.lol exercises.com.cn

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
multi-tasking	several processes held in memory at once, the processor switching between them so that they appear to run simultaneously
process	a program that has been loaded into memory and is being executed (or is ready to be)
running / ready / blocked	has the processor / waiting for the processor / cannot continue until an event such as I/O completes
scheduling	deciding which ready process gets the processor next, and for how long
pre-emptive scheduling	the running process can be interrupted and moved to ready so that another process runs
virtual memory	using secondary storage to extend RAM, holding only the pages currently needed in physical memory
paging	dividing memory and programs into fixed-size pages that are moved between disk and RAM as needed
segmentation	dividing a program into variable-sized logical segments, each mapped to memory by a segment table
disk thrashing	pages being swapped between RAM and disk so often that little useful processing is done
interpreter	translates and executes a program one statement at a time, without producing a translated version
compiler	translates a whole high-level program into machine (object) code before it is run
lexical analysis	converts the source code into tokens, removing white space and comments, and builds the symbol table
syntax analysis	checks that the tokens obey the grammar of the language and builds a parse tree
Backus–Naur Form	a notation for the grammar of a language: rules of the form `<name> ::= alternatives` built from terminals and non-terminals

exercises.lol exercises.com.cn

Term	Definition
Reverse Polish Notation	a way of writing expressions with each operator after its operands, so they can be evaluated with a stack and without brackets

Exam tips

- The OS questions are marked on named mechanisms: scheduling, memory management, I/O buffering and spooling, file management; for the interface, file names not addresses, clicks not commands, drivers, GUI.
- Process states with their transitions and the reason for each; scheduling routines as function plus benefit plus drawback; the kernel saves state, identifies the interrupt, services it, restores.
- Virtual memory: disk extends RAM, pages swapped, address translation; paging is fixed-size and invisible, segmentation is variable-size and logical; thrashing is swapping instead of working.
- Interpreter: one statement at a time, translated then executed, nothing stored. Compiler stages: tokens and symbol table, grammar and parse tree, code, optimisation.
- BNF: a rule per diagram, `|` for choice, recursion for repetition, terminals bare and non-terminals in angle brackets. Say which rule a string breaks.
- RPN: operators after operands, evaluate with a stack, show every step; convert by fully bracketing; when converting back, keep the brackets.

Common mistakes

- Describing multi-tasking as “running several programs at the same time” without saying the processor switches between them.
- Sending a blocked process straight to running, or giving “time slice ended” as the reason for running to blocked.
- Confusing shortest job first (non-pre-emptive) with shortest remaining time (pre-emptive), or round robin with priority.
- Defining virtual memory as “using the hard disk as RAM” with no mention of pages being swapped.
- Saying an interpreter “converts the program to machine code and then runs it”; that is a compiler.
- Putting syntax checking in lexical analysis, or optimisation before code generation in the matching question.
- Writing BNF repetition as `<letter>*` or with an ellipsis; use recursion. Leaving angle brackets off non-terminals.

- Reversing the operands of `-` or `/` when evaluating RPN, or writing the RPN of $a * b + c$ as `a b c + *`.

exercises.lol exercises.com.cn

Security

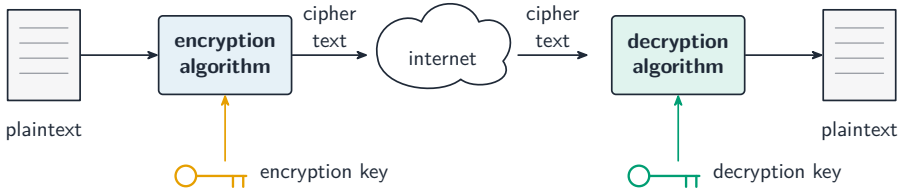
A-Level Computer Science

How encryption works

Encryption 加密 turns readable **plaintext** 明文 (plain text) into unreadable **ciphertext** 密文 (cipher text) using a maths operation that depends on a **key**. Only someone with the right key can reverse it – **decryption** 解密 – to get the plaintext back. An attacker who intercepts the ciphertext without the key sees only meaningless data, because trying every possible key would take far too long. A newer approach, **quantum cryptography** 量子密码学, uses quantum physics to share a key in a way that reveals any eavesdropper.



The Enigma machine encrypted messages in the Second World War – an early, mechanical cipher device

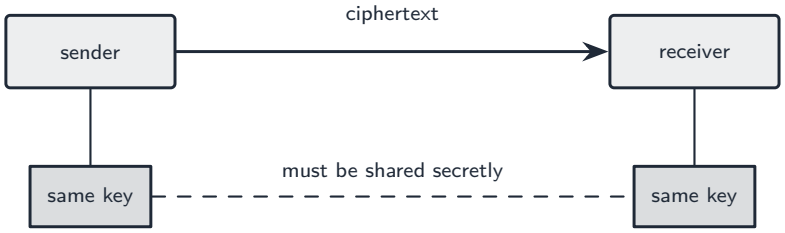


Encryption scrambles plaintext with a key; decryption reverses it

Symmetric encryption

Symmetric encryption 对称加密 (symmetric key cryptography) uses the **same key** for both encryption and decryption, so sender and receiver must both hold the secret key. It is **fast** and good for **bulk data** (a whole disk, a video stream). Its problem is **key distribution** 密钥分发: how do you share the key safely in the first place? Asymmetric encryption solves this.

"Describe what is meant by symmetric key encryption" (two marks). *The same key is used to encrypt the plaintext and to decrypt the ciphertext*, so the key must be **shared** between sender and receiver and kept **secret** from everyone else. **Two drawbacks.** The key has to be **exchanged** before the message can be sent, and if it is intercepted in transit the interceptor can read every message; a separate key is needed for **every pair** of correspondents; and it gives no proof of **who** sent the message, because both ends hold the same key. **"Give two reasons for using key cryptography":** so that data is **unreadable** by anyone who intercepts it (confidentiality); so that the receiver can be sure the data came from the claimed sender and was **not altered** (authenticity and **integrity** 完整性). The two methods are symmetric and asymmetric key cryptography.

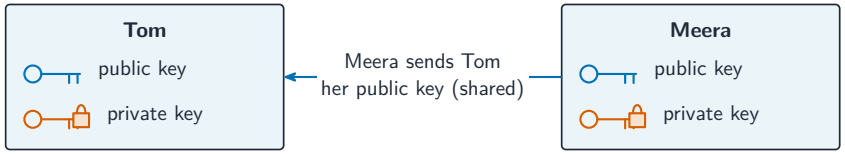


Symmetric encryption uses the same secret key at both ends

Asymmetric encryption (public-key)

Asymmetric encryption 非对称加密 (asymmetric key cryptography) gives each user a **pair** of related keys: a **public key** 公钥 they publish, and a **private key** 私钥 they keep secret. Data encrypted with the public key can be decrypted **only** with the matching private key, and vice versa.

exercises.lol exercises.com.cn



encrypt with the recipient's **public key** — only their matching **private key** can decrypt

Each user has a public key to share and a private key to keep secret

To send a secret message to Alice: get her published public key, encrypt with it, and send. Only Alice – holding the matching private key – can decrypt. No prior key exchange is needed. The trade-off is that it is **much slower** than symmetric, so it is not used for large data.

"State what is meant by a private key." *A key known only to its owner (never transmitted), used to decrypt data that was encrypted with the matching public key, and to create digital signatures.* **"Describe the process of asymmetric encryption" (four marks):** (1) the receiver generates a **pair** of keys, a public key and a private key, mathematically related; (2) the **public key** is made available to anyone who wants to send to them; (3) the sender **encrypts** the plaintext with the receiver's **public key**; (4) the ciphertext can only be **decrypted with the receiver's private key**, which never leaves the receiver, so nobody who intercepts the message can read it.

> **Worked example.** Fred wants to send Sheila a confidential document. Explain how asymmetric encryption is used.

Sheila has a key pair; she sends Fred her **public** key (or he obtains it from her certificate). Fred encrypts the document with **Sheila's public key** and sends the ciphertext. Only Sheila's **private key** can decrypt it, and only Sheila holds that, so nobody else, including Fred once it is encrypted, can read the document. The keys are used the **receiver's** way round: her public key to lock, her private key to unlock. An organisation that holds a key pair "to receive secure transmissions" does exactly this: it publishes the public key, keeps the private key, and decrypts what arrives.

Two differences between symmetric and asymmetric encryption. Symmetric uses **one** key for both directions; asymmetric uses **two** related keys, one to encrypt and the other to decrypt. In symmetric encryption the key must be **kept secret by both** parties and exchanged securely; in asymmetric encryption the public key can be **published** and only the private key is secret. Symmetric encryption is much **faster** and suits large amounts of data; asymmetric is slower, so it is used for keys and signatures rather than bulk data.

A private key must stay secret, so it is sometimes kept on a small **hardware security key** 硬件安全密钥. You plug it in or tap it to prove who you are, and the secret key never leaves the device.



A hardware security key stores a secret key to prove who you are

Hybrid approach (used by almost every real system)

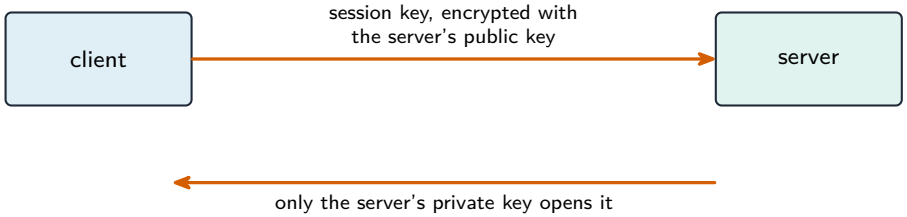
Use asymmetric encryption to exchange a fresh **session key** 会话密钥, then use that symmetric key for the data:

1. the client makes a random session key.
2. it encrypts the session key with the server's public key.
3. the server decrypts it with its private key.
4. both ends now share the session key and use fast symmetric encryption for the rest.

This is how HTTPS and SSH work.

The exam's version of the key-exchange problem. "A symmetric key is to be exchanged before the message is sent. Explain how the key can be exchanged securely." The sender encrypts the **symmetric key** with the **receiver's public key** and sends it; the receiver decrypts it with their **private key**; both now hold the symmetric key, which was never exposed in transit, and use it for the messages. Asymmetric encryption solves the distribution problem; symmetric encryption then does the fast work.

exercises.lol exercises.com.cn

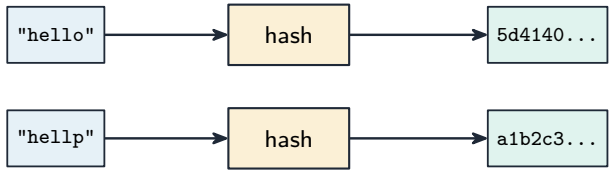


both ends now share the session key → fast **symmetric** encryption for all the data

The hybrid approach: asymmetric crypto shares a session key once, then fast symmetric encryption protects the data

Hashing (related, not encryption)

A **cryptographic hash** 密码散列 function takes any input and gives a fixed-size **digest** 摘要 such that the same input always gives the same digest, it is infeasible to find two inputs with the same digest, and a tiny change in input changes the digest completely. Hashing is **one-way** – you cannot get the input back. It is used for storing password checks, **integrity** checks, and digital signatures.



one letter changed → a completely different digest; one-way (cannot reverse)

A cryptographic hash gives a fixed digest; a tiny input change changes it completely, and it cannot be reversed

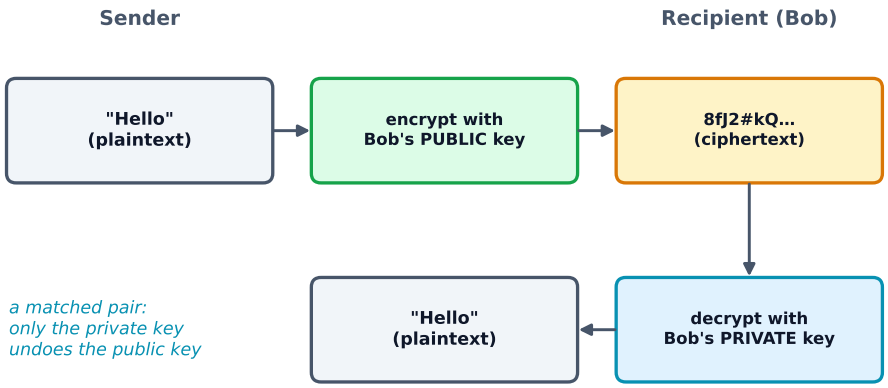
Quantum cryptography

Quantum cryptography uses the physics of light to distribute keys: the bits of a key are sent as **photons** whose quantum states encode the values. **"Describe its purpose":** to transmit an encryption key **securely**, in such a way that any attempt to intercept it can be **detected**, because measuring a photon **changes its state**; an **eavesdropper** 窃听者 therefore leaves evidence, and the corrupted key is thrown away

and a new one sent. **Benefits:** interception is always detectable; the key cannot be copied without being altered; it is secure against future advances in computing power (a mathematical key can eventually be cracked, a quantum one cannot be read without disturbing it). **Drawbacks:** it needs specialised, **expensive** equipment; it works only over **limited distances** on dedicated optical fibre (or line of sight), not across the existing internet; it distributes the **key only**, so ordinary encryption still protects the message; and it is a **new** technology with few suppliers and little experience.

SSL / TLS

TLS 传输层安全 (Transport Layer Security, the successor to the **Secure Socket Layer**, SSL) is a **protocol** that gives encryption and authentication for data sent over a network. It **encrypts** the data in transit, **authenticates** the server with a certificate, and provides **integrity** (detecting tampering).



Asymmetric encryption: encrypt with the public key, decrypt with the private key

Outline of a TLS handshake:

1. the client connects and proposes cipher options.
2. the server picks one and sends its **digital certificate** (with its public key) – issuing and validating these certificates is **digital certification**.
3. the client checks the certificate.
4. the two ends exchange a fresh **session key** using asymmetric crypto.
5. all later traffic uses fast symmetric encryption with the session key.

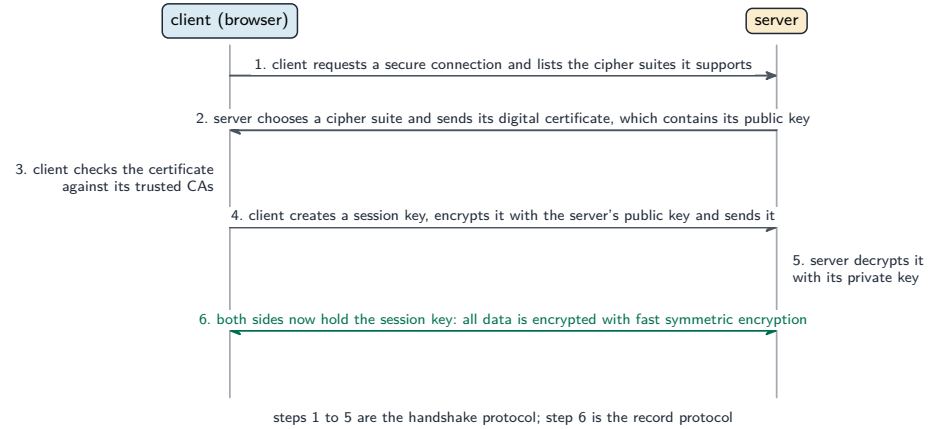
The result is an encrypted, authenticated, integrity-checked tunnel for higher-level protocols (HTTP, SMTP). It is appropriate wherever **sensitive information** is sent: HTTPS web browsing, online banking and payments, secure email, and VPNs.

"Describe the purpose of SSL/TLS" and "state two functions." The purpose

exercises.lol exercises.com.cn

is to provide **secure communication** between a client and a server over a network. Its functions: it **encrypts** the data sent, so that it cannot be read if intercepted; it **authenticates** 认证 the server (and optionally the client) by means of a digital certificate, so the client knows it is talking to the genuine site; and it **checks the integrity** of the data, so that changes in transit are detected. Two examples of where it is appropriate: **online banking** and **online shopping** (card payments); also logins, private email, file transfer, VoIP and instant messaging: any transaction in which private data crosses the internet.

The two protocols that make up TLS. The **handshake** 握手 protocol sets up the session: it agrees the encryption algorithms (cipher suite), authenticates the server with its certificate, and exchanges the session key. The **record** protocol then carries the data: it encrypts each message with the session key, adds an integrity check, and passes it to the transport layer.

