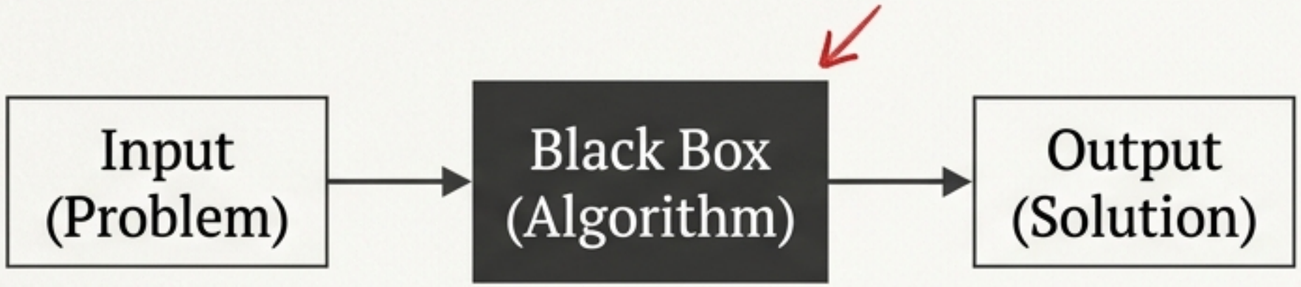
演算法複雜度分析：從零到精通
Algorithm Complexity Masterclass

ITP4510 Chapter 4.1 | The Feynman Approach

Instructor: 資深教授 (Senior Professor)
Goal: 從基礎概念到完全掌握 Big-O 標記法
(From Zero to Mastery in Big-O Notation)

什麼是演算法？ (What is an Algorithm?)



費曼說 (Feynman says):
「如果你不能用簡單的話解釋，代表你還沒真正懂。」
(If you can't explain it simply, you don't understand it well enough.)

Definition: An algorithm is a sequence of computational steps for solving a problem.
(解決問題的計算步驟序列) 。

Good Data Structure + Good Algorithm = Perfect Solution

Enrichment (增潤項目)	
Why Pseudo-code? 真實的 Java 程式碼包含繁雜的實作細節。虛擬碼 (Pseudo-code) 幫我們過濾細節，只專注於「高階思維」(High-level ideas)。	**Data Structures:** 組織大量資料的方式。好的資料結構加上好的演算法，才能產出完美的解決方案！

衡量效能的兩大指標 (Program Performance)

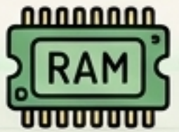
寫出能跑的程式不夠，我們需要「跑得快」且「省資源」的程式。

本章核心

Time Complexity (時間複雜度) ★

Definition: The amount of computer time it needs to run to completion.
(程式執行完畢所需的電腦時間)。

費曼比喻：煮好一頓飯需要花多少分鐘？



Space Complexity (空間複雜度)

Definition: The amount of memory it needs to run to completion.
(程式執行時所需佔用的記憶體空間)。

增潤補充：Merge Sort 需要額外的陣列空間，Space Complexity 較差；而 Quick Sort 是 in-place 排序，Space 效能較佳。

如何評估效能？ (Algorithm Analysis Techniques)

Experimental Approach (實驗法)

- **方法**: Implement and run it! (寫成程式碼在電腦上跑)。
- **缺點**: 結果極度依賴硬體設備與 OS 環境。
- **缺點**: 資料量小 (Small n) 時看不出差異。
- **缺點**: 無法做到絕對客觀。

Too dependent on hardware!

