Course Companion

AP Computer Science Principles

Every topic handout in one volume

全部专题讲义合集

exercises.lol

Contents 目录

Creative Development

AP Computer Science Principles

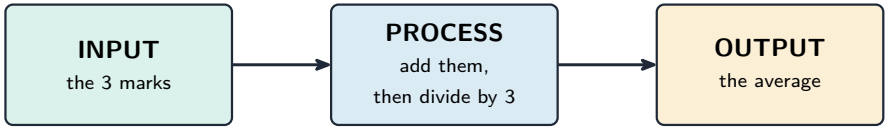
Collaboration

A jigsaw in progress: collaboration and modular design piece the solution together

Computing is a **collaborative** 协作 activity. Working in a team brings more perspectives, catches more errors, and produces better programs than working alone. Good collaboration uses **consensus building**, clear communication, and each member's strengths. **Pair programming** 结对编程 – two people at one computer, one typing and one reviewing – is a common practice. On the exam, you should be able to explain how collaboration improved a program (more ideas, fewer bugs, wider testing).

Program Function and Purpose

Every program is written for a **purpose** – it solves a problem or pursues an interest. A program takes **input** 输入, processes it, and produces **output** 输出. Inputs can come from a user, a device, a file, or another program; outputs can be visual, audible, textual, or a signal to a device. Being able to state a program's purpose, and describe its inputs and outputs clearly, is a core skill (and part of the Create performance task).



Every program decomposes into input, processing, and output

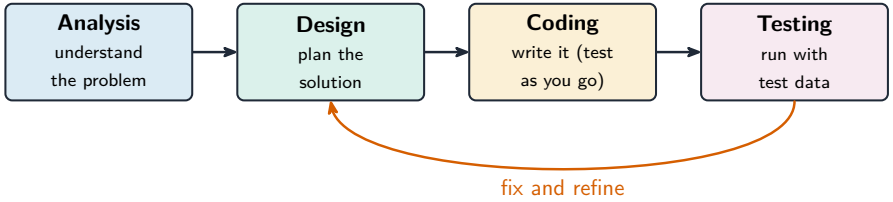
computing system model: Input → Process → Output (IPO)

Every program follows the input-processing-output model

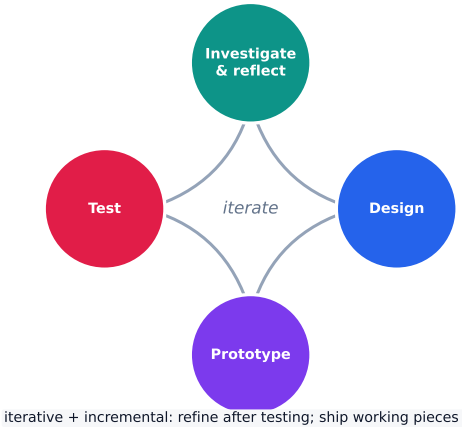
Program Design and Development

A programmer debugging at a multi-monitor workstation — iterative design and testing

Programs are built through an **iterative** 迭代 process, not in one straight line: investigate the problem and users, design (often with a **diagram** or written plan), implement in code, and test – then repeat. A large problem is broken into smaller pieces (**decomposition** 分解). **Comments** 注释 and clear naming document the design so others (and your future self) can understand it. Development is **incremental** – build and test a small piece, then add the next.



The stages of program development, with testing feeding back to fix and refine



iterative + incremental: refine after testing; ship working pieces early

Software is built by an iterative, incremental development process

Investigating what users actually need

Before any code is written, the developer investigates the problem and the people who will use the program. Three ways to do that:

- **surveys** 调查问卷 sent to potential users, which collect data from many people quickly;

- **interviews** and direct observation of users doing the task by hand;
- **studying existing solutions** to see what already works and what frustrates people.

The findings are turned into a design. Two artefacts do that: a **program requirements** list saying exactly what the program must do, and **diagrams representing the layout of the user interface** 用户界面 — sketches showing which controls appear where, and what each one does when used. Designing the interface on paper first is cheaper than discovering after coding that the buttons are in the wrong place.

Events, and programs that wait

Not every program runs straight through from top to bottom. An **event** 事件 is generated when a key is pressed, a mouse is clicked, a program is started, or any other defined action occurs — and an event changes the **flow of execution**: the program pauses what it was doing and runs the code attached to that event, called an **event handler** 事件处理程序.

This is why a program with a graphical interface can appear to be doing nothing: it is waiting for the next event. The order in which those events arrive is decided by the user, not by the programmer, so the same program can run its blocks in a different order each time it is used.

Identifying and Correcting Errors

A **bug** is an error in a program; **debugging** 调试 is finding and fixing it. Three kinds:

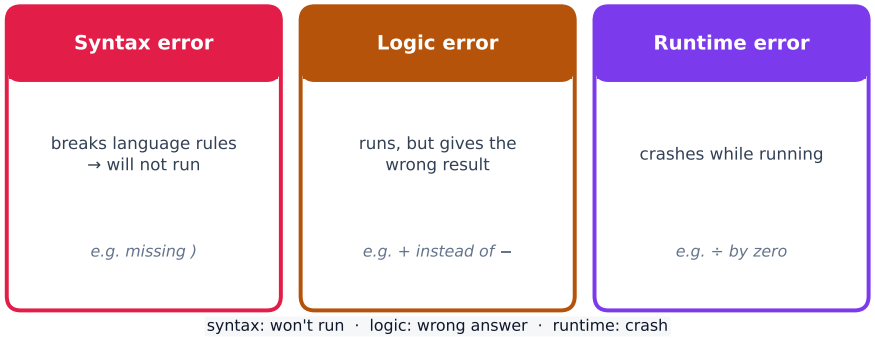
count	total	output
1	5	
2	12	
3	20	20

A trace table records each variable's value as the program runs, to find bugs

- a **syntax error** 语法错误 breaks the language's rules, so the program will not run;
- a **runtime error** 运行时错误 crashes the program while it runs (e.g. dividing by zero);
- a **logic error** 逻辑错误 lets it run but gives the wrong result.

