課程名稱：演算法設計與分析

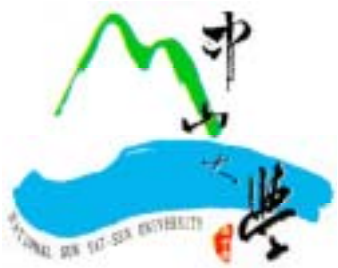
Design and Analysis of Algorithms

Outline of This Lecture:

Divide and Conquer

1) Merge Sort and Quick Sort

2) Linear Time Selection



授課教師: 周孜燦

國立中山大學電機系

聯絡方式: ztchou@ee.nsysu.edu.tw

Divide-and-Conquer Approach

- **Divide** the problem into a number of independent subproblems.
  注意：如果可以將一個問題切成很多 subproblems，但是這些 subproblems 之間若有重複，就必須採用 dynamic programming 的策略，將計算的結果存起來，以避免重複計算。

- **Conquer** the subproblems by solving them recursively.
  - **Base case**： If the subproblems are small enough, just solve them by brute force.（一般來說，當我們使用 recursive 的方式來設計 divide-and-conquer algorithm，base case 可確保程式有終止的時候）

- **Combine** the solutions of sub-problems to deliver the solution of the original problem.

The Sorting Problem

Sorting problem

- Input: A sequence of n numbers $\boldsymbol{A} = \langle \boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n \rangle$
- Output: A permutation of the input sequence

$$\boldsymbol{A'} = \langle \boldsymbol{a'}_1, \boldsymbol{a'}_2, \ldots, \boldsymbol{a'}_n \rangle \text{ such that } \boldsymbol{a'}_1 \leq \boldsymbol{a'}_2 \leq \ldots \leq \boldsymbol{a'}_n.$$

How John von Neumann Solves Sorting Problem ?

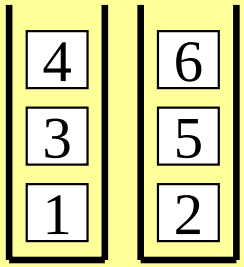
John von Neumann（1903年－1957年），奧匈帝國人，獲得數學博士學位（輔修實驗物理），曾經參與美國曼哈頓計畫

1945年6月，John von Neumann發表了一篇長達101頁紙的報告，將電腦分成五大組件，並明確規定用二進制替代十進制運算，被譽為「電腦之父」。同年（1945年），John von Neumann 發明了 merge sort 演算法

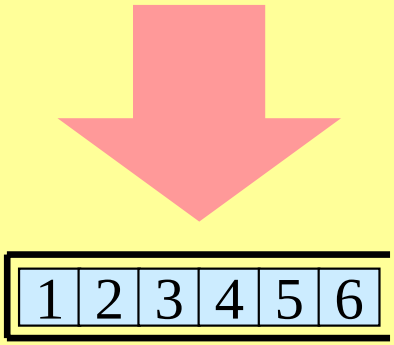
1944年出版「Theory of Games and Economic Behavior」一書，被稱為「博弈論之父」

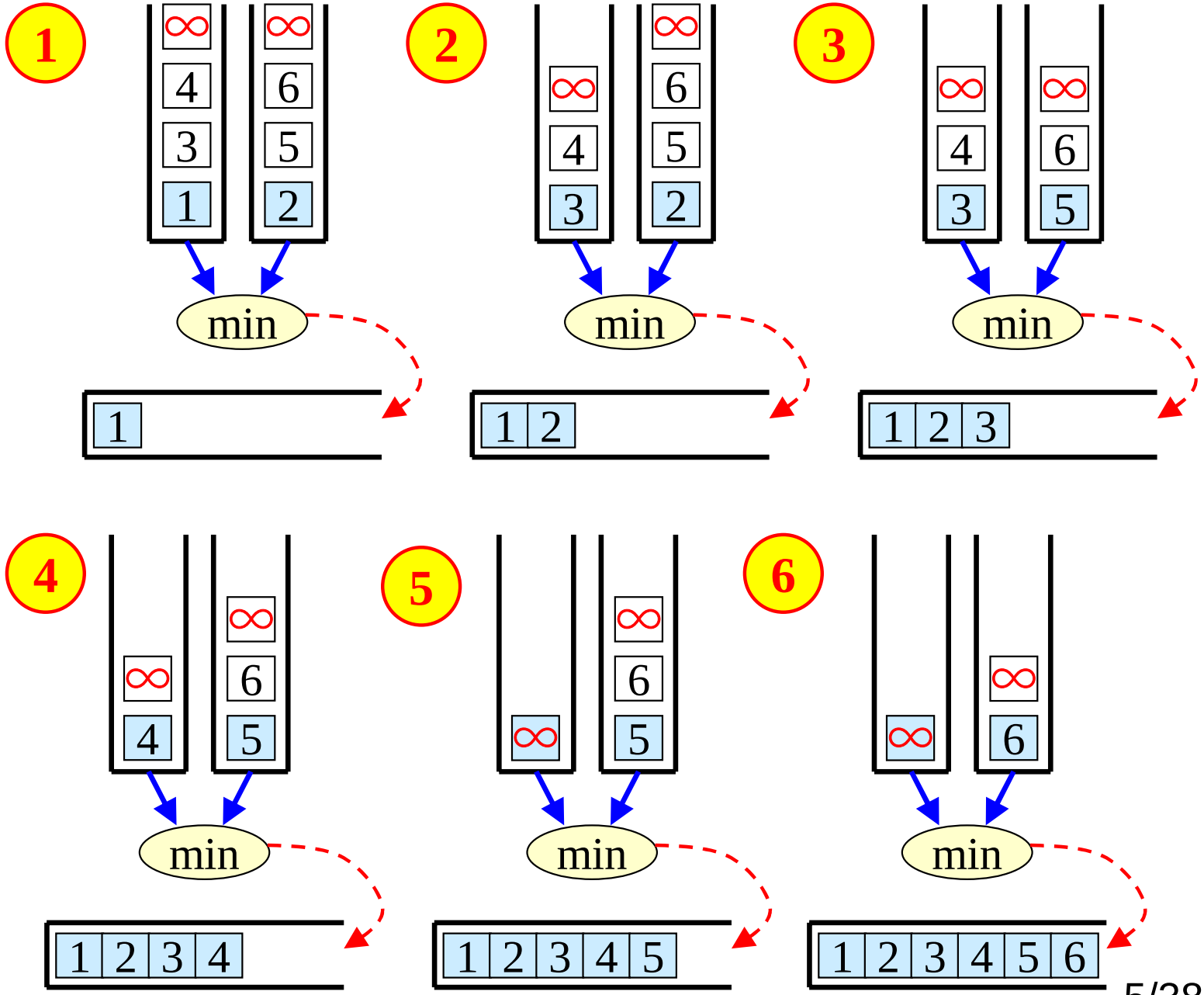
Let's See How to Merge Two Sorted Arrays