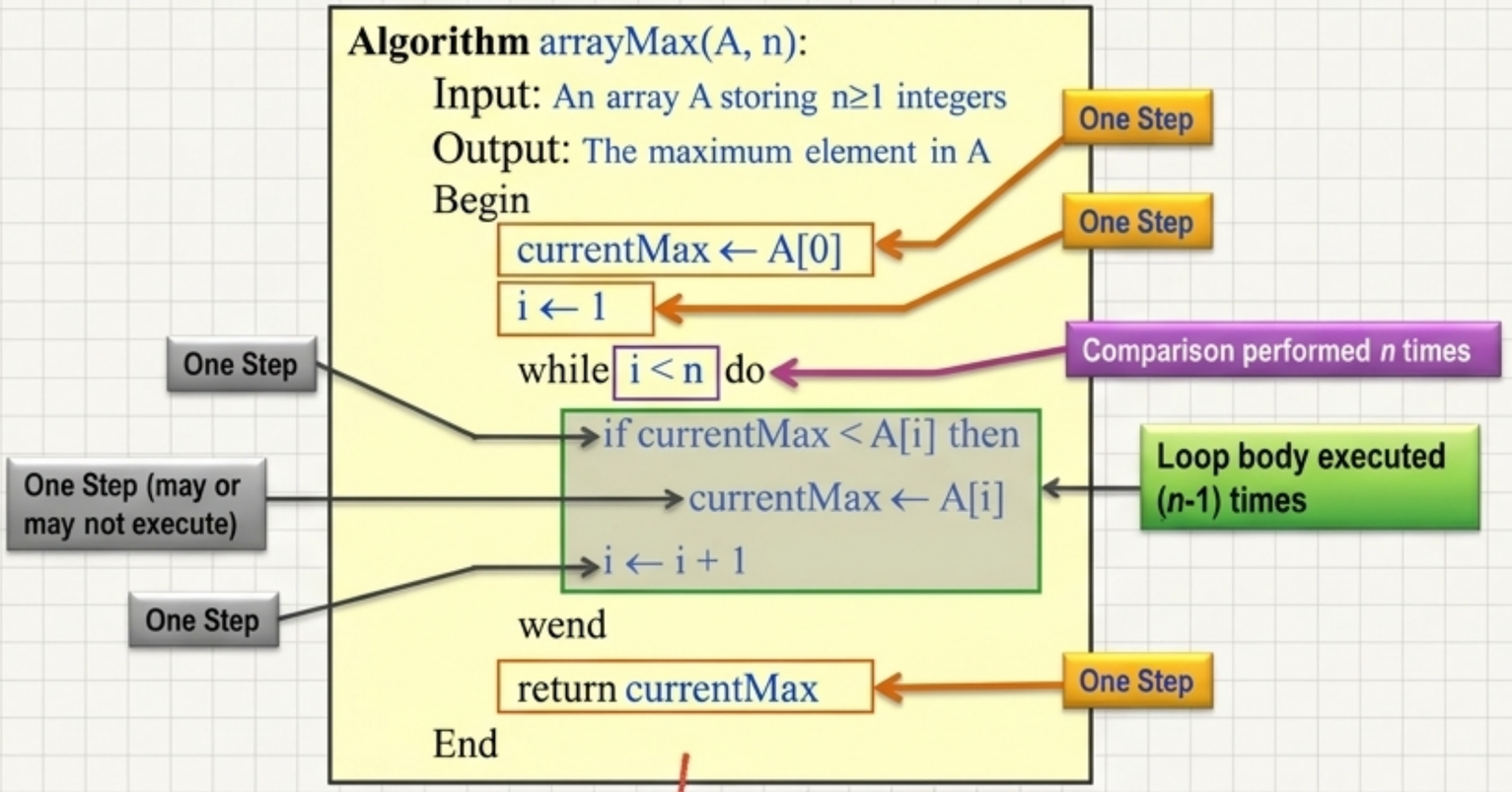
Analytical Approach (分析法) ⭐ (We use this!)

- **方法**: Study the pseudo-code using mathematical techniques. (透過數學分析虛擬碼)。
- **優點**: 允許在「不寫程式」的情況下，客觀比較不同演算法。
- **核心思維**: 專注於當資料量非常龐大時 (n is very large)，時間增長的趨勢 (Growth Rate)。

We choose this one!
我們選擇它!

分析第一步：計算步驟 (Counting the Steps)

數學分析的第一個動作：計算演算法執行了多少個「步驟」(Steps)。以找出陣列最大值（arrayMax）為例：



- 初始化 (currentMax, i) → 各執行 1 次 (One Step)。
- while 迴圈條件判斷 → 執行 n 次。
- 迴圈內部邏輯 (if 判斷 , i+1) → 執行 (n-1) 次。

