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

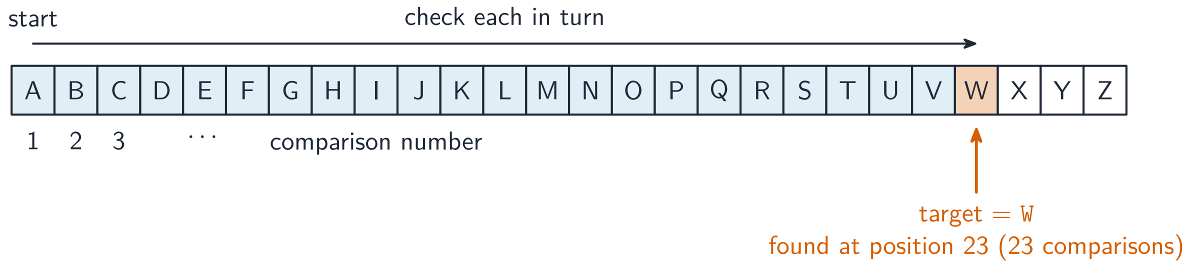
Image: © 2010 by Tomasz Sienicki [user: tsca, mail: tomasz.sienicki at gmail.com], CC BY 3.0 (commons.wikimedia.org)

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
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.



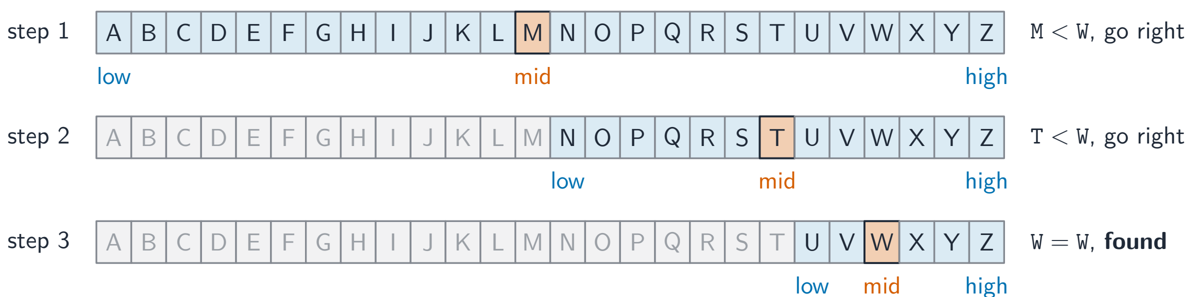
Linear search checks every letter in turn —23 comparisons to find W

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
    mid ← (low + high) DIV 2
    IF A[mid] = target THEN RETURN mid
    IF A[mid] < target THEN
        low ← mid + 1
    ELSE
        high ← mid - 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.



Binary search halves the range each step (low / mid / high) —just 3 comparisons to find W

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 EXIT FOR   // already sorted
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