前提：給定兩個 **sorted arrays**



目標：合併成一個 **sorted array**



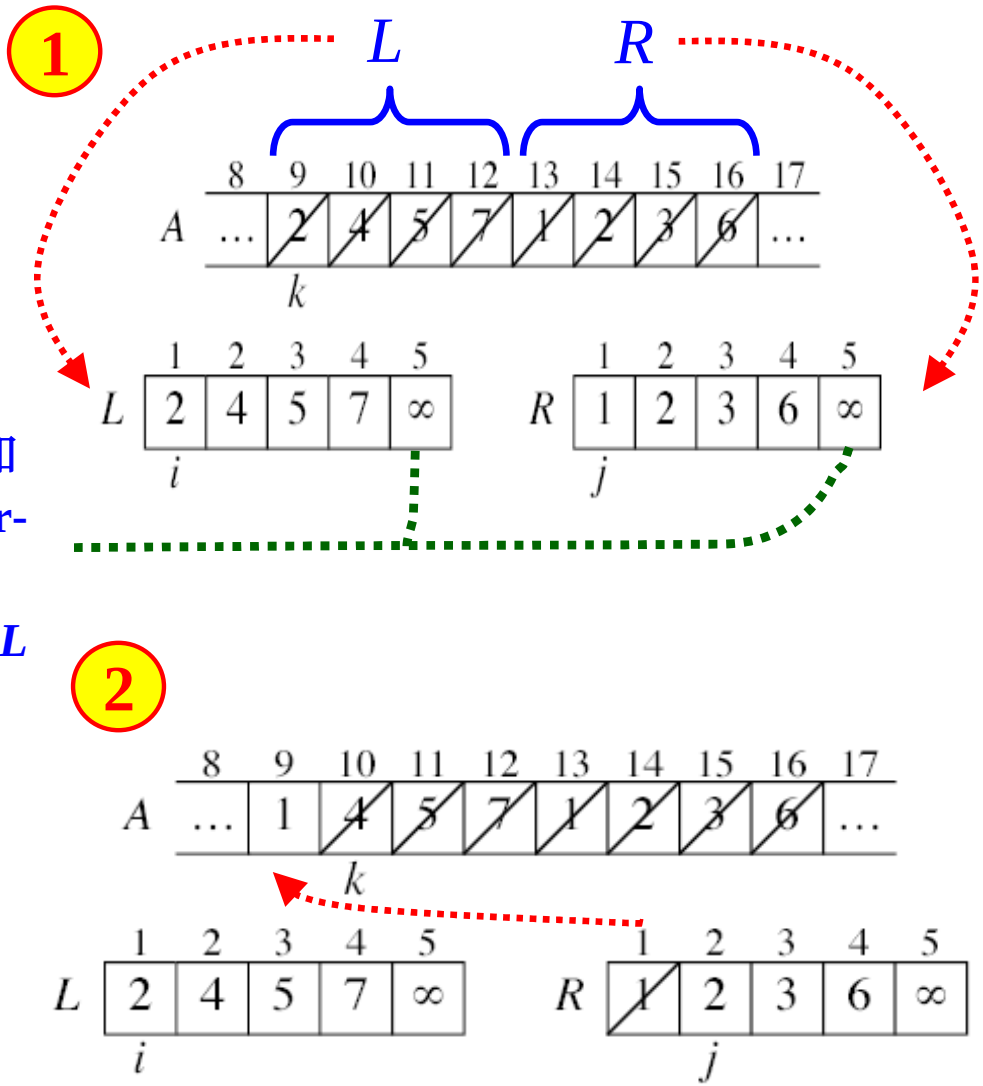


Pseudo Code of Merge Procedure

```
MERGE(A, p, q, r)
n1 ← q − p + 1        (n1 = 12 - 9 + 1 = 4)
n2 ← r − q            (n2 = 16 - 12 = 4)
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
    do L[i] ← A[p + i − 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i + 1
          else A[k] ← R[j]
               j ← j + 1
```