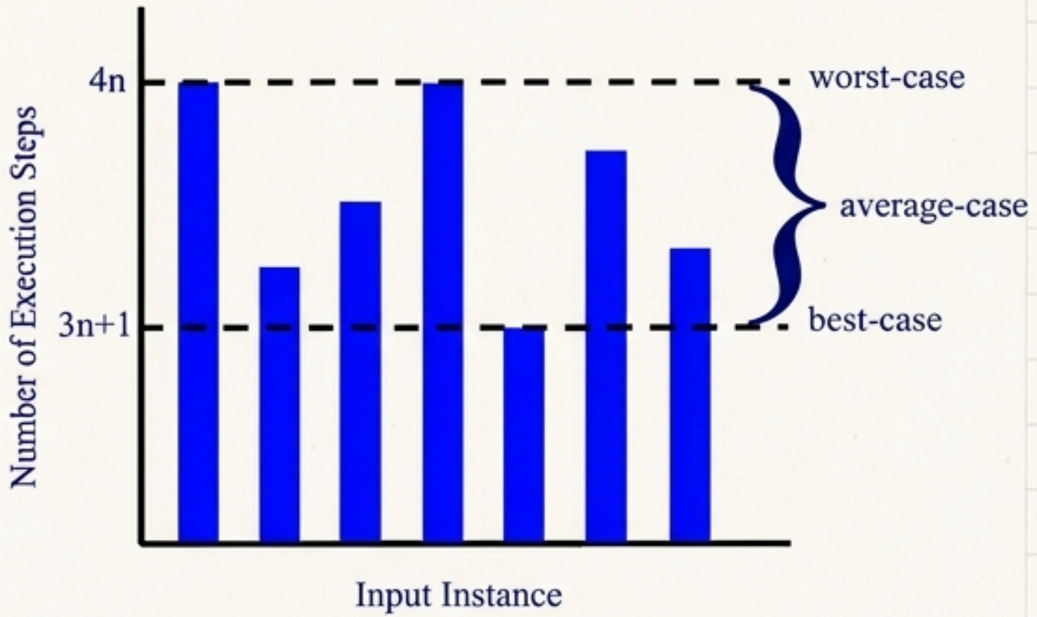
Professor's Insight: 發現了嗎？程式的總步驟數，會隨著輸入的資料內容與大小 (Input Instance) 而劇烈改變！

最佳、最壞與平均情況 (Scenarios in Analysis)

Scenario 1: Best-case (最佳情況)

Definition: Minimum number of steps. (需要最少步驟的情況)。

Example: 最大值剛好在 A[0]，$T = 3n + 1$ 步。



Scenario 2: Worst-case (最壞情況) ⭐

Definition: Maximum number of steps. (需要最多步驟的情況)。

Example: 陣列由小到大排序，每次都要更新最大值。$T = 4n$ 步。

Scenario 3: Average-case (平均情況)
落在兩者之間，但數學推導過於複雜，實務上較少作為唯一指標。

Professor's Rule: 我們永遠最關注 Worst-case。因為它提供了效能的「保證上限 (Upper Bound)」— 再怎麼糟，也不會比這更慢！

為什麼需要漸進複雜度？ (Why Asymptotic Complexity?)

Feynman's Question Box

如果 $T = 3n^2 + 4n + 1000$，當資料量 n 變成一千萬時，後面的 $4n$ 和 1000 還重要嗎？

n (Data Size)	n^2 (Dominant Term)	Other Terms (4n + 1000)
10	100	1,040
1,000	1,000,000	5,000
10,000,000	100,000,000,000,000	40,000,1000

佔據 99.9% (Approaches 99.9%)

Asymptotic Complexity (漸進複雜度):

- **核心概念**: 簡化函數！消除不會實質改變數量級的常數與低階項。
- Dominant Term (**主導項**): 當 n 趨近無限大時，最高次方項（如 n^2）會佔據 99.9% 的影響力。
- 結論: 忽略瑣碎細節，專注於主導項的成長趨勢。這就是 Big-O 標記法 的由來！

Big-O 標記法詳解 (Big-O Notation)

Upper Bound (最壞情況的漸進上限)