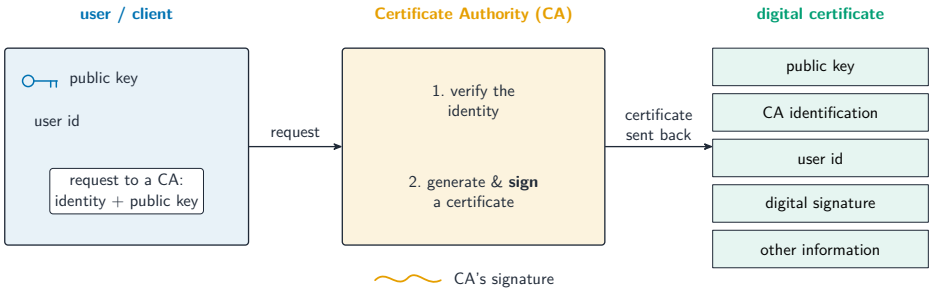


How a secure session starts: the certificate proves who the server is, the server's public key protects the session key on its way over, and the session key protects everything after that

"Explain how SSL/TLS is used when client–server communication is initiated" (six marks). (1) The client (browser) sends a request to the server for a **secure connection**, saying which encryption methods it supports. (2) The server sends back its **digital certificate**, which contains its **public key**. (3) The client **checks the certificate** is valid (issued by a trusted Certificate Authority, not expired, for the right domain). (4) The client generates a **session key**, encrypts it with the server's **public key** and sends it. (5) The server **decrypts** the session key with its **private key**. (6) Both sides now hold the session key and all further data is sent using **symmetric encryption** with it. Give the steps in this order; the marks are for the certificate, the public key, the session key and the switch to symmetric encryption.

Digital certificates

A **digital certificate** 数字证书 binds an identity (a domain, an organisation) to a public key, and is signed by a trusted **Certificate Authority** 证书颁发机构 (CA). It contains the subject (who it identifies), the subject's public key, the issuer (the CA), a validity period, and the CA's signature over all of it.



A Certificate Authority issues a digital certificate binding an identity to a public key

To verify one, the client (which holds a list of trusted root CAs):

1. checks the expiry dates.
2. checks the subject name matches the URL.
3. checks it is **signed by a trusted CA**, using the CA's public key to verify the signature.
4. follows the certificate chain up to a trusted root.

If anything fails, the browser shows the "Your connection is not private" warning. When it verifies cleanly, the client knows the identity was vetted by a trusted CA, the public key really belongs to that identity, and the certificate is current.

"Describe what is meant by a digital certificate" (two marks). *An electronic document, issued by a Certificate Authority, that verifies the identity of its owner (a person, organisation or website) and contains the owner's public key.* **Items found in one:** the **serial number**; the **name of the owner** (subject) and, for a website, its domain; the owner's **public key**; the name of the **issuing CA**; the **validity period** (dates); the signature **algorithm** used; and the **CA's digital signature** of the whole certificate.

"Explain how an organisation acquires a digital certificate" (four marks). (1) The organisation generates its own **key pair**, a public key and a private key. (2) It sends a **request** containing its public key and its **identity details** to a Certificate Authority. (3) The CA **verifies the identity** (checks that the applicant really is the organisation or owns the domain). (4) The CA creates the certificate containing the public key and the identity, **signs it with the CA's own private key**, and returns it. (5) The organisation installs the certificate on its server so that it can be sent to clients. The private key never leaves the organisation.

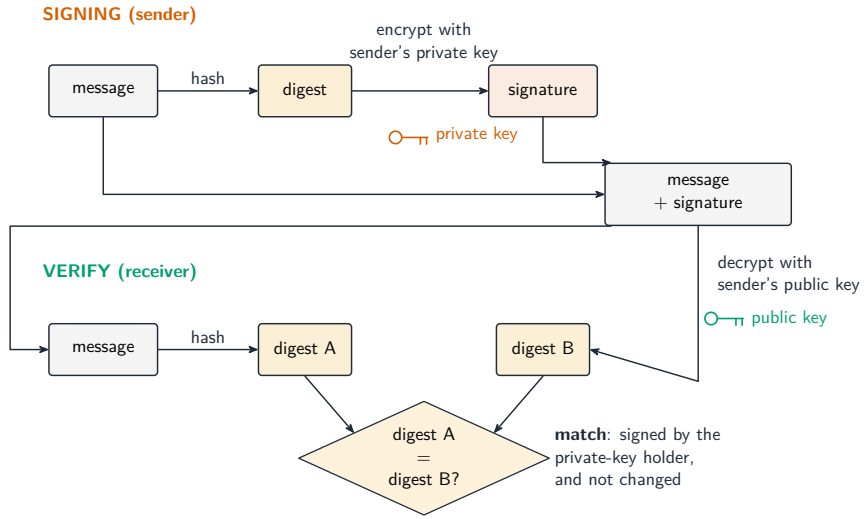
"Explain why a digital certificate is required to validate a digital signature." To check a signature the receiver needs the sender's **public key**, and needs to be sure that the key really **belongs to the claimed sender**; the certificate supplies the public key together with the identity, and because the certificate is **signed by a trusted CA** the receiver can trust that binding. Without it an impostor could publish a public key in someone else's name and sign messages as them. The same reasoning answers "what should be included with a program downloaded from the internet to prove it is genuine": a **digital signature**, checked against the publisher's certificate.

Digital signatures

A **digital signature** 数字签名 proves who signed a message **and** that it was not changed. To sign:

1. compute a cryptographic hash of the message.
2. **encrypt the hash with the sender's private key** – that is the signature.
3. send the message and the signature.

To verify: compute the hash of the received message; **decrypt the signature with the sender's public key** to get the sender's hash; compare. If they match, the message was signed by the holder of the private key (**authentication** 身份验证) and was not changed (integrity). A signature does **not** hide the message – for confidentiality as well, encrypt **and** sign.



Signing hashes the message and encrypts the digest with the private key; the receiver checks it with the public key

exercises.lol exercises.com.cn

"Explain the role of a digital certificate in creating a digital signature" (three marks). The sender's certificate was issued by a CA and contains the sender's **public key** together with the sender's identity; the sender produces the signature by **hashing** the message and **encrypting the hash with their private key**, the partner of the key in the certificate; the receiver uses the **public key from the certificate** to decrypt the hash and, because the certificate binds that key to the sender, the signature proves **who** signed.

"Explain how a digital signature is used to verify a message" (four marks). (1) The receiver **decrypts the signature** with the sender's **public key** (taken from the sender's certificate), which yields the hash that the sender computed. (2) The receiver **hashes the received message** with the same hash algorithm. (3) The two hashes are **compared**. (4) If they **match**, the message came from the holder of the private key (**authentic**) and has **not been altered** since it was signed (**integrity**); if they differ, the message is rejected. A banker receiving confidential data with a signature does exactly this before trusting it; the data itself may separately be encrypted with the banker's public key for confidentiality.

■ Putting it together

A secure request to `https://www.bank.com`: the server sends its certificate; the client verifies it against trusted CAs; the client uses the server's public key to exchange a session key; then data flows encrypted with that key. Encryption stops eavesdroppers, the certificate proves the server's identity, and integrity checks stop a **man-in-the-middle** 中间人攻击 altering the data.

> **Worked example.** Alice sends Bob a contract. She wants Bob to be certain it came from her and was not altered, **and** she wants nobody else to be able to read it. Which keys does she use, and in which direction? These are two different jobs needing two different key pairs. For the **signature** (authentication and integrity): Alice hashes the contract and encrypts that hash with **her own private key**; Bob decrypts it with **Alice's public key** and compares it against his own hash of the message. Only Alice holds her private key, so only she could have produced it. For **confidentiality**: Alice encrypts the contract itself with **Bob's public key**, so only **Bob's private key** can open it. One rule keeps all four straight: you **sign with your own private key** and **encrypt with the recipient's public key**. A signature on its own does **not** hide the message.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

exercises.lol exercises.com.cn

Term	Definition
encryption	converting plaintext into ciphertext using an algorithm and a key so that it cannot be understood if intercepted
plaintext / ciphertext	the original readable data / the encrypted, unreadable form of it
symmetric key cryptography	the same secret key is used to encrypt and to decrypt, so it must be shared securely by both parties
asymmetric key cryptography	a pair of related keys is used: the public key encrypts and only the matching private key decrypts
public key	a key made available to anyone, used to encrypt messages to its owner and to verify the owner's signatures
private key	a key known only to its owner, used to decrypt messages encrypted with the public key and to sign
SSL/TLS	protocols that provide secure (encrypted, authenticated, integrity-checked) communication between a client and a server
digital certificate	an electronic document issued by a Certificate Authority that verifies the owner's identity and contains their public key
digital signature	a hash of a message encrypted with the sender's private key, proving who sent it and that it is unaltered
Certificate Authority	a trusted organisation that verifies identities and issues and signs digital certificates
quantum cryptography	the use of quantum states of photons to distribute keys so that any interception is detected

Exam tips

- Symmetric: one shared secret key, fast, key exchange is the weakness. Asymmetric: public key to encrypt, private key to decrypt, slow, no exchange problem. Two differences, two drawbacks, two reasons: the exam asks for them in pairs.
- Confidentiality uses the **receiver's** keys (public to lock, private to unlock); a signature uses the **sender's** keys (private to sign, public to check). Say whose key every time.
- The TLS start-up is six steps: request, certificate with public key, check, session key encrypted with the public key, decrypted with the private key, symmetric encryption from then on.

- A certificate is identity plus public key, signed by a CA; acquisition is key pair, request, verification, signing, installation. It is needed to validate a signature because it proves whose public key it is.
- A signature is a hash encrypted with the private key; verification is decrypt, re-hash, compare. Integrity and authenticity are the two things it proves.
- Quantum cryptography distributes keys and detects eavesdropping; its limits are cost, distance and novelty.

■ Common mistakes

- Saying a message is encrypted with the sender's public key; the receiver's public key encrypts, the receiver's private key decrypts.
- Describing a signature as "encrypting the message with the private key" instead of encrypting its hash.
- Claiming a certificate contains the private key; it holds the public key and the identity, signed by the CA.
- Listing "the server sends its private key" in the TLS handshake; only the public key travels, inside the certificate.
- Giving "SSL/TLS makes the connection faster" as a function; its functions are encryption, authentication and integrity.
- Confusing hashing with encryption: a hash cannot be reversed and has no key; encryption is reversible with the key.
- Answering "why is a certificate needed for a signature" with "to encrypt it"; it is needed to trust the public key.

exercises.lol exercises.com.cn

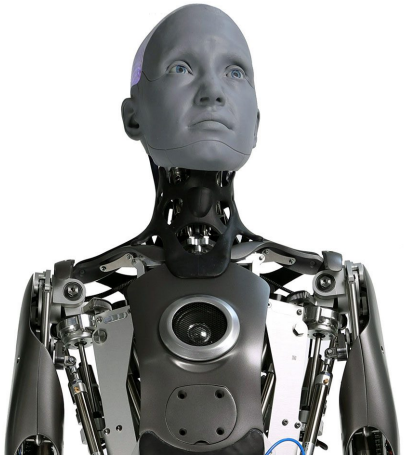
Artificial Intelligence (AI)

A-Level Computer Science

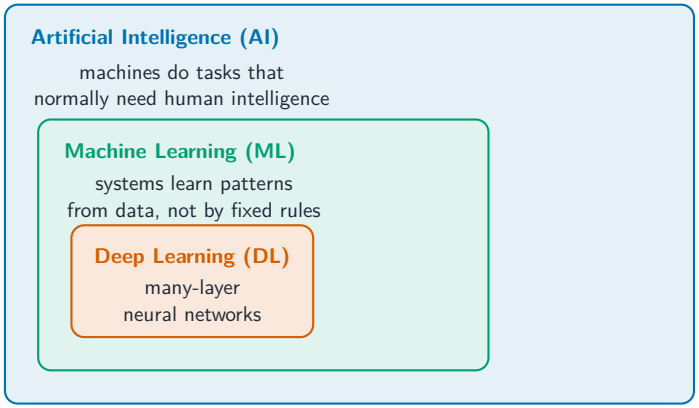
What AI is

Artificial intelligence 人工智能 (AI) builds systems that do tasks normally needing human intelligence – recognising speech and images, translating, playing games, driving, generating text. Most modern AI uses **machine learning** 机器学习 – algorithms that learn patterns from data instead of being programmed step by step. Within it, **deep learning** 深度学习, using **neural networks** 神经网络 with many layers, has been dominant since the 2010s.

A **humanoid robot** 人形机器人 puts many of these abilities into one body: it uses AI to see faces, understand speech and move its face and arms in a lifelike way.



A humanoid robot uses AI to see, listen and respond like a person

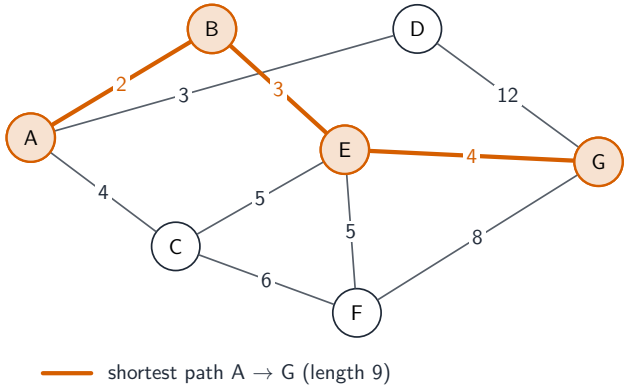


Deep learning is part of machine learning, which is part of AI

Graphs in AI

Many AI problems sit on a **graph** 图 – **nodes** 节点 (states, places) joined by **edges** 边 (moves, relationships).

- **pathfinding**: roads form a graph; the shortest route is a graph search (**Dijkstra's algorithm**, the **A* algorithm**).
- **game playing**: each board position is a node, each move an edge; **minimax** 极小化极大 with alpha-beta pruning searches the game tree.
- **state-space search**: a planning problem is moving between states by applying operators to reach a goal.
- **knowledge representation**: a **semantic network** 语义网络 has concepts as nodes and relationships as edges ("dog IS-A animal"); a **knowledge graph** 知识图谱 stores facts about the world for search engines and assistants.

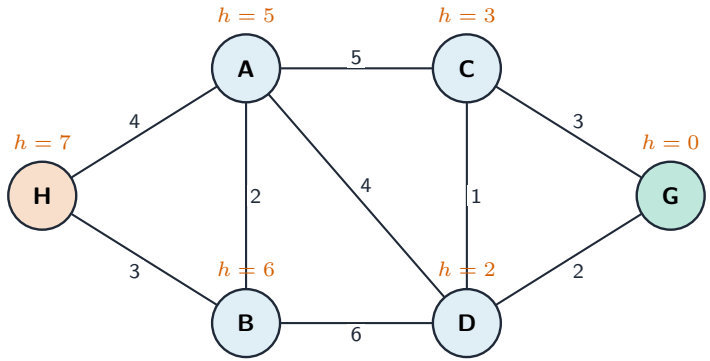


AI problems often sit on a graph; here the shortest path is highlighted

Standard tools for navigating graphs include **breadth-first search** 广度优先搜索 and **depth-first search** 深度优先搜索.

"Describe the purpose and structure of a graph in an AI system." *Purpose:* to **represent a problem** as a set of states (or places) and the possible moves between them, so that an **algorithm can search** it for a solution, such as the shortest or cheapest route, or the best next move. *Structure:* a set of **nodes** (vertices), each representing a state, location or item, joined by **edges** representing the connections between them; each edge may carry a **weight** (a cost, distance or time), and edges may be **directed** (one-way) or undirected. **"Explain the use of graphs to aid AI":** the graph is the model on which the AI's search algorithms run: A* and Dijkstra's algorithm find optimal paths through it (navigation, routing), game positions form a tree searched for the best move, and knowledge stored as a graph lets a system reason about how facts are related.

exercises.lol exercises.com.cn



edge numbers: actual distances; h (red): the heuristic estimate of the distance still to go to G, used only by A*. Start at H (Home), goal G (School)

The graph used below: the edge numbers are real distances; the red numbers are each node's heuristic 启发式 estimate of how far the goal still is, which only A uses*

Dijkstra's algorithm. It finds the **shortest distance from the start to every node**. Keep a table of the best distance found so far to each node (start 0, all others infinity). Repeatedly take the unvisited node with the **smallest** distance, mark it visited, and for each neighbour check whether going through this node gives a shorter distance; if so, update it and record where it came from. Stop when every node is visited (or the target is).

