Computational Thinking

A-Level Computer Science ▪ Topic 19

A-Level Computer Science

What we will cover

1. Searching algorithms
2. Sorting algorithms
3. Complexity
4. Recursion
5. Recap

1 Searching

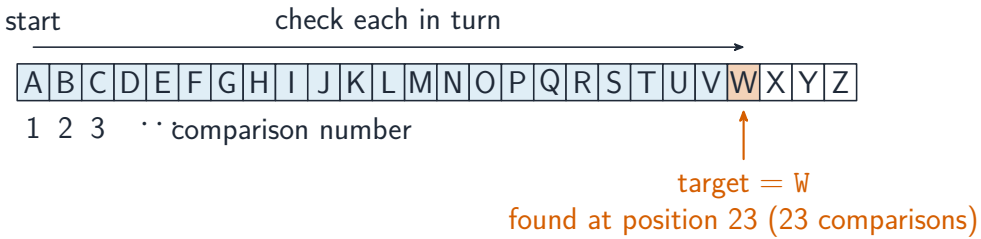
Linear search – check every item

A **linear search** 线性查找 walks from start to end until it finds the target (or the end). `FOR i ← 1 TO n IF A[i] = target THEN RETURN i RETURN −1` Works on any list. Worst case $O(n)$; best case 1 comparison. Use it on unsorted data or small lists.

> The returned -1 is a *sentinel*: an impossible position that means “not found”.

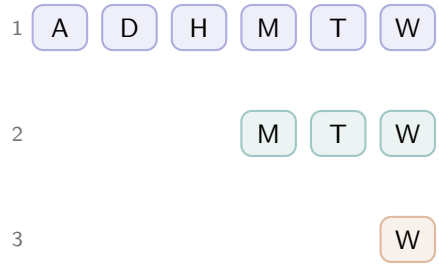
Linear search, step by step

Check each item in turn. Worst case is n comparisons, and the list need not be sorted.



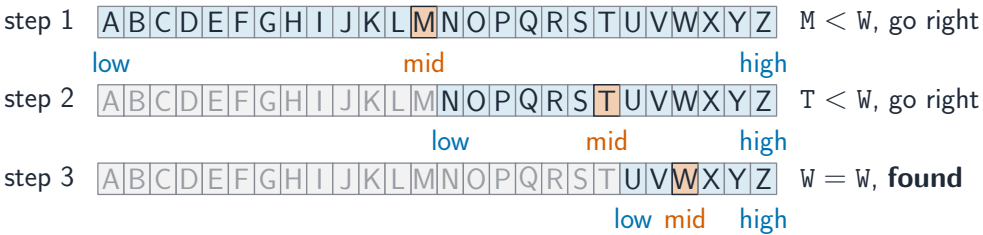
Binary search – halve the range

A **binary search** 二分查找 needs the data sorted. Look at the middle; if the target is smaller, search the left half, else the right. Worst case $O(\log_2 n)$ – a million items, about 20 comparisons. You must sort first (a one-off $O(n \log n)$ cost), worth it if you search many times.



Binary search halves the list

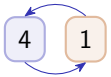
Compare with the middle, then discard half. $\log_2 n$ steps – but the list must be sorted.



2 Sorting

Bubble sort – largest bubbles to the end

A **bubble sort** 冒泡排序 walks the array, swapping adjacent pairs that are out of order. After each pass the next-largest item is in place. Best case $O(n)$ (already sorted, with an early-exit flag); average/worst $O(n^2)$. Simple, slow for large n.



swap

Insertion sort – grow a sorted prefix

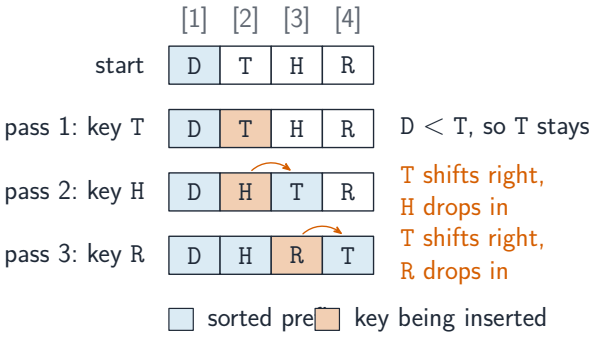
An **insertion sort** 插入排序 builds a sorted prefix from the left, inserting each new key by shifting larger items right. Best $O(n)$ (already sorted); worst $O(n^2)$. Good for small or nearly-sorted arrays. It sorts **in place** 原地 and is **stable** 稳定.