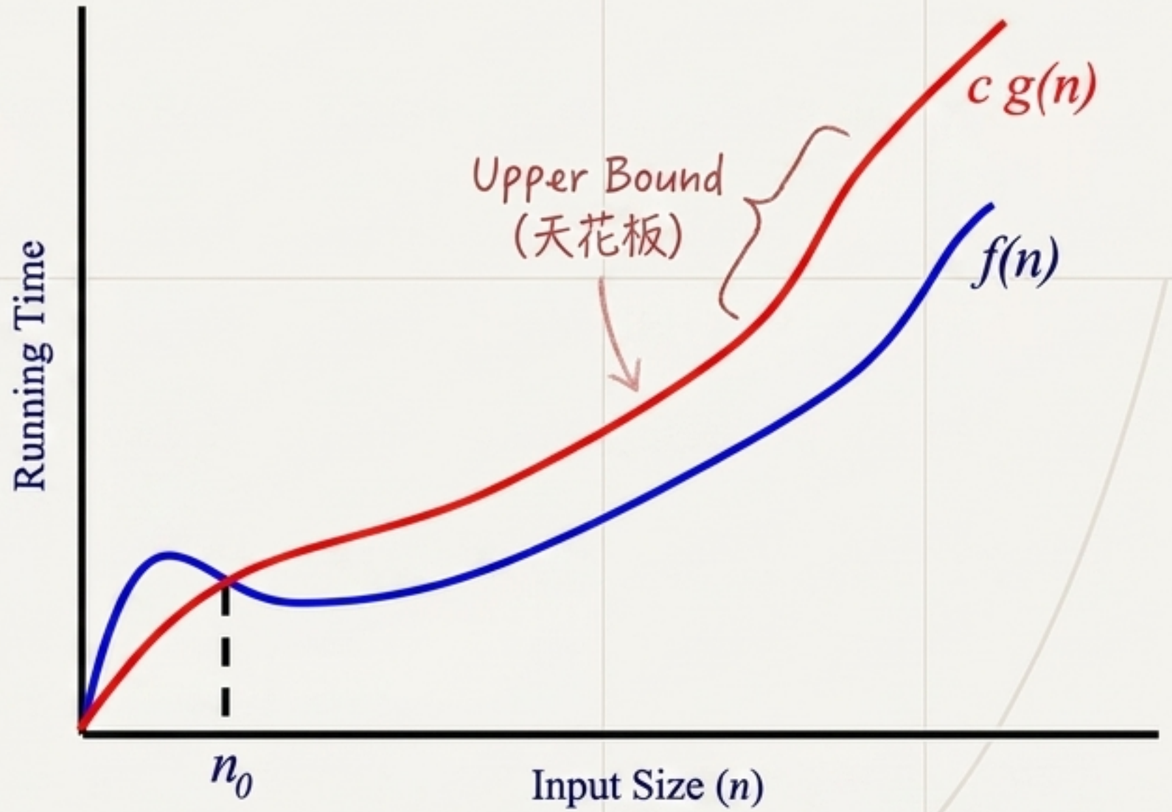
Formal Definition (嚴格定義)：

$f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that:

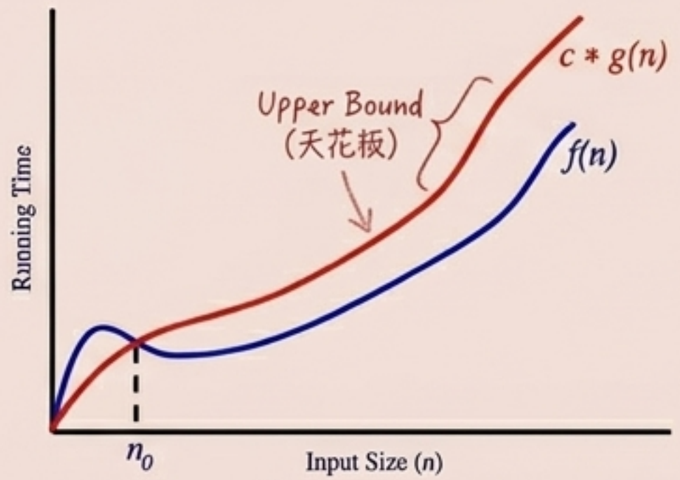
```
f(n) <= c * g(n) for all n >= n_0
```

費曼白話文翻譯 (The Feynman Translation)：

- $f(n)$ 是你實際演算法的時間。$g(n)$ 是標準函數 (如 n^2)。
- 只要我們能找到一個倍數 (c) 和一個起點 (n_0)，讓 $c * g(n)$ 在起點後 永遠蓋在 $f(n)$ 的上面（畫一條天花板），我們就說 $f(n)$ 是 $O(g(n))$。
- 意義：演算法成長速度「至多 (at most)」跟 g(n) 一樣快，保證了效能的最壞極限。