Find bugs by **testing** with different inputs, adding **print statements** to see values, and hand-tracing the code. Choose the test inputs deliberately: they should demonstrate the different expected outcomes **at or just beyond the extremes** — the minimum and maximum values the program should accept, and a value just outside each of them. A program that works on ordinary data very often fails on an empty list, a zero, or a value one past the end of a range, so those are the inputs worth trying first. Fixing one bug at a time and re-testing is the reliable method.

Exam skill: be able to name the type of an error and describe a testing strategy that would catch it – a recurring multiple-choice and Create-task theme.



Three kinds of programming error: syntax, logic, and runtime

Worked example. A program meant to print the average of two numbers instead runs `avg = a + b / 2`. Tracing the order of operations, `/` runs before `+`, so it computes $a+\frac{b}{2}$ rather than the average. Add parentheses to fix it: `avg = (a + b) / 2`. Testing with $a = 4$, $b = 6$ confirms the fix — the buggy line gives $4+3 = 7$, the corrected line gives $\frac{10}{2} = 5$. Testing with known inputs is exactly how you find and confirm a **logic error**.

Exam tips

- Much of CSP is assessed through the **Create** and written performance tasks — explain your reasoning clearly, not just your result.
- Know the benefits of collaboration and how diverse perspectives reduce bias in a program.
- Use precise vocabulary (iterative development, program requirements) when you describe a design process.
- Give and take feedback constructively; credit collaborators and sources.
- Break a large problem into smaller modules that a team can build in parallel.

Data

AP Computer Science Principles

Binary Numbers

Binary digits on a display — all digital data is ultimately stored as 0s and 1s

Computers store everything as **bits** 位 – each a `0` or `1`. A group of 8 bits is a **byte** 字节. Numbers are stored in **binary** 二进制 (base 2), where each place is a power of two $(1, 2, 4, 8, 16, \dots)$ instead of the powers of ten in **decimal** 十进制. For example, binary `1011` is $8 + 2 + 1 = 11$.

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

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

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

Worked example. To convert binary `1101` to decimal, write the place values `8 4 2 1` under the bits `1 1 0 1` and add the ones that have a `1`: $8 + 4 + 0 + 1 = 13$. Going the other way, convert `19` to binary by subtracting the largest power of two that fits: $19 - 16 = 3$, then $3 - 2 = 1$, then $1 - 1 = 0$, so the bits sit at the `16`, `2`, and `1` places $\rightarrow$ `10011` (check: $16 + 2 + 1 = 19$).

Because a computer has a **finite** number of bits, it can represent only a limited range of values. This causes two effects tested on the exam:

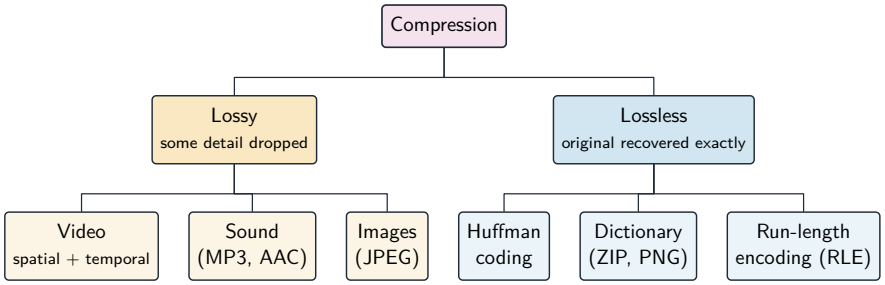
- **Overflow error** 溢出错误: a number too large for the available bits cannot be stored correctly.
- **Round-off (rounding) error** 舍入错误: numbers with **decimals** (real numbers) can only be **approximated**, because infinitely many real values must map onto finitely many bit patterns.

All data – text, images, sound – is ultimately encoded as binary. An image is a grid of **pixels** 像素, each stored as numbers for its colors; sound is stored as numbers sampled many times per second.

Data Compression

Hard-disk platters and head: data compressed and stored as magnetic patterns

Compression 压缩 reduces the number of bits needed to store or send data. Two kinds:



Compression methods: lossless versus lossy, with common examples

- **Lossless compression** 无损压缩 lets you restore the **exact** original data (used for text and programs, where every bit matters).
- **Lossy compression** 有损压缩 throws away some data to shrink the size **further** (used for photos, music, video, where a small quality loss is acceptable).

Choosing between them trades **size against fidelity**: lossless keeps everything but saves less; lossy saves more but loses detail permanently. Prefer lossless when the data must be exact.

Extracting Information from Data

Data 数据 becomes useful when we extract **information** 信息 from it – patterns, trends, and answers to questions. Large data sets can reveal correlations a small one cannot, but data must be **cleaned** (fixing errors and inconsistencies) and often **transformed** or **filtered** first. A **correlation** 相关性 between two things does not prove that one **causes** the other – a key caution. **Metadata** 元数据 (data about data, like a photo's date and location) helps organize and search large collections.