array 的最尾巴添加「哨兵」讓整個 for-loop 比較 $r-p+1$ 次即可，而不需顧慮 L 或 R 何時變空

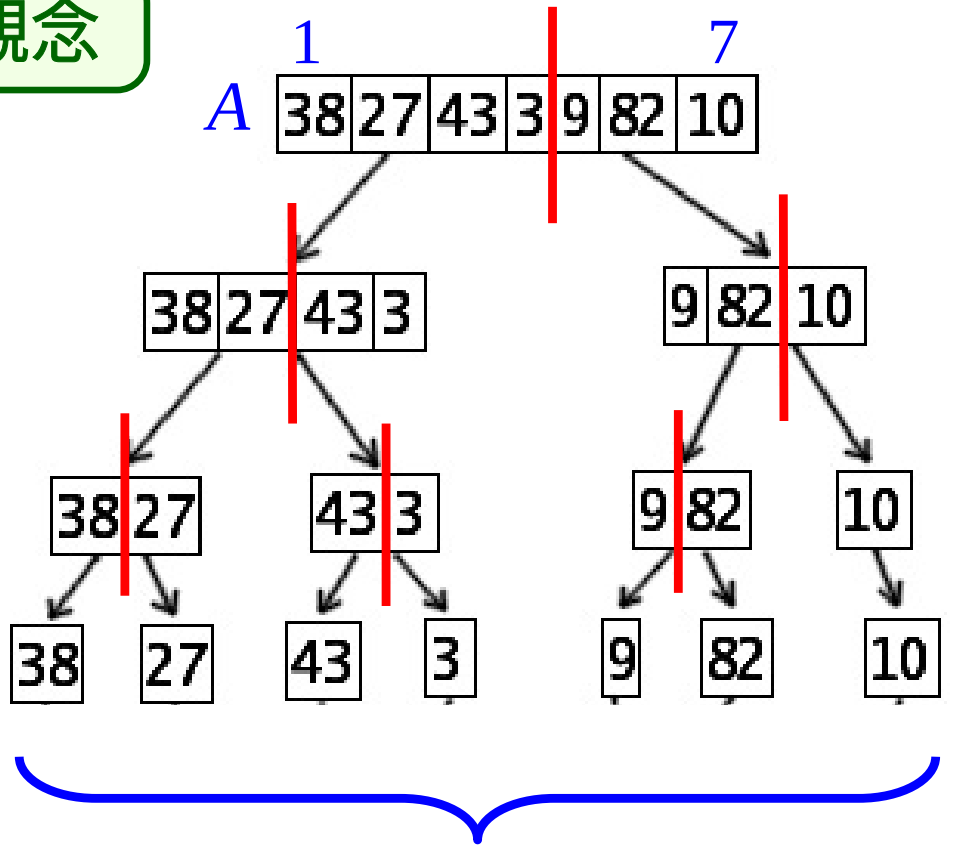
呼叫 Merge(*A*, 9, 12, 16)



1st Phase of Merge Sort: Divide

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

觀念



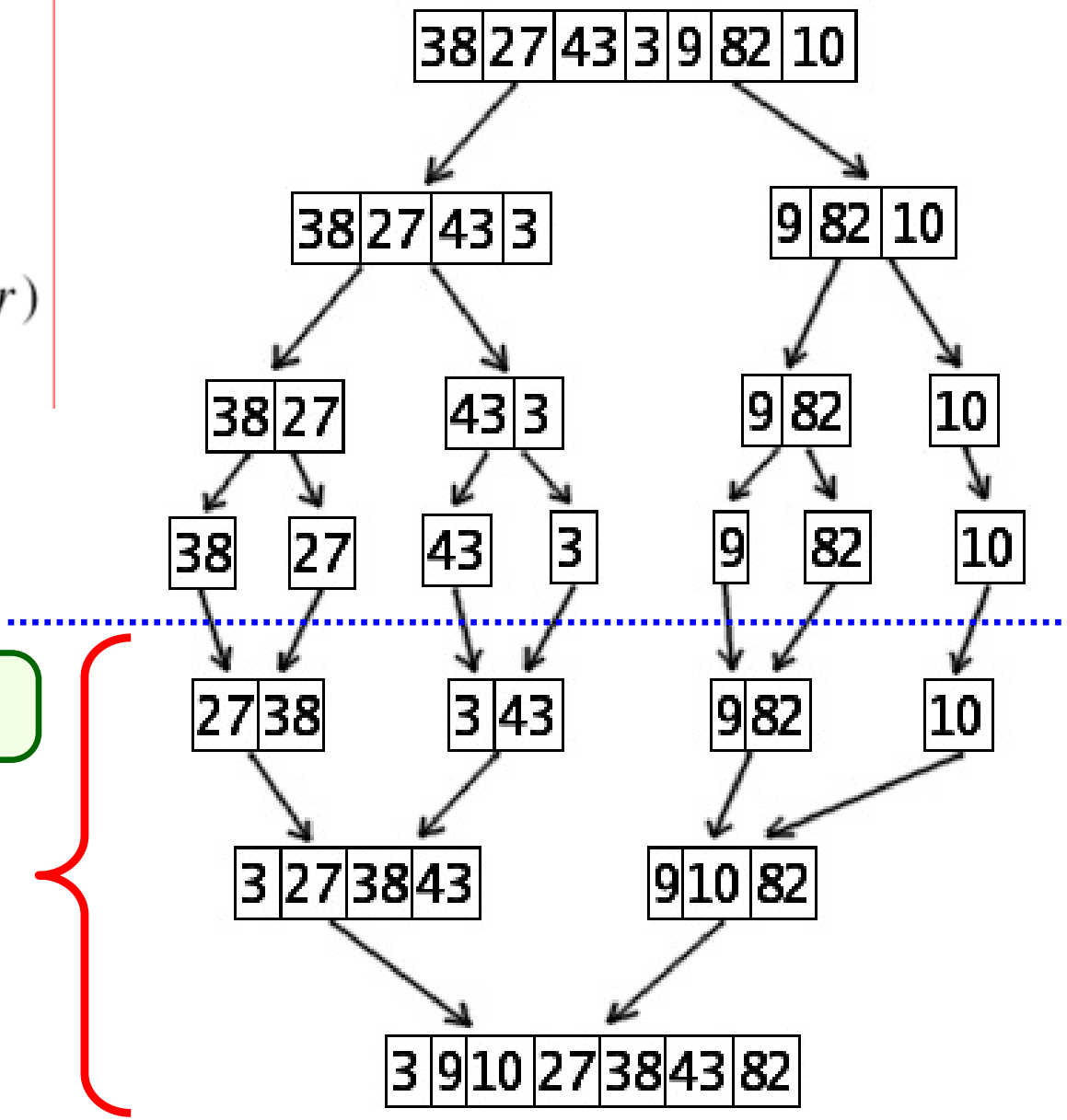
此圖顯示 **merge sort** 的切割方式，而非程式的執行步驟。詳細執行次序參見下下頁