Worked example. Find the shortest distances from H to every other node in the graph above.

step	visit	H	A	B	C	D	G
start		0	∞	∞	∞	∞	∞
1	H (0)	0	4 (H)	3 (H)	∞	∞	∞
2	B (3)	0	4 (H)	3	∞	9 (B)	∞
3	A (4)	0	4	3	9 (A)	8 (A)	∞
4	D (8)	0	4	3	9 (A)	8	10 (D)
5	C (9)	0	4	3	9	8	10 (D)
6	G (10)						

Shortest distances: A 4, B 3, D 8, C 9, G 10, and the path to G is H–A–D–G (read the "came from" labels backwards). At step 3, A offers D a distance of $4+4=8$, better than the 9 found through B, so D is updated; at step 5, C could reach G at $9+3=12$, worse than 10, so nothing changes. Showing these comparisons is the "working" the question asks for.

exercises.lol exercises.com.cn

The A* algorithm. Dijkstra explores in every direction. A* adds a **heuristic** h, an estimate of the distance still to go, and always expands the node with the smallest $f = g + h$, where g is the distance travelled so far. With a sensible heuristic (never over-estimating), it finds the same shortest path while looking at far fewer nodes, which is why satnavs and games use it. The exam gives h for each node and a table to fill in.

Worked example. Find a path from H to G with A*, showing the working.

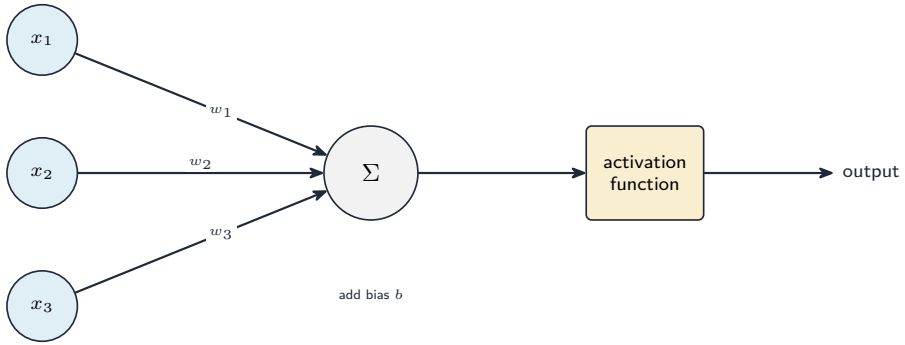
node expanded	g so far	h	$f = g + h$	neighbours added (node: g, h, f)
H	0	7	7	A: 4, 5, 9; B: 3, 6, 9
B (tie with A; either)	3	6	9	D via B: 9, 2, 11
A	4	5	9	C: 9, 3, 12; D via A: 8, 2, 10 (better than 11, keep)
D	8	2	10	G: 10, 0, 10; C via D: 9 (no better)
G	10	0	10	goal reached

Path H–A–D–G, length 10, the same as Dijkstra's, but C was never expanded. Each time a node is reached by a second route, keep the smaller g; the search ends when the **goal** is the node with the smallest f. State the g, h and f values in every row: those are the marks.

Artificial neural networks (ANNs)

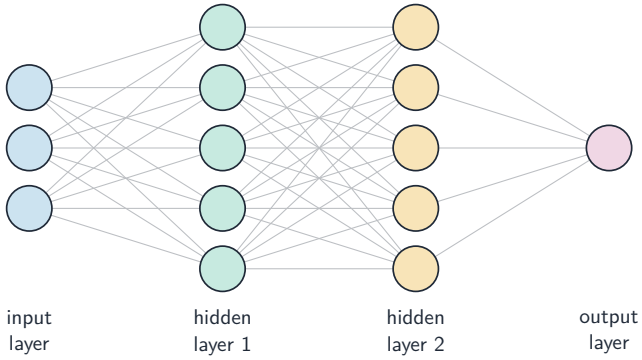
An ANN is inspired by the brain's neurons. An **artificial neuron** 人工神经元:

- takes several **input values**, multiplies each by a **weight** 权重, and adds them up with a **bias term** 偏置项.
- applies an **activation function** 激活函数 (a non-linear function such as ReLU) to the sum.
- outputs the result, which feeds neurons further on.



A single neuron: each input times its weight, summed with a bias, then an activation function

Neurons sit in layers: an **input layer**, one or more **hidden layers** 隐藏层 (where useful internal patterns are learned), and an **output layer**. With many hidden layers it is a **deep neural network** 深度神经网络, and training it is deep learning.



A neural network with an input layer, two hidden layers and an output layer

ANNs let models learn complex patterns straight from raw data (pixels, audio, text) without hand-designed features – driving breakthroughs in **image recognition** 图像识别, **speech recognition** 语音识别, **machine translation** 机器翻译, and game playing. They do well with large amounts of data, noisy or very complex input, and patterns too hard to capture with explicit rules.

"Explain what is meant by an artificial neural network." *A model of the brain's network of neurons, made of layers of connected nodes: an input layer, one or more hidden layers and an output layer.* Each connection has a **weight**; each node sums its weighted inputs and passes the result through an activation function to the next layer. **"Explain how ANNs enable machine learning" (three marks):** the network is **trained** on many examples; for each example the output is compared with the expected result and the **error** is used to **adjust the weights** (back propagation) so that the error falls; after enough examples the weights encode the **patterns** in the data, and the network can then classify or predict for **new** data it has never seen. **"State the reason for multiple hidden layers":** each additional layer combines the features found by the layer before it into **more complex, more abstract** features, so the network can learn **more complex** relationships (edges, then shapes, then objects); that is what makes a network deep.

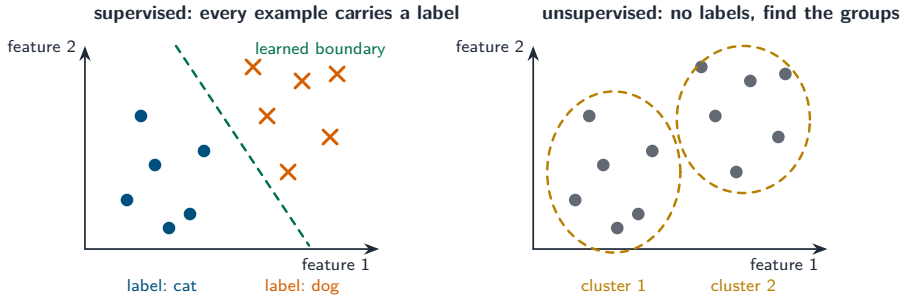
Machine learning, deep learning, reinforcement learning

Machine learning

The umbrella term – any algorithm that learns from data. Three paradigms:

- **supervised learning** 监督学习 – the data has **labels** 标签 (images tagged "cat"/"dog"); the algorithm learns input → label. Used for **classification** 分类 (a category) and regression.
- **unsupervised learning** 无监督学习 – no labels; the algorithm finds structure, e.g. a **cluster** 聚类 of similar customers.
- reinforcement learning (below).

Use ML when explicit rules would be impractical (spam filters, recommendations, fraud detection).

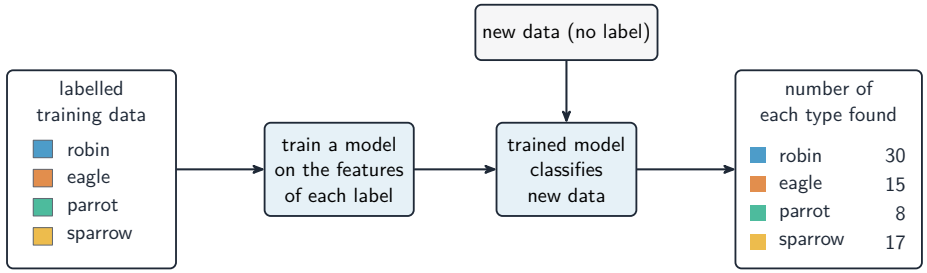


The same data seen two ways: with labels, the task is to learn what separates the classes; without labels, the task is to discover that there are groups at all

"Describe supervised learning and unsupervised learning" (the marked wordings). *Supervised learning:* the algorithm is trained on **labelled training data** 训练数据, each example paired with the **correct output** (the target); it learns the **relationship between inputs and outputs** and uses it to classify or predict for new inputs; the answers are known while training, so the error can be measured. *Unsupervised learning:* the data is **unlabelled**, with no correct answers given; the algorithm

exercises.lol exercises.com.cn

looks for **patterns, structure or groupings** in the data by itself (clustering similar items, finding associations); the output is a set of categories or relationships that were not defined in advance. **How they differ:** labelled against unlabelled data; known outputs against discovered structure; supervised is used to **predict** (classification, regression), unsupervised to **explore** (clustering, anomaly detection). Both are categories of machine learning; the third is reinforcement learning.



Supervised learning: a model is trained on labelled data, then recognises new data

Deep learning

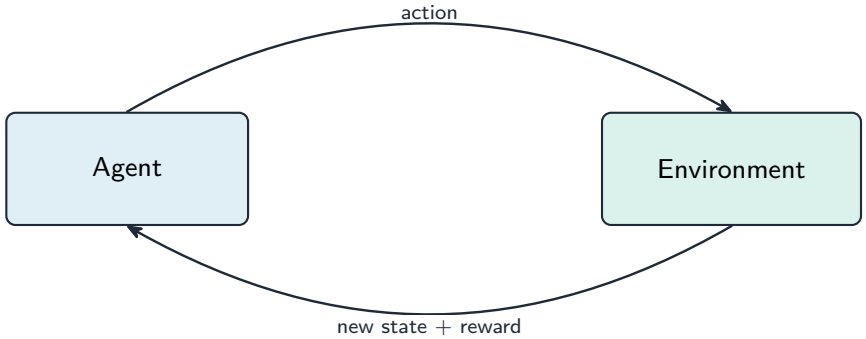
A subset of ML using deep neural networks. Lower layers learn simple patterns (edges, phonemes), higher layers combine them into abstract concepts. It needs **lots of data** and **lots of compute** (GPUs); for small datasets, simpler ML methods often do better.

"Explain what is meant by deep learning" (three marks). *Machine learning that uses artificial neural networks with **many hidden layers** (deep networks); the network is trained on **very large amounts of data**, and each layer **extracts features** from the output of the layer below, so that the network learns the features it needs by itself rather than having them specified by the programmer.* **Reasons for using it:** it can solve problems too **complex** for hand-written rules or shallow models (recognising faces, understanding speech, translating text); it **improves as more data** becomes available; it removes the need for human **feature engineering**; and it can handle **unstructured** data such as images, sound and text. **How it is made more effective:** more (and better-labelled) training data; more layers or nodes, within the limits of overfitting; more processing power (GPUs) and training time; tuning the learning rate and other parameters. **Examples:** speech recognition in voice assistants, image recognition in medical scans and self-driving cars, machine translation, recommendation systems.

Reinforcement learning

In **reinforcement learning** 强化学习, an **agent** 智能体 acts in an environment; each action changes the state and returns a **reward** 奖励. The agent learns a **policy** 策略 (a strategy) that maximises the total reward over time, by **trial and error** with no labels up front. Used for sequential-decision problems – games, robot control, autonomous driving.

"Explain what is meant by reinforcement learning" (three marks). *An **agent** learns by **interacting with its environment**: it takes an action, the environment moves to a new state and returns a **reward** (or penalty), and the agent adjusts its behaviour so as to **maximise the total reward** over time.* There is **no labelled data**: the agent learns by **trial and error**, discovering which actions are good from the rewards it collects, and gradually forms a **policy** that says what to do in each state. Used where the right answer is not known in advance but the result of an action can be scored: game playing (chess, Go), robot control, traffic-light timing, resource allocation. A computer playing a board game against a user learns in this way, or searches the game tree with minimax to choose the move whose worst outcome is best.



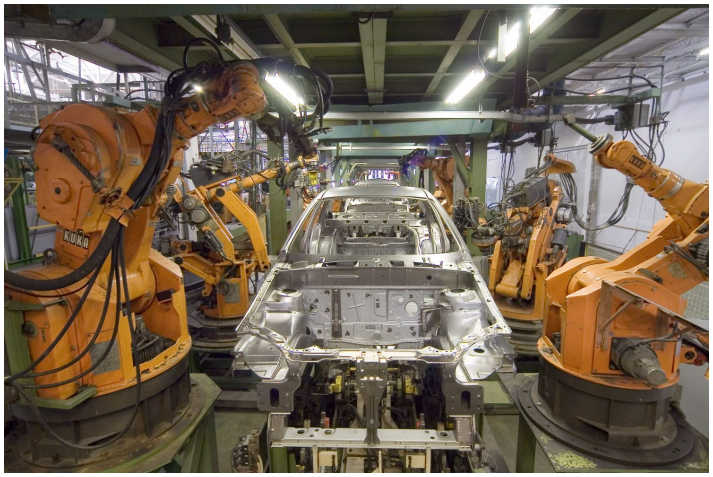
Reinforcement learning: the agent acts, the environment returns a new state and a reward, and the agent learns from it

A **self-driving car** 自动驾驶汽车 is a real example. **Lidar** 激光雷达 and camera sensors (the spinning unit on the roof) build a live picture of the road, and a learned policy decides how to steer, speed up and brake safely.

exercises.lol exercises.com.cn



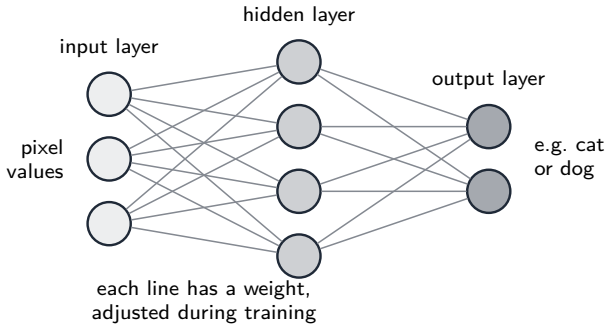
A self-driving car uses cameras and lidar sensors to see the road around it



Industrial robot arms on a production line: reinforcement learning can teach a robot to control its movements

Training an ANN: backpropagation

Training adjusts the **weights** so outputs match the targets. The standard method is **backpropagation** 反向传播 (back propagation of errors) with **gradient descent** 梯度下降. For each training example:



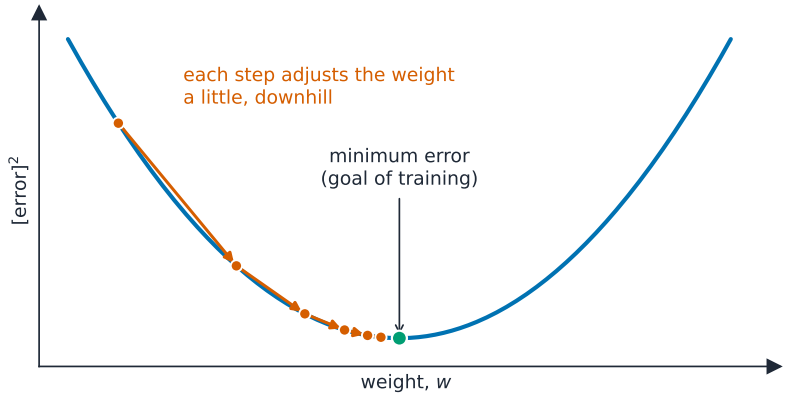
Training adjusts the weights until the outputs match the labels

1. **forward pass** – feed the input through to the output.
2. **compute the error** with a **loss function** 损失函数 (a single number for how wrong the output is).
3. **backward pass** – propagate the error **backwards**, finding each weight's **gradient** (how much it contributed to the error) using the chain rule.
4. **update the weights** by a small step (set by the **learning rate** 学习率) that reduces the error.

Repeat over many examples and many passes (**epochs** 训练轮次) until the error stops shrinking. The name "back" comes from step 3: the error flows from the output **back** towards the input, so every weight's gradient is found in one sweep. After training, a new input needs only one forward pass to get a prediction.

"Describe the back propagation of errors method" (four marks). (1) An input is fed **forward** through the network and its output is compared with the **expected (target) output**; (2) the difference is the **error**; (3) the error is passed **backwards** through the network, layer by layer from the output to the input, and each weight's share of the error is calculated; (4) the **weights are adjusted** in proportion to their contribution, in the direction that **reduces** the error; (5) the process is **repeated** with many examples until the error is as small as required. The point of the method is that a network with hidden layers has no direct way of knowing which internal weight caused an output error; back propagation apportions the blame.

exercises.lol exercises.com.cn



gradient descent: update w opposite the slope of the error surface

Training adjusts the weights to reach the minimum error

Regression

Some tasks predict a **number** (a house price, tomorrow's temperature) – **regression** 回归, as opposed to classification (a category).