Using Programs with Data

Programs process data at scales humans cannot. Common operations are **filtering** 过滤 (keeping only rows that meet a condition), **cleaning** (removing errors), and **visualizing** 可视化 (charts and graphs that make patterns visible). Combining data from multiple sources can reveal more, but raises **privacy** 隐私 concerns. Interactive tools and visualizations let people explore data and draw their own conclusions.

Exam skill: be able to explain how a program helps find information in a large data set, and why correlation shown in the data does not establish causation.

Exam tips

- Convert confidently between **binary, decimal**, and (where asked) hexadecimal — practise until it is quick.
- Remember a bit is one binary digit and a byte is 8 bits; n bits represent 2^n values.
- Explain that all data — numbers, text, images, sound — is stored as binary, and that finite bits cause **overflow** and **round-off**.
- Distinguish **lossless** from **lossy** compression and when each is appropriate.
- Show the analog-to-digital idea: **sampling** turns a continuous signal into discrete values.

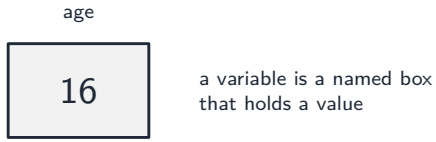
Algorithms and Programming

AP Computer Science Principles

Code below uses the **AP CSP pseudocode** – the exam's language-neutral reference. Assignment is written `a ← expression`, and list indices start at **1**.

Variables and Assignments

A **variable** 变量 is a named place that holds a value. The **assignment** 赋值 operator stores the value on the right into the variable on the left:



A variable is a named store whose value can change

```
a ← 5
b ← a + 3      // b is now 8
```

A variable holds one value at a time; assigning again **replaces** it. Variables let a program store input, remember results, and reuse them.

Data Abstraction

Data abstraction 数据抽象 lets you manage complexity by giving a single name to a collection of data – for example, a **list** rather than dozens of separate variables. It hides detail: you use the named collection without worrying about how it is stored. Lists (below) are the course's main data abstraction.

Mathematical Expressions

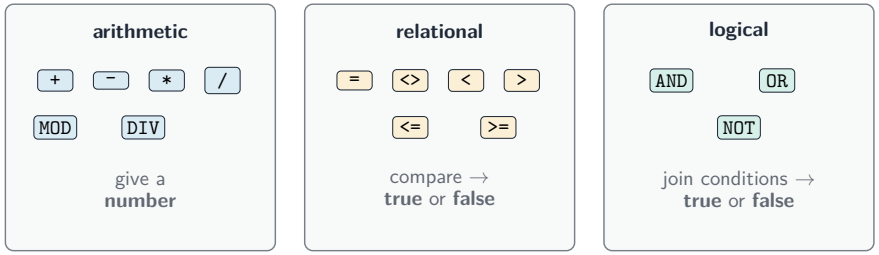
Programs compute with the operators `+`, `-`, `*`, `/`, and `MOD` (the **remainder** 余数 of a division, e.g. `17 MOD 5` is `2`). Expressions follow the usual order of operations. `MOD` is especially useful for testing divisibility (`n MOD 2 = 0` means `n` is even) and for wrapping values around a range.

Strings

A **string** 字符串 is an ordered sequence of characters, like `"hello"`. Programs join strings (**concatenation** 拼接) and find their **length**. Strings represent text – names, messages, sequences – and are a common program input and output.

Boolean Expressions

A **Boolean expression** 布尔表达式 evaluates to `true` or `false`. It uses **relational operators** (`=`, ≠, `<`, `>`, ≤, ≥) and **logical operators** `NOT`, `AND`, `OR`:



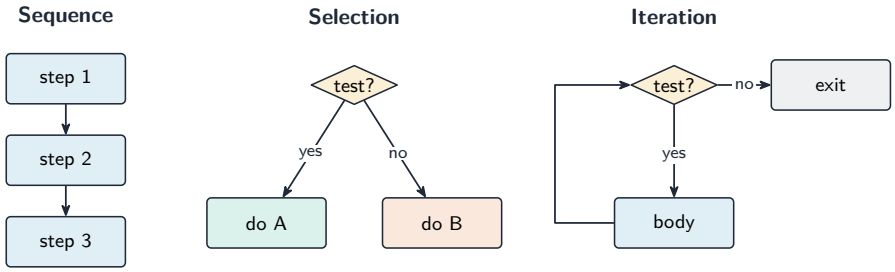
The three families of operators: arithmetic, relational, and logical

- `NOT` reverses a value,
- `AND` is true only when **both** sides are true,
- `OR` is true when **at least one** side is true.

These conditions drive every decision and loop.

