資料結構與演算法入門：第 1 章

演算法複雜度分析

悠太翼 Yuuta Tsubasa

July 16, 2025

同樣是乘法，哪個比較快？

寫法一：用加法模擬乘法

```
int multiply(int a, int b) {
    int result = 0;
    for (int i = 0; i < b; i++) {
        result += a;
    }
    return result;
}
```

寫法二：直接用乘法運算子

```
int multiply(int a, int b) {
    return a * b;
}
```

問題：

- 如果 $a = 12$, $b = 1000000$，兩者誰比較快？為什麼？
- 我們可以怎麼評估程式「快不快」？

各種基本運算所需的指令數（假設）

為了更真實地比較不同程式的效率，我們先來假設：

操作類型	**指令數（估計值）**
變數初始化（如 `int x = 0`）	1
加法（`x + y`）	1
乘法（`x * y`）	2
比較（`x < y`）	1
賦值（`x = y`）	1
函式呼叫（不含內容）	1
迴圈迭代一次	每次約 3（包含比較、執行、遞增）

- 我們會用這些估計，來推算整段程式「大概」會執行幾個基本指令
- 雖然實際上會更複雜，但這是理解效率的第一步

怎麼比較程式的效率？

加法版本的乘法：估算指令數

- `int result = 0;` → 初始化：1 指令
- `int i = 0;` → 初始化：1 指令
- `for` 迴圈跑 `b` 次，每次包含：
  - 比較（`i < b`）：1 指令
  - 加法（`result += a`）：1 指令
  - 遞增（`i++`）：1 指令

  → 每次迴圈共 3 指令 × `b` 次
- `return result;` → 1 指令

總共：$1 + 1 + 3b + 1 = 3b + 3$ 次指令

- 而直接乘法如 `return a * b;` 只需約 3 指令（乘法 + return）
- 所以我們可以透過「基本指令總數」來估算效率差異

比較兩種乘法方式的指令數

根據我們對「基本運算所需指令數」的假設：

- **方式一（用加法實作）**：總共約 $3b + 3$ 指令
- **方式二（直接使用乘法運算子）**：約 3 指令（乘法 + return）

代入不同參數來估算指令總數：

a	b	加法實作 $(3b + 3)$	乘法運算子
1	5	18	3
10	100	303	3
5000	6	21	3
3000	1,000,000	3,000,003	3
5	2,147,483,647	6,442,450,944	3

結論：當輸入變大，**使用加法的版本會產生極大量的重複操作**，而直接乘法始終只需常數次指令。

我們想要找「上限」

- 剛才我們得到：用加法做乘法需要 $3b+3$ 次操作
- 現在我們想問：
  - 有沒有一個比較簡單的式子可以「估過它」？
  - 我們不需要知道「準確」幾次，只要知道「最多大概多少」
- 這就帶出「Big O」的想法！

Big O 的數學定義（非正式）

$$f(n) = O(g(n)) \Longrightarrow \text{存在常數} c\text{，當} n \text{ 很大時，}$$

$$\text{可以使得} f(n) \leq c \cdot g(n)$$

- 舉例：$3b+3 \leq 4b$，當 b 很大時就成立
- 所以我們說：$3b+3 = O(b)$

從指令數推導 Big O：幾個簡單例子

我們已經知道：可以估算每個指令的成本，現在我們試著從估出來的「總指令數」來找出 Big O！

範例一：一次加法

```
1 int result = a + b;
```

指令數：約 1（加法）+ 1（賦值）= 2 → Big O：$O(1)$

範例二：for 迴圈相加

```
1 int sum = 0;
2 for (int i = 0; i < n; i++) {
3     sum += i;
4 }
```

初始： 2 指令（sum, i）**每次迴圈：** 3 指令 × n → **總指令數：約** $3n + 2$ → **Big O：** $O(n)$

從指令數推導 Big O：巢狀迴圈

```
1 for (int i = 0; i < n; i++) {
2     for (int j = 0; j < n; j++) {
3         count++;
4     }
5 }
```