一直對切，直到 **size** 等於 **1** 為止

當 **array size** 為 **1** 時，很自然就是一個 **sorted array**

2nd Phase of Merge Sort: Conquer and Combine

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```



觀念

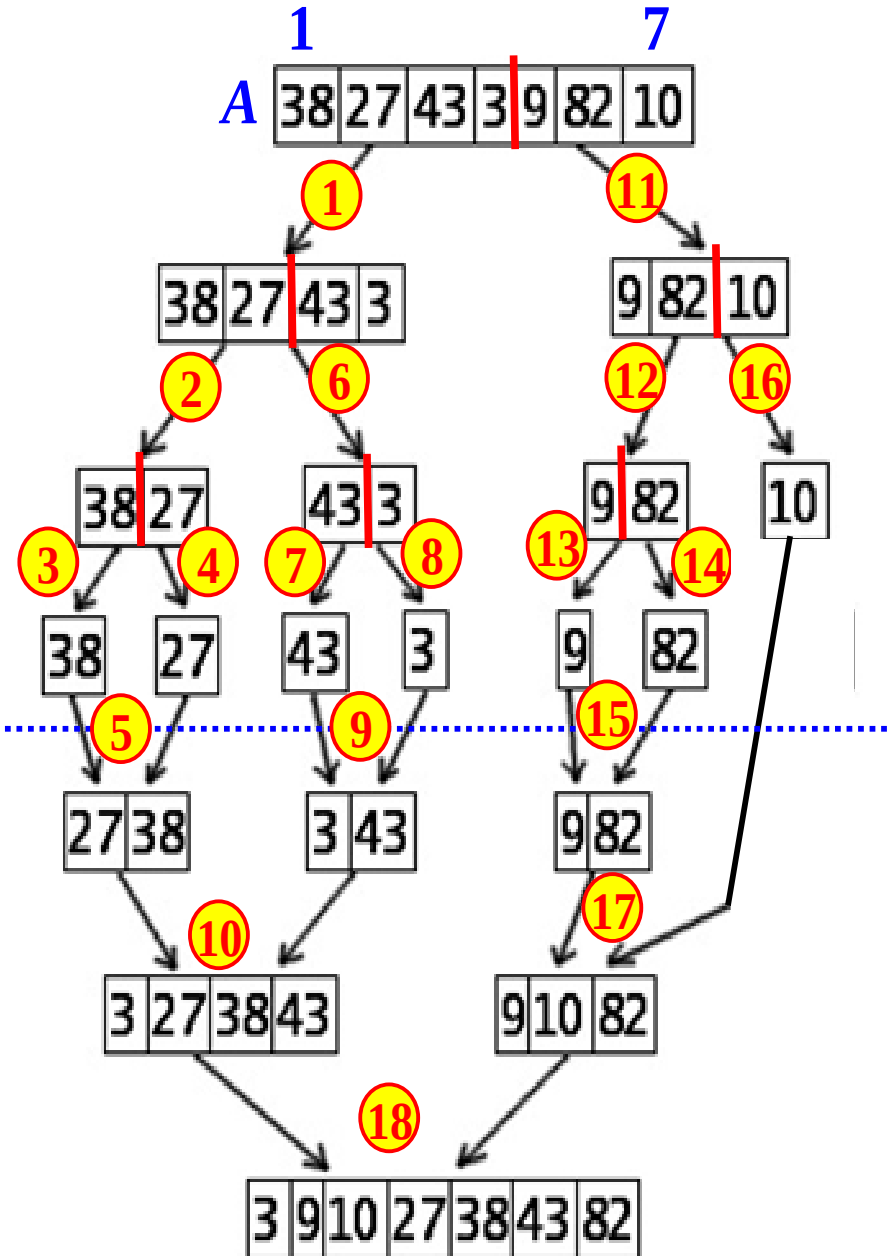
一直 merge，一直 merge
最後就合成為一個 sorted array

Execution Steps by Recursive Merge Sort

真正執行步驟

```
MERGE-SORT(A, p, r)
if p < r
  then q ← ⌊(p + r)/2⌋
       MERGE-SORT(A, p, q)
       MERGE-SORT(A, q + 1, r)
       MERGE(A, p, q, r)
```

