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

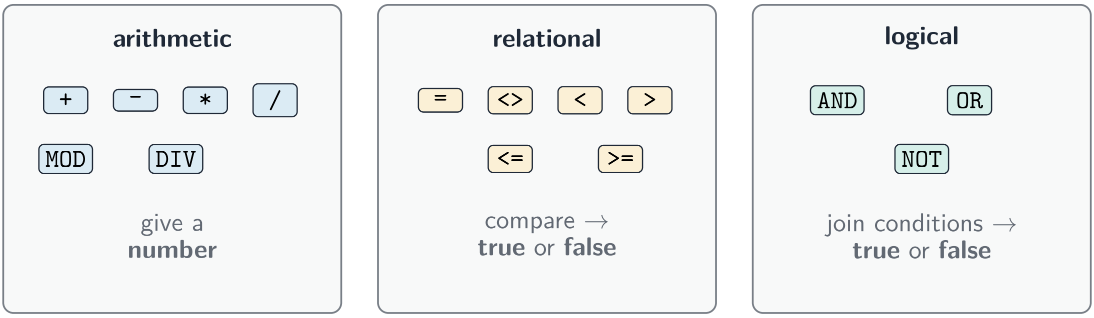
Programs compute with the operators `+`, `-`, `*`, `/`, and `MOD` (the **remainder** 余数 of a division, e.g. `17 MOD 5` is `2`). Expressions follow the usual order of operations. `MOD` is especially useful for testing divisibility (`n MOD 2 = 0` means `n` is even) and for wrapping values around a range.

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`:



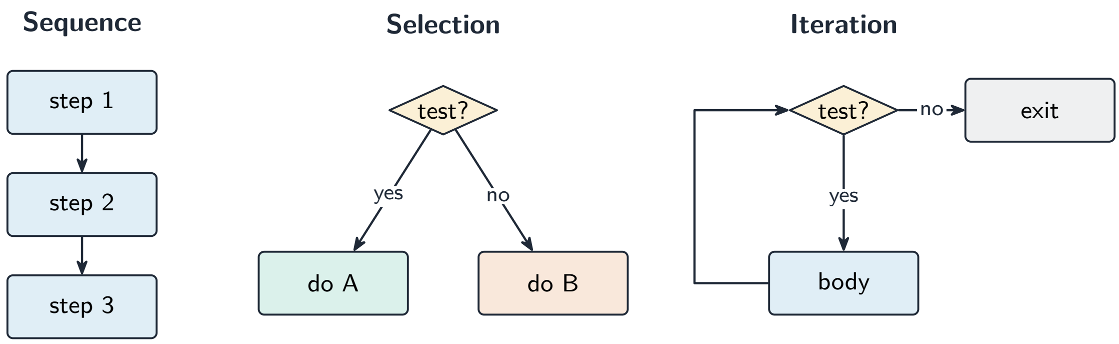
The three families of operators: arithmetic, relational, and logical

