課程名稱：演算法設計與分析

Design and Analysis of Algorithms

Outline of This Lecture
1) Insertion Sort
2) Growth of Functions

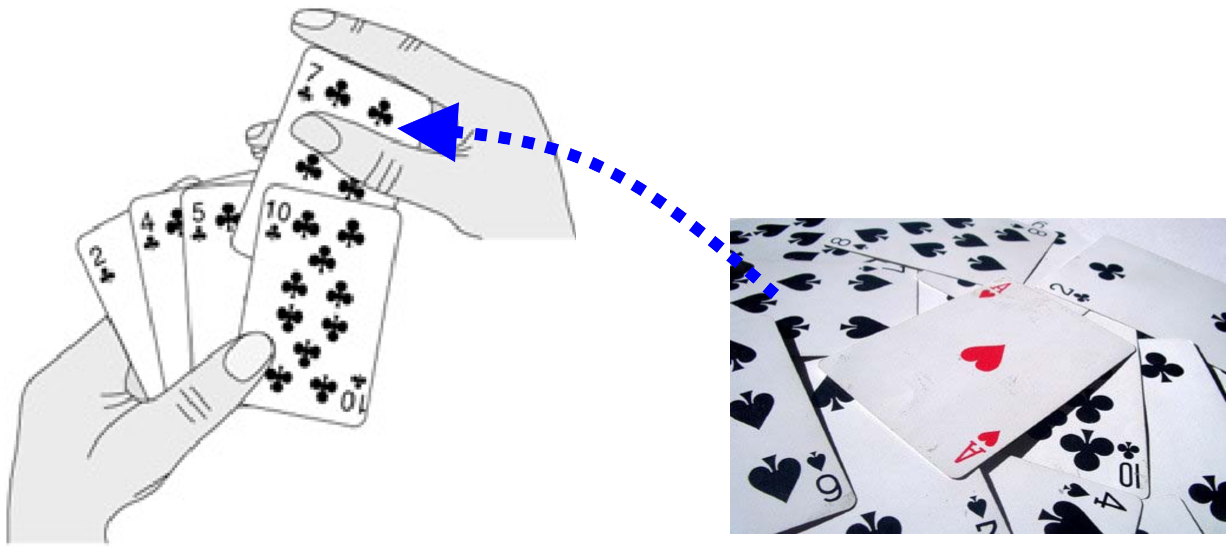
Instructor: 周孜燦 助理教授
國立中山大學電機系
Email: ztchou@ee.nsysu.edu.tw

Let Us Consider the Sorting Problem

Sorting problem

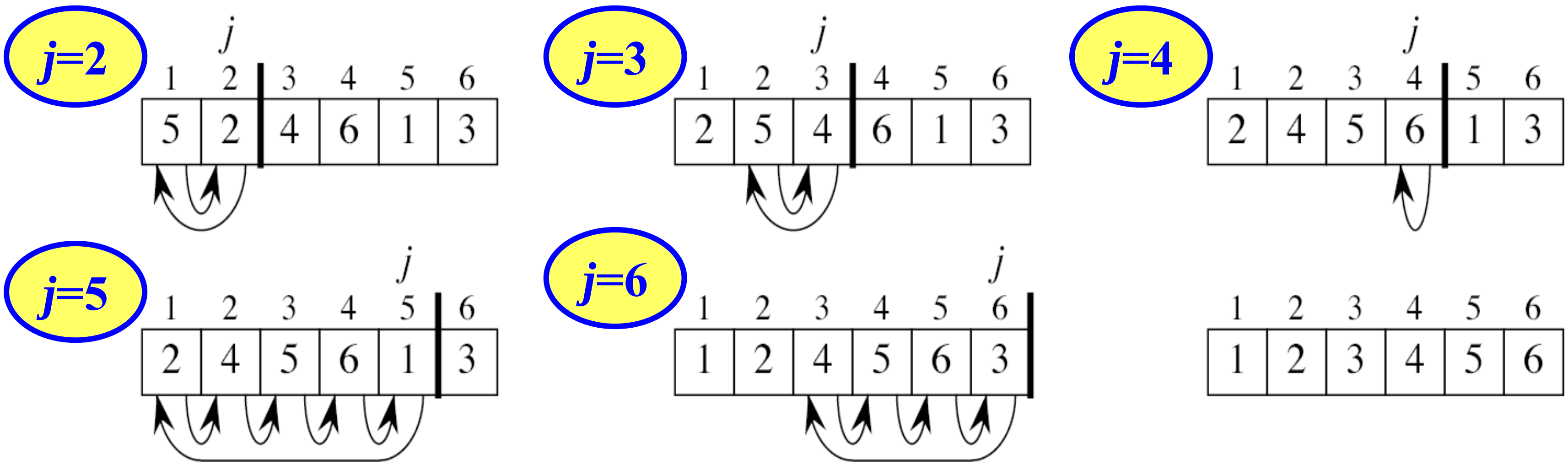
– Input: A sequence of n numbers $A = \langle a_1,\ a_2, \cdots, a_n \rangle$

– Output: A permutation $A' = \langle a'_1,\ a'_2, \cdots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.



方法之一：每次一拿到新的撲克牌，就安插到「正確」的位置

Insertion Sort



INSERTION-SORT(A)

for $j \leftarrow 2$ **to** n

  do $key \leftarrow A[j]$

    ▷ Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.

    $i \leftarrow j - 1$

    while $i > 0$ and $A[i] > key$

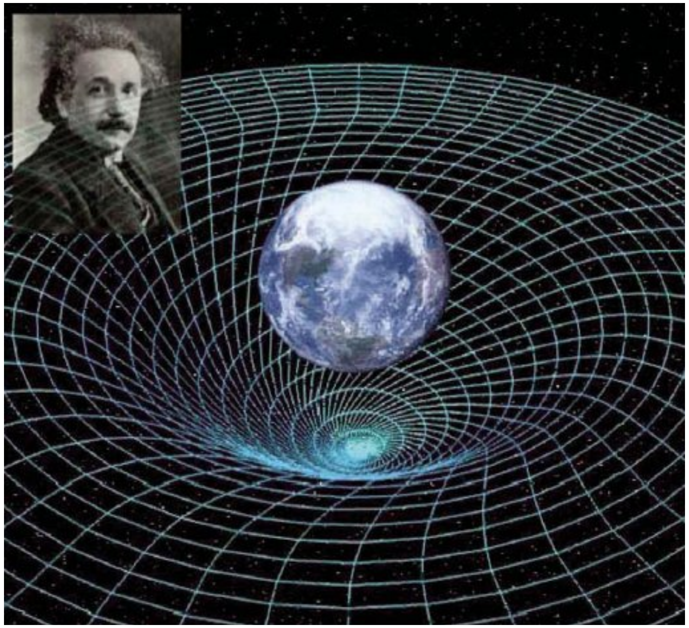
      do $A[i+1] \leftarrow A[i]$

        $i \leftarrow i - 1$

    $A[i+1] \leftarrow key$

How to Measure the Performance of an Algorithm？

- Time
- Space（memory）



現在 memory 愈來愈便宜，所以我們最 care 的還是時間

但如何分析一個 algorithm 所花費的時間？在不同的 CPU，不同的程式語言、不同的作業系統之下，執行時間都不一樣，怎麼辦？

爲了要公平的比較，我們需要建立一個 machine independent 的 model。這個 model 必須夠抽象，以便我們能夠用數學來證明。這個 model 必須夠具體，以免偏離 real-world 太遠。在這兒我們採用 random access model（RAM）。

Random Access Model (RAM)

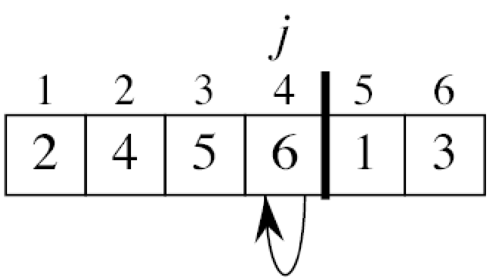
We analyze algorithms based on the RAM model.