分析指令數：

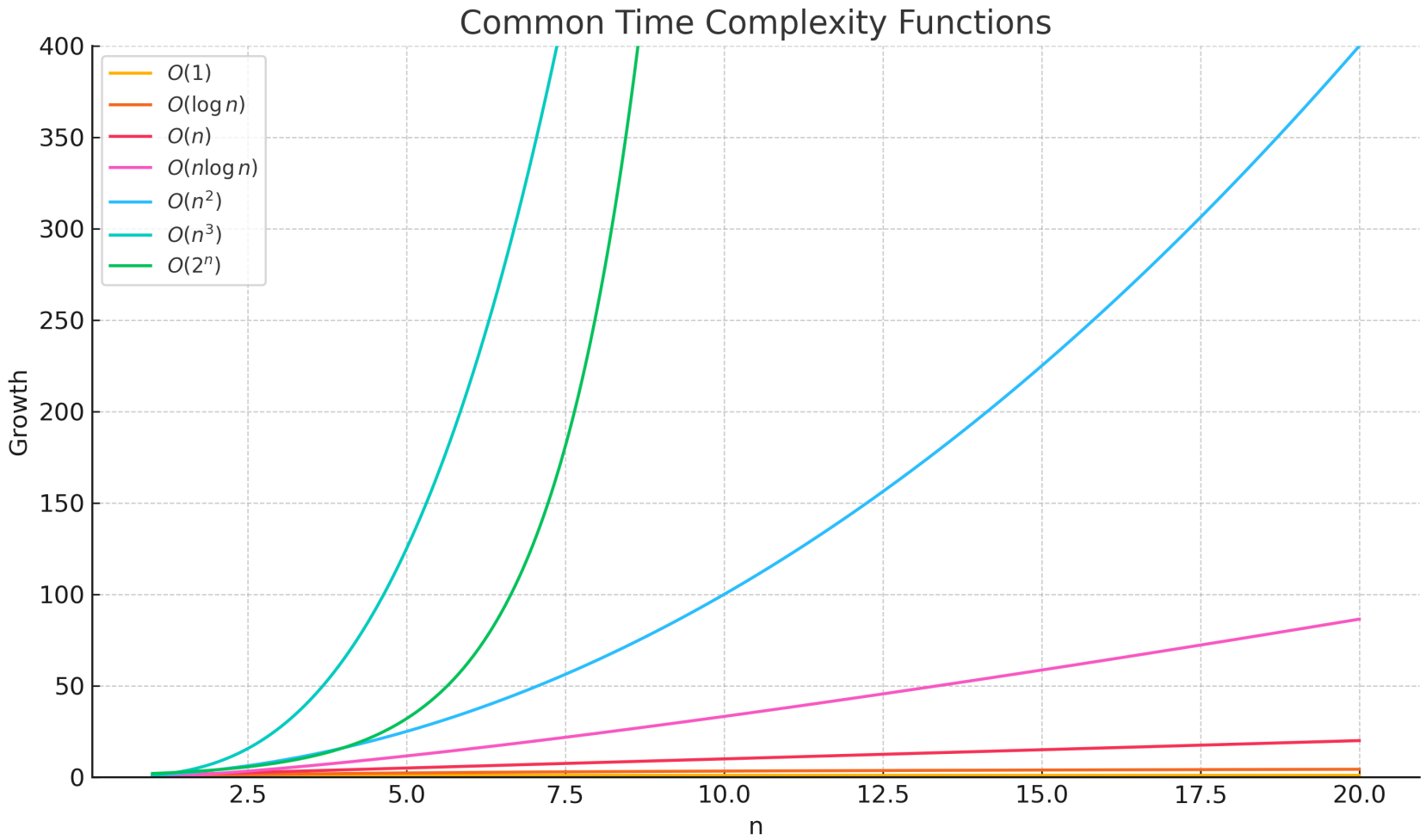
- 外層：
  - 初始化 `int i = 0;`：1 指令
  - 比較 `i < n`：約 $n+1$ 次
  - 遞增 `i++`：約 n 次
- 內層（每次外層執行 `n` 次）：
  - 初始化 `int j = 0;`：n 次
  - 比較 `j < n`：$n(n+1)$ 次
  - 遞增 `j++`：n^2 次
  - 主體 `count++`：n^2 次