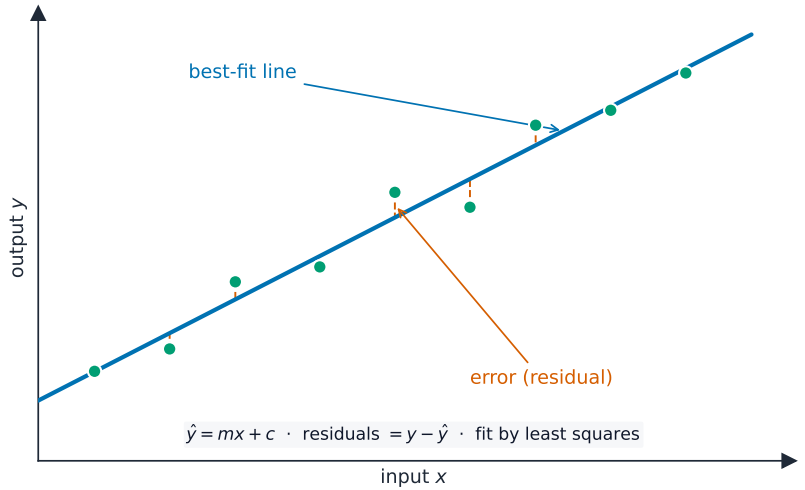
Linear regression 线性回归 fits a straight line (or hyperplane):

$$y = m_1x_1 + m_2x_2 + \ldots + m_nx_n + c.$$

Choose the coefficients to **minimise the sum of squared errors** against the training data. Use it when the relationship looks roughly linear and you want an interpretable model. For curved data, use polynomial, decision-tree, or neural-network **regression methods** – same idea: define a model, define a loss, and adjust the parameters to minimise it. Regression and classification are both supervised; the choice depends on whether the answer is a number or a category.

"Describe regression methods in machine learning" (two marks). *Statistical methods that find the* ***relationship*** *between input variables and a* ***continuous*** *output, by fitting a function (a line or curve) to the training data with the smallest total error; the fitted function is then used to* ***predict*** *the output for new inputs.* Linear regression fits a straight line; other methods fit curves. Regression predicts a **value** (a price, a temperature, a time); classification predicts a **category**, which is the distinction the exam asks for.

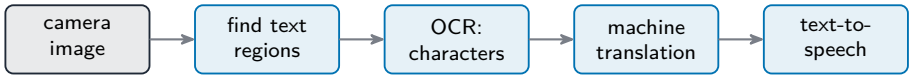
exercises.lol exercises.com.cn



Linear regression fits the line that makes the total squared error (the dashed gaps) as small as possible

How AI is used in a real scenario

Many exam scenarios use the same pattern – a **deep-learning model trained on labelled data**, often several combined into a pipeline:



each blue stage: a model trained on labelled data, now only doing fast inference

A real system chains several trained models.

- **customer identification** at an automated shop: the system is trained on labelled face images; a camera captures a face; image recognition extracts a representation; it is matched against registered customers; the closest match identifies the person.
- **reading text from images**: image recognition finds text regions; **optical character recognition** 光学字符识别 extracts the characters; machine translation converts them; **text-to-speech** 文本转语音 reads them aloud.
- **checkout item-detection**: object-detection AI, trained on labelled product images, sees which items go into a basket and charges the account.

By the time a user interacts with the system, the model is fast – it only does forward-pass inference; the intelligence is in the patterns learned during training.

Model answers for the scenario questions. *A car-park camera reads registration numbers:* the camera captures an image; an AI trained on many **labelled images** of number plates locates the plate in the image; **character recognition** (a deep-learning classifier, again trained on labelled characters) converts the plate into text; the text is stored with the time and matched when the car leaves. *A CCTV system detects and tracks a person:* image-recognition software trained on labelled images of people identifies a person in each frame; the system compares successive frames to follow their movement; unusual movement can trigger an alert. *Speech turned into commands:* speech recognition trained on many recorded voices converts the sound into text; the system matches the text to a set of known commands; it improves as it is corrected. *A camera that focuses on faces:* a face-detection model trained on labelled faces finds the face region, and the lens is adjusted to bring that region into focus. *A bank's face-recognition login:* the app captures the face, a deep network extracts its features, and they are compared with the stored features for that customer. In every case the pattern is: **trained on labelled examples, extracts features, matches or classifies new input**.

> **Worked example.** For each task, say whether it needs **regression** or **classification**, and what the output layer of an ANN would look like: (a) predict tomorrow's temperature; (b) decide whether an email is spam. Ask what **kind of thing** is being predicted. (a) A temperature is a **number** on a continuous scale, so this is **regression**, and the output layer is a **single** neuron holding that value. (b) Spam or not-spam is a **category**, so this is **classification**, and the output gives a probability per class. Both are **supervised** learning: each needs labelled examples to train on, and training adjusts the **weights** by **backpropagation** to reduce the error. The deciding question is simply number-or-category - not how difficult the task feels.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
graph (in AI)	a set of nodes representing states or places, joined by edges representing connections, often weighted, that a search algorithm can explore
Dijkstra's algorithm	finds the shortest distance from a start node to every other node by always visiting the unvisited node with the smallest distance so far
A* algorithm	a shortest-path search that expands the node with the smallest total of distance so far plus a heuristic estimate of the distance to the goal

exercises.lol exercises.com.cn

Term	Definition
artificial neural network	a model of the brain's neurons: layers of nodes joined by weighted connections, trained by adjusting the weights
machine learning	algorithms that learn from data and improve with experience rather than following fixed rules
supervised learning	learning from labelled training data in which the correct output for each input is known
unsupervised learning	learning from unlabelled data by finding patterns, groupings or structure in it
reinforcement learning	an agent learns by trial and error, choosing actions in an environment to maximise the rewards it receives
deep learning	machine learning using neural networks with many hidden layers, trained on large amounts of data, each layer extracting features from the one below
back propagation of errors	comparing the network's output with the target, passing the error back through the layers and adjusting each weight to reduce it
regression	fitting a function to training data in order to predict a continuous output value from inputs

Exam tips

- Graph answers name nodes, edges and weights, and what they represent; then the algorithm. Dijkstra: table of distances, visit the smallest, update neighbours. A*: g, h and $f = g + h$ in every row, expand the smallest f.
- ANN answers name the layers, the weighted connections and training; deep learning adds many hidden layers, large data and automatic feature extraction, with a reason and an example.
- The three categories in one line each: labelled data and known outputs; unlabelled data and discovered structure; agent, environment, actions and rewards.
- Back propagation: compare with the target, error backwards through the layers, adjust weights to reduce it, repeat. Regression predicts a value; classification predicts a category.
- Scenario questions want the pipeline: trained on labelled examples, extracts features, recognises or classifies new input; name the type of AI (image recognition, speech recognition, deep learning).

Common mistakes

- Describing a graph as “a chart”; in AI it is nodes and edges.
- Running Dijkstra by picking the nearest neighbour of the current node rather than the smallest overall distance not yet visited; or forgetting to update a node when a shorter route appears.
- Adding h into g for the next step in A*; g is only the real distance, h is recomputed from the table.
- Saying deep learning is “learning a lot”; it is the many hidden layers.
- Confusing unsupervised learning with reinforcement learning; the first finds structure in data, the second learns from rewards.
- Describing back propagation without the comparison with the expected output or without saying the weights are adjusted.
- Calling a prediction of a price “classification”; a continuous value is regression.

exercises.lol exercises.com.cn

Computational thinking and Problem-solving

A-Level Computer Science

Searching algorithms

A **search** finds a **target value** in a collection (often an **array** 数组) and returns its position, or “not found”.



Searching a sorted list, like a phone book, is far faster than checking every entry one by one

Linear search

A **linear search** 线性查找 walks from start to end, comparing each element with the target:

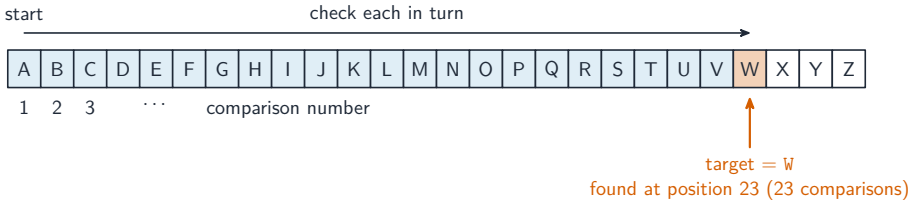
```
FOR i ← 1 TO n
    IF A[i] = target THEN
        RETURN i
    ENDIF
NEXT i
RETURN -1   // not found
```

No preparation is needed, so it works on any list. Worst case O(n) (target at the end or absent); best case 1 comparison. Use it on **unsorted** data or small lists. (The returned `-1` is a **sentinel** value – an impossible position that means “not found”; the caller tests `IF result = -1`.)

The exam's version. Paper 3 asks you to complete a linear search written with a flag and a `WHILE` loop, and Paper 4 to write a function that returns the index or a count. Both look like this:

```
FUNCTION LinearSearch(Data : ARRAY OF INTEGER, Target : INTEGER) RETURNS
↪  INTEGER
    DECLARE Index, Count : INTEGER
    Count ← 0
    FOR Index ← 1 TO 100
        IF Data[Index] = Target THEN
            Count ← Count + 1
        ENDIF
    NEXT Index
    RETURN Count          // how many times Target occurs; 0 means not
    ↪  found
ENDFUNCTION
```

To stop at the **first** match instead, use a `WHILE Index <= 100 AND NOT Found` loop that sets `Found ← TRUE` and remembers the index. The marks are for the loop over every element, the comparison, and what is returned when the value is absent.



Linear search checks every letter in turn – 23 comparisons to find W

Binary search

A **binary search** 二分查找 needs the data **sorted**. Look at the middle element; if it is the target, done; if the target is smaller, search the left half, else the right half – halving the range each time:

```
low ← 1
high ← n
WHILE low <= high DO
    MiddleIndex ← (low + high) DIV 2
    IF A[MiddleIndex] = target THEN
        RETURN MiddleIndex
    ENDIF
    IF A[MiddleIndex] < target THEN
        low ← MiddleIndex + 1
    ELSE
        high ← MiddleIndex - 1
    ENDIF
ENDWHILE
RETURN -1
```

Worst case $O(\log_2 n)$ – for a million items, about 20 comparisons. Much faster than linear search on large sorted arrays, but you must sort first (a one-off $O(n \log n)$ cost), worth it if you search many times.

"State the condition necessary for a binary search." *The data must be in order* (sorted, ascending or descending, on the key being searched). **"Describe how to perform a binary search" (three marks):** (1) find the **middle** item of the list (or of the current range) and compare it with the target; (2) if it matches, the search ends; if the target is **smaller**, repeat on the **lower half**, if larger, on the **upper half**; (3) keep **halving** the range until the item is found or the range is empty, which means it is not present.

exercises.lol exercises.com.cn

The exam's version, with the bounds and a flag, is the one to reproduce when asked to complete the algorithm:

```
DECLARE Lower, Upper, MiddleIndex : INTEGER
DECLARE Found : BOOLEAN
Lower ← 0
Upper ← 99
Found ← FALSE
WHILE Lower <= Upper AND NOT Found
    MiddleIndex ← (Lower + Upper) DIV 2
    IF Names[MiddleIndex] = Target THEN
        Found ← TRUE
    ELSE
        IF Names[MiddleIndex] < Target THEN
            Lower ← MiddleIndex + 1
        ELSE
            Upper ← MiddleIndex - 1
        ENDIF
    ENDIF
ENDWHILE
IF Found THEN
    OUTPUT MiddleIndex
ELSE
    OUTPUT "Not found"
ENDIF
```

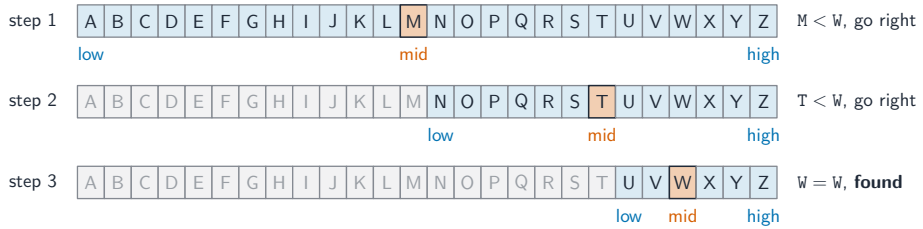
Worked example (tracing a binary search). `Names[0:9]` holds `Ali, Ben, Cai, Dev, Eva, Fay, Gus, Hal, Ivy, Jo`. Trace the exam's version for `Target = "Ivy"`, then for `Target = "Cat"`, giving `Lower`, `Upper` and `MiddleIndex` at each step.

For "Ivy":

Lower	Upper	MiddleIndex	Names[MiddleIndex]	Result
0	9	4	Eva	Eva < Ivy, so Lower ← 5
5	9	7	Hal	Hal < Ivy, so Lower ← 8
8	9	8	Ivy	Found ← TRUE

The output is `8`, after **three** comparisons. For "Cat": middle 4 (Eva > Cat, Upper ← 3); middle 1 (Ben < Cat, Lower ← 2); middle 2 (Cai < Cat, Lower ← 3); middle 3 (Dev > Cat, Upper ← 2). Now `Lower` (3) is greater than `Upper` (2), so the loop ends and "Not found" is output after **four** comparisons, the worst case for ten items ($\lceil \log_2 10 \rceil = 4$). `DIV` rounds down, so the middle of 0 and 9 is 4, not 4.5 or 5. Strings compare in alphabetical (character code) order.

"Explain how the performance varies with the number of items." Each comparison **halves** the number of items left, so the maximum number of comparisons is about $\log_2 n$: doubling the size of the list adds only **one** more comparison. This is **O($\log n$)**. **"Compare linear and binary search":** a linear search needs up to n comparisons (O(n)) and, on average, half that, but works on **unsorted** data; a binary search needs at most $\log_2 n$ (O($\log n$)) and is far faster for large lists, but the data must first be **sorted** and it must allow **direct access** to the middle item (an array, not a linked list). For 1000 items: 1000 against 10 comparisons.



Binary search halves the range each step (low / mid / high) – just 3 comparisons to find W



A card catalogue: sorted records are what make a binary search possible – halve, look, halve again

exercises.lol exercises.com.cn

Sorting algorithms

Bubble sort

A **bubble sort** 冒泡排序 repeatedly walks the array, swapping adjacent pairs that are out of order, so the largest "bubbles" to the end each pass:

```
FOR pass ← 1 TO n - 1
    swapped ← FALSE
    FOR i ← 1 TO n - pass
        IF A[i] > A[i + 1] THEN
            temp ← A[i]
            A[i] ← A[i + 1]
            A[i + 1] ← temp
            swapped ← TRUE
        ENDIF
    NEXT i
    IF swapped = FALSE THEN      // already sorted
        EXIT FOR
    ENDIF
NEXT pass
```

Best case $O(n)$ (already sorted, with the early exit); average/worst $O(n^2)$. Simple but slow for large n.

Insertion sort

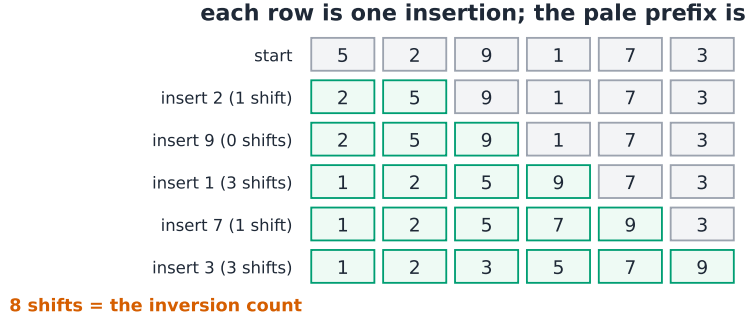
An **insertion sort** 插入排序 builds a sorted prefix from the left, inserting each new element into place by shifting larger ones right:

```
FOR i ← 2 TO n
    key ← A[i]
    j ← i - 1
    WHILE j >= 1 AND A[j] > key DO
        A[j + 1] ← A[j]
        j ← j - 1
    ENDWHILE
    A[j + 1] ← key
NEXT i
```

Best case $O(n)$ (already sorted); worst $O(n^2)$. Good for **small** or **nearly-sorted** arrays. It sorts **in place** 原地 and is **stable** 稳定 (keeps the order of equal elements).

Tracing a sort

A common task is to show the array after each outer pass. For `[D, T, H, R]` with insertion sort: pass 1 (key T) no change; pass 2 (key H) → `[D, H, T, R]`; pass 3 (key R) → `[D, H, R, T]`.