Conditionals

A **conditional (selection)** 条件语句 chooses which code to run. `IF` runs a block only when its condition is true; `ELSE` gives an alternative:



Selection chooses between paths based on a condition

```
IF (score ≥ 60)
{
    DISPLAY("Pass")
}
ELSE
{
    DISPLAY("Fail")
}
```

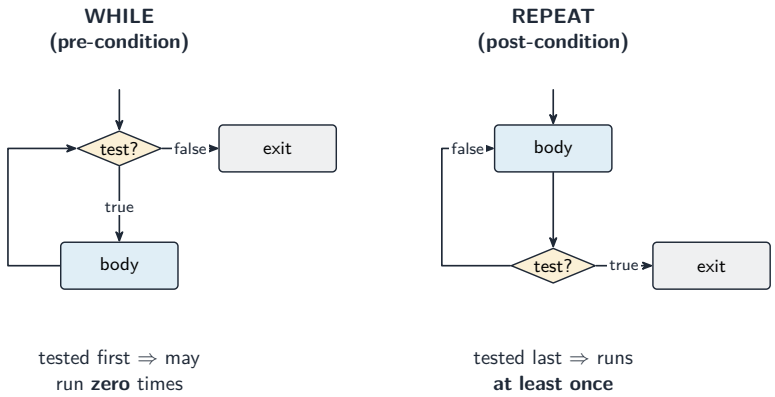
Nested Conditionals

A **nested conditional** 嵌套条件 places one `IF` inside another (or chains `ELSE IF`) to choose among **more than two** paths. Only the first matching branch runs:

```
IF (g ≥ 90)      { grade ← "A" }
ELSE IF (g ≥ 80) { grade ← "B" }
ELSE             { grade ← "C" }
```

Iteration

Iteration (a loop) 迭代 repeats instructions. AP pseudocode has two forms:



tested first ⇒ may run **zero** times

tested last ⇒ runs **at least once**

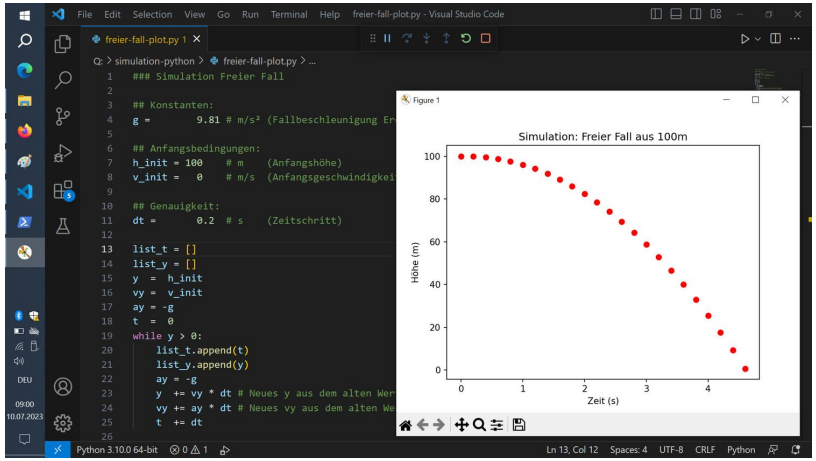
A pre-condition (WHILE) loop tests before the body, so it may run zero times

```
REPEAT 5 TIMES        // a fixed count
{
    DISPLAY("hi")
}

REPEAT UNTIL (found)  // until a condition becomes true
{
    ...
}
```

A loop that never meets its stopping condition is an **infinite loop** 无限循环.

Developing Algorithms

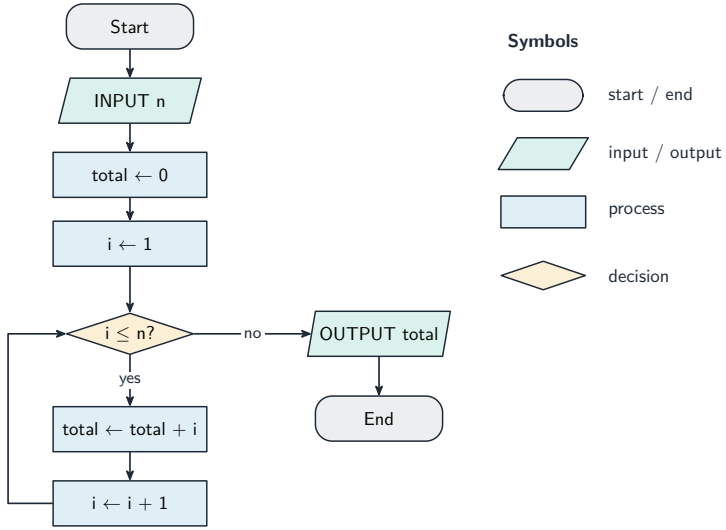


Python source code on a screen — algorithms are precise, ordered instructions

An algorithm is not the same thing as code. Beyond visual and textual programming languages, an algorithm can be expressed in a **variety of ways**: in **natural language** (ordinary sentences), as a **diagram** such as a flowchart, or in pseudocode. Those forms are for people — they let you check the logic and agree on it before any language is chosen, and the same algorithm can then be written in any language.

When you do write it in a programming language, **clarity and readability are important considerations**, not decoration: meaningful variable names, consistent indentation and comments explaining *why* rather than *what*. The program has to be read and modified later by someone — often you — and an algorithm nobody can follow cannot be maintained or debugged.

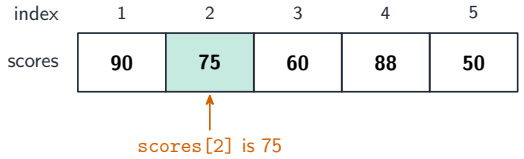
An **algorithm** 算法 is a finite sequence of steps that solves a problem, built from **sequencing**, **selection**, and **iteration**. Different algorithms can solve the same problem, and you should be able to combine and modify existing algorithms (for example, count the values in a list that meet a condition, or find the largest). Trace an algorithm by hand to check it is correct.