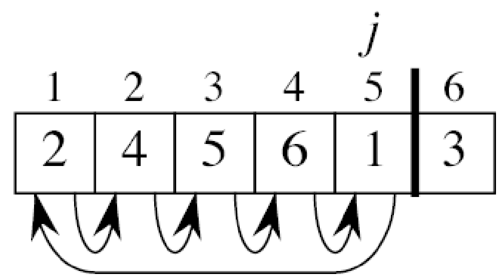
- All basic instructions (e.g., $+$, $-$, $\times$, $\div$, $:=$) take the same time.
  - This implies that "sort" is not a basic instruction.
- All memory accesses cost equally.
  - We don't distinguish cache or virtual memory.
- No concurrent operations.
  - Instructions should be executed step by step.
- The size of each basic type (e.g., **`int/double`**) variable is roughly the same.
  - This implies that the size of **`array[n]`** is n times that of a basic type variable.

Running Time Analysis of the Insertion Sort

	cost	times
INSERTION-SORT(A)		
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
▷ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

***t_j* 表示當 for loop 在第 *j* 個迴圈時，while loop 的執行次數**





best case：***A*[1..*j*－1] 已經排序完畢**
發覺 ***A*[*i*] > key** 這個條件不成立
→ 只比 **1** 次

worst case： ***A*[*j*] 比 *A*[1..*j*－1] 都小**
***i* 的值遞減**，直到 ***i* = 0** 爲止
→ 共執行 ***j*** 次

Best Case Running Time

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1 \mathinner{.\,.} j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Let $T(n)$ = running time of INSERTION-SORT.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1).$$

所以在 best case（$t_j = 1$），insertion sort 的執行時間 $T(n)$ 爲（$c_1 + c_2 + c_4 + c_5 + c_8$）$n$ -（$c_2 + c_4 + c_5 + c_8$）$= an + b$

Worst Case Running Time

Let $T(n)$ = running time of INSERTION-SORT.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

在 worst case 時，$t_j = j$。

因為 $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = 2 + 3 + \cdots + n = \frac{(n-1)(n+2)}{2}$

$$\sum_{j=2}^{n} (t_j - 1) = 1 + 2 + 3 \cdots (n-1) = \frac{n(n-1)}{2}$$

所以此時 insertion sort 的執行時間 $T(n)$ 為

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$$

$$= a \cdot n^2 + b \cdot n + c$$

Worst Case Analysis

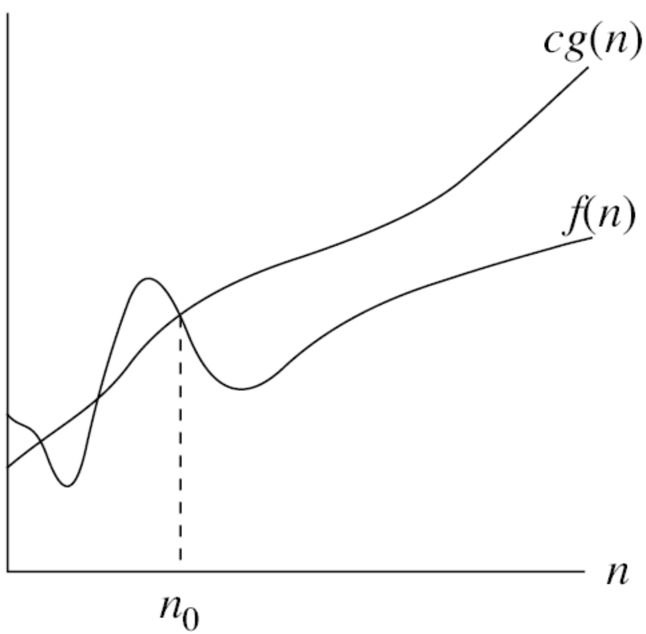
- We usually focus on the worst-case running time. The reasons are as follows:
  - For real-time applications, we need to know what running time the algorithm can guarantee under any circumstances.
  - The worst case occurs quite often such as query an absent item in a database.
  - The average case is often as bad as the worst case. For example, on average, we need to check half of $A[1..j]$. In this case, $t_j = j/2$ and $T(n)$ will be still the form of $an^2 + bn + c$, just like the worst case running time.
- 以後投影片裡頭提到 running time，若無特別聲明，就是指 worst case running time

Upper Bound Notation：Big-O

雖然我們知道 insertion sort 的 worst case running time 爲

$$T(n) = \left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2 + \left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n - (c_2+c_4+c_5+c_8)$$

但這太囉唆了 → 有時候我們只想知道 insertion sort 執行時間的大致趨勢。



A function $f(n)$ is a member of the set $O(g(n))$ if there exists positive constants c and n_0 such that $f(n) \le c \times g(n)$ for all $n \ge n_0$.

例如：$T(n) = 3n^2 + 4n + 3$，我們說

$T(n) = O(n^2)$ 註：用「等號」表示「屬於」，帶給我們符號操作上的方便（以後會看到）。

因爲 當 $c = 4$，$n_0 = 5$ 時，$T(n) \le 4n^2$ 。

意思是說，$T(n)$ 的成長速率不會超過 n^2 這種規模的程度

Order of Growth

- **Highest-order term** is what we counts
  - As the input size grows larger, the running time will be dominated by the highest order term.
- A polynomial of degree k is $O(n^k)$

多項式 $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots. + a_1 n^1 + a_0$

令 $b_i = |a_i|$ 且 $c = b_k + b_{k-1} + \ldots. + b_1 + b_0$

那麼 $f(n) \le b_k n^k + b_{k-1} n^{k-1} + \ldots. + b_1 n^1 + b_0$

$$= n^k \sum_{i=0}^{k} b_i \frac{n^i}{n^k} \le n^k \sum_{i=0}^{k} b_i \le cn^k$$

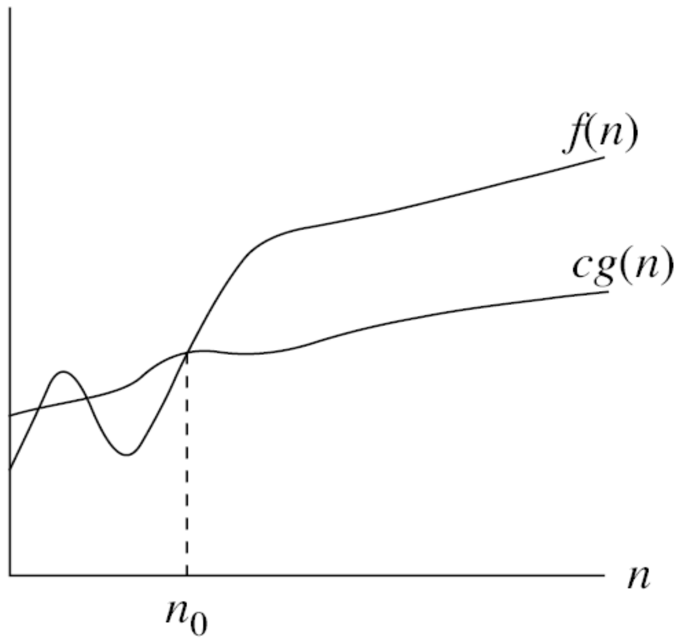
所以 $f(n) = O(n^k)$

當然，$f(n) \le cn^k \le cn^{k+1}$，所以也可以寫成 $f(n) = O(n^{k+1})$

但是這樣的 upper bound 不夠 tight

Lower Bound Notation：Big-Omega

- A function $f(n)$ is a member of the set $\Omega(g(n))$ if there exists positive constants c and n_0 such that $0 \le c \times g(n) \le f(n)$ for all $n \ge n_0$.
- Hence the running time of insertion sort is $\Omega(n)$.



