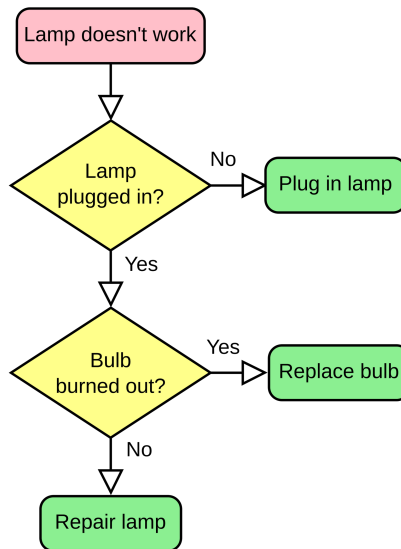


Selection and Iteration

AP Computer Science A

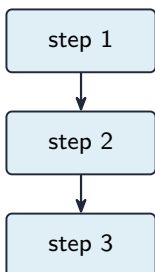
Selection and Repetition in Algorithms



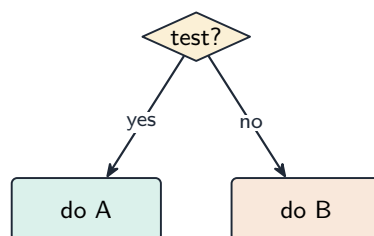
A flowchart with a decision diamond: selection chooses which path the algorithm takes

Algorithms are built from three **control structures** 控制结构: **sequence** (steps in order), **selection** 选择 (choosing a path), and **iteration** 迭代 (repeating steps). This topic covers selection and iteration – the tools that let a program make decisions and loop.

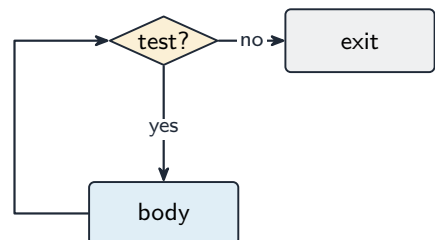
Sequence



Selection



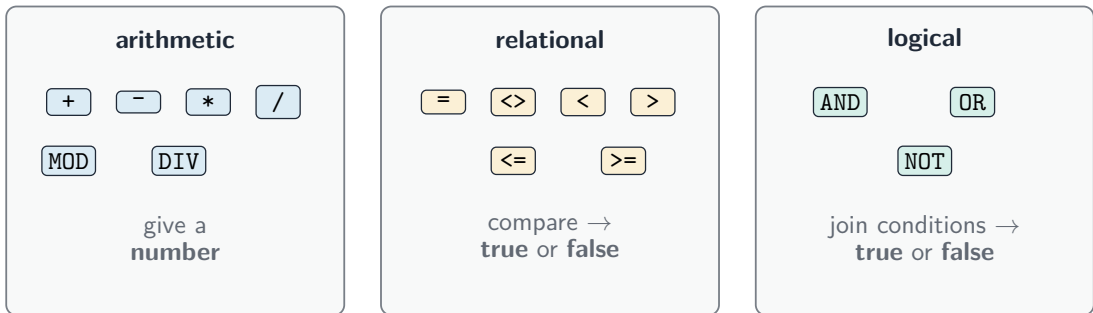
Iteration



The three control structures: sequence, selection, and iteration

Boolean Expressions

A **boolean expression** 布尔表达式 evaluates to **true** or **false**, using **relational operators** 关系运算符: `==` (equal), `!=` (not equal), `<`, `>`, `<=`, `>=`. Note `==` compares primitive values but **object references** for objects, so use `.equals` for `Strings`.