A flowchart lays out an algorithm using the standard symbols

Lists

A **list** 列表 is an ordered collection of values under one name, the course's key data abstraction. AP pseudocode indexes from **1**:



A list holds many values in one variable, each found by its index

```
scores ← [88, 74, 95]
DISPLAY(scores[1])          // 88
scores[2] ← 80              // replace the 2nd value
APPEND(scores, 60)          // add to the end
INSERT(scores, 1, 100)      // insert at index 1
REMOVE(scores, 3)           // delete the 3rd element
LENGTH(scores)              // how many elements
```

Traverse a list with a loop to sum, count, search, or find a maximum:

```
FOR EACH x IN scores
{
    total ← total + x
}
```

Binary Search

A phone book: binary search halves the remaining pages each step

Binary search 二分搜索 finds a value in a **sorted** list far faster than checking each element. It looks at the middle element, then discards the half that cannot contain the target, repeating until found. Each step **halves** the search space, so a list of n items takes about $\log_2 n$ steps. It **requires the data to be sorted** first.

Binary search halves the range at each step (the list must be sorted)

Worked example. Searching a sorted list of 8 items, binary search halves the range each step: $8 \to 4 \to 2 \to 1$, at most 3 comparisons ($\log_2 8 = 3$), whereas a linear search could take up to 8. The advantage grows explosively: about 1,000 items need only ≈ 10 binary-search steps (but up to 1,000 linear ones), and 1,000,000 items need just ≈ 20. Halving is what makes it a **reasonable-time** algorithm.

Calling Procedures

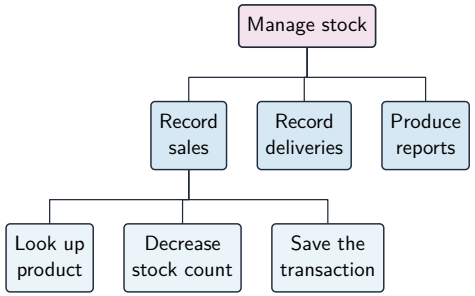
A **procedure (function)** 过程 is a named, reusable block of code. **Calling** it runs its code with the **arguments** you supply, and it may **return** a value:

```
sum ← Add(3, 4)      // call, passing 3 and 4
```

Procedures let you use code without knowing its inner workings – **procedural abstraction** 过程抽象.

Developing Procedures

You **define** a procedure with a name, **parameters** (inputs), and a body, and optionally `RETURN` a result:



Decomposing a program into procedures and sub-procedures

```
PROCEDURE Add(a, b)
{
    RETURN(a + b)
}
```

Writing your own procedures reduces repetition, breaks a big problem into named pieces, and makes programs readable and easier to test – the essence of **abstraction** 抽象.

Libraries

A **library** 库 is a collection of ready-made procedures that others can reuse. An **API** (Application Program Interface) 应用程序接口 documents what each procedure does, its parameters, and its result – so you can use it without seeing its code. Libraries save time and let you build on existing, tested work.

The documentation is part of the library. Documentation for an API or library is **necessary** in order to understand the behaviours it provides and how to use them — what each procedure expects as parameters, what it returns, and what it does at the edges. Without it you would have to read the source, which defeats the point of abstraction; with it you can use a procedure correctly without knowing how it works inside.

Random Values

`RANDOM(a, b)` returns a random integer from `a` to `b` (inclusive), letting a program produce **unpredictable** results – for games, sampling, or simulations. Each call may give a different value, so a program using randomness behaves differently each run.

Simulations

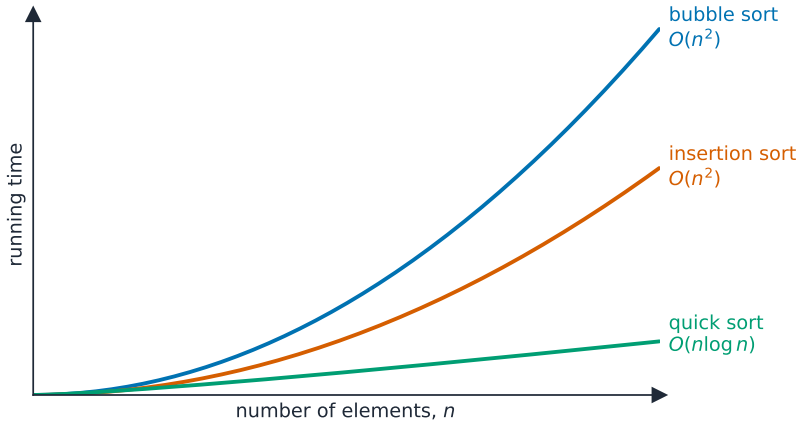
A **simulation** 模拟 is a program that models a real-world process to study it safely and cheaply. Simulations **simplify** reality (they leave out detail) and often use **randomness** to imitate chance events. They let you test scenarios that would be too costly, slow, or dangerous in real life – but their results are only as good as their assumptions.

A simulation is a way of doing science, not just a picture. Because it can be run many times, cheaply and with one variable changed at a time, a simulation **facilitates the formulation and refinement of hypotheses** about the object or phenomenon under consideration: you propose an explanation, run the model, compare the result with reality, and adjust either the hypothesis or the model. That is why a simulation's simplifications matter — a result only supports a hypothesis about the real world to the extent that what was left out does not matter.