Insertion sort on the course dataset: each row is one insertion; the shaded prefix is already sorted

The same array with **bubble sort** moves differently: pass 1 compares D/T (no swap), T/H (swap) and T/R (swap), giving `[D, H, R, T]` with the largest item at the end; pass 2 makes **no** swaps, so the flag stops the sort. Show the whole array after each pass, and count the comparisons if asked: 3 in pass 1 and 2 in pass 2 (the version above stops one place earlier each pass), 5 in all.

exercises.lol exercises.com.cn

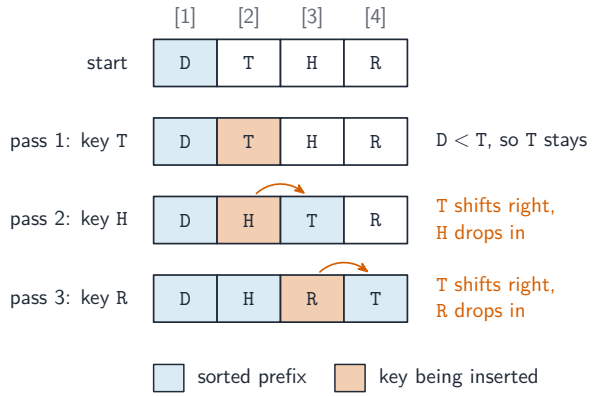
Writing a sort from scratch. "Write pseudocode to sort `DataArray[1:1000]` into ascending order" is answered by a complete bubble sort with the early-exit flag, or an insertion sort, declared and indented; either scores full marks if it works for every input:

```
DECLARE Pass, Index, Temp : INTEGER
DECLARE Swapped : BOOLEAN
Pass ← 1
REPEAT
    Swapped ← FALSE
    FOR Index ← 1 TO 1000 - Pass
        IF DataArray[Index] > DataArray[Index + 1] THEN
            Temp ← DataArray[Index]
            DataArray[Index] ← DataArray[Index + 1]
            DataArray[Index + 1] ← Temp
            Swapped ← TRUE
        ENDIF
    NEXT Index
    Pass ← Pass + 1
UNTIL Swapped = FALSE OR Pass = 1000
```

For **descending** order change `>` to `<`; to sort records or a 2D array by one field, compare that field but swap the **whole** record (or every column). Asked to write an insertion sort "that performs the same task" as a given bubble sort, keep the same array name and direction and reproduce the insertion sort above with the comparison reversed if the order is descending.

"Describe two ways the performance of a sort is affected by the data" (two marks). (1) The **number** of items: an $O(n^2)$ sort takes four times as long for twice as many items. (2) How far the data is already **in order**: a bubble sort with a flag, or an insertion sort, finishes in one pass over already-sorted data ($O(n)$) and does the most work on data in reverse order; the number of swaps depends on how many pairs are out of order. (Also accepted: the range or number of duplicate values, and whether the items are large records that are expensive to move.) Bubble and insertion sort are both $\mathrm{O}(n^2)$ in the worst and average cases and $\mathrm{O}(n)$ at best; quicksort and merge sort are $\mathrm{O}(n \log n)$, which is why they are used for large data.

exercises.lol exercises.com.cn



An insertion sort of `[D, T, H, R]`, shifting each key into its place pass by pass

ADTs in algorithms

The **Abstract Data Types** (ADTs) from Topic 10 appear inside many algorithms: a **stack** 栈 drives depth-first traversal and undo; a **queue** 队列 drives breadth-first traversal and print ordering; a **linked list** 链表 lets data grow and shrink.

ADTs can be built from other ADTs, not just from arrays: a queue from **two stacks**; a stack from a linked list (push = prepend a head **node** 节点); a queue from a linked list with head and tail **pointers** 指针; a **binary tree** 二叉树 from nodes with two child pointers; a **dictionary** 字典 stores key→value pairs (often on a hash table). Layering this way separates concerns – the algorithm using the ADT need not know how it is built.

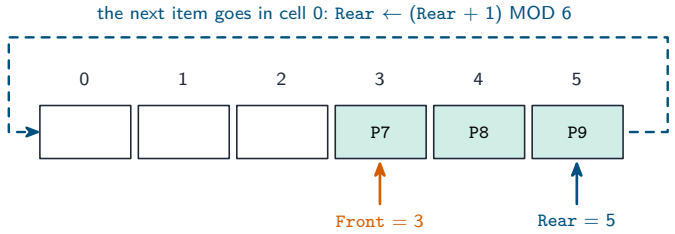
The ADTs the exam asks you to describe and implement

Stack (last in, first out): items are added (**pushed**) and removed (**popped**) at the same end, the **top**; a pointer `TopOfStack` holds the index of the top item. Implemented with an array and that one pointer: push checks the stack is not full, increments the pointer and stores the item; pop checks it is not empty, returns the top item and

decrements the pointer.

```
FUNCTION Push(Item : INTEGER) RETURNS BOOLEAN
    IF TopOfStack = 9 THEN      // full (array 0 to 9)
        RETURN FALSE
    ENDIF
    TopOfStack ← TopOfStack + 1
    StackData[TopOfStack] ← Item
    RETURN TRUE
ENDFUNCTION
FUNCTION Pop() RETURNS INTEGER
    IF TopOfStack = -1 THEN      // empty
        RETURN -1
    ENDIF
    TopOfStack ← TopOfStack - 1
    RETURN StackData[TopOfStack + 1]
ENDFUNCTION
```

Queue (first in, first out): items join at the **rear** (**enqueue**) and leave from the **front** (**dequeue**); two pointers and a count. In a **linear** queue the front pointer creeps along the array until the space at the start is wasted; a **circular queue** 循环队列 wraps both pointers round with `MOD`, so every cell is reused.



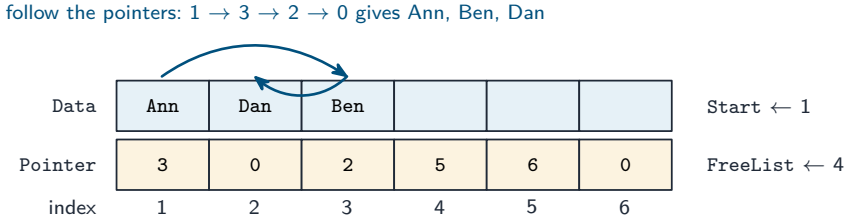
three items P7, P8, P9 after earlier items were dequeued from cells 0 to 2;
dequeue takes `Queue[Front]` then `Front` ← (`Front` + 1) MOD 6; a count (or a flag) tells full from empty

A circular queue: the rear and front pointers step forward with MOD, so the array's first cells are reused once their items have left

exercises.lol exercises.com.cn

```
FUNCTION Enqueue(Item : STRING) RETURNS BOOLEAN
    IF Count = 6 THEN      // full
        RETURN FALSE
    ENDIF
    Rear ← (Rear + 1) MOD 6
    QueueArray[Rear] ← Item
    Count ← Count + 1
    RETURN TRUE
ENDFUNCTION
FUNCTION Dequeue() RETURNS STRING
    IF Count = 0 THEN      // empty
        RETURN ""
    ENDIF
    DECLARE Item : STRING
    Item ← QueueArray[Front]
    Front ← (Front + 1) MOD 6
    Count ← Count - 1
    RETURN Item
ENDFUNCTION
```

Linked list: a sequence of **nodes**, each holding a data item and a **pointer** to the next node; a **start pointer** gives the first node and a null pointer (0 or −1) ends the list. In an array implementation two parallel arrays hold the data and the pointers, and unused cells are chained into a **free list** 空闲列表 so that an insertion knows where to put the new node.



a pointer of 0 marks the end of a list; unused cells are chained into the free list 4 → 5 → 6
to insert Cal: take cell 4 from the free list, store Cal, set `Pointer[4]` ← 2, `Pointer[3]` ← 4
and `FreeList` ← 5 (the next free cell)

A linked list in two arrays: the order of the list is in the pointers, not in the positions; inserting a name means taking a cell from the free list and re-linking two pointers

```
FUNCTION FindInList(Target : STRING) RETURNS INTEGER   // index, or 0 if
↪  absent
    DECLARE Current : INTEGER
    Current ← Start
    WHILE Current <> 0
        IF Data[Current] = Target THEN
            RETURN Current
        ENDIF
        Current ← Pointer[Current]
    ENDWHILE
    RETURN 0
ENDFUNCTION
```

To **insert** into an ordered list: take the first free cell (`NewNode ← FreeList`, `FreeList ← Pointer[FreeList]`), store the item, then walk the list with a `Previous` and `Current` pointer until `Data[Current] > Item` or the end; set `Pointer[NewNode] ← Current` and `Pointer[Previous] ← NewNode` (or `Start ← NewNode` if it goes first). To **delete**, re-link the previous node past the deleted one and return the cell to the free list.

Binary tree: a **root** node, each node holding data, a **left pointer** to a subtree of smaller values and a **right pointer** to a subtree of larger values. Implemented as a 2D array (or three 1D arrays) `Tree[Index, 0..2]` for left pointer, data, right pointer, with a root pointer and a next-free pointer.

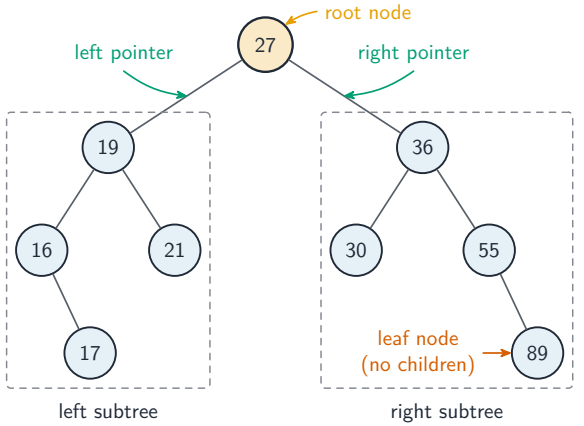
```
FUNCTION FindInTree(Target : INTEGER) RETURNS INTEGER   // index, or -1
    DECLARE Current : INTEGER
    Current ← Root
    WHILE Current <> -1
        IF Tree[Current, 1] = Target THEN
            RETURN Current
        ENDIF
        IF Target < Tree[Current, 1] THEN
            Current ← Tree[Current, 0]      // go left
        ELSE
            Current ← Tree[Current, 2]      // go right
        ENDIF
    ENDWHILE
    RETURN -1
ENDFUNCTION
```

To **insert**: store the item in the next free node with both pointers -1; if the tree is empty make it the root; otherwise walk down from the root, going left or right by comparison, until the pointer you would follow is -1, and set that pointer to the new node. **An ADT from another ADT:** a stack is a linked list where push and pop both work at the start; a queue is a linked list with a start and an end pointer; a queue

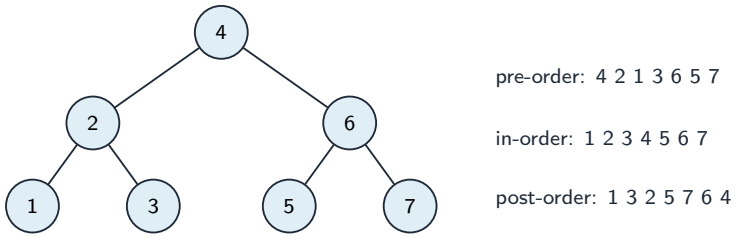
exercises.lol exercises.com.cn

can be made from **two stacks** (push onto one, pop from the other, moving everything across when the second is empty); a binary tree's nodes are records or objects linked by pointers, so it is built from a linked structure of nodes. Say which operations of the new ADT map onto which operations of the old one.

The first figure shows a binary search tree: each node has up to two child nodes, values smaller than a node go to its left, and larger values go to its right. The second figure shows the three ways to traverse a tree.



A binary search tree: each node has up to two child nodes



Three depth-first traversals of a binary tree: pre-order, in-order (sorted order) and post-order

Comparing algorithms

Time complexity

Time complexity 时间复杂度 is how the running time grows with input size n, written in **Big-O notation** 大 O 表示法 (the dominant term): O(1) constant, O($\log n$)

binary search, O(n) linear search, O($n \log n$) good sorts, O(n^2) bubble/insertion sort. A smaller order is better at scale, even if another algorithm is faster for small n.

To make that concrete: to sort a million items, an $O(n \log n)$ sort finishes in a fraction of a second, while an $O(n^2)$ sort can take minutes.

> **Worked example.** A sorted list holds 1000 items. How many comparisons does each search need in the worst case?

A linear search checks items one at a time, so it may need up to 1000 comparisons – this is $O(n)$. A binary search halves the list each step, so it needs at most $\lceil \log_2 1000 \rceil = 10$ comparisons – this is $O(\log n)$. Doubling the list to 2000 items adds only **one** comparison to the binary search, but up to another 1000 to the linear search – which is why the order of growth, not raw speed, decides the winner at scale.

Describing an order. *O(1):* the time is **constant**, independent of the number of items (pushing onto a stack, reading an array element). *O(log n):* the time grows with the **logarithm** of the number of items, so **doubling** the data adds only a fixed extra step (binary search). *O(n):* the time grows **in proportion** to the number of items (linear search, one pass through a list). *O(n log n):* a little worse than linear (efficient sorts). *O(n^2):* the time grows with the **square** of the number of items, so doubling the data quadruples the time (bubble and insertion sort). "State the Big O of a binary search of `Names[0:99]`" is answered $O(\log n)$, and "describe its meaning" as above; Big O measures how the time or memory **scales**, not the actual time.

exercises.lol exercises.com.cn

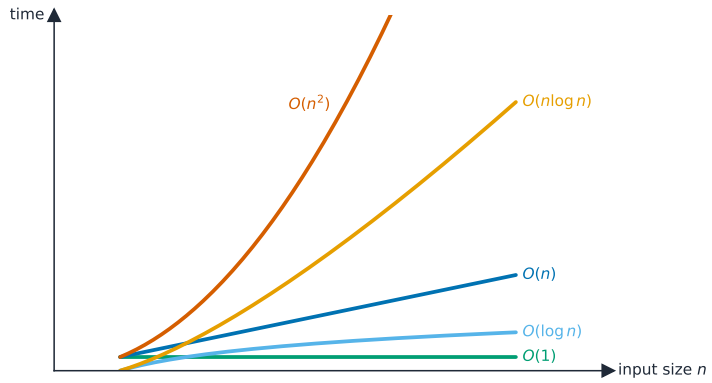
Worked example (the order of a piece of code). State the time complexity of each fragment, where `n` is the number of items.

```
// Fragment A
FOR i ← 1 TO n
    FOR j ← 1 TO n
        Total ← Total + Grid[i, j]
    NEXT j
NEXT i

// Fragment B
Count ← 0
WHILE n > 1
    n ← n DIV 2
    Count ← Count + 1
ENDWHILE

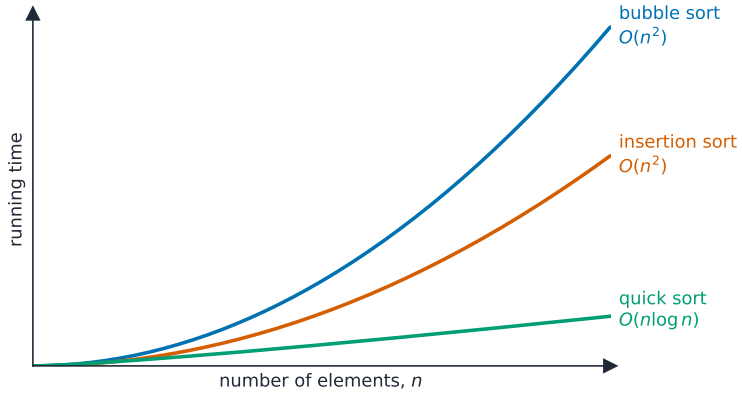
// Fragment C
FOR i ← 1 TO n
    Total ← Total + A[i]
NEXT i
FOR i ← 1 TO n
    OUTPUT A[i] / Total
NEXT i
```

A is $O(n^2)$: the inner statement runs n times for each of the n outer passes, $n \times n$ in all. **B** is $O(\log n)$: n halves each pass, so the loop runs about $\log_2 n$ times (for $n = 64$, six passes). **C** is $O(n)$: two loops **one after the other** take $n + n = 2n$ steps, and constants are dropped. The rule: nested loops **multiply**, loops in sequence **add**, and only the fastest-growing term is kept, so $3n^2 + 5n + 2$ is $O(n^2)$.



for large n: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$

How the common orders of growth compare: a smaller order wins at scale



for large n: $O(n \log n)$ (quick/merge) beats $O(n^2)$ (bubble/insertion)

How sorting time grows with the number of elements n: $O(n^2)$ sorts climb away from an $O(n \log n)$ sort

exercises.lol exercises.com.cn

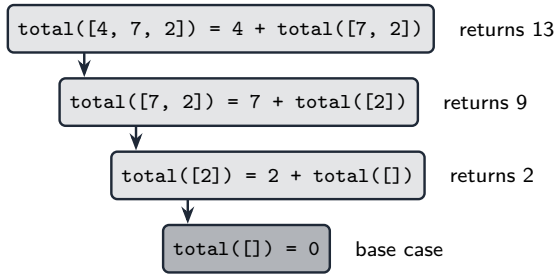
Space complexity

Space complexity 空间复杂度 is the extra memory needed. Bubble and insertion sort use O(1) extra (in place); merge sort uses O(n); recursion uses stack memory proportional to its depth. There is often a time–memory trade-off.