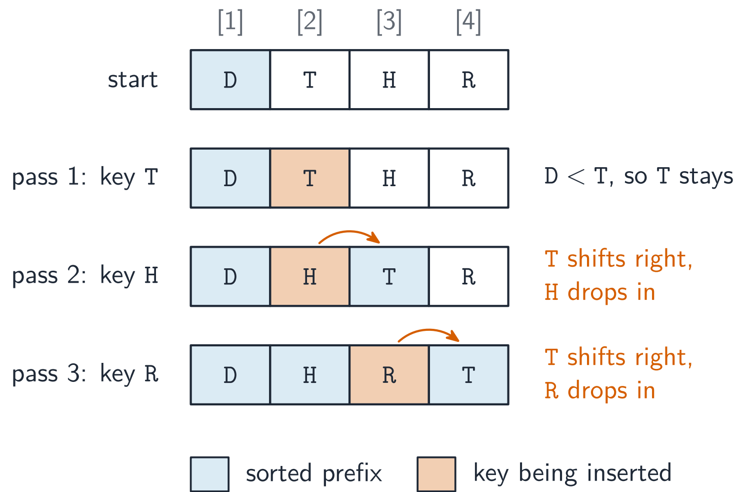
An **insertion sort** 插入排序 builds a sorted prefix from the left, inserting each new element into place by shifting larger ones right:

```
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]`.

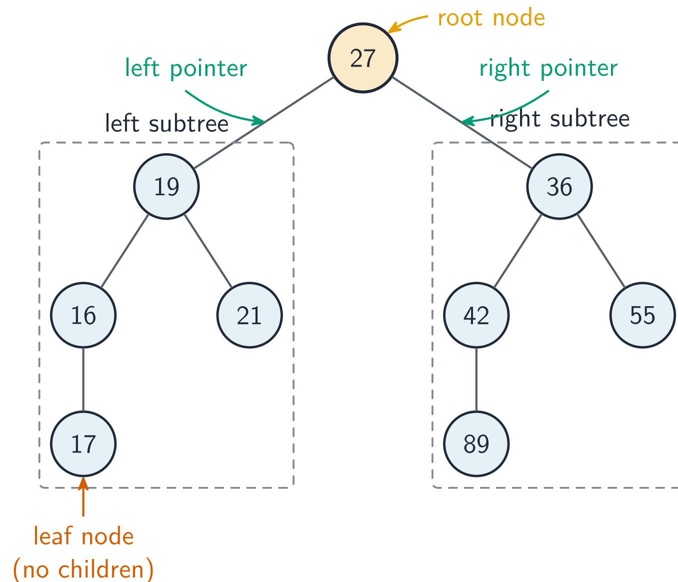


An insertion sort of [D, T, H, R], shifting each key into its place pass by pass

ADTs in algorithms

The 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. Layering this way separates concerns —the algorithm using the ADT need not know how it is built.

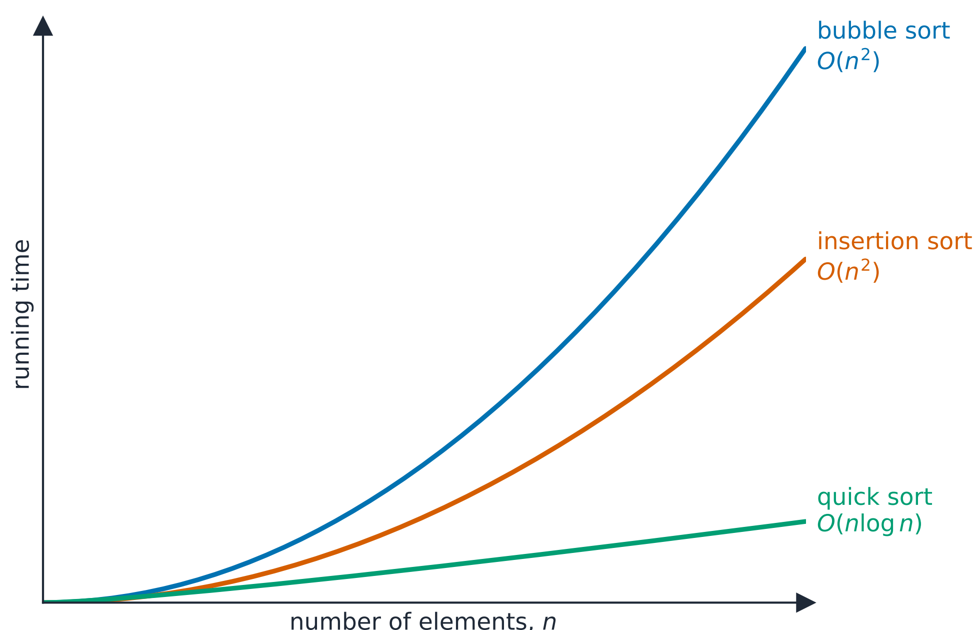


A binary tree: each node has up to two child nodes

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.



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

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

A **recursion** 递归 algorithm 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.

Tracing a recursive call

For `Factorial(4)`: the calls go down to `Factorial(1)=1`, then unwind: `2*1=2`, `3*2=6`, `4*6=24`. Final result `24`. Track each pending call on a stack.

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.

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”.