Algorithmic Efficiency

Efficiency 效率 is how much time (or memory) an algorithm needs as its input grows. A **reasonable-time** algorithm's work grows like a polynomial of the input size (e.g. linear or quadratic); an **unreasonable-time** algorithm grows far faster (e.g. doubling with each added item), becoming impractical for large inputs. A faster algorithm can

make a previously impossible problem solvable. Sometimes an exact answer takes too long, so a **heuristic** 启发式 – an approach that finds a good-enough answer quickly – is used instead.



for large n: $O(n \log n)$ (quick/merge) beats $O(n^2)$ (bubble/insertion)

How the running time of an algorithm grows with the input size n

Undecidable Problems

Some problems are **undecidable** 不可判定: no algorithm can solve **every** case of them with a correct yes/no answer. This is a fundamental limit of computing – not a matter of needing a faster computer, but a proof that no such algorithm can exist.

Exam skill: be able to determine a code segment's result by tracing it, compare two algorithms' efficiency (reasonable vs unreasonable time), and recognize procedural and data abstraction in a program.

Exam tips

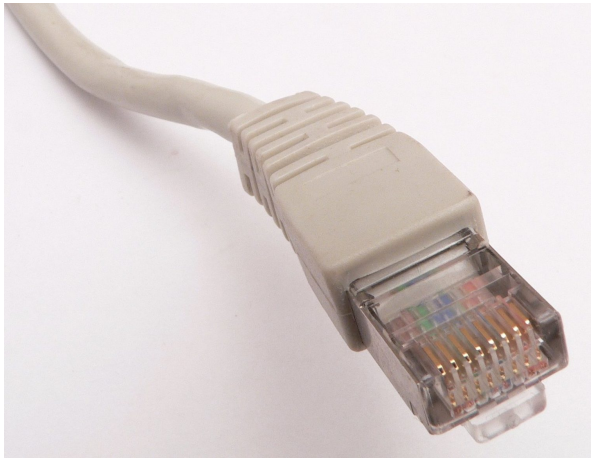
- Know a variable is a named store for a value and trace how **assignment** updates it step by step.
- Read the AP **pseudocode** carefully — `a <- expression` assigns, and lists are **1-indexed** on the exam reference sheet.
- Distinguish a variable from a **list** (a collection accessed by index) and use list operations correctly.

- Evaluate expressions with the right precedence and boolean logic (`AND`, `OR`, `NOT`).
- Pick clear, meaningful variable names — the written tasks reward readable code.

Computer Systems and Networks

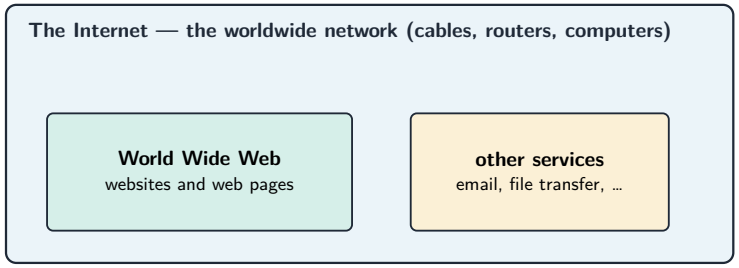
AP Computer Science Principles

The Internet



An RJ45 Ethernet connector — physical links carry packets across the Internet

The **Internet** 互联网 is a global network of networks. Data travels as **packets** 数据包 – small chunks that are sent separately and reassembled at the destination. Two ideas make it work at scale:



The internet is the worldwide network; the web is one service running on it

- **Protocols** 协议 are agreed rules for communication. **IP** (Internet Protocol) addresses and routes packets; **TCP** reassembles them in order and re-requests lost

ones; **HTTP** carries web pages; **DNS** translates a name like `example.com` into an IP address.

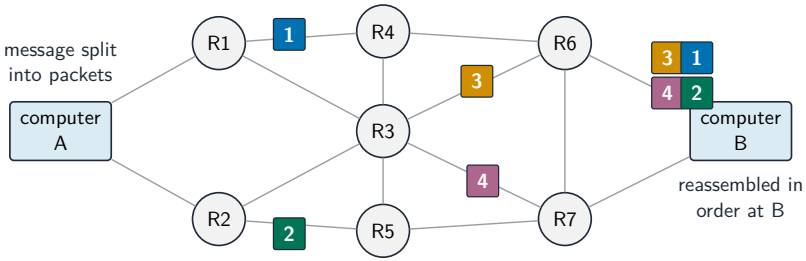
- **Redundancy** 冗余 and routing: there are **many** possible paths between two points, so if one path fails, packets take another. This makes the Internet **fault-tolerant** 容错.

The Internet is designed to be **open** and **scalable** – built on standards anyone can use, so it keeps working as billions of devices join. **Bandwidth** 带宽 is the amount of data a connection can carry per second.

Fault Tolerance

A Wi-Fi router: the local gateway that forwards packets toward the wider Internet

A system is **fault-tolerant** if it keeps working even when part of it fails. The Internet achieves this through **redundant** connections: because packets can be routed along multiple paths, the failure of one router or cable does not stop communication – traffic simply reroutes. Fault tolerance costs extra resources (the redundant paths) but greatly improves **reliability** 可靠性. A single path with no backup is not fault-tolerant.