Trace [D, T, H, R]

pass 1 (T): no change
pass 2 (H): [D, H, T, R]
pass 3 (R): [D, H, R, T]

Insertion sort, traced

> Each new item is slid back into the part already sorted.



ADTs inside algorithms

The Abstract Data Types from Topic 10 show up inside many algorithms.

stack 栈

depth-first traversal, undo; push = prepend

queue 队列

breadth-first traversal, print order

linked list 链表

grow and shrink; a tree is nodes with two child pointers

dictionary 字典

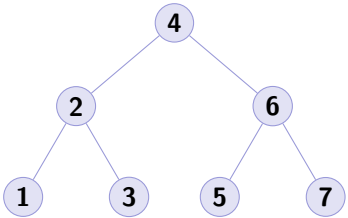
key → value, often on a hash table

Each one is chosen for how it is entered and left: a stack for depth-first work, a queue for breadth-first, a dictionary for lookup by key.

Binary trees and three traversals

A **binary tree** 二叉树: each node has up to two children. Three depth-first visits:

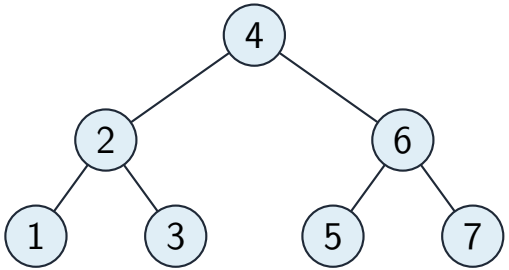
- pre-order – root, left, right
- in-order – left, root, right (sorted on a BST)
- post-order – left, right, root



in-order: 1 2 3 4 5 6 7

The three traversals

Pre-, in- and post-order differ only in when the node itself is visited; in-order on a binary search tree comes out sorted.



pre-order: 4 2 1 3 6 5 7

in-order: 1 2 3 4 5 6 7

post-order: 1 3 2 5 7 6 4

3 Complexity

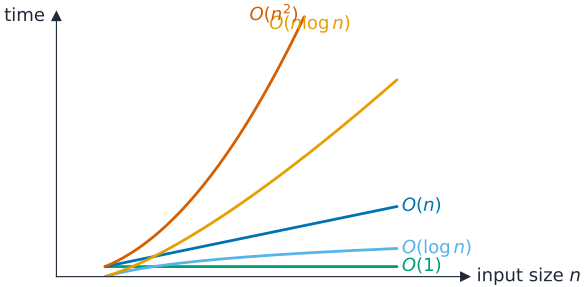
Time complexity – how it grows with n

Time complexity 时间复杂度 is written in **Big-O** 大 O 表示法: the dominant term.

$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
constant	binary search	linear search	good sorts	bubble, insertion

> A smaller order wins at scale, even if another algorithm is faster for small n.

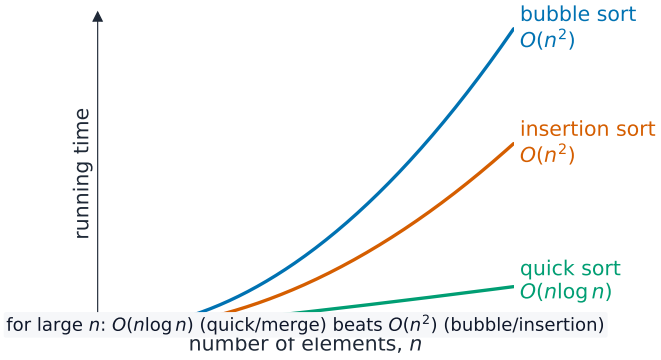
How the orders of growth compare



for large n: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$

Every curve is beaten eventually by the one below it – which is why $O(n^2)$ sorting is unusable on real data volumes.

Time complexity



How sorting time grows with the number of elements n: $O(n^2)$ sorts climb away from an $O(n \log n)$ sort

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.