例如：$T(n) = 3n^2 + 4n + 3$，我們說

$T(n) = \Omega(n^2)$　註：用「等號」表示「屬於」

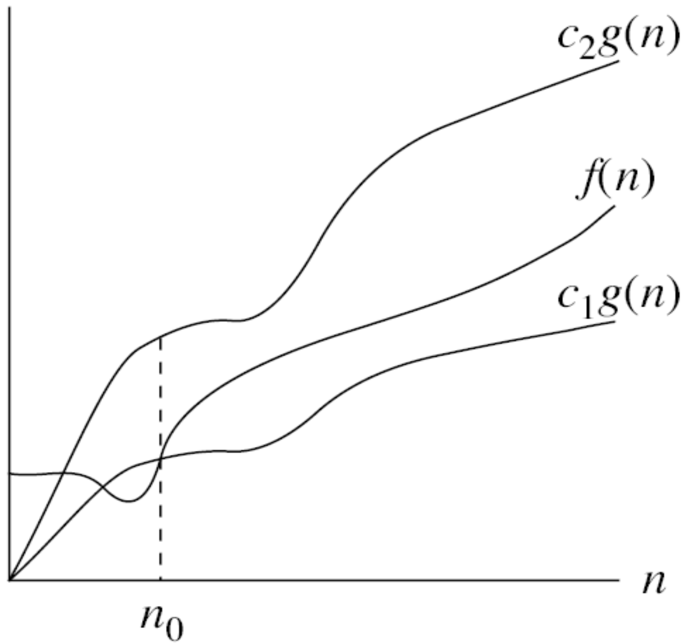
因為 當 $c = 3$，$n_0 = 1$ 時，$T(n) \ge 3n^2$ 。

意思是說，$T(n)$ 的成長速率至少大於 n^2

這種規模的程度

Bounded Range Notation：Big-Theta

- A function $f(n)$ is a member of the set $\Theta(g(n))$ if there exists positive constants c_1, c_2, and n_0 such that $c_1\, g(n) \le f(n) \le c_2\, g(n)$ for all $n \ge n_0$.
- $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$



例如：$T(n) = 3n^2 + 4n + 3$，我們說

$T(n) = \Theta(n^2)$　註：用「等號」表示「屬於」

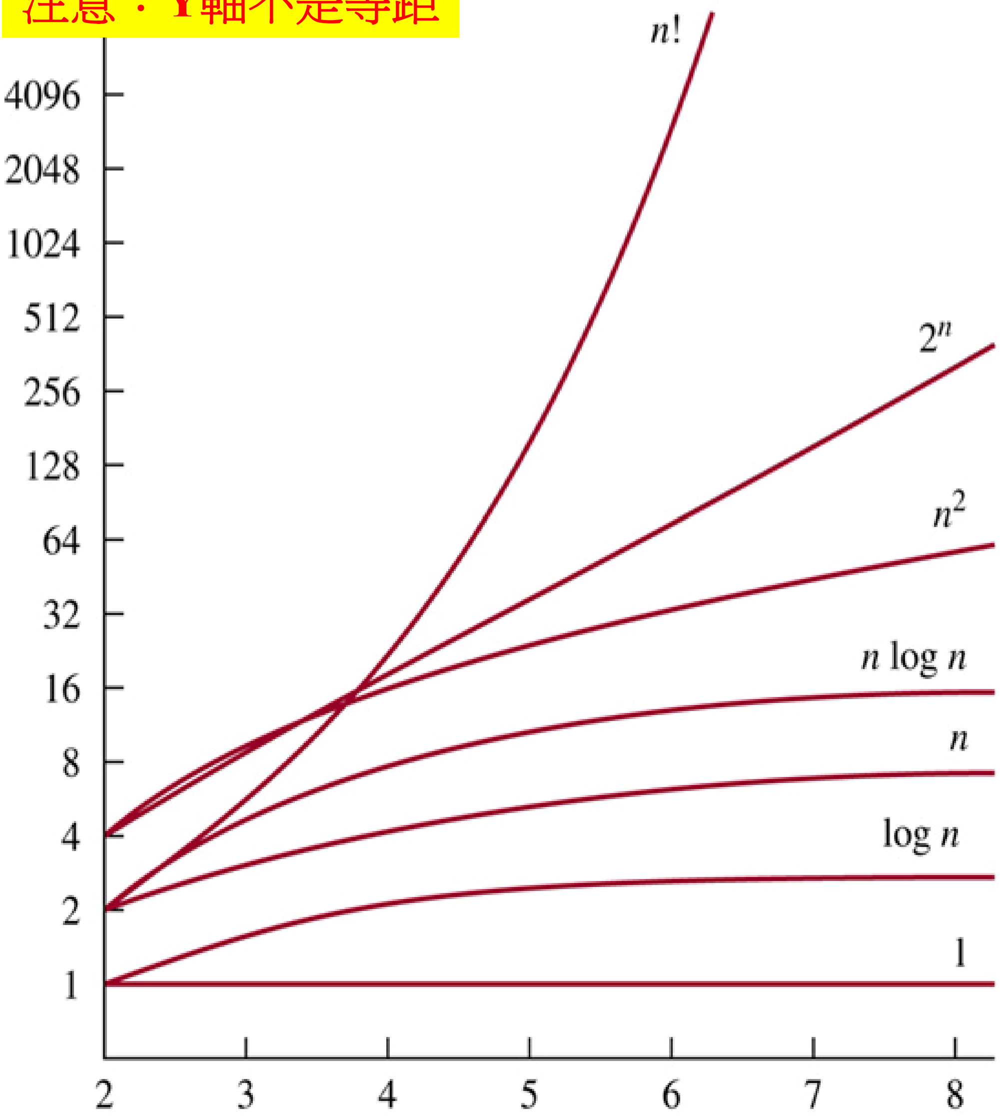
因爲 當 $c_1 = 3$， $c_2 = 4$，$n_0 = 5$ 時，

$3n^2 \le T(n) \le 4n^2$ 。

意思是說，$T(n)$ 的成長速率大約就是 n^2 這種規模的程度

Common Used Big-O Functions

注意：Y軸不是等距



Complexity
$\Theta(1)$
$\Theta(\log n)$
$\Theta(n)$
$\Theta(n \log n)$
$\Theta(n^b)$
$\Theta(b^n)$, where $b > 1$
$\Theta(n!)$

神秘的 $O(1)$？

Definition $O(g(n))$：A function $f(n) = O(g(n))$ if there exists positive constants c and n_0 such that $f(n) \le c \times g(n)$ for all $n \ge n_0$.