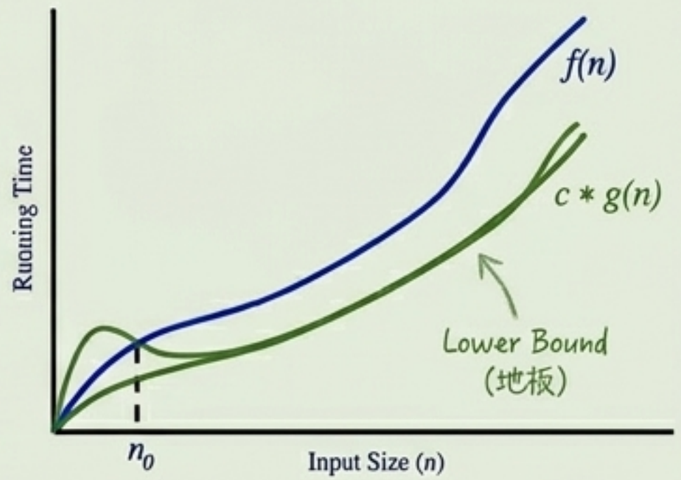
複雜度的三種視角：O, Ω, Θ (Asymptotic Bounds)



1. Big-O Notation (O)

Upper Bound (上限)

設定「天花板」(Worst-case)。

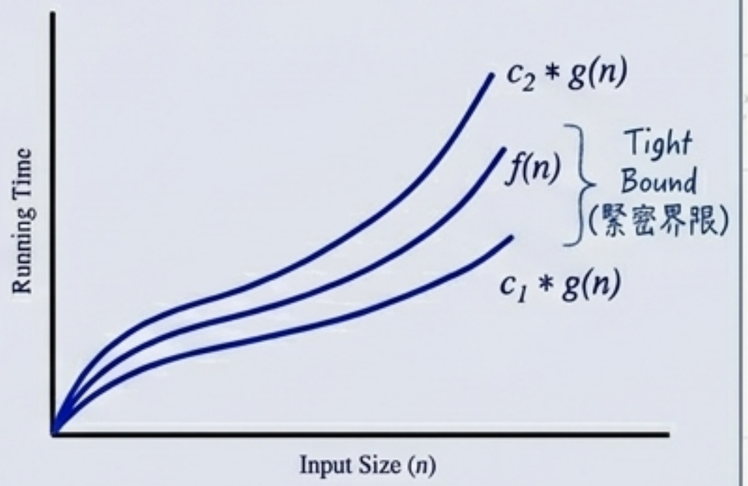


2. Big-Omega Notation (Ω)

Lower Bound (下限)