Why order of growth matters

A million-name book: linear search averages 500,000 checks, binary search about 20.

Worked example – 1000 sorted items

A sorted list holds 1000 items. Worst-case comparisons for each search?

Linear search checks items one at a time: up to 1000 – $O(n)$.

Binary search halves each step: at most $\lceil \log_2 1000 \rceil = 10$ – $O(\log n)$.

Double the list to 2000: binary search adds one comparison; linear search adds up to another 1000. That is why the order of growth, not raw speed, decides the winner at scale.

Space, stability, simplicity

Space complexity 空间复杂度 is the extra memory.

- bubble / insertion: $O(1)$ extra (in place)
- merge sort: $O(n)$
- recursion uses stack memory proportional to its depth

Also weigh simplicity, stability (equal items keep their order), and adaptiveness (faster on nearly-sorted data). The right algorithm depends on the data and the constraints.

4 Recursion

A routine that calls itself

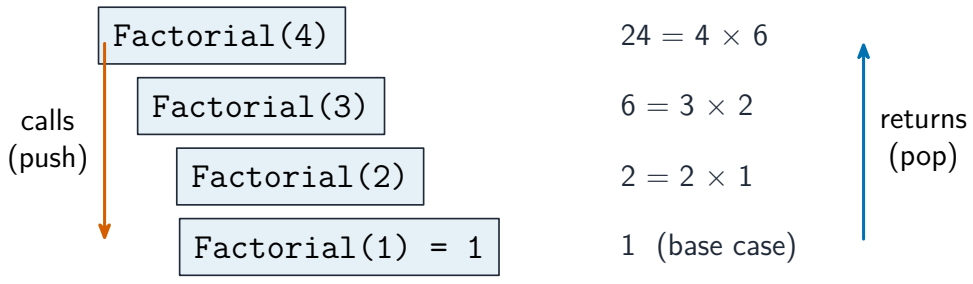
Recursion 递归 has two parts: a **base case** 基本情形 (small enough to solve directly) and a **recursive case** 递归情形 (reduce the input and call itself).

```
FUNCTION Factorial(n)
 IF n = 0 OR n = 1 THEN
  RETURN 1
 ELSE
  RETURN n * Factorial(n − 1)
 ENDIF
```

Without a base case the recursion never stops – a **stack overflow** 栈溢出.

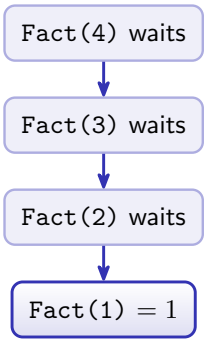
What the stack holds

Each call adds a frame with its own parameters and return address; the base case is what lets the stack unwind.



Tracing Factorial(4) on the call stack

Each call has its own **stack frame** 栈帧: parameters, locals, **return address** 返回地址.



Unwinding: $2 \times 1 = 2$, $3 \times 2 = 6$, $4 \times 6 = 24$. There is no special "recursion mechanism" – it is ordinary call-and-return.

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**.

When recursion is a poor fit

infinite

miss the base case
→ stack overflow

deep

high memory use
for a tall call stack

repeated work

naive Fibonacci
is exponential
– use a loop or
memoisation 记忆化

Exam tips

- Match each algorithm to its **Big-O**: linear search $O(n)$, binary search $O(\log n)$, bubble/insertion $O(n^2)$, good sorts $O(n \log n)$.
- **Binary search needs a sorted list** and halves the search space each step.
- A **recursive** routine needs a base case and a call to itself; explain how deep recursion overflows the call stack.
- Trace a sort or search with a table when asked, showing each pass.

5 Recap

Topic 19 – key takeaways

1 Search

linear $O(n)$; binary $O(\log n)$, needs sorted

2 Sort

bubble / insertion $O(n^2)$; trace each pass

3 Big-O

dominant term; smaller order wins at scale

4 Recursion

base case + recursive case; frames on the stack

Check yourself

1. What must be true of a list before binary search?
2. Best-case time for insertion sort on an already-sorted list?
3. In-order traversal of a BST gives the keys in what order?
4. What happens if a recursive routine has no base case?

Answers **1.** it must be sorted **2.** $O(n)$ **3.** sorted order **4.** stack overflow