Definition $O(1)$：A function $f(n) = O(1)$ if there exists positive constants c and n_0 such that $f(n) \le c$ for all $n \ge n_0$.

Example 1：Function $f(n) = 567 = O(1)$ since we can set $c = 567$ and $n_0 = 1$ such that $f(n) \le 567$ for all $n \ge n_0 = 1$.

Example 2：給定 **input size n**，是否存在一個問題能在 **$O(1)$** 時間解決？ **Yes：**給定 **$n-1$** 個整數及 **n** 的值，請問 **n** 是奇數？還是偶數？

Time Complexity to Sort `array[10]`？

我們知道 insertion sort 的執行時間

$$T(n)=\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n-\left(c_2+c_4+c_5+c_8\right)$$

$$=an^2+bn+c=O(n^2)$$

假設 $a=5$ 、$b=6$ 、$c=7$ ，那麼當 $n=10$ ，$T(10)=567$

剛才我們知道：$f(n)=567=O(1)$，又由於 $T(10)=f(n)=567$，所以 $T(10)=O(1)$

一公斤的棉花

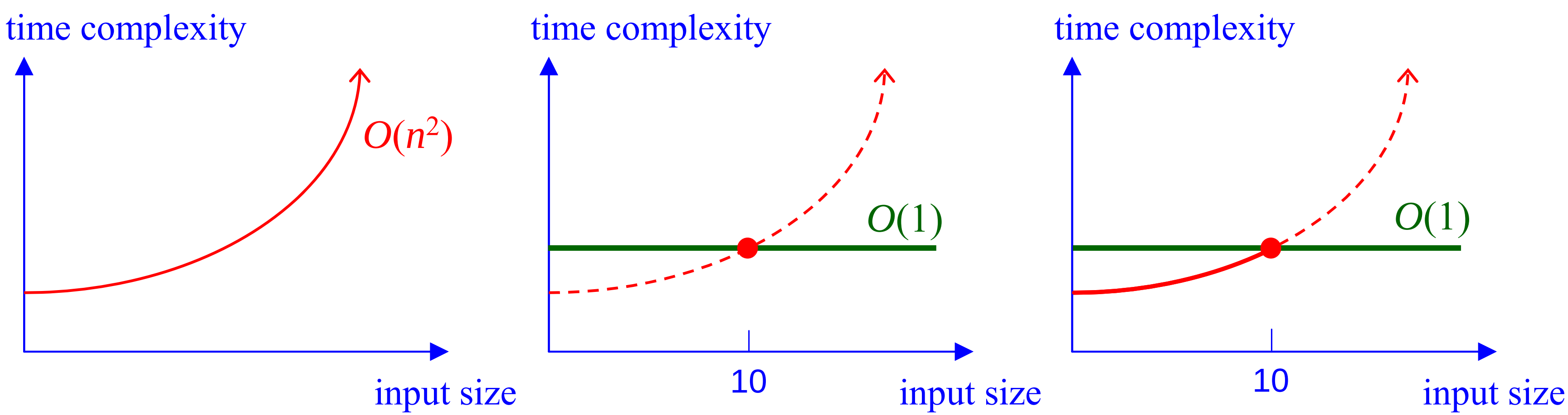
$f(n)=567$

一公斤的鐵

$T(10)=567$

一公斤的棉花和一公斤的鐵哪個比較重？

Time Complexity for Bounded Input Size Problem



特定的 input size

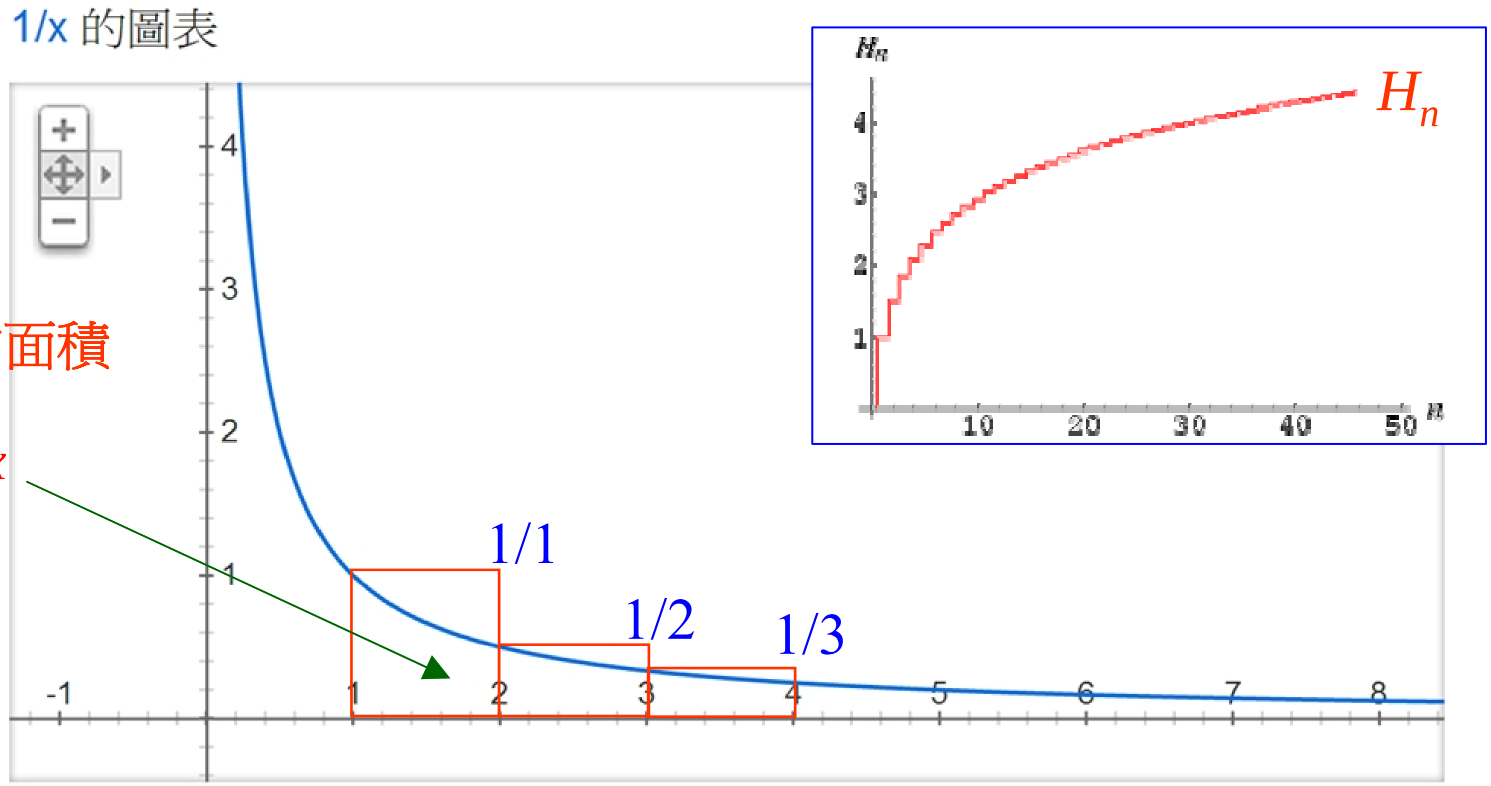
bounded input size

結論：對於特定或 **bounded** 的 **input size**，任何演算法的 **time complexity** 都是 $O(1)$。

Harmonic Number H_n Is Unbounded

觀察曲線下的面積