Other criteria

Simplicity (easier to code and maintain), stability, and adaptiveness (faster on nearly-sorted data). The right algorithm depends on the data and the constraints.

Recursion



Calls go down to the base case; results come back up

Recursive algorithms use **recursion** 递归: the routine calls itself with a smaller version of the same problem, until a **base case** 基本情形 ends the chain. It has two parts: the base case (small enough to solve directly – without it the recursion never stops) and the **recursive case** 递归情形 (reduce the input and call itself).

Factorial 阶乘:

```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
    IF n = 0 OR n = 1 THEN
        RETURN 1
    ELSE
        RETURN n * Factorial(n - 1)
    ENDIF
ENDFUNCTION
```

Recursion is natural for self-similar problems: trees, **divide-and-conquer** 分治 (binary search, merge sort), and nested data. When it is a poor fit, a loop is usually cleaner.

"Describe what is meant by recursion" (two marks). *A function or procedure that is defined in terms of itself: it **calls itself** from within its own body, with a **smaller** version of the problem each time, until a base case is reached.* **"State three**

essential features of recursion": (1) a **base case** (stopping condition) that returns a value without a further call; (2) a **general case** 一般情形 in which the routine **calls itself**; (3) each call moves the problem **closer to the base case** (the parameter is reduced), so that the recursion terminates. Some schemes add: values are returned as the calls **unwind**.

"Describe when the use of recursion is beneficial, and give an example." When the problem is **naturally defined in terms of smaller versions of itself**, so that the recursive solution is **shorter, clearer and closer to the mathematical definition** than a loop would be: a factorial or Fibonacci number, a binary search, traversing a binary tree, merge sort or quicksort, and processing nested structures such as folders within folders. It is a poor choice when the depth is large (the stack may overflow) or when the same sub-problem is computed many times (naive Fibonacci).

Tracing a recursive call

For `Factorial(4)`: the calls go down to `Factorial(1)=1`, then **unwinding** multiplies back up: `2*1=2`, `3*2=6`, `4*6=24`. Final result `24`. Track each pending call on a stack.

Worked example. The function below is given without an explanation. Trace `Unknown(3, 5)` and state its output and return value.

```
FUNCTION Unknown(BYVAL X, BYVAL Y : INTEGER) RETURNS INTEGER
    IF X < Y THEN
        OUTPUT X + Y
        RETURN Unknown(X + 1, Y - 1) + 1
    ELSE
        RETURN 0
    ENDIF
ENDFUNCTION
```

Call 1: $X = 3, Y = 5$: $3 < 5$, output **8**, call `Unknown(4, 4)`. Call 2: $4 < 4$ is false, return **0**. Unwinding: call 1 returns $0 + 1 = 1$. Output 8, return value **1**. Write the trace as a table with a row per call (parameters, condition, output, what it returns), and do the returns from the deepest call upwards: that is the unwinding the mark scheme looks for.

> **Worked example (Fibonacci).** `Fib(n)` returns `n` when `n < 2`, otherwise `Fib(n - 1) + Fib(n - 2)`. Find `Fib(5)`.

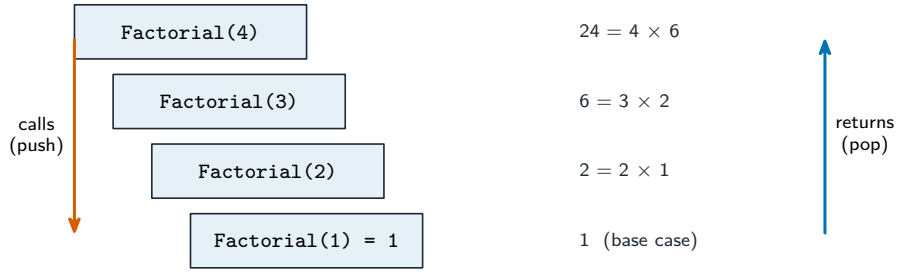
`Fib(5) = Fib(4) + Fib(3)`; `Fib(4) = Fib(3) + Fib(2)`; `Fib(3) = Fib(2) + Fib(1)`; `Fib(2) = Fib(1) + Fib(0) = 1 + 0 = 1`. So `Fib(3) = 1 + 1 = 2`, `Fib(4) = 2 + 1 = 3`, `Fib(5) = 3 + 2 = 5`. The base case is reached **many** times (`Fib(2)` is computed three times), which is why this version is slow: it makes 15 calls for $n = 5$ and roughly doubles the calls for every increase in n.

exercises.lol exercises.com.cn

Converting recursion to iteration. Every recursive routine can be rewritten with a loop, which uses less memory and is faster: keep a running result and loop from the base case upwards. Factorial as a loop:

```
FUNCTION Factorial(N : INTEGER) RETURNS INTEGER
    DECLARE Result, Count : INTEGER
    Result ← 1
    FOR Count ← 2 TO N
        Result ← Result * Count
    NEXT Count
    RETURN Result
ENDFUNCTION
```

Asked to change a recursive insertion sort or search into an iterative one, replace the self-call with a loop over the index that the recursion was stepping through, and turn the base case into the loop's exit condition.



Recursion uses the call stack: calls push frames down to the base case, then returns unwind back up

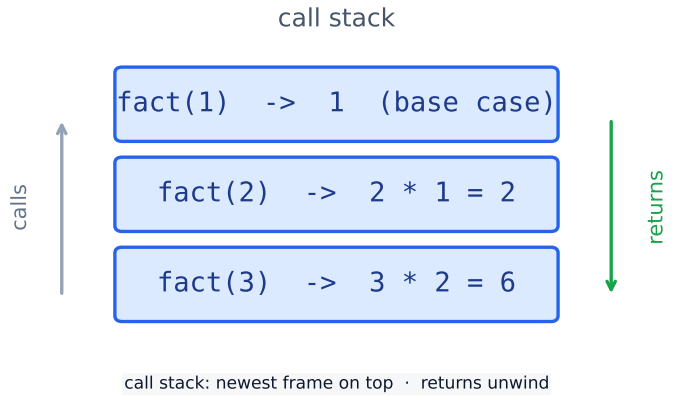
Risks

- **infinite recursion** if the base case is missed – crashes with a **stack overflow** 栈溢出.
- high memory use for deep recursion.
- slow if it repeats work (naive Fibonacci is exponential – use a loop or **memoisation** 记忆化).

What the compiler does for recursive code

Recursion needs **each call to have its own copy** of its **parameters** 参数 and **local variables** 局部变量. The compiler keeps these on the **call stack** 调用栈. For each call it pushes a **stack frame** 栈帧 holding the parameters, the local variables, and the **return address** 返回地址 (where to resume in the caller). When the function returns, the return value is handed back, the frame is popped, and control resumes at

the return address.



The call stack for factorial(3): each call waits, then returns in reverse order

Because each call has its own frame, recursive calls don't trample each other's variables. The stack can grow large for deep recursion, which is why very deep recursion may overflow it. This is the same call-and-return mechanism used for ordinary (non-recursive) calls – there is no special "recursion mechanism".

"Explain why a stack is suitable for implementing recursion" (three marks). Each recursive call must **save** its **return address**, its parameters and its local variables, and the calls are completed in the **reverse** order to that in which they were made (the last call made is the first to finish), which is exactly the **last in, first out** behaviour of a stack: each new call **pushes** a frame, and each return **pops** the most recent frame, restoring the caller's state and telling it where to continue. This is the compiler's job when it translates recursive code: it generates the push of a stack frame on every call and the pop on every return, and the frames are unwound as the results come back.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
linear search	checking each item in turn from the start until the target is found or the end is reached

exercises.lol exercises.com.cn

Term	Definition
binary search	repeatedly comparing the target with the middle item of a sorted list and discarding the half that cannot contain it
bubble sort	repeatedly passing through the list, swapping adjacent items that are in the wrong order, until a pass makes no swaps
insertion sort	taking each item in turn and inserting it into its correct place among the items already sorted
abstract data type	a collection of data and the operations that can be performed on it, defined independently of how it is stored
stack	a last-in-first-out structure with push and pop at the top
queue	a first-in-first-out structure with items added at the rear and removed from the front
linked list	a sequence of nodes, each holding data and a pointer to the next node, with a start pointer
binary tree	nodes each holding data and pointers to a left subtree of smaller values and a right subtree of larger values
Big O notation	a way of classifying the time (or memory) an algorithm needs by how it grows with the size of the input
recursion	a routine that calls itself with a smaller version of the problem until a base case stops the calls
base case	the condition under which a recursive routine returns without calling itself
unwinding	the returns of a chain of recursive calls, from the deepest call back to the first, as the stack frames are popped

Exam tips

- Searches: linear needs no order and $O(n)$; binary needs a sorted array, halves each time and is $O(\log n)$. Know both algorithms by heart, including the bounds and the flag.
- Sorts: bubble with a swapped flag, insertion with a key that shifts larger items right; both $O(n^2)$ worst, $O(n)$ on sorted data. Performance depends on the number of items and how ordered they are.
- ADT implementations are pointer bookkeeping: a top pointer; front, rear and count with MOD; start, pointers and a free list; root with left and right pointers. Always check for full and empty.

- Big O is about scaling: constant, logarithmic, linear, square. Say "doubling the data adds one comparison" for a binary search.
- Recursion: base case, general case, progress towards the base case; beneficial when the problem is defined in terms of itself; a stack holds the return addresses and variables because calls return in reverse order. Trace with a table and unwind from the deepest call.

■ Common mistakes

- Writing $O(2n)$ or $O(n^2 + n)$: drop constants and smaller terms, giving $O(n)$ and $O(n^2)$.
- Computing the middle index as $(Lower + Upper)/2$ in a trace: the exam's version uses `DIV`, so round **down**.
- Using a binary search on unsorted data, or on a linked list; and setting `Lower ← MiddleIndex` instead of `MiddleIndex + 1`, which loops for ever.
- A bubble sort inner loop that runs to the end of the array every pass, or a swap without a temporary variable.
- A push or enqueue that does not test for full, or a pop or dequeue that does not test for empty.
- Moving the queue's front pointer without MOD in a circular queue, or treating front = rear as always meaning empty.
- Inserting into a linked list by shifting the array contents; only the pointers change.
- A recursive function with no base case, or one whose recursive call does not make the problem smaller.
- Tracing a recursive call but forgetting to add the pending work on the way back up.
- Answering "why a stack" with "because it is fast"; the reason is the last-in-first-out order of the returns.

exercises.lol exercises.com.cn

Further Programming

A-Level Computer Science

Programming paradigms

A **programming paradigm** 编程范式 is a style of programming – a way of structuring programs, with its own ideas and language features. Four **programming paradigms** are in this syllabus.

"Describe what is meant by an imperative (procedural) language" (two marks). *A language in which the program is a* ***sequence of instructions*** *that are executed in order and that* ***change the program's state****; the programmer says* ***how*** *the task is done, using procedures, sequence, selection and iteration.* **"Describe what is meant by a declarative language":** *the program states* ***facts and rules*** *(what is known and what is wanted) and the language's* ***inference engine*** *works out how to find the answer; the programmer does* ***not*** *give the sequence of steps.*

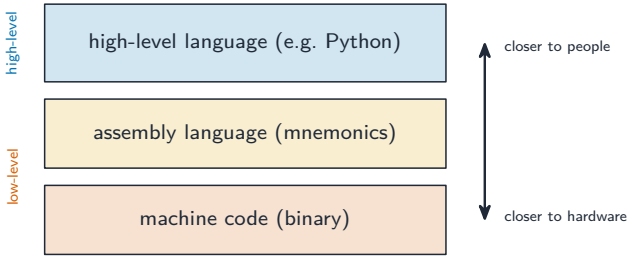
Identify the paradigm from a code sample (a regular Paper 3 question): `LDD 200`, `ADD #5`, `STO 201` is **low-level** (mnemonics, registers, memory addresses); `FOR Count ← 1 TO 10 … NEXT Count` with procedures and assignments is **imperative**; `CLASS Dog … PRIVATE Name : STRING … PUBLIC PROCEDURE NEW(…)` is **object-oriented**; `type(lion, wild).` and `dangerous(X) IF type(X, wild)` is **declarative** (logic). In the matching question: low-level pairs with "mnemonics that correspond directly to machine instructions", imperative with "a sequence of statements that change the state", OOP with "objects that combine attributes and methods", declarative with "facts and rules, with no order of execution given".



Four paradigms: low-level, imperative, object-oriented and declarative

Low-level programming

Programming **close to the hardware** in **machine code** 机器码 or **assembly language** 汇编语言, where each instruction maps to what the CPU runs. It gives direct access to **registers** 寄存器 and **memory addresses** 内存地址, using different **addressing modes** 寻址方式 (immediate, direct, indirect, indexed and relative). It is very fast and compact, but architecture-specific, tedious, and hard to maintain. This is **low-level** 低级 programming, used for device drivers, firmware and bootloaders.



High-level languages are close to human language and run on any computer; low-level languages (assembly and machine code) are close to the hardware and tied to one type of CPU

The five addressing modes. The syllabus asks for low-level code that uses each **addressing mode** (the instruction set is in Topic 4). The operand of a load instruction can be read five ways, and the exam gives you the memory contents and asks what the accumulator holds. The figure uses one memory and shows what each mode does with the operand:

address	contents
27	91
105	27
106	64
107	43

index register IX = 2
`JMR +6` is at address 40

mode	instruction	what happens
immediate	`LDM #105`	ACC = 105 (the operand itself)
direct	`LDD 105`	ACC = 27 (the contents of 105)
indirect	`LDI 105`	ACC = 91 (105 holds 27, so the contents of 27)
indexed	`LDX 105`	ACC = 43 (the contents of 105 + IX = 107)
relative	`JMR +6`	ACC does not change; jump to 40 + 6 = 46

The same operand, 105, read five ways: as a value, as an address, as the address of an address, as an address plus the index register, and as an offset from the current instruction

- **immediate** (`LDM #105`): the operand **is** the value; ACC becomes 105.

- **direct** (`LDD 105`): the operand is the **address** of the value; ACC becomes the contents of 105, here 27.
- **indirect** (`LDI 105`): the operand is the address of an address; ACC becomes the contents of 27, here 91. Used for pointers and for data whose position is decided at run time.
- **indexed** (`LDX 105`): the address is the operand **plus the index register** IX; with IX = 2, ACC becomes the contents of 107, here 43. Used to step through an array by incrementing IX.
- **relative** (`JMR +6`): the target is an **offset** from the address of the current instruction; at address 40, `JMR +6` jumps to 46. ACC does not change. This makes the code relocatable.

> **Worked example.** Memory: 105 holds 27, 106 holds 64, 200 holds 0. Write code to add the contents of 105 and 106, store the result in 200 and output it. `LDD 105` (ACC = 27), `ADD 106` (ACC = 91), `STO 200`, `OUT`. To double the value in 105 instead: `LDD 105`, `ADD 105`, `STO 105`. State the register contents after each line when asked to trace.

Imperative (procedural) programming

In **imperative programming** 命令式编程 the programmer writes a **sequence of commands** that change the program's state – assignments, conditionals, loops, function calls. **Variables** 变量 hold state; statements change it; code is organised into procedures and functions (also called **structured** or **structural programming**). This is the style of Topics 9 and 11 (Python, C). Strong when the algorithm has clear sequential steps.