Step 0: MergeSort(1, 7)
Step 1: MergeSort(1, 4)
Step 2: MergeSort(1, 2)
Step 3: MergeSort(1, 1)
Step 4: MergeSort(2, 2)
Step 5: Merge(1, 1, 2)
Step 6: MergeSort(3, 4)
Step 7: MergeSort(3, 3)
Step 8: MergeSort(4, 4)
Step 9: Merge(3, 3, 4)
Step 10: Merge(1, 2, 4)
Step 11: MergeSort(5, 7)
Step 12: MergeSort(5, 6)
Step 13: MergeSort(5, 5)
Step 14: MergeSort(6, 6)
Step 15: Merge(5, 5, 6)
Step 16: MergeSort(7, 7)
Step 17: Merge(5, 6, 7)
Step 18: Merge(1, 4, 7)



Running Time Analysis

```
MergeSort(A, left, right) {                          T(n)
   if (left < right) {                               Θ(1)
      mid = floor((left + right) / 2);               Θ(1)
      MergeSort(A, left, mid);                       T(n/2)
      MergeSort(A, mid+1, right);                    T(n/2)
      Merge(A, left, mid, right);                    Θ(n)
   }
}
```

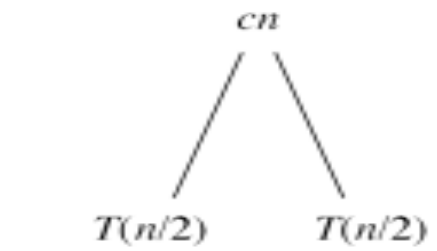
所以我們有 $T(n) = 2T(n/2) + \Theta(n)$, when $n > 1$