總指令數估算：約 $3n^2 + 4n + 1$

結論：主導項為 n^2，因此這段程式的複雜度為 $\boxed{O(n^2)}$

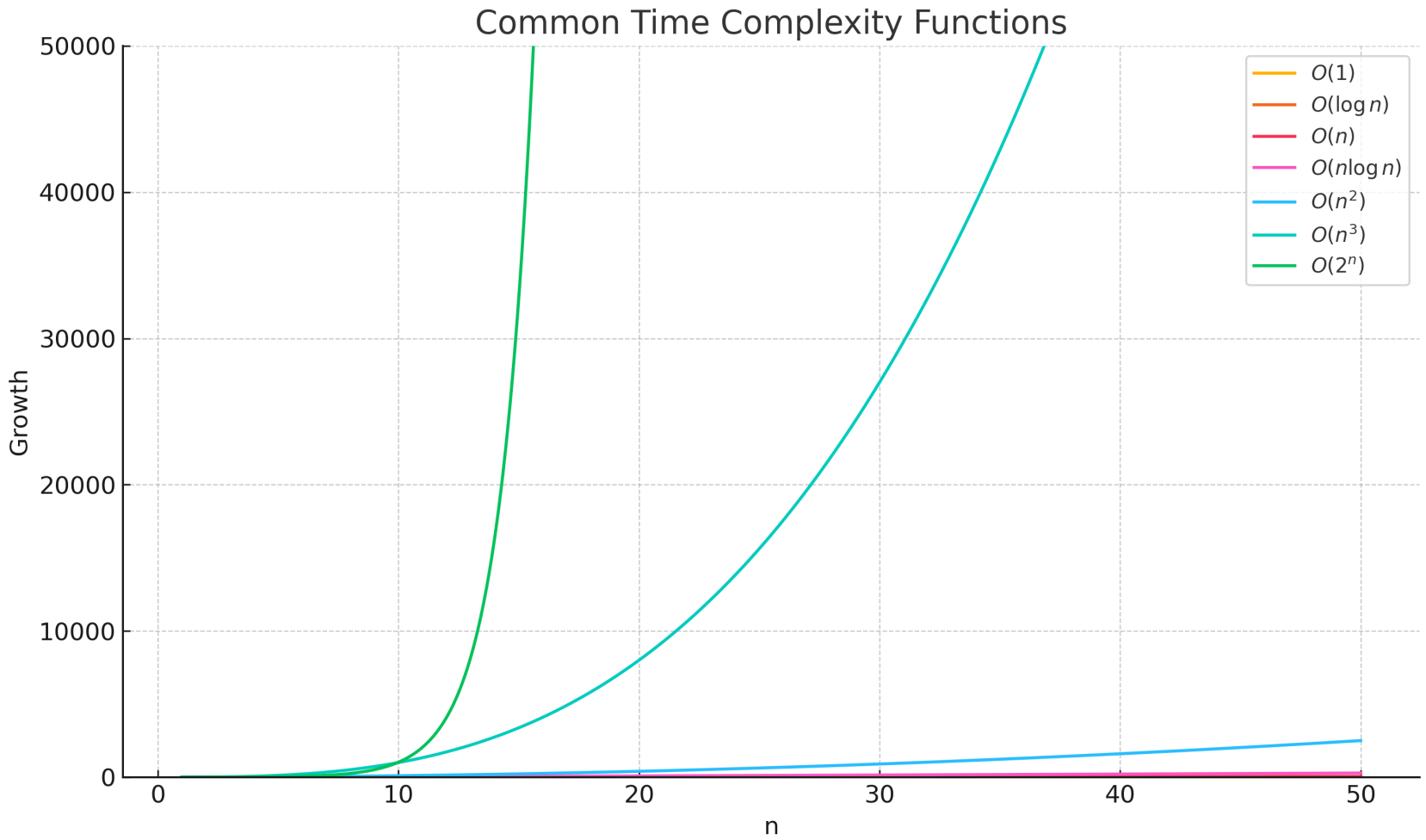
常見時間複雜度的成長速度比較

透過 Big O 的表示法，我們可以將不同演算法的效率分門別類，依據「輸入大小 n 增加時，執行時間成長的快慢」來做比較。



為什麼要把 n 放大來觀察？

在前一張圖中，我們看到 $O(n^3)$ 好像比 $O(2^n)$ 還大？但那只是因為我們看的範圍太小，2^n 的成長是「爆炸性的」，需要把 n 放大才能看出來！



空間複雜度也是 Big O 的應用

除了分析「演算法執行多久」，我們也可以分析「需要多少記憶體」這稱為：**空間複雜度（Space Complexity）**

幾個範例：

- 宣告一個變數：　$O(1)$ 空間
- 宣告一個長度為 n 的陣列：　$O(n)$ 空間
- 宣告一個 $n \times n$ 的二維陣列：　$O(n^2)$ 空間
- 遞迴呼叫 n 層函式，每層都有參數與區域變數：　$O(n)$ 空間（呼叫堆疊）

結論： Big O 不只能用來估時間，**也可以估演算法的空間需求**

進階補充：五種常見漸進符號的數學定義（可略過）

以下是 Big O 與其他常見漸進符號的正式數學定義：

符號	**數學定義**
$O(g(n))$	存在正實數 c 和 n_0，使得對所有 $n \geq n_0$，都有 $f(n) \leq c \cdot g(n)$
$o(g(n))$	對所有正數 c，存在 n_0，使得對所有 $n \geq n_0$，都有 $f(n) < c \cdot g(n)$
$\Omega(g(n))$	存在正實數 c 和 n_0，使得對所有 $n \geq n_0$，都有 $f(n) \geq c \cdot g(n)$
$\omega(g(n))$	對所有正數 c，存在 n_0，使得對所有 $n \geq n_0$，都有 $f(n) > c \cdot g(n)$
$\Theta(g(n))$	同時滿足 $O(g(n))$ 與 $\Omega(g(n))$ 的條件

備註：這些定義有助於理解演算法在「最佳、最壞、平均」等不同情況下的行為。

本章總結

- **時間複雜度（Time Complexity）**
  - 分析程式在不同輸入大小下，會執行多少「基本操作」
  - 幫助我們預估程式的執行效率
- **空間複雜度（Space Complexity）**
  - 分析演算法需要佔用多少額外記憶體空間
  - 特別重要於資料量龐大或遞迴深層的情況
- **Big O 表示法**
  - 用來描述演算法的「上限成長速度」
  - 忽略常數與低次項，聚焦在輸入變大時的趨勢
  - 也有 Big Ω、Θ 等進階表示法

搞懂這些，才是真正開始學演算法的第一步！