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

The if Statement

An **if statement** 条件语句 runs a block only when its condition is true; an optional **else** gives an alternative:

```
if (score >= 60) {  
    System.out.println("Pass");  
} else {  
    System.out.println("Fail");  
}
```



Traffic lights: selection chooses which branch runs, just as if statements choose code paths

Nested if Statements

Placing an **if** inside another, or chaining with **else if**, tests several cases in order. Only the **first** matching branch runs:

```
if (g >= 90) grade = 'A';  
else if (g >= 80) grade = 'B';  
else grade = 'C';
```

Compound Boolean Expressions

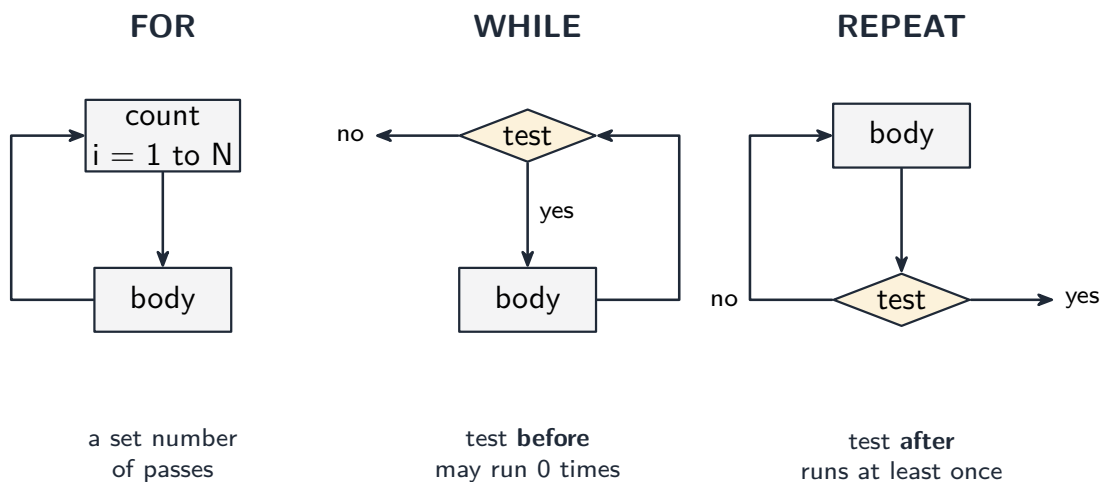
Logical operators 逻辑运算符 combine conditions: `&&` (**and** – both true), `||` (**or** – at least one true), `!` (**not** – reverse). Java uses **short-circuit evaluation** 短路求值: `&&` stops if the left side is false, and `||` stops if the left side is true – useful to guard against errors, e.g. `if (n != 0 && total / n > 5)`.

Comparing Boolean Expressions

De Morgan's laws 德摩根定律 rewrite negations: `!(a && b)` equals `!a || !b`, and `!(a || b)` equals `!a && !b`. Two boolean expressions are **equivalent** if they give the same result for every input – a truth table proves it. Simplifying conditions this way is a common exam task.

while Loops

A **while loop** 循环 repeats **while** its condition stays true, testing **before** each pass. You must change something inside so the loop eventually stops, or it becomes an **infinite loop** 无限循环:



The three loop types differ in where the condition is tested

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

```
}
```

for Loops

A **for loop** packs initialization, condition, and update into one line – best when you know the count:

```
for (int i = 0; i < n; i++) {  
    // runs n times, i = 0..n-1  
}
```

A **for** and an equivalent **while** do the same work; be able to convert between them.



An assembly line: loops repeat a process for every item, like for and while

Building Complete Selection and Iteration Algorithms

Combine loops and conditions to solve real problems – count, sum, find a maximum, or test a property:

```
int max = arr[0];  
for (int k = 1; k < arr.length; k++) {  
    if (arr[k] > max) max = arr[k];  
}
```

Two integer patterns the exam tests directly use % and /. To **read the digits of an integer** one at a time, repeatedly take `n % 10` (the last digit) and then `n = n / 10` (drop it). To test **divisibility**, `n % d == 0` means `n` is evenly divisible by `d`. Combine them with a counter to **find the frequency** with which some criterion is met.

Standard patterns like a running total, a counter, or a **flag** 标志 (a boolean that records whether something happened) recur throughout the course.