→ 意即存在 constant c，使得當 $n > n_0$ 時，$T(n) = 2T(n/2) + cn$

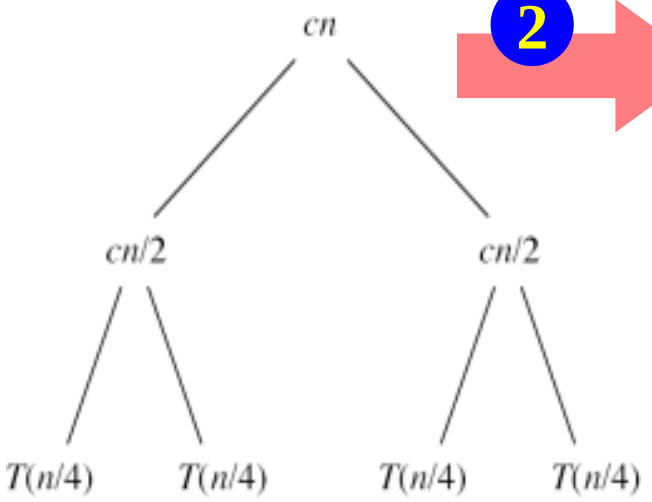
現在，我們將推導出 $T(n)$

Informal Proof：Recursion Tree Method

$T(n) = 2T(n/2) + cn$

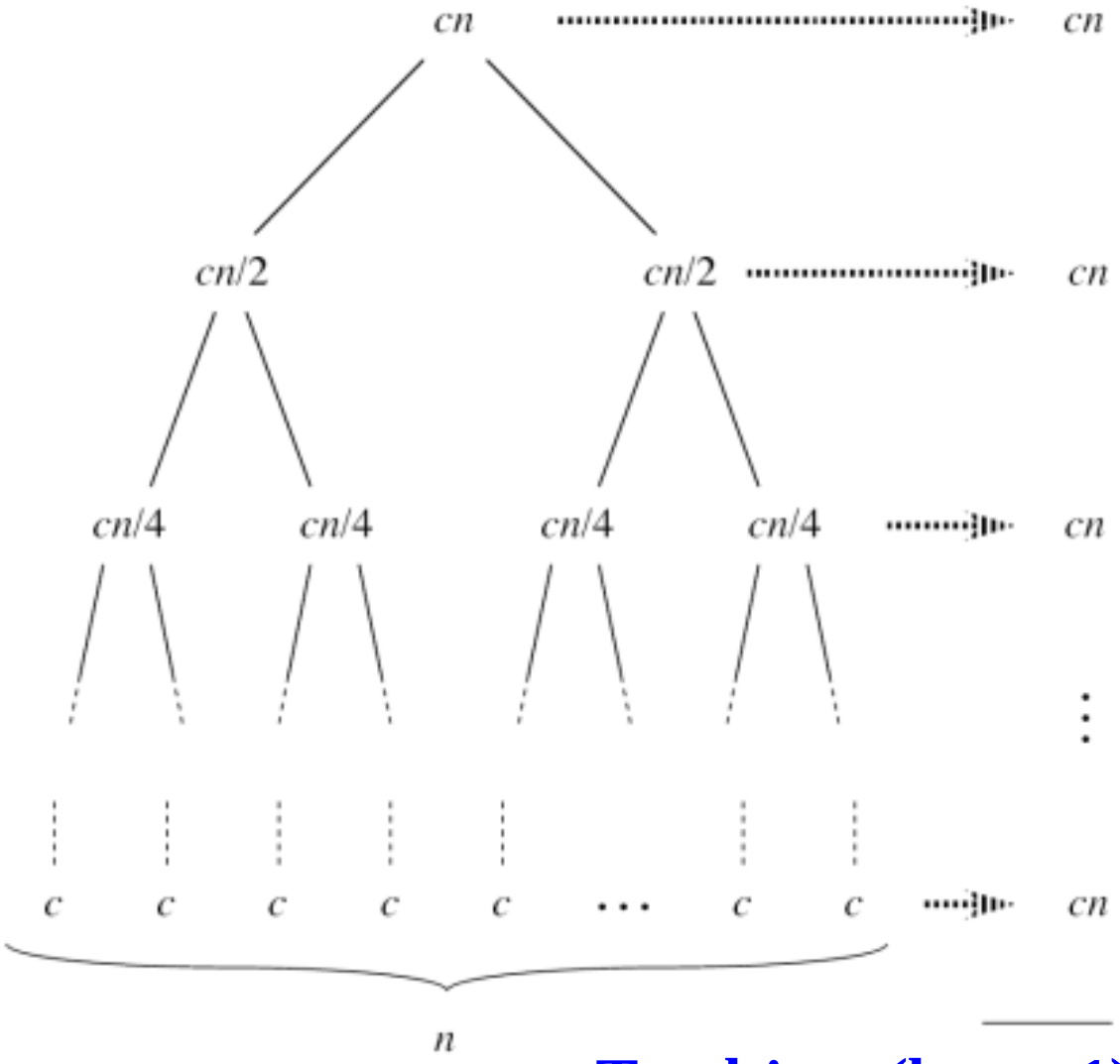


1



2

樹高
log n



Total：cn (log n+1)

$T(n) = O(n \log n)$

Formal Proof by Substitution Method

已知 $T(n)=2T(n/2)+\Theta(n)$，我們猜測 $T(n)=O(n\log n)$

$\Rightarrow$ 我們採用 substitution 的技巧來推導看看，

看是否存在常數 c 和 n_0，使得當 $n\geq n_0$，我們有 $T(n)\leq cn\log n$

假如 $T(n)\leq cn\log n$ 是正確的，那麼代入 $T(n)=2T(n/2)+\Theta(n)$ 的遞迴式

我們有 $$T(n) \leq 2\times\left[c\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right)\right]+\hat{c}n = cn\left(\log n-\log 2\right)+\hat{c}n$$

$$= cn\log n-\left(c\log 2-\hat{c}\right)n$$

要讓 $T(n)\leq cn\log n$，我們必須讓 $\left(c\log 2-\hat{c}\right)\geq 0$ $\Rightarrow$ 亦即 $c\geq\frac{\hat{c}}{\log 2}$

因此當 $n\geq 1$ 且 $c\geq\frac{\hat{c}}{\log 2}$ 時，$T(n)\leq cn\log n$ 成立。故得證。

The Practical Value of Quick Sort

從上次的 lecture，我們得知：任何一個 comparison-based sorting algorithm，其 running time 至少 $\Omega(n \log n)$，而 merge sort 在最差情況之下，其 running time 為 $O(n \log n)$。因此 merge sort 為 optimal sorting algorithm。

然而若觀察一下 Turbo C++ 2.0 所提供的 library，我們會發現 Turbo C++ 2.0 的 library 只提供 quick sort 讓我們呼叫。這意味著就實用價值來說，quick sort 是目前已知的所有 sorting algorithms 中最高的。 現在讓我們看看 quick sort 是如何運作的

實驗結果

Sort type	Input is...	Sort time (sec.)
Quicksort	unsorted	55.73
Merge sort	unsorted	60.04
Quicksort	already sorted	28.87
Merge sort	already sorted	30.29

http://www.ai.mit.edu/projects/reinventing_computing/papers/theses/pshuang/meng/node20.html#SECTION00710000000000000000