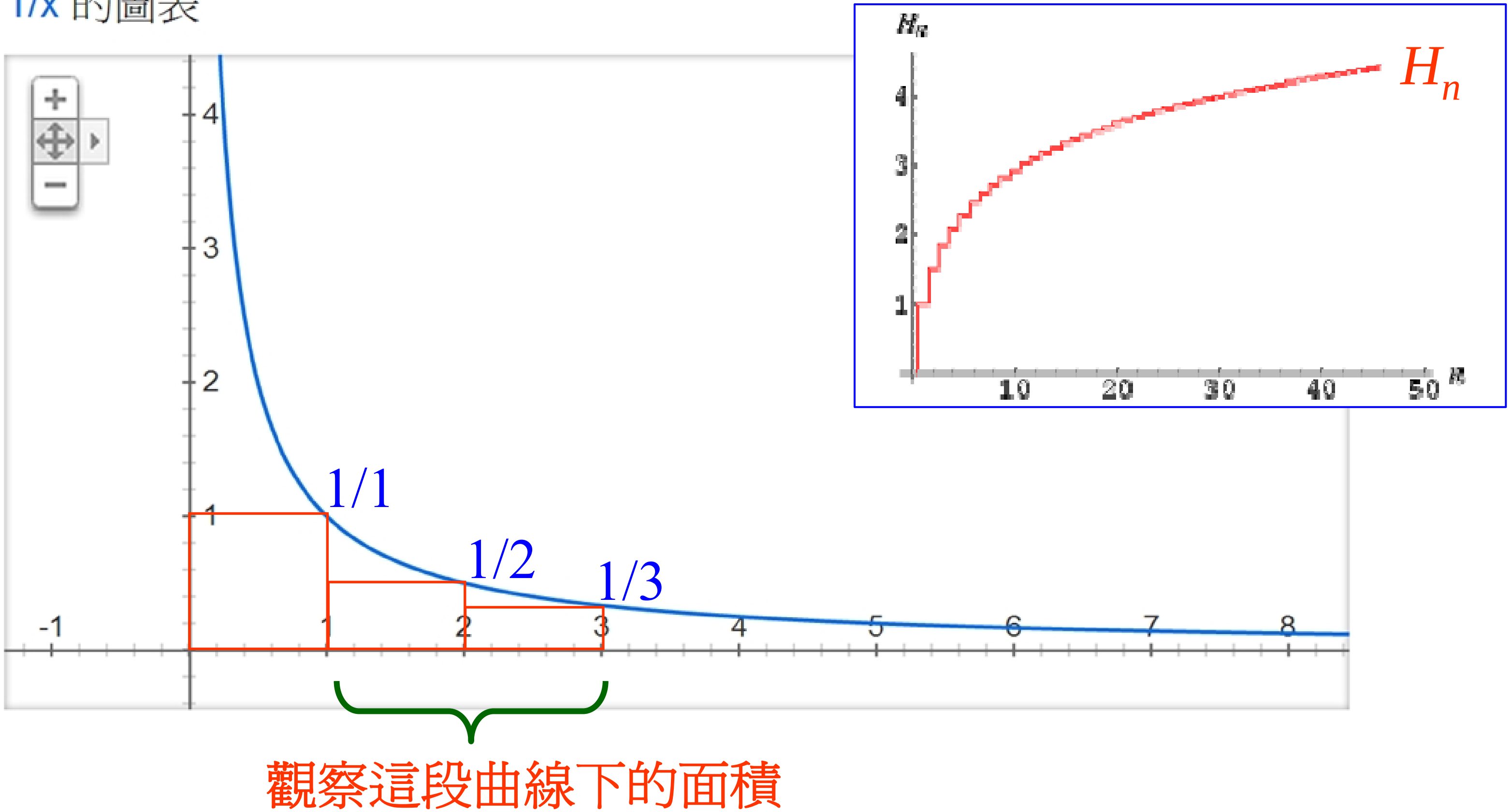
$$\frac{1}{1}+\frac{1}{2} \geq \int_1^3 \frac{1}{x}\,dx$$



$$H_n = \frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n} \geq \int_1^{n+1} \frac{1}{x}\,dx = \ln(n+1)$$

所以 $\lim\limits_{n\to\infty} H_n \geq \lim\limits_{n\to\infty} \ln(n+1) = \infty$

Time Complexity of Harmonic Number H_n



觀察上圖，我們有$\frac{1}{1}+\frac{1}{2}+\frac{1}{3} \le 1+\int_1^3 \frac{1}{x}dx$

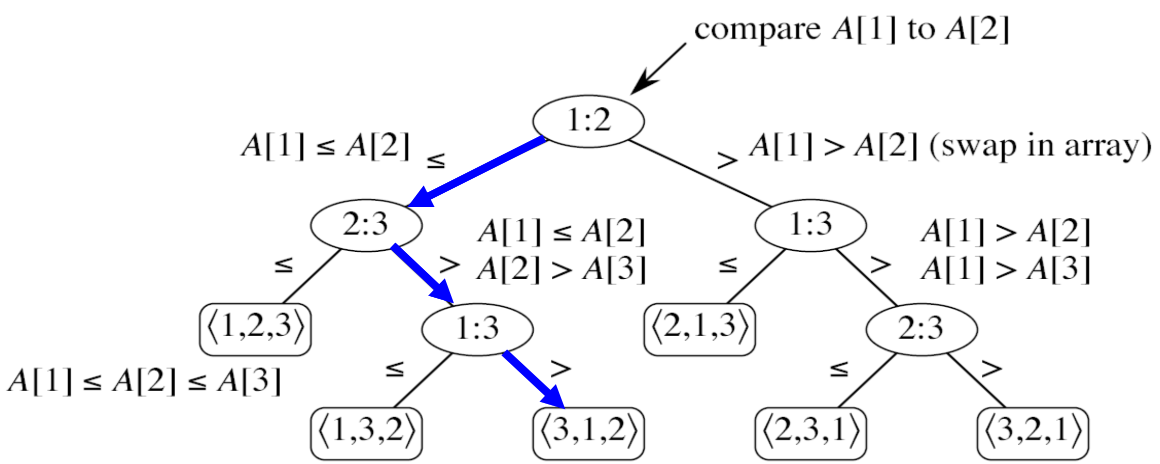
$$T(n)=H_n=\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n} \le 1+\int_1^n \frac{1}{x}dx = 1+\ln n = O(\ln n)$$

Lower Bound for Sorting（1/3）

現在，我們想知道，insertion sort 到底是不是一個夠好的 sorting algorithm。
一般證明一個 algorithm 是 optimal 的二個步驟：
(1) 先指出 the lower bound of any algorithms to solve this problem.
(2) The worst case running time matches this lower bound.

所以我們必須先知道 sorting 的 **lower bound** （在 **worst case** 的情況之下，最厲害的演算法，到底能多快？）是多少。我們假設 (1) sorting algorithm is based on the "**comparison**" operation. (2) All elements in $A[1..n]$ are **distinct**.