- `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")
}
```

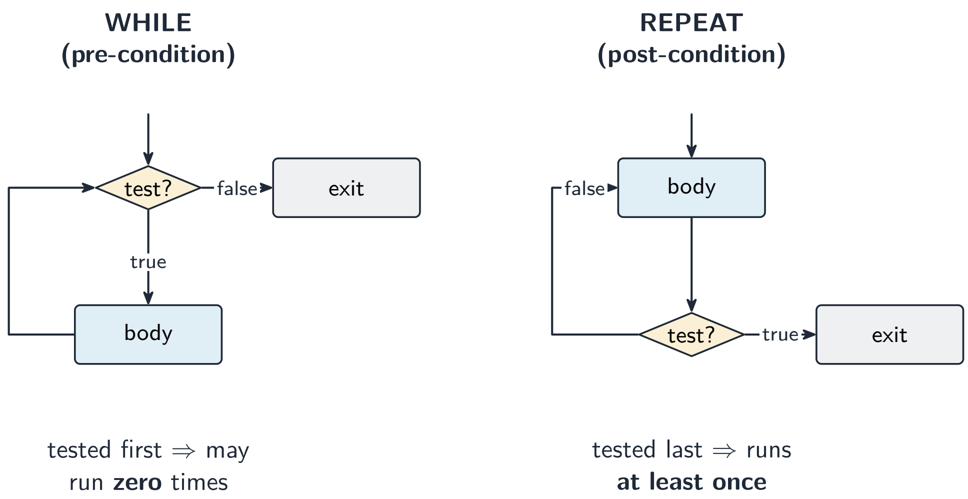
Nested Conditionals

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:



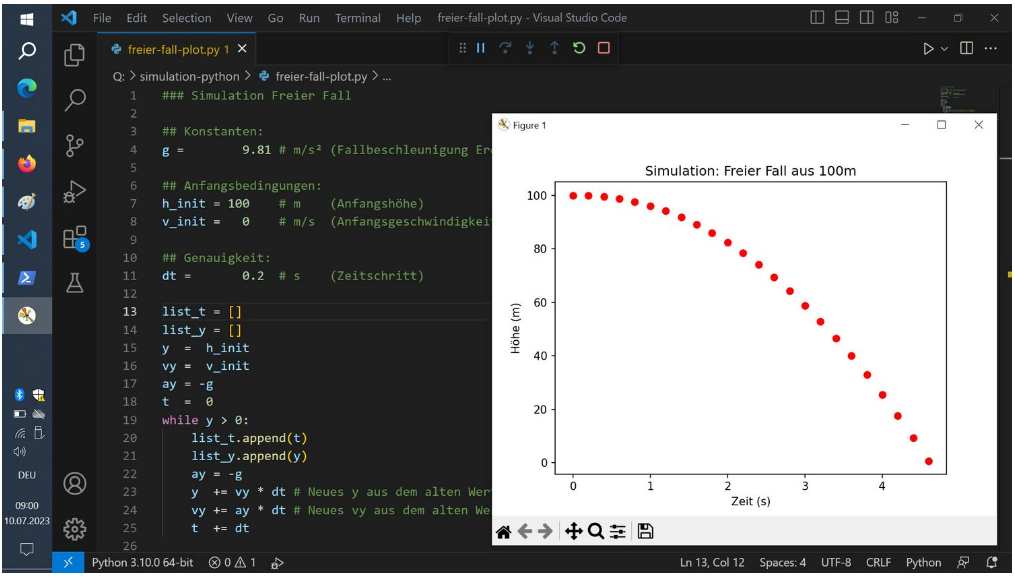
A pre-condition (WHILE) loop tests before the body, so it may run zero times

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

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

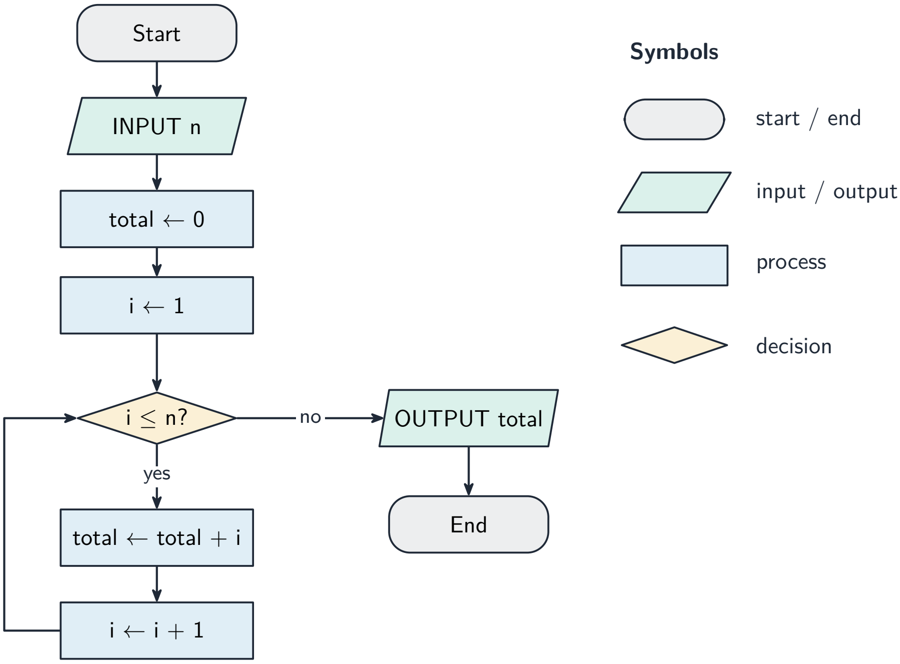
A loop that never meets its stopping condition is an **infinite loop** 无限循环.

Developing Algorithms



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

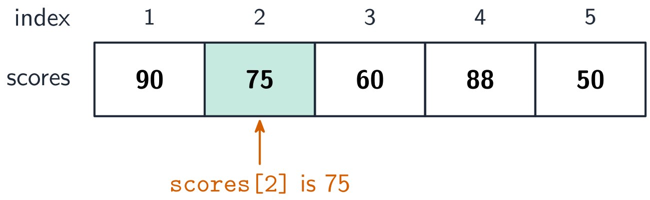
An **algorithm** 算法 is a finite sequence of steps that solves a problem, built from **sequencing**, **selection**, and **iteration**. Different algorithms can solve the same problem, and you should be able to combine and modify existing algorithms (for example, count the values in a list that meet a 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
```