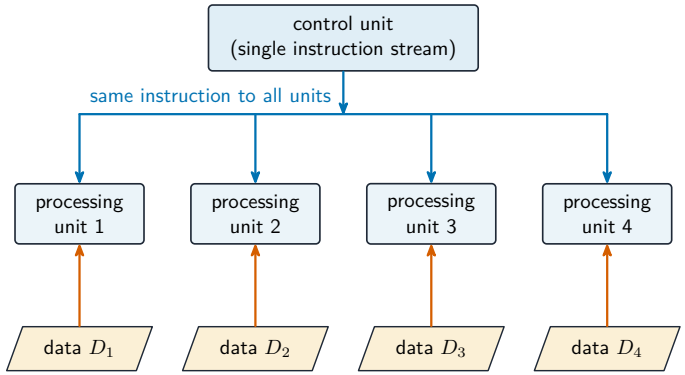


Packet switching sends packets by independent routes, so the network tolerates a failed link

Parallel and Distributed Computing

Data-centre server racks: distributed computing spreads work across many machines

- **Sequential computing** 顺序计算 runs one operation at a time.
- **Parallel computing** 并行计算 splits a task into parts that run **at the same time** on multiple processors, finishing faster.
- **Distributed computing** 分布式计算 uses **many computers** connected by a network to work on one problem – essential for problems too big for a single machine.



Parallel computing: many processors work at the same time

A parallel solution's **speedup** 加速比 is the sequential time divided by the parallel time. Speedup is limited: parts that **must** run in sequence cannot be sped up by adding processors, so doubling the processors rarely doubles the speed.

Worked example. A task has a part that **must** run sequentially, taking 40 seconds, plus a parallelizable part that takes 60 seconds on one processor – so on a single processor the whole task takes $40 + 60 = 100$ seconds. Spread the parallel part across 3 processors and it takes $\frac{60}{3} = 20$ seconds, so the total parallel time is $40 + 20 = 60$ seconds and the speedup is $\frac{100}{60} \approx 1.67$. The 40-second sequential part is a floor: even with infinitely many processors the task can never finish in under 40 seconds.

Exam skill: given the times for the sequential and parallel portions of a task, be able to calculate the total parallel time and the speedup.

Exam tips

- Describe how data travels in **packets** over a redundant, fault-tolerant network of independent routers.
- Know that **protocols** (IP, TCP, HTTP) are agreed rules, and that open standards let different systems interoperate.
- Explain **scalability** and how the Internet grows without central control.
- Contrast **bandwidth** (rate) with **latency** (delay), and describe the **DNS** name-to-address lookup.
- Discuss the **digital divide** and security basics (encryption, authentication) in plain terms.

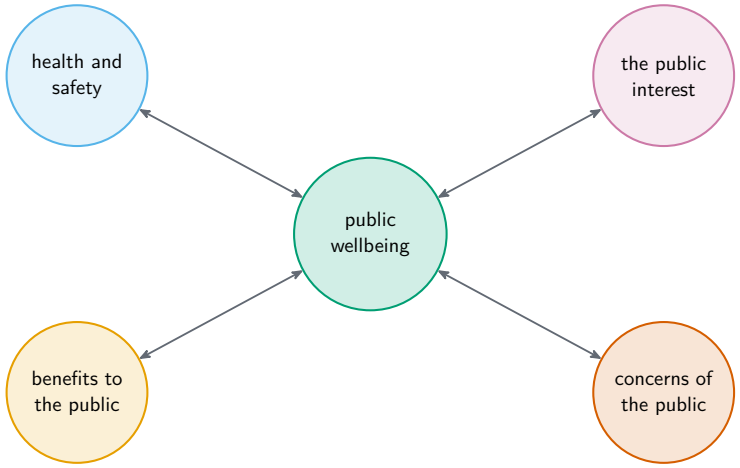
Impact of Computing

AP Computer Science Principles

Beneficial and Harmful Effects

A pile of e-waste — computing has environmental costs as well as benefits

Every computing innovation can be used in ways that **help** and ways that **harm** – often the same technology does both. A social network connects people **and** can spread misinformation; automation raises productivity **and** can remove jobs. Effects are frequently **unintended**: creators cannot foresee every use. When you evaluate a computing innovation, weigh its benefits and harms on people and society, and remember that harms are not always deliberate.

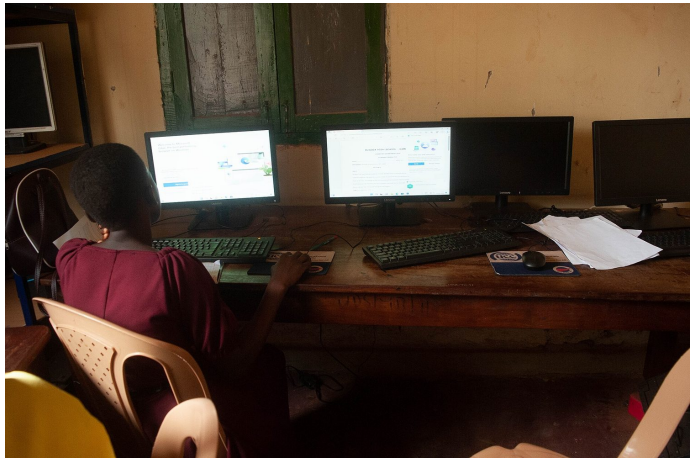


Computing affects the public's wellbeing in several ways

Computing also generates creativity in other fields, which the CED asks for as a benefit in its own right, not merely as convenience: modelling and imaging in **medicine**, simulation in **engineering**, new forms in the **arts** and music, and new kinds of **communication** entirely. An innovation's beneficial effects are often in a field far from computing.