首先，讓我們先看看 sorting 的對象是 $A[1..3]$

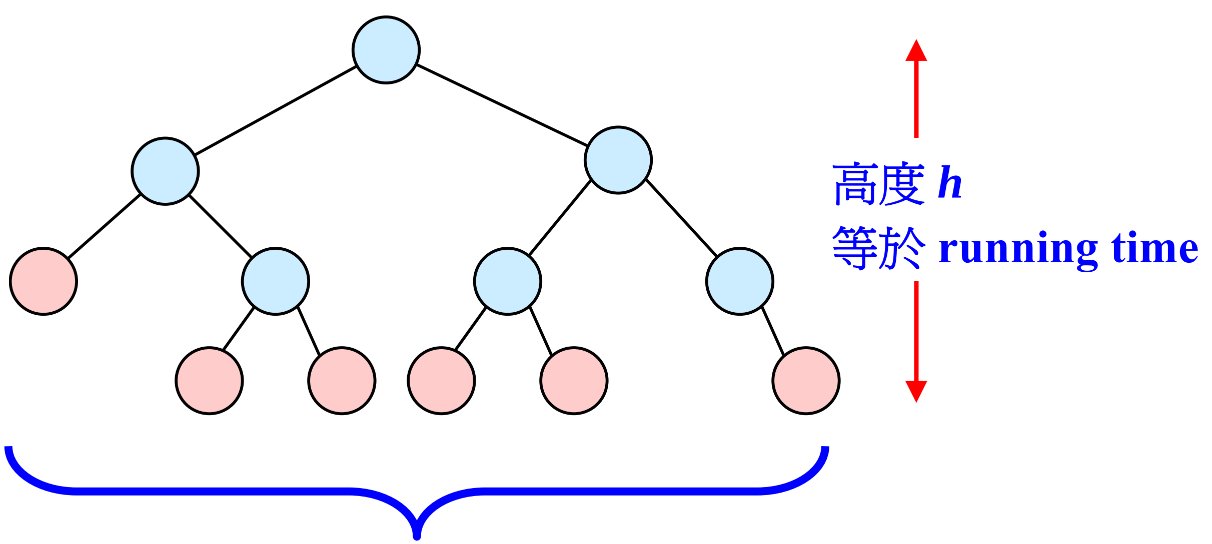


decision binary tree 的高度就是 sorting 所需的步驟

Lower Bound for Sorting（2/3）

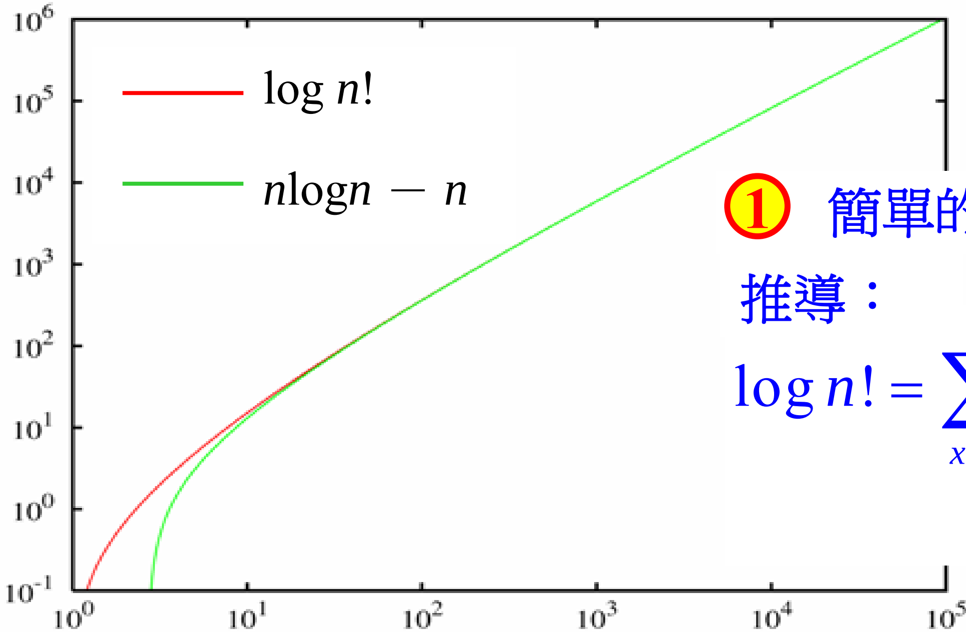
因爲我們要 sort 的對象總共有 n 個 elements，所以這顆 decision binary tree 的 leaves 總共有 $n!$ 個。

對於一個 binary tree 來說，假設高度爲 h ，那麼這顆 tree 的 leaves 最多不超過 2^h 個。但這 2^h 個 leaves 必須涵蓋所有可能的 permutations。所以我們有 $\mathbf{2^h \geq n!}$ 。這個 h 相當於 running time，所以我們可得 $\mathbf{h \geq \log n!}$ 。剩下來的工作就試求出 $\log n!$ 的 order，並用 Ω () 表示。



leaves 個數至多 2^h 個，這些 leaves 必須包含所有的 permutations（共 $n!$ 個）

Lower Bound for Sorting（3/3）



① 簡單的來說，**log *n*! ～ *n* log *n* - *n***

推導：

$$\log n! = \sum_{x=1}^{n} \log x \approx \int_{1}^{n} \log x dx \approx n \log n - n$$

$$= \Omega(n \log n)$$

②

或者查一下 Wiki 百科，我們有 Stirling's approximation： $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

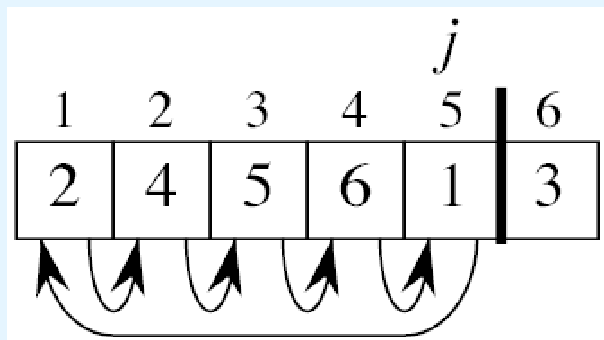
所以

$$h \geq \log n! \approx \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) = \log n^n - n \log e + \frac{1}{2}\log n + \log\sqrt{2\pi} = \Omega(n \log n)$$

→ comparison-based sorting algorithms 的 lower bound 爲 Ω (*n* log *n*)

What's Wrong with Insertion Sort？

咦，insertion sort 的 worst case running time 為 $O(n^2)$，遠大過 sorting 的 lower bound $O(n\log n)$，是不是當初我們有什麼地方沒考慮周詳的？ 仔細觀察一下 worst case



我們發覺即使 $A[1..j-1]$ 已經 sorted 完畢，$A[j]$ 還要一個一個比才能找到正確位置，這會不會太慢？

讓我們考慮 divide-and-conquer（個個擊破）的策略

Divide-and-Conquer：Binary Search

目標：我們想尋找 *A*[1..*n*] 裡頭的一個 element *x*。假設 *A*[1..*n*] 已經 sorted 完畢。我們先問問看 *x* 有沒有比**中間**的元素大 或 相等？（例如：***x* =3**）

1	2	3	4	5	6	7	**8**	9	10	11	12	13	14	15
1	3	5	7	9	11	13	**15**	17	19	21	23	25	27	29