定義：`f(n) >= c * g(n) for all n >= n_0`

意義：設定「地板」。成長速度「至少 (at least)」一樣快，代表最佳情況最少需要多少時間。



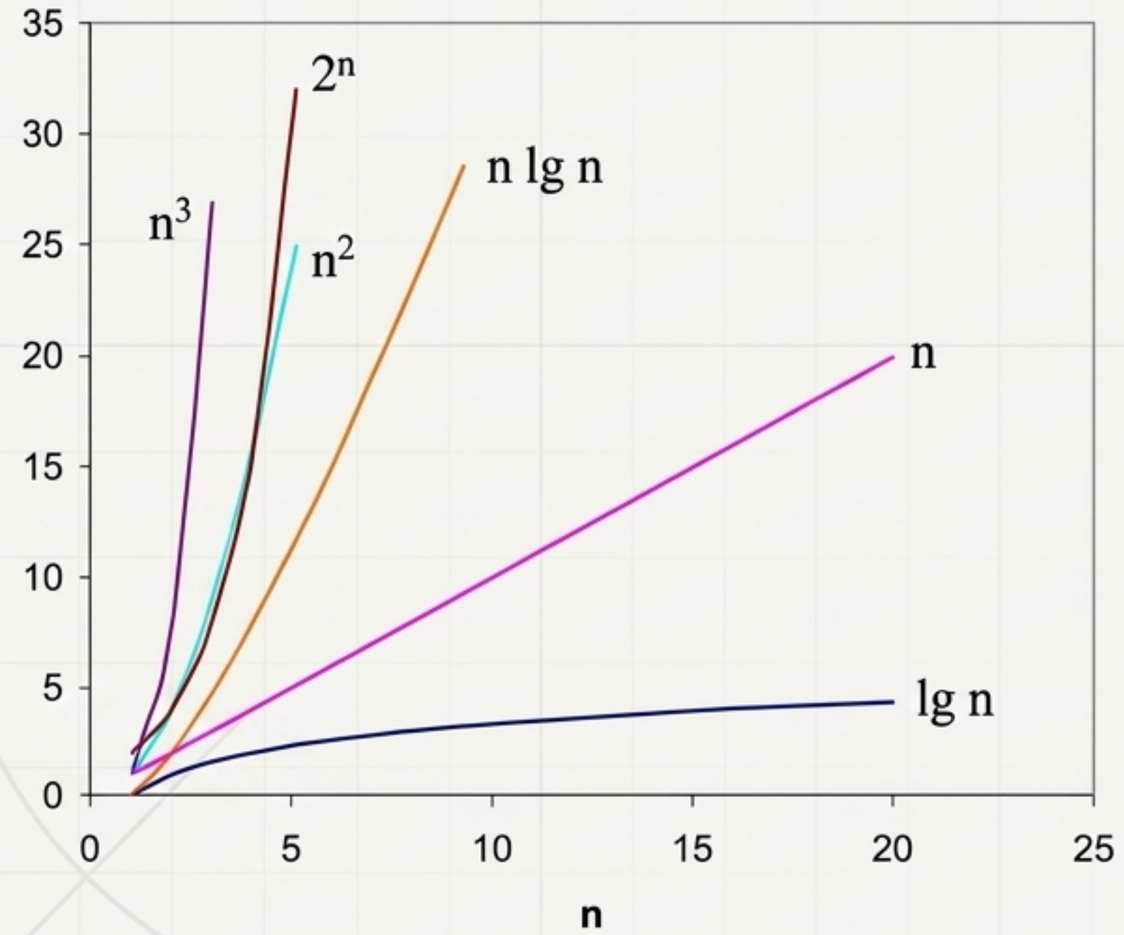
3. Big-Theta Notation (Θ)

Tight Bound (緊密界限)

定義：`c_1 * g(n) <= f(n) <= c_2 * g(n)`

意義：同時被天花板和地板夾住。代表兩個函數的成長速度完全同級。

複雜度階層圖 (The Hierarchy of Growth Rates)



The Ladder (背誦此階層，一眼看出演算法優劣)：

[Green Zone - Excellent]

- O(1) : Constant (常數級) - 執行時間不變 (如：讀取 A[0])
- O(log n) : Logarithmic (對數級) - 資料加倍，時間只加一點

[Yellow Zone - Good]

- O(n) : Linear (線性級) - 單層迴圈
- O(n log n) : Linearithmic - 高階排序法 (如 Merge/Quick Sort)

[Red Zone - Poor]

- $O(n^2)$: Quadratic (平方級) - 雙層巢狀迴圈 (如 Bubble Sort)
- $O(2^n)$ / O(n!) : Exponential (指數級) - 電腦殺手

實戰心法：Big-O 的兩大拇指法則 (Rules of Thumb)

在實務與考試中，我們不需要每次都畫圖找 c。記住這兩個法則：

Rule 1: Drop the constants. (捨棄所有常數)

不管常數是相加還是相乘，全部劃掉。

$$O(2n + 3) \rightarrow O(n)$$

Rule 2: Select the 'most significant' term. (只保留最高階項)

在多項式中，只留下成長最快的主導項 (Dominant term)。

$$20n^3 + 10n \log n + 5 \rightarrow O(n^3)$$

Professor's Hint: 任何多項式 (Polynomial) $a_k n^k + \ldots + a_0$ 的複雜度永遠是 $O(n^k)$。

實戰演練：單層迴圈分析 (Linear Single Loops)

Example A (from Past Paper 16/17):

```
for (int i=2; i<=n; i+=2)
```

- 分析: i 每次加 2，執行次數是 n/2 次。
- 套用 Rule 1: 捨棄常數 1/2，複雜度為 O(n)。

Example B (from Past Paper 22/23):

```
for (int i=n+1000; i>=1; i-=10)
```

- 分析: 執行次數大約是 (n+1000)/10 = n/10 + 100 次。
- 套用 Rule 1: 捨棄常數 1/10 和 100，結果依然是 O(n)。