The Digital Divide

The **digital divide** 数字鸿沟 is the unequal access to computing and the Internet across groups – by income, geography, age, or country. Those with access gain education, jobs, and services; those without fall further behind. The divide is shaped by economic, social, and geographic factors, and efforts to close it (affordable devices, public access, infrastructure) aim to make computing's benefits fairer.



Public library computers: digital divide is about unequal access to devices, connectivity and skills

A rural satellite dish: geography still shapes who gets fast internet and who waits

Computing Bias

A self-driving car on a city street — autonomous systems raise safety and bias questions

Bias 偏见 can be built into computing systems – often unintentionally. If the **data** used to build a system reflects existing prejudice, or if the designers' assumptions are one-sided, the system can produce **unfair** results (for example, a hiring tool that favors one group). Bias can enter at every stage – data collection, design, and use – so systems should be tested for fairness across different groups. Recognizing that "the computer said so" is not the same as "fair" is an important habit.

Crowdsourcing

Crowdsourcing 众包 obtains input, ideas, or funding from a **large group of people**, usually online. It harnesses the knowledge and effort of many – mapping projects, product reviews, citizen science, and **crowdfunding** all rely on it. The Internet makes crowdsourcing possible at a scale and speed never before achievable, letting a project draw on contributors worldwide.

A Wikipedia edit-a-thon: crowdsourcing pools many people's work into a shared resource

Legal and Ethical Concerns

A CCTV control room: surveillance systems trade safety gains against privacy concerns

Computing raises questions of law and ethics:

- **Intellectual property** 知识产权 and **copyright** 版权 protect creators' work; using it may require permission or a license. **Open-source** 开源 and **Creative Commons** licenses let creators share work under stated terms.
- **Plagiarism** 抄袭 – using others' work as your own – is unethical and often illegal.

- Collecting and using personal data raises **privacy** questions about consent and misuse.

The three "open" terms, which are not the same thing

Term	What it means
open source	programs that are **made freely available and may be redistributed and modified** by anyone. The licence grants those rights explicitly — free of charge is not the same as open source, and a free program you may not modify is not open source.
open access 开放获取	research and other content made available online **without charge**, so a reader does not need a subscription. It says nothing about the right to modify.
Creative Commons 知识共享	a family of licences a creator applies to their own work to grant specific permissions in advance — for example "you may reuse this if you credit me" or "you may reuse this but not commercially".

All three are ways of **granting** rights the creator holds by default under copyright. That is why they matter for the exam: copyright is automatic, so anything not explicitly licensed is restricted, and using it needs permission.

Just because something is technically possible does not make it legal or ethical.

What is recorded while you browse

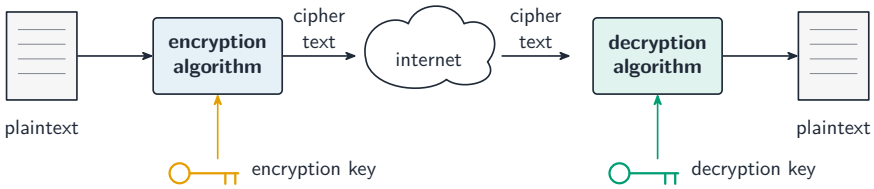
Two collection routes the CED names, and neither needs you to type anything:

- **Websites can record and maintain a history of the individuals who have viewed their pages** — which pages, when, how long, and from which device.
- **Search engines can use your search history** to suggest websites, and to sell **targeted marketing**: the advertisement follows the search, which is why a product looked up once then appears for weeks.

Neither is inherently malicious, and both are the mechanism behind services people find useful. The point the exam wants is that **data collected for one purpose can be combined and used for another**, often without the person realising they agreed to it.

Safe Computing

Protecting personal data is a shared responsibility. Key ideas:



Encryption scrambles plaintext with a key; only the key can decrypt it

- **Personally identifiable information (PII)** 个人身份信息 (name, address, ID numbers) should be shared carefully, because it can be misused for **identity theft** 身份盗窃.
- Threats include **phishing** 网络钓鱼 (tricking you into revealing information), **malware** 恶意软件, and weak passwords.
- **A malicious link can be disguised** on a web page or in an email message: the text you see and the address it actually goes to are separate, so a link reading `www.yourbank.com` can point anywhere. Hover to see the real destination before clicking, and be most suspicious of a link that arrives unexpectedly and creates urgency.
- Defenses include strong, unique **passwords**, **multi-factor authentication** 多因素认证, **encryption** 加密 (scrambling data so only authorized people can read it), and keeping software updated.

Encryption is the central tool for keeping data private in transit and storage. Being a responsible computer user means protecting your own and others' information.

Exam skill: be able to identify the beneficial and harmful effects of a given innovation, explain a privacy or security risk, and name a safe-computing practice that addresses it.

Worked example. A hiring algorithm is trained on a company's past hires, who were mostly one group, and it then rejects qualified applicants from other groups. Name the problem and its cause: this is **computing bias**, caused by biased training data — the model learned the historical pattern instead of a fair rule. A full-mark exam answer states the harm (qualified people are unfairly rejected) **and** its cause (the bias came from the data, not the code).

Exam tips

- Argue **both** the beneficial and harmful effects of a computing innovation — a balanced answer scores best.
- Use correct terms for data concerns: **PII**, privacy, security, and **algorithmic bias**.
- Explain how **crowdsourcing** and large data sets create value and raise new risks.
- Distinguish the **digital divide** (access) from bias (fairness) and give a concrete example of each.
- Tie every claim to a specific innovation and effect, as the written response demands.