String Algorithms

Loop through a string by index to process each character:

```
for (int i = 0; i < s.length(); i++) {  
    char c = s.charAt(i);  
    // count vowels, reverse, check for a substring, ...  
}
```

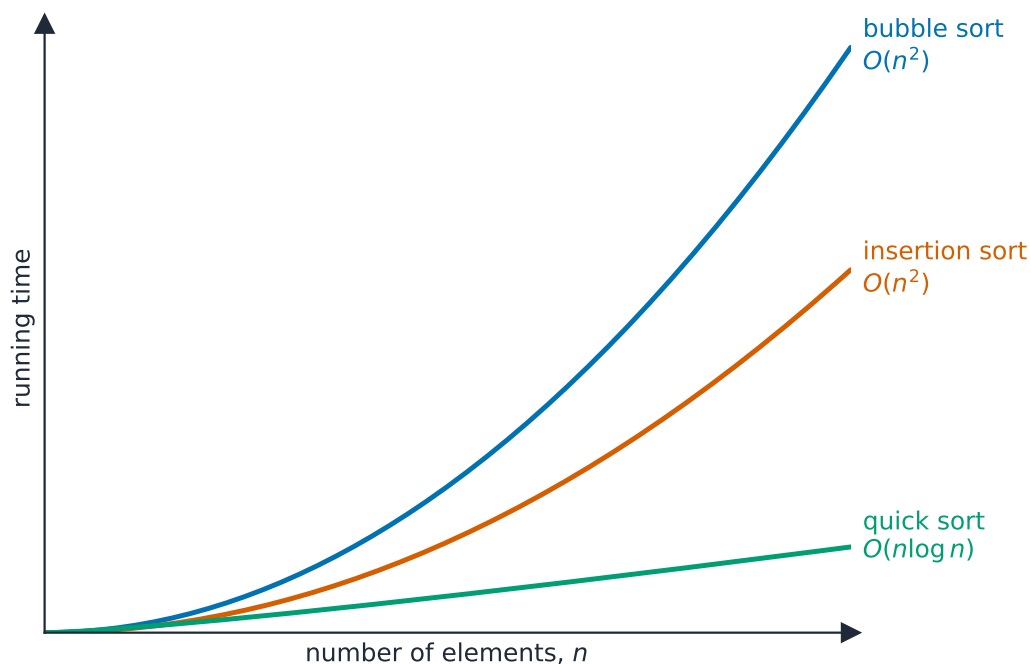
Typical tasks: count occurrences, build a reversed or filtered copy, or test whether one string contains another.

Nested Iteration

A **nested loop** 嵌套循环 puts one loop inside another; the inner loop completes fully for **each** pass of the outer. If the outer runs n times and the inner m times, the body runs $n \times m$ times – the basis for processing grids and comparing all pairs.

Informal Run-Time Analysis

Run-time analysis 运行时间分析 counts how many basic steps an algorithm takes as the input size n grows. Count the executions of the innermost statement: a single loop over n items is **linear** (n steps); two nested loops over n are **quadratic** (n^2). This informal counting lets you compare two algorithms' efficiency.



How the running time grows with the number of elements n

Exam skill: for a nested loop, be able to state how many times the inner statement runs in terms of the loop bounds – a frequent multiple-choice question.

Worked example. How many stars does this print?

```
for (int i = 0; i < 4; i++)  
    for (int j = 0; j < i; j++)  
        System.out.print("*");
```

The inner loop runs i times for each outer i : $0 + 1 + 2 + 3 = 6$ stars. When the inner bound is the *outer* variable, the total is the triangular sum $0 + 1 + \dots + (n - 1) = \frac{n(n - 1)}{2}$ – here $\frac{4 \times 3}{2} = 6$ – not the full $n^2 = 16$ of a rectangular nested loop.

Exam tips

- Get boundary conditions right: use `<` vs `<=` deliberately, and watch the first and last iteration of every loop (off-by-one is the classic bug).
- Build compound conditions with `&&`, `||`, `!` and remember **short-circuit** evaluation (put the null check first).
- Trace nested loops by counting how many times the **inner** body runs in total.
- Choose the right structure — `if/else if` for ranges, a loop for repetition — and avoid an infinite loop by updating the loop variable.
- Apply **De Morgan’s laws** when you simplify or negate a boolean condition.