Object-oriented programming (OOP)

In **object-oriented programming** 面向对象编程 programs are built from **objects** 对象 – units combining **data** (**attributes** 属性) and **operations** (**methods** 方法). Objects are **instances** 实例 of **classes** 类. The four pillars:

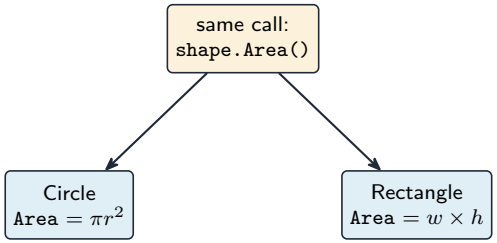
- **encapsulation** 封装 – an object's data is hidden behind its methods; outside code uses the public methods only, not the data directly. This protects the object and lets its internals change without breaking callers. For example, a `BankAccount` hides its `balance`; you change it only through `deposit()` and `withdraw()`, which can enforce a rule like "never go below zero".
- **inheritance** 继承 – a **subclass** 子类 specialises a **superclass** 父类, inheriting its attributes and methods and adding or **overriding** 重写 them. Models "is-a" ("a Manager is an Employee").
- **polymorphism** 多态 – different objects respond to the **same method call** differently; the caller need not know the exact type. Every `Shape` has `Area()`, and a `Circle` and a `Rectangle` each implement it their own way.
- **abstraction** 抽象 – show a simple interface and hide the implementation.

exercises.lol exercises.com.cn

Other terms:

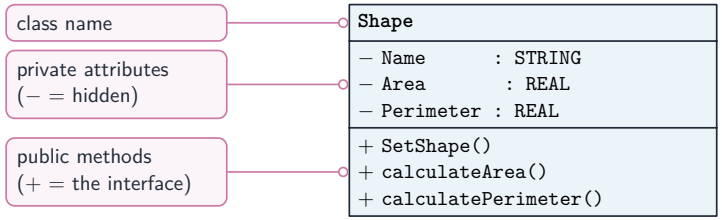
- a **constructor** 构造函数 is a special method run when an object is created, to set up its attributes.
- **getters** and **setters** read and write an object's attributes (its **properties**) through methods.
- **aggregation** 聚合 and **containment** 包含 build an object from other objects (a "has-a" relationship).

OOP is used for large systems, GUIs, simulations and games.



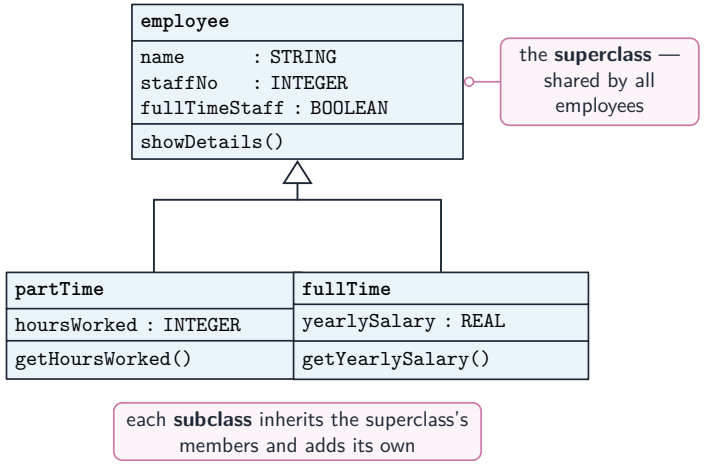
same method name — different code runs

Polymorphism: the same method call runs each object's own code

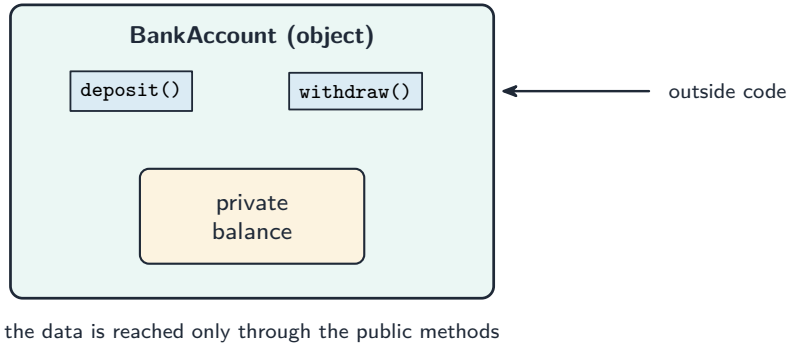


A class diagram for a Shape: private attributes and public methods

exercises.lol exercises.com.cn



Inheritance: partTime and fullTime are subclasses of employee



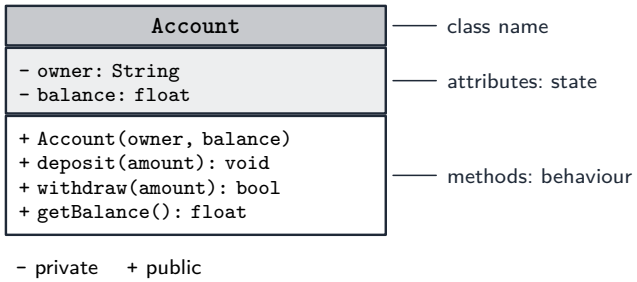
the data is reached only through the public methods

Encapsulation: an object's data is private, reached only through its public methods

■ OOP as the examiner marks it

Definitions. *Class:* a **template** (blueprint) that defines the attributes and methods of the objects of that type. *Object:* an **instance** of a class, created from it, with its own values for the attributes ("an occurrence of an object" is the exam's phrase for an instance). *Attribute (property):* a data item belonging to a class. *Method:* a procedure or function belonging to a class that acts on its attributes. *Encapsulation:* combining the attributes and methods in one class and **restricting external access** to the data: the attributes are **private** and can only be read or changed through **public** methods. *Inheritance:* a subclass **acquires** the attributes and methods of its parent (super)

class and can add its own or **override** them. *Polymorphism:* methods with the **same name** that behave **differently** in different classes; typically a subclass **redefines** a method of its parent, and the right version runs for each object. *Containment:* a class has an object of another class as an attribute (a car **has** an engine). "Identify the feature that restricts external access to the data" is **encapsulation**; "the term for an occurrence of an object" is **instance**.



A UML class diagram: name, attributes, methods; - private, + public

"Outline the structure of a class" (three marks): attributes (properties) that hold the object's data, usually declared **private**; methods (procedures and functions) that act on those attributes, usually **public**; and a **constructor**, a method that runs when an object is created to initialise the attributes. **"Give three benefits of OOP":** code is **reused** through inheritance; data is **protected** by encapsulation, so it can only be changed by the class's own methods; a large program is split into classes that are written and tested **independently**, so it is easier to maintain and extend; classes model **real-world** entities, so the design is easier to understand; polymorphism lets the same call work for different objects.

The class in pseudocode, as Paper 3 sets it:

```
CLASS Car
    PRIVATE Registration : STRING
    PRIVATE Year : INTEGER
    PRIVATE Mileage : INTEGER
    PUBLIC PROCEDURE NEW(NewReg : STRING, NewYear : INTEGER)
        Registration ← NewReg
        Year ← NewYear
        Mileage ← 0
    ENDPROCEDURE
    PUBLIC FUNCTION GetMileage() RETURNS INTEGER
        RETURN Mileage
    ENDFUNCTION
    PUBLIC PROCEDURE AddMileage(Extra : INTEGER)
        Mileage ← Mileage + Extra
    ENDPROCEDURE
ENDCLASS
```

An object is created with `MyCar ← NEW Car("AB12 CDE", 2020)` and used with `MyCar.AddMileage(150)` and `OUTPUT MyCar.GetMileage()`. A **subclass** reuses the parent's constructor through `SUPER`:

```
CLASS ElectricCar INHERITS Car
    PRIVATE BatteryCapacity : REAL
    PUBLIC PROCEDURE NEW(NewReg : STRING, NewYear : INTEGER, NewCapacity
    ↪  : REAL)
        SUPER.NEW(NewReg, NewYear)
        BatteryCapacity ← NewCapacity
    ENDPROCEDURE
ENDCLASS
```


exercises.lol exercises.com.cn

The same class in Python, as Paper 4 expects it: attributes are made private with a double underscore, the constructor is `__init__`, and a subclass names its parent in brackets and calls `super().__init__(…)`:

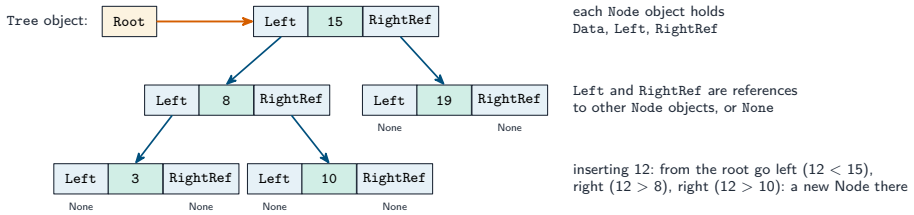
```
class Car:
    def __init__(self, reg, year):
        self.__registration = reg
        self.__year = year
        self.__mileage = 0
    def get_mileage(self):
        return self.__mileage
    def add_mileage(self, extra):
        self.__mileage = self.__mileage + extra

class ElectricCar(Car):
    def __init__(self, reg, year, capacity):
        super().__init__(reg, year)
        self.__capacity = capacity

cars = []
cars.append(Car("AB12 CDE", 2020))
cars.append(ElectricCar("EV21 XYZ", 2023, 75.0))
cars[1].add_mileage(150)
print(cars[1].get_mileage())
```

In Java the same ideas are `private`/`public` fields, a constructor with the class's name, `extends` and `super(…)`; in VB.NET `Private`/`Public`, `Sub New`, `Inherits` and `MyBase.New`. A polymorphic method is written in the parent and **overridden** in the child with the same name; a call through a parent-type variable runs the child's version.

Data structures as objects. Paper 4 builds a stack, linked list or binary tree from a `Node` class whose attributes are the data and one or two **references** to other nodes; a `Tree` (or `LinkedList`) class holds the root (or start) and the methods.



A binary tree built from objects: each Node holds Data plus Left and RightRef references, and the Tree holds the Root; inserting walks down the references

exercises.lol exercises.com.cn

```
CLASS Node
    PUBLIC Data : INTEGER
    PUBLIC Left : Node          // NULL when there is no child
    PUBLIC RightRef : Node
    PUBLIC PROCEDURE NEW(NewData : INTEGER)
        Data ← NewData
        Left ← NULL
        RightRef ← NULL
    ENDPROCEDURE
ENDCLASS

CLASS Tree
    PRIVATE Root : Node
    PUBLIC PROCEDURE Insert(NewData : INTEGER)
        DECLARE NewNode, Current : Node
        DECLARE Placed : BOOLEAN
        NewNode ← NEW Node(NewData)
        IF Root = NULL THEN
            Root ← NewNode
        ELSE
            Current ← Root
            Placed ← FALSE
            WHILE NOT Placed
                IF NewData < Current.Data THEN
                    IF Current.Left = NULL THEN
                        Current.Left ← NewNode
                        Placed ← TRUE
                    ELSE
                        Current ← Current.Left
                    ENDIF
                ELSE
                    IF Current.RightRef = NULL THEN
                        Current.RightRef ← NewNode
                        Placed ← TRUE
                    ELSE
                        Current ← Current.RightRef
                    ENDIF
                ENDIF
            ENDWHILE
        ENDIF
    ENDPROCEDURE
ENDCLASS
```

A **find** method walks the same path and returns `TRUE` when `Current.Data = Target`, `FALSE` when it reaches `NULL`; an in-order **output** method is recursive: output the left subtree, the node, then the right subtree. For a **linked list** the node has one reference,

`Next`, and the list class holds `Start`; for a **stack** built from a list, push and pop both work at `Start`.

> **Worked example.** A game has characters. Each has a name, health (starting at 100) and a position given by X and Y. Write a class `Character` with a constructor and a method `Move(DX, DY)`; then a subclass `Wizard` that adds `Mana` (starting at 50) and a method `CastSpell()` that takes 10 mana and returns `TRUE` if there was enough.

```
CLASS Character
    PRIVATE Name : STRING
    PRIVATE Health : INTEGER
    PRIVATE X : INTEGER
    PRIVATE Y : INTEGER
    PUBLIC PROCEDURE NEW(NewName : STRING, StartX : INTEGER, StartY :
    ↪ INTEGER)
        Name ← NewName
        Health ← 100
        X ← StartX
        Y ← StartY
    ENDPROCEDURE
    PUBLIC PROCEDURE Move(DX : INTEGER, DY : INTEGER)
        X ← X + DX
        Y ← Y + DY
    ENDPROCEDURE
ENDCLASS

CLASS Wizard INHERITS Character
    PRIVATE Mana : INTEGER
    PUBLIC PROCEDURE NEW(NewName : STRING, StartX : INTEGER, StartY :
    ↪ INTEGER)
        SUPER.NEW(NewName, StartX, StartY)
        Mana ← 50
    ENDPROCEDURE
    PUBLIC FUNCTION CastSpell() RETURNS BOOLEAN
        IF Mana >= 10 THEN
            Mana ← Mana - 10
            RETURN TRUE
        ELSE
            RETURN FALSE
        ENDIF
    ENDFUNCTION
ENDCLASS
```

The marks are for private attributes, a constructor that sets every attribute, the inheritance line, the call to the parent's constructor, and a method that uses and changes the object's own data. When the question asks for a **class diagram**, draw a box in three parts (name; attributes with `-` for private; methods with `+` for public) and join a subclass to its parent with an arrow pointing at the parent.

Declarative programming

In **declarative programming** 声明式编程 you say **what** to compute, not **how** – the runtime works out the steps. Two kinds:

- **functional programming** 函数式编程 – built from **pure functions** 纯函数 (no **side effects** 副作用; same input always gives the same output) composed together. Examples: Haskell, Lisp.
- **logic programming** 逻辑编程 – state facts and rules; the engine answers a **goal** (query) by inference. Example: Prolog.

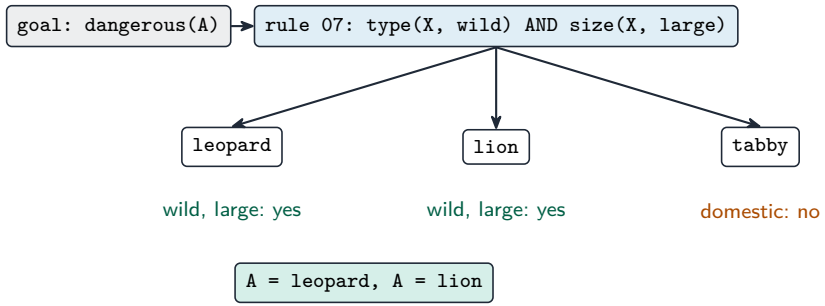
A familiar declarative example is **SQL** 结构化查询语言: `SELECT * FROM Customer WHERE Country = 'UK'` says what you want, not how to walk the records.

Facts, rules and goals are what the exam tests in the declarative paradigm. Given these **facts** 事实 (statements that are true) and a **rule** 规则 (a conclusion that holds when its conditions hold):

```
01 type(leopard, wild).
02 type(lion, wild).
03 type(tabby, domestic).
04 size(leopard, large).
05 size(lion, large).
06 size(tabby, small).
07 dangerous(X) IF type(X, wild) AND size(X, large).
```

"Write the result of the goal `type(X, wild)`*":* `X = leopard, X = lion`. The engine matches the goal against each fact in turn; every match is a solution, and a capital letter is a **variable** that the match fills in. *"Write a fact to show that a cheetah is wild":* `type(cheetah, wild).` *"Explain what line 07 does":* it defines a rule with the conclusion `dangerous(X)`, which is true for any `X` that is both wild and large, so `dangerous(A)` returns `A = leopard, A = lion`. *"Write a rule: a feature* `F` *may be available for a body style* `B` *if* `F` *is a feature and* `B` *is a body style and* `F` *is not unavailable for* `B`*":* `may_be_available(F, B) IF feature(F) AND body_style(B) AND NOT unavailable(F, B)`. Copy the exact predicate names and argument order used in the question's facts; a new fact ends with a full stop, and a rule's conditions are joined with `AND`.

exercises.lol exercises.com.cn



How the engine answers the goal dangerous(A) through rule 07

Comparing paradigms

Paradigm	Strength	Typical languages
Low-level	maximum control, speed	assembly
Imperative	direct, intuitive	C, Python
Object-oriented	modular, models entities	Java, C#, Python
Functional	clear, no side effects	Haskell, F#
Logic	inference, rules	Prolog
Database	data queries	SQL

Modern languages often **mix paradigms** – Python supports all of procedural, OOP and functional. The right one depends on the problem.

File processing

This extends the **file** 文件 handling from Topic 10, processing **serial**, **sequential** and **random** (direct-access) files. Pseudocode operations: `OPENFILE name FOR READ | WRITE | APPEND` (READ opens an existing file, WRITE creates/overwrites, APPEND adds to the end); `READFILE name, line`; `WRITEFILE name, value`; `CLOSEFILE name`; and `EOF(name)` which is TRUE at the end.