比 *A*[8] 小，所以我們只考慮 *A*[1..7] 的部分。言下之意，*x* 不用再跟 *A*[9..15] 一個一個去比了（**當場省掉一半的功夫**）

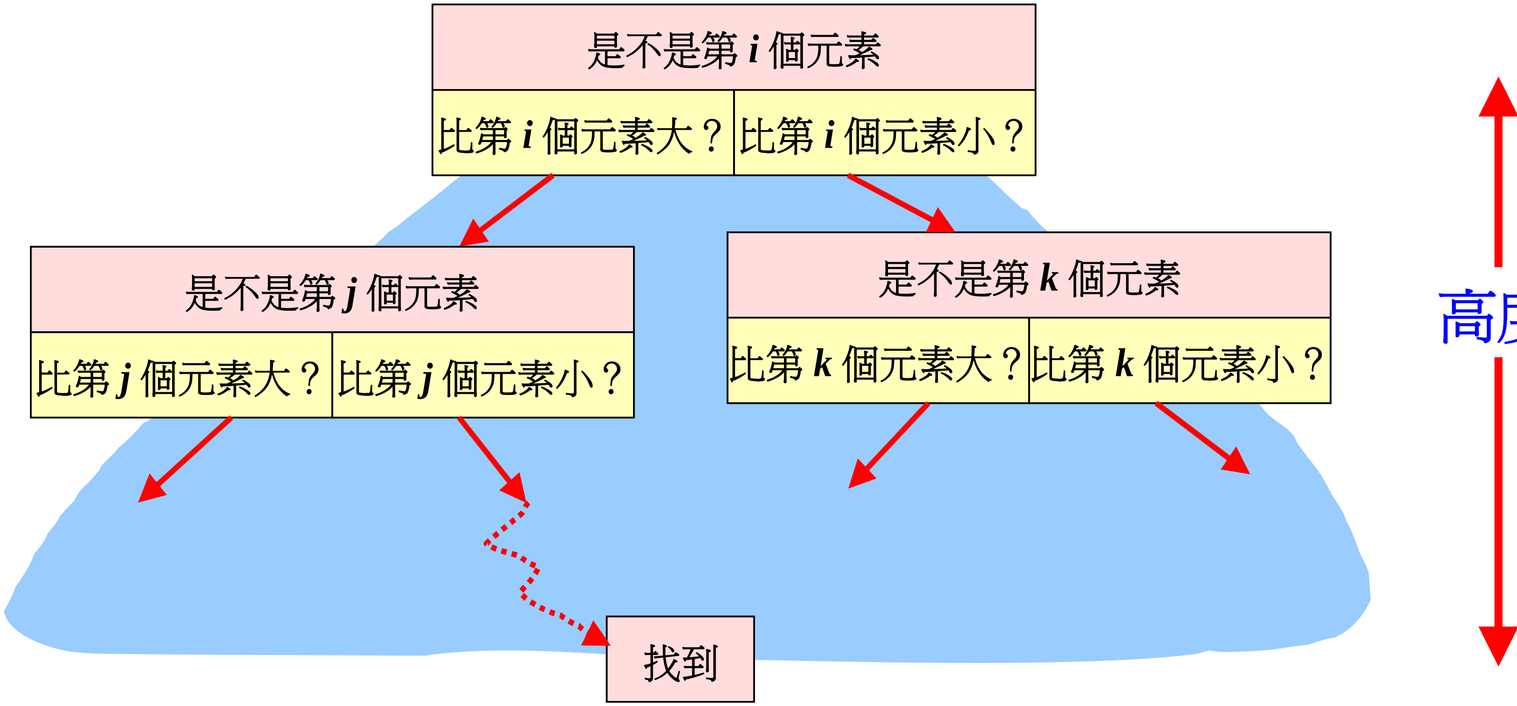
1	2	3	**4**	5	6	7
1	3	5	**7**	9	11	13

比 *A*[4] 小，所以我們只考慮 A[1..3] 的部分。

1	**2**	3
1	**3**	5

剛好等於 *A*[2]，所以我們找到了！ 3 在 *A*[2] 的位置

Is Binary Search Optimal for Sorted Array？



一個高度爲 h 的 comparison tree 總共包含 $2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$ 個 nodes

所有的陣列元素必需落在這些 nodes 裡頭

所以 $n \le 2^{h+1} - 1 \quad \Rightarrow$ search 的執行時間 $h \ge \log(n+1) - 1 = O(\log n)$

Binary search 的執行時間爲 $O(\log n)$ ，所以是 optimal

Is Binary Search Better Than Sequential Search？

	執行時間	運作前提	是否爲 optimal？
sequential search	$O(n)$	陣列未排序	yes
binary search	$O(\log n)$	陣列已排序	yes

給定 unsorted array，要執行 binary search，必需先將陣列排序完畢，所以此時 binary search 的執行時間爲 $O(n\log n) + O(\log n) = O(n\log n)$，有比 sequential search 好嗎？

前提一樣才能比較。Binary search 比 sequential search 好？
前提不一樣，如何比較？ 就像香蕉比蘋果，亂比一通！

Recursive Approach to Find the Right Position

```
FindPosition(A, x, low, high) {
  mid = floor((low + high)/2);
  if (( A[ mid ] <= a) && ( a <= A[ mid + 1 ] ))
     return mid + 1;
  else if ( a < A[ mid ] )
     FindPosition(A, x, low, mid);
  else
     FindPosition(A, x, mid + 1, high);
}
```

現在我們來看看，假設我們想安插 x 到 $A[low..high]$ 裡頭，應該安插在什麼位置才正確。假設想安插 $\boldsymbol{x = 4}$ 到 $A[1..15]$

1	2	3	4	5	6	7	**8**	9	10	11	12	13	14	15
1	3	5	7	9	11	13	**15**	17	19	21	23	25	27	29

落在這兒嗎？

由於 4 並沒有落在 $A[8]$ 和 $A[9]$ 之間，所以呼叫 FindPosition(A, 4, 1, 8)

1	2	3	**4**	5	6	7	8
1	3	5	**7**	9	11	13	15

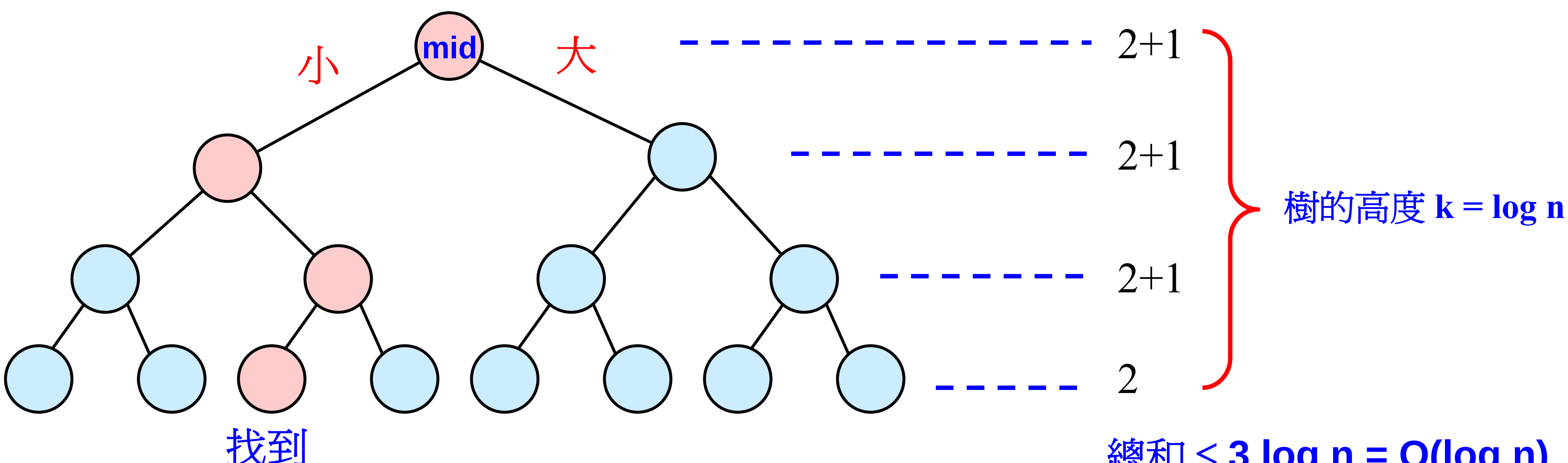
落在這兒嗎？

1	**2**	3	4
1	**3**	5	7

確實落在這兒（**return 3**）

Performance Analysis by Recursion-Tree Method

```
FindPosition(A, x, low, high) {
  mid = floor((low + high)/2);
  if (( A[ mid ] <= a) && ( a <= A[ mid + 1 ] ))
     return mid + 1;
  else if ( a < A[ mid ] )
     FindPosition(A, x, low, mid);
  else
     FindPosition(A, x, mid + 1, high);
}
```