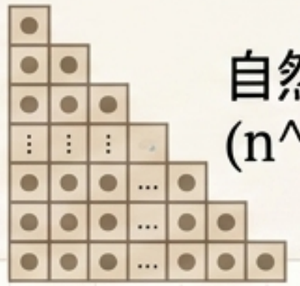
實戰演練：巢狀迴圈分析 (Quadratic Nested Loops)

Example C (from Past Paper 16/17):

```
for (int i=1; i<=n; i++)
   for (int j=1; j<=i; j++)
```

教授的 Feynman 拆解法 (Professor's Breakdown):

1. 外層迴圈 i 執行 n 次。
2. 內層迴圈 j 的執行次數 依賴於 i (1次, 2次, 3次... 直到 n 次)。
3. 總執行次數 (Arithmetic Series) $= 1 + 2 + 3 + 3 + \ldots \cdots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \cancel{\frac{1}{2}n}$
4. 套用 Rule 2: 保留最高階, 捨棄低階項與常數 -> **O(n^2)**。



自然形成半個正方形 (n^2 的一半)

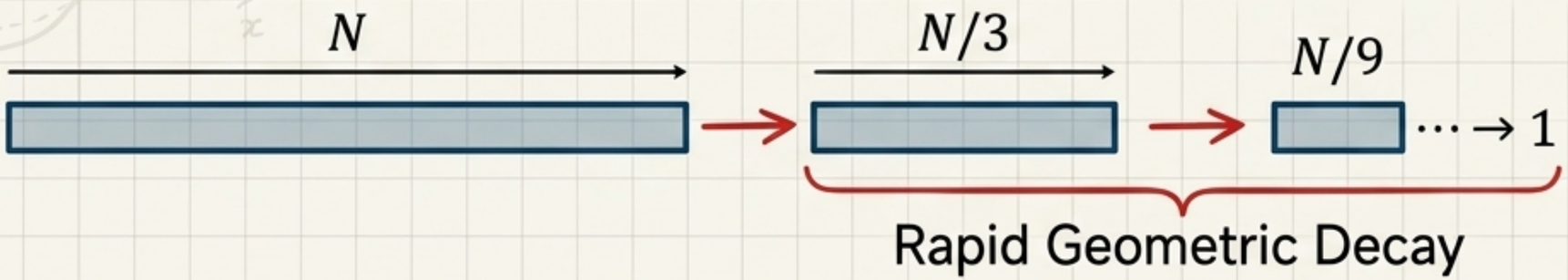
實戰演練：對數級迴圈 (Logarithmic Loops)

Context: 看到迴圈不要直接猜 $O(n)$，注意 乘法與除法 陷阱！

Example D (from Past Paper 22/23):

```
for (int i=1; i<=n; i*=3)
```

- **分析:** i 的變化是 $1, 3, 9, 27, \ldots, 3^k$。
- 當 $3^k = n$ 時，迴圈結束。解方程式得到 $k = \log_3 n$。
- 捨棄底數常數 $\rightarrow O(\log n)$。



Example E (from Past Paper 17/18):

```
while (i<=n) { sum += i; i *= 4; }
```

- **分析 :** 同理，每次乘以 4。執行次數為 $\log_4 n \rightarrow O(\log n)$。

課程總結 (Mastery Checklist)

- [x] 我能解釋 Time Complexity (時間) 和 Space Complexity (記憶體) 的差異。
- [x] 我明白為何 Analytical Approach (分析法) 優於 Experimental (實驗法)。
- [x] 我能分辨 Best-case 與 Worst-case，並知道為何業界最重視 Worst-case。
- [x] 核心理論: 我能解釋 Big-O Notation 的定義 (尋找漸進上限 c * g(n))。
- [x] 我熟悉常見的時間複雜度階層 (從 O(1) 到 O(2^n) 排序)。
- [x] 實戰能力: 給我一段 Java 程式碼，我能透過 Rule 1 & 2 迅速推導出它的 Big-O 時間複雜度！

Professor's Closing Thought: Complexity analysis is not just math; it is the fundamental language used to evaluate the elegance and efficiency of your code!
(複雜度分析不只是數學，它是評估程式碼優雅與效率的基礎語言！)