Read a whole file:

```
OPENFILE "names.txt" FOR READ
WHILE NOT EOF("names.txt") DO
    READFILE "names.txt", thisName
    OUTPUT thisName
ENDWHILE
CLOSEFILE "names.txt"
```

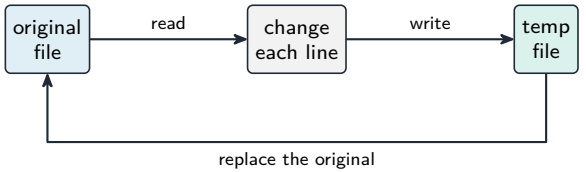

exercises.lol exercises.com.cn

Search a file (stop when found):

```
found ← FALSE
OPENFILE "people.txt" FOR READ
WHILE NOT EOF("people.txt") AND NOT found DO
    READFILE "people.txt", line
    IF line = target THEN
        found ← TRUE
    ENDIF
ENDWHILE
CLOSEFILE "people.txt"
```

Updating a file in place

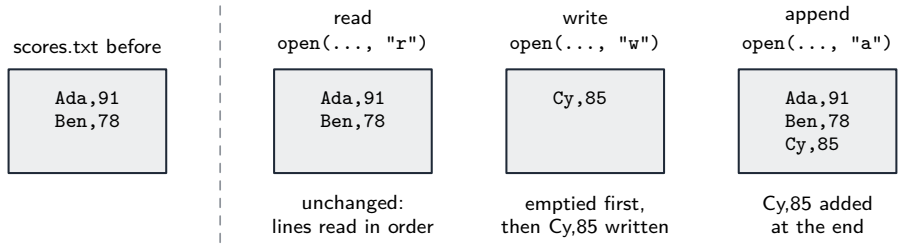
Most languages can't edit a text file in place. Instead: open the original for READ and a temporary file for WRITE; for each line, write the new version if it should change, else the original; close both; then replace the original with the temp file. The same pattern handles deleting lines (skip them) and inserting lines.



Updating a file in place: read the original, write changes to a temp file, then replace the original

Records and random-access files

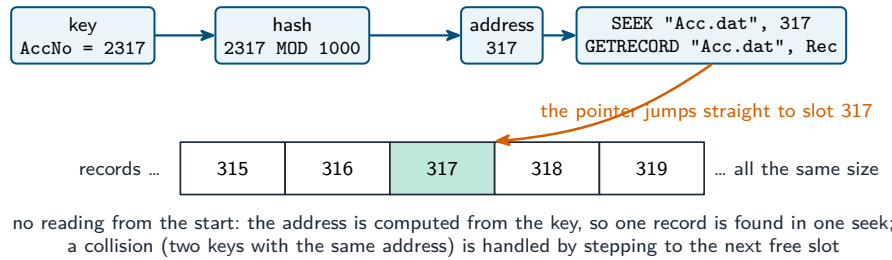
Opening modes. `READ`: the file must exist and reading starts at the beginning. `WRITE`: a **new** file is created, and an existing file of that name is **overwritten**. `APPEND`: writing adds to the **end** of an existing file. Every file that is opened is closed with `CLOSEFILE`, and `EOF(name)` is `TRUE` when the last item has been read.



Mode "w" destroys what was there; mode "a" keeps it

Three file organisations. In a **serial** file the records are in the order they were added; in a **sequential** file they are in **key** order; both are read from the start. A **random file** 随机文件 (direct-access file) stores each record at an **address** calculated from its key by a **hashing** 哈希 function, so one record is found without reading the others. Records are declared as a user-defined type:

```
TYPE AccountRecord
    DECLARE AccNo : INTEGER
    DECLARE Name : STRING
    DECLARE Balance : REAL
    DECLARE Active : BOOLEAN
ENDTYPE
```



Finding one record in a random file: the key is hashed to an address, the file pointer seeks straight to that slot and the record is read; no other record is touched

The random-file operations in pseudocode are `OPENFILE "Acc.dat" FOR RANDOM`, `SEEK "Acc.dat", Address` (move the file pointer to that record), `GETRECORD "Acc.dat", Rec` (read the record there) and `PUTRECORD "Acc.dat", Rec` (write the record there). Finding a customer by account number, as Paper 3 sets it:

```
DECLARE Rec : AccountRecord
DECLARE Target, Address : INTEGER
INPUT Target
Address ← Target MOD 1000              // the hashing function
OPENFILE "Acc.dat" FOR RANDOM
SEEK "Acc.dat", Address
GETRECORD "Acc.dat", Rec
WHILE Rec.AccNo <> Target AND Rec.AccNo <> 0    // 0 marks an empty slot
    Address ← Address + 1               // a collision: try the next slot
    SEEK "Acc.dat", Address
    GETRECORD "Acc.dat", Rec
ENDWHILE
IF Rec.AccNo = Target THEN
    OUTPUT Rec.Name, Rec.Balance
ELSE
    OUTPUT "No such account"
ENDIF
CLOSEFILE "Acc.dat"
```

To **store** a record, hash its key, `SEEK` to the address and `PUTRECORD`, stepping on past any slot already occupied. Marks go to the hash, the SEEK before the GET or PUT, the comparison with the target, the handling of a collision, and closing the file.

Worked example. `ActiveFile.dat` holds `AccountRecord` records. Write pseudocode that copies every record whose `Active` field is `FALSE` to the end of `ArchiveFile.dat`.

```
DECLARE Rec : AccountRecord
OPENFILE "ActiveFile.dat" FOR READ
OPENFILE "ArchiveFile.dat" FOR APPEND
WHILE NOT EOF("ActiveFile.dat")
    READFILE "ActiveFile.dat", Rec
    IF Rec.Active = FALSE THEN
        WRITEFILE "ArchiveFile.dat", Rec
    ENDIF
ENDWHILE
CLOSEFILE "ActiveFile.dat"
CLOSEFILE "ArchiveFile.dat"
```

Text files in Python (Paper 4): `file = open("HighScore.txt", "r")`, then `for line in file:` with `line.strip()` and `line.split(",")` to separate the fields, `int(…)` to convert a score, and `file.close()`; to write, `open(name, "w")` (or `"a"` to append) and `file.write(str(score) + "\n")`. A high-score table is read into a list of records, the new score inserted at its place, and the whole list written back. The examiner marks the open with the correct mode, a loop that reads every line, the

exercises.lol exercises.com.cn

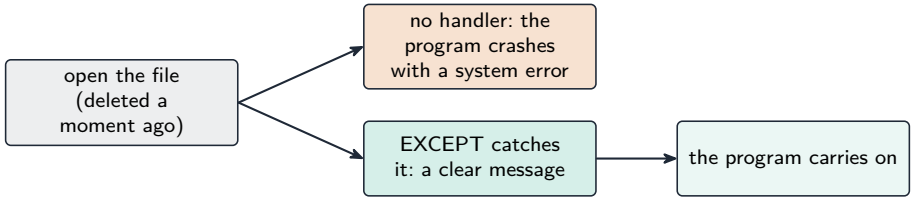
conversion of text to numbers, and the close.

Pitfalls

Forgetting to close a file (data may be lost); opening for WRITE when you meant APPEND (overwrites everything); reading past `EOF`; hard-coded paths – a path like `/Users/Admin/data.txt` breaks on another machine, so use a relative constant such as `DataFile = "./data/scores.txt"`.

Exception handling

An **exception** 异常 is an error or unexpected condition during execution – divide by zero, file not found, network failure, an **array** 数组 index out of range. **Exception handling** 异常处理 lets a program **detect** it and respond gracefully instead of crashing.



The same run-time error without and with a handler

It matters because real programs face errors that **cannot be prevented up front** (files moved, networks down, bad input); without it, every operation needs its own `IF` check; and it **separates** the normal flow from the error handling, so the main path reads cleanly. For example, a file may be deleted by another user between your program checking it exists and actually opening it – you cannot prevent that, only handle the failure when it happens.

"Describe, with an example, what is meant by an exception" (two marks). *An **unexpected event or error** that occurs **during the execution** of a program (at run time) and **interrupts its normal flow**; for example dividing by zero, opening a file that does not exist, converting non-numeric input to an integer, an array index out of range, or running out of memory.* **"Identify two possible causes of exceptions"** is answered from that list, plus "a device or network is not available" and "invalid data type entered".

"State the reasons for including exception handling" (three marks). To stop the program **crashing** (terminating unexpectedly); to output a **meaningful message** to the user rather than a system error; to allow the program to **recover** and continue, for example by asking for the input again, or to close files safely before it ends; and because some errors **cannot be predicted** when the program is written. **"Describe how program termination due to an exception can be avoided":** put the

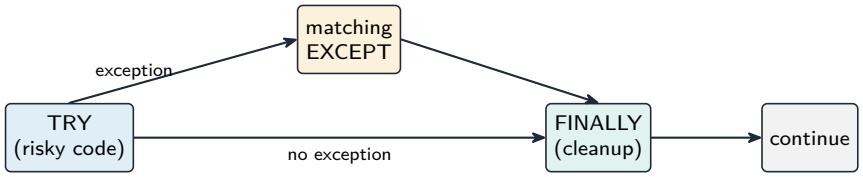
exercises.lol exercises.com.cn

statements that might raise the exception inside a **TRY** block; write an **EXCEPT** (catch) block for that exception that handles it, for example by outputting a message, so that execution continues after the block instead of stopping. **"Explain what is meant by exception handling":** detecting an exception when it occurs and running code (the **handler**) that deals with it so that the program continues.

Pattern

```
TRY
    OPENFILE "data.txt" FOR READ
    READFILE "data.txt", line
    OUTPUT line
    CLOSEFILE "data.txt"
EXCEPT FileNotFound
    OUTPUT "Sorry, the file does not exist."
EXCEPT ReadError
    OUTPUT "Sorry, error reading the file."
ENDTRY
```

The `TRY` block holds the code that might fail; the first matching `EXCEPT` block runs. Real languages also have a catch-all `EXCEPT` and a `FINALLY` block that runs whether or not an exception happened – useful for cleanup (closing files).



Exception flow: an exception jumps to the matching EXCEPT; FINALLY always runs before the program continues

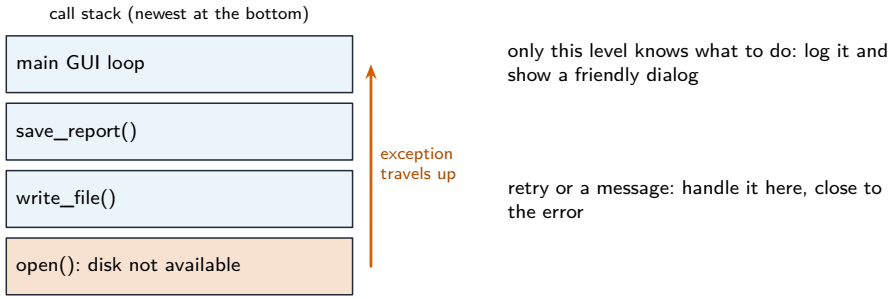
Raising an exception

A subroutine that detects an error can **raise** 抛出 an exception so the caller handles it:

```
FUNCTION Divide(a : INTEGER, b : INTEGER) RETURNS INTEGER
    IF b = 0 THEN
        RAISE DivideByZero
    ENDIF
    RETURN a DIV b
ENDFUNCTION
```

Where to handle exceptions

Handle them **close to the error** if the response is simple (a message, a retry), or higher up the **call stack** 调用栈 if only the outer code knows what to do (a top-level GUI loop logs the error and shows a friendly dialog). **Don't swallow exceptions silently** – at least log them, or debugging becomes impossible.



An exception travels up the call stack to the level that knows what to do

Common exceptions: `FileNotFound`, `IOError`, `DivisionByZero`, `IndexOutOfRange`, `InvalidArgument`, `NullReference`, `OutOfMemory`. Wrapping each failing operation in a `TRY` with the right `EXCEPT` handlers gives a program that **degrades gracefully** instead of crashing.

Worked example (Paper 4). Write a function that reads whole numbers, one per line, from a file whose name is passed as a parameter and returns them in a list. It must not crash if the file does not exist or a line is not a whole number.

```
def read_scores(filename):
    scores = []
    try:
        file = open(filename, "r")
        for line in file:
            scores.append(int(line))
        file.close()
    except FileNotFoundError:
        print("The file", filename, "does not exist")
    except ValueError:
        print("A line in the file was not a whole number")
    return scores
```

The `try` block holds the code that can fail (the open and the conversion); each `except` names one exception and does something useful; the function still returns a list, so the caller continues. In Java the same shape is `try { … } catch (FileNotFoundException e) { … } catch (NumberFormatException e) { … }`; in VB.NET `Try … Catch ex As FileNotFoundException … End Try`. Marks: the risky statements

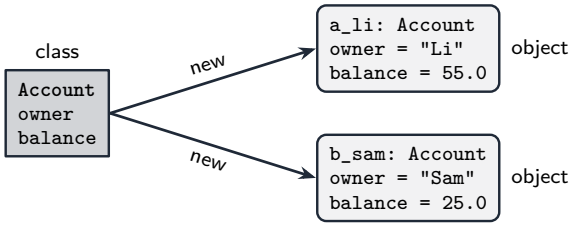
exercises.lol exercises.com.cn

inside the try, the correct exception names, a message for each, and the program **continuing** afterwards; a catch-all `except:` gets the crash mark but not the "appropriate exception" mark.

> **Worked example.** A text file of members needs one member's phone number changed. Why can the program not simply overwrite that line, and what is the pattern? A text file's lines are **different lengths**, and the file has no gaps to absorb a difference: a longer replacement would run into the next record, and a shorter one would leave part of the old line behind. So the pattern is to open the original for **READ** and a **temporary** file for **WRITE**, read every line in turn, writing the **new** version for the line that changes and the **original** line for all the others, close both, then **replace** the original with the temporary file. The same shape handles deleting (skip the line) and inserting (write the extra line). Note that **every** line gets written, not only the changed one - writing just the new record and losing the rest of the file is the classic slip.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.



One class, many objects: the same attributes, each with its own values

Term	Definition
programming paradigm	a style or way of programming, with its own way of structuring a program
imperative language	the program is a sequence of statements that change the program's state; the programmer says how the task is done
declarative language	the program states facts and rules and the inference engine works out how to find the answer
class	a template defining the attributes and methods of the objects of that type

Term	Definition
object (instance)	an occurrence of a class, with its own values for the attributes
attribute	a data item that belongs to a class
method	a procedure or function that belongs to a class and acts on its attributes
encapsulation	keeping the attributes and methods together in a class and restricting external access to the data, so that it is changed only through public methods
inheritance	a subclass acquires the attributes and methods of its parent class and can add or override them
polymorphism	methods with the same name that behave differently for different classes
constructor	a method that runs when an object is created and initialises its attributes
containment	a class has an object of another class as one of its attributes
fact	a statement in a declarative program that is true
rule	a conclusion that holds when its conditions are true
serial, sequential, random file	records in the order added; records in key order; each record at an address calculated from its key
exception	an unexpected error or event during execution that interrupts the normal flow
exception handling	detecting an exception when it occurs and running code that deals with it so that the program continues

Exam tips

- Paradigms: know the one-line description of each and be ready to name the paradigm from a code sample; low-level questions want the five addressing modes and what the accumulator receives.
- OOP definitions come up every session: class, object, attribute, method, encapsulation, inheritance, polymorphism, constructor. Write a class in pseudocode with PRIVATE attributes, a PUBLIC NEW and getters; a subclass with INHERITS and SUPER.NEW.

exercises.lol exercises.com.cn

- Declarative: a goal with a variable returns every matching fact; a rule is a conclusion IF conditions joined with AND; copy the question's predicate names exactly.
- Files: the three modes and what each does to an existing file; READFILE in a WHILE NOT EOF loop; random files use a hash, SEEK, GETRECORD and PUTRECORD, with a step-on for collisions.
- Exceptions: definition with an example, three reasons for handling them, and TRY with a named EXCEPT that lets the program continue.

Common mistakes

- Describing a declarative program as "a sequence of steps that gives the answer"; it states what is true and what is wanted, not how.
- Confusing an object with a class, or an instance with an attribute; the question "an occurrence of an object" wants instance.
- Declaring the attributes PUBLIC, or reaching them from outside the class instead of through a getter, which loses the encapsulation marks.
- A subclass constructor that sets the parent's attributes directly instead of calling SUPER.NEW.
- Explaining polymorphism as "many objects"; it is the same method name behaving differently for different classes.
- Opening a file FOR WRITE to add a record, which destroys the existing contents; use APPEND.
- Reading a random file from the start; SEEK to the hashed address first.
- Putting the exception handler around code that cannot fail, or catching everything with no message, or describing exception handling as "checking the input with IF".