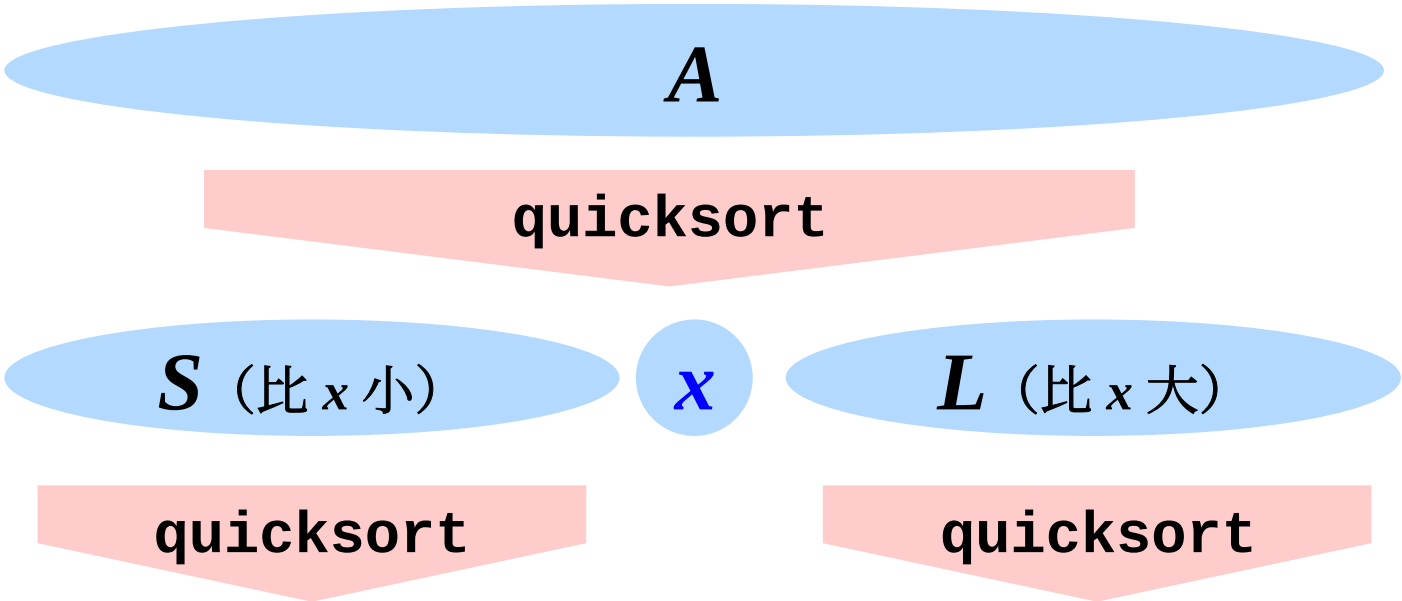
The Inventor of Quick Sort

Tony Hoare，**1956** 年在牛津大學取得西洋古典學學士學位，**1956** 年至 **1958** 年間，在英國皇家海軍服役。他為了學習俄語，至蘇聯莫斯科大學留學，**1960** 年取得數學博士學位。同年（**1960**年），發明 **quick sort**，並開發出 **ALGOL 60** 編譯器。**1980** 年獲得 **Turing award**。

至今仍然活著，在劍橋微軟研究院擔任研究員。

Quick Sort：Concept

```
quicksort(A) {
  if (length[A] == 0)
    return;
  select an element x from A;
  S:={(y 屬於 A) and (y ≤ x)};
  L:={(z 屬於 A) and (z > x)};
  quicksort(S);
  print x;
  quicksort(L);
}
```