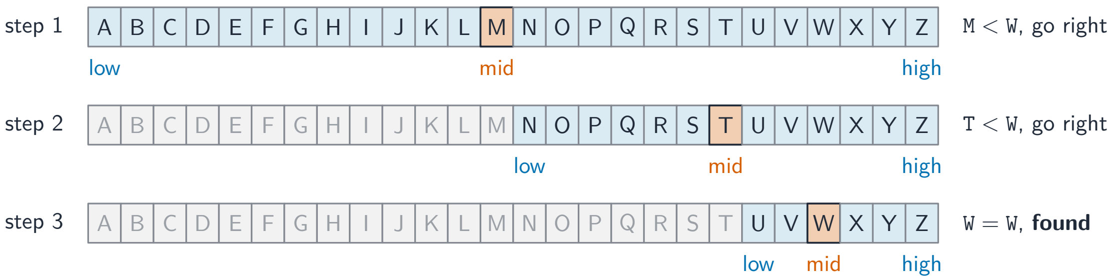
Traverse a list with a loop to sum, count, search, or find a maximum:

```
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

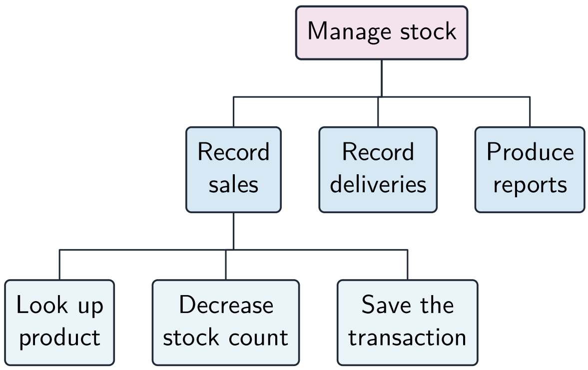
A **procedure (function)** 过程 is a named, reusable block of code. **Calling** it runs its code with the **arguments** you supply, and it may **return** a value:

```
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.

Random Values

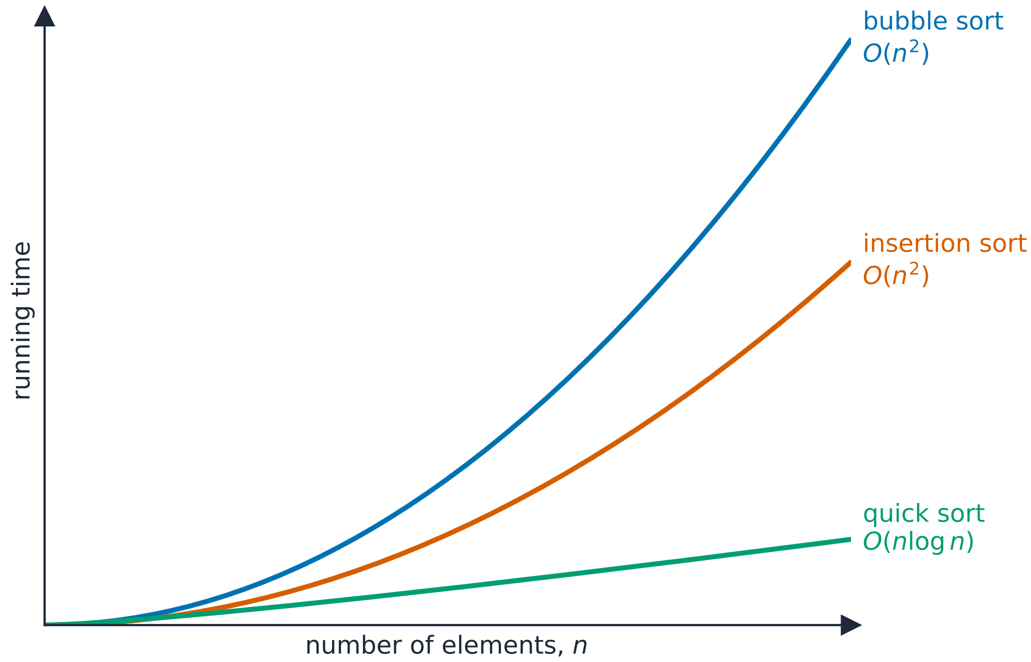
`RANDOM(a, b)` returns a random integer from `a` to `b` (inclusive), letting a program produce **unpredictable** results – for games, sampling, or simulations. Each call may give a different value, so a program using randomness behaves differently each run.

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.

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.



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.