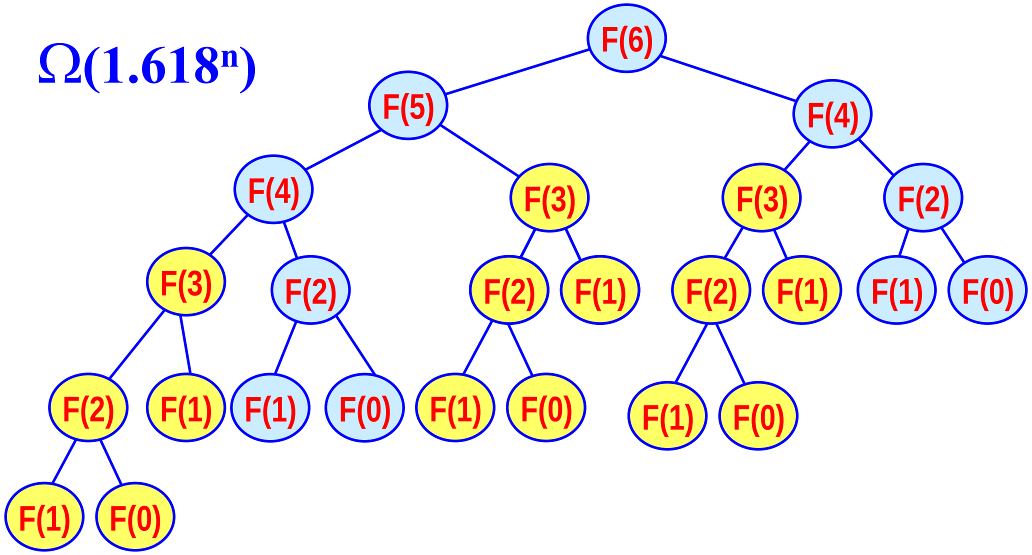
總和 ≤ 3 log n = O(log n)

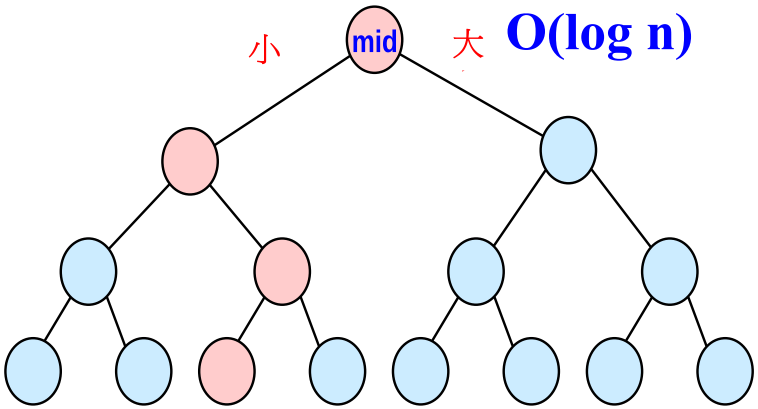
What Causes Difference in Running Time？

```
int F(n) {
  if ((n==0) or (n==1))
    return 1;
  else
    return F(n-1) + F(n-2);
}
```

```
FindPosition(A, x, low, high) {
  mid = floor((low + high)/2);
  if (( A[ mid ] <= a) && ( a <= A[ mid + 1 ] ))
     return mid + 1;
  else if ( a < A[ mid ] )
     FindPosition(A, x, low, mid);
  else
     FindPosition(A, x, mid + 1, high);
}
```

$\Omega(1.618^n)$





$O(\log n)$

並非 **recursive programs** 的 **performance** 都很差。左邊是因爲發生「重複計算」的關係，才導致 **performance** 嚴重下降。

Binary Insertion Sort

現在我們假設使用 FindPosition（使用 binary search）來取代 insertion sort 裡頭一個一個比較的動作，這樣的 algorithm 稱之爲 binary insertion sort（BINSORT）。那麼 BINSORT 總共會比較多少次？

最外層的 for loop 會執行 $n-2$ 次，裡頭的 while loop 用 FindPosition 取代，花 $O(\log n)$ 次比較，所以總比較次數爲

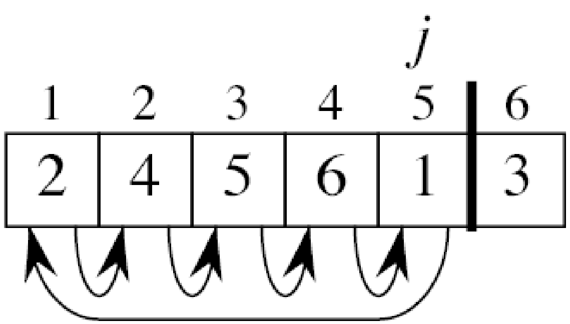
$(n-2) \times O(\log n) = n \times f(n)$ **// 註：$f(n)$ 爲 $O(\log n)$ 的函數**

$= O(n \log n)$

所以就「**比較次數**」而言，binary insertion sort 確實是 optimal sorting。 但是等一下！ 「比較次數」幹嘛要特別刮號起來？

The Running Time of BINSORT Is Still $O(n^2)$

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] \leftarrow key$	c_8	$n-1$



以左圖來說，雖然 *A*[*j*] 可以在 *O*(log *n*) 之內找到正確的位置，但是要將 *A*[2..*j*] **搬移**到正確位置卻需 *O*(*n*) 的時間。言下之意，由於 insertion sort 牽涉到 **move** 的動作，*A*[2..*j*] 裡頭的 element 必須**一個一個的搬動**，導致幾乎沒有改善的空間。所以總結的來說，**insertion sort 還是需要 *O*(n^2) 的時間。**

In the next lecture, we will introduce two optimal sorting algorithms whose running time are $O(n \log n)$.

自我評量

1. 令$T(n)=O(f(n))$ ，請針對下列式子求出$f(n)$ 。
註：$f(n)$ 必須是 simplest 及 tighest 的形式

(a) $T(n)=\frac{1}{1}+\frac{1}{2}+\cdots+\frac{1}{n}=\sum_{i=1}^{n}\frac{1}{i}$

(b) $T(n)=\log 1+\log 2+\cdots+\log n=\sum_{i=1}^{n}\log i$

(c) $T(n)=2T\left(\left\lfloor\sqrt{n}\right\rfloor\right)+\log n$

2. 請問 comparison-based sorting algorithms 的 lower bound 是多少？
試證明之。

3. 請問 comparison-based searching algorithms 的 lower bound 是多少？
試證明之。