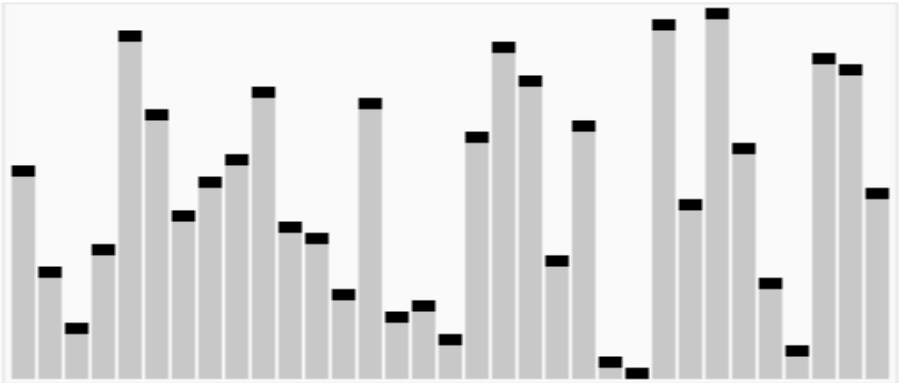


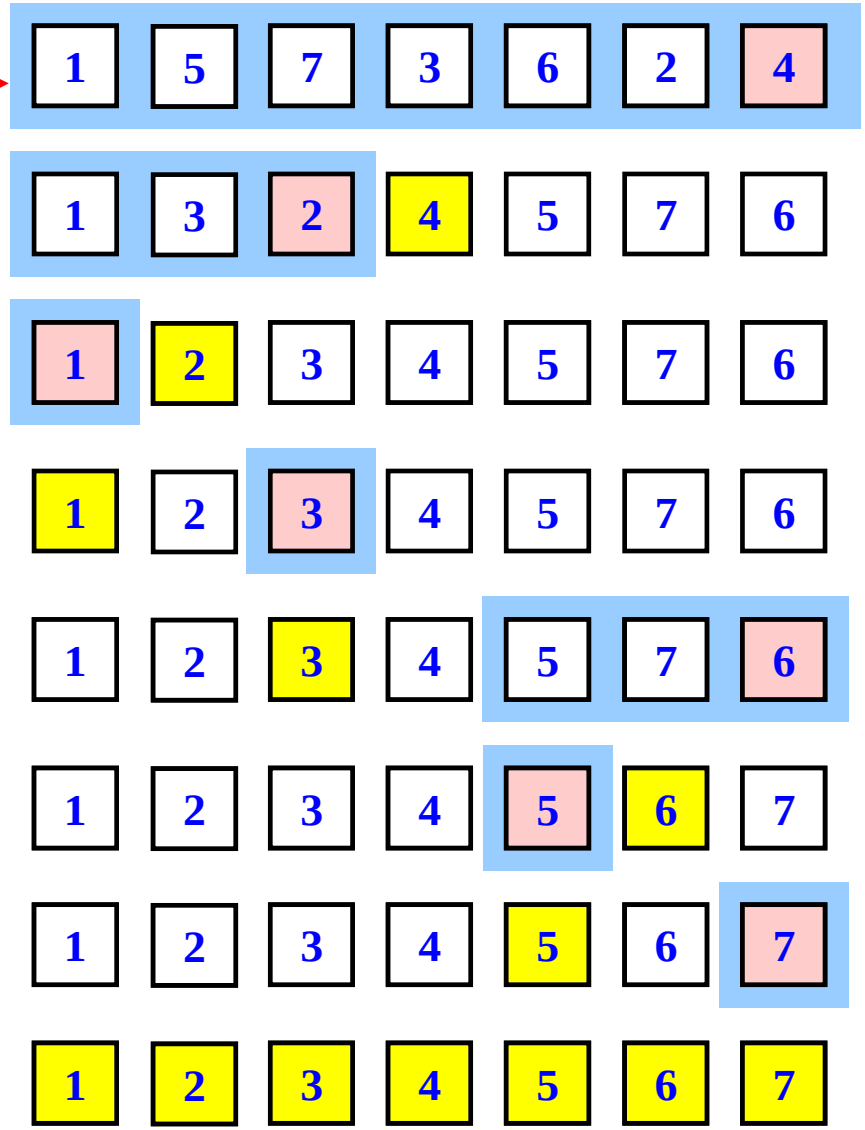
Quick Sort：Example

假設每次都選擇 *A* 的最後一個 **element**

```
quicksort(A) {
  if (length[A] == 0)
    return;
  select an element x from A;
  S:={(y 屬於 A) and (y ≤ x)};
  L:={(z 屬於 A) and (z > x)};
  quicksort(S);
  print x;
  quicksort(L);
}
```

按一下觀賞動畫





Running Time Analysis

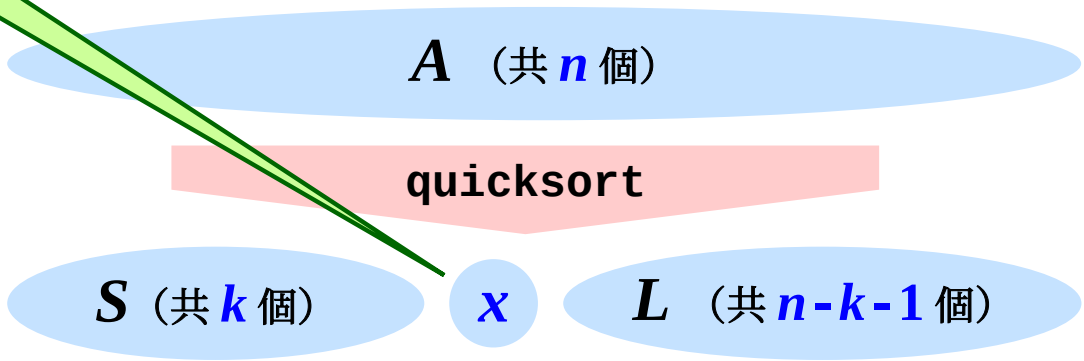
```
quicksort(A) {
  if (length[A] == 0)
    return;
  select an element x from A;
  S:={(y 屬於 A) and (y ≤ x)};
  L:={(z 屬於 A) and (z > x)};
  quicksort(S);
  print x;
  quicksort(L);
}
```

$O(n)$：意即存在 constant c，使得當 $n > n_0$ 這部分的 steps $\le cn$

quicksort(S); ⋯⋯▶ $T(k)$

quicksort(L); ⋯⋯▶ $T(n-k-1)$

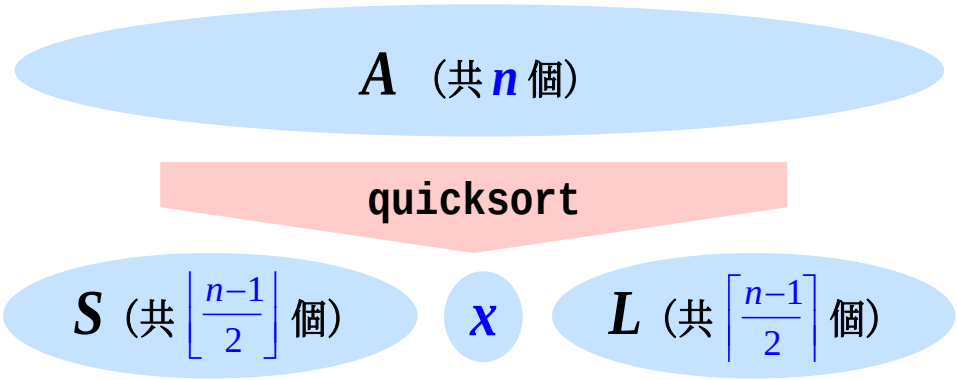
critical step



所以 quick sort 的 runnung time $T(n)$ 為

$$T(n)=\begin{cases}1 & \text{if } n=1\\ T(k)+T(n-k-1)+cn & \text{if } n>1\end{cases}$$

Best Case Running Time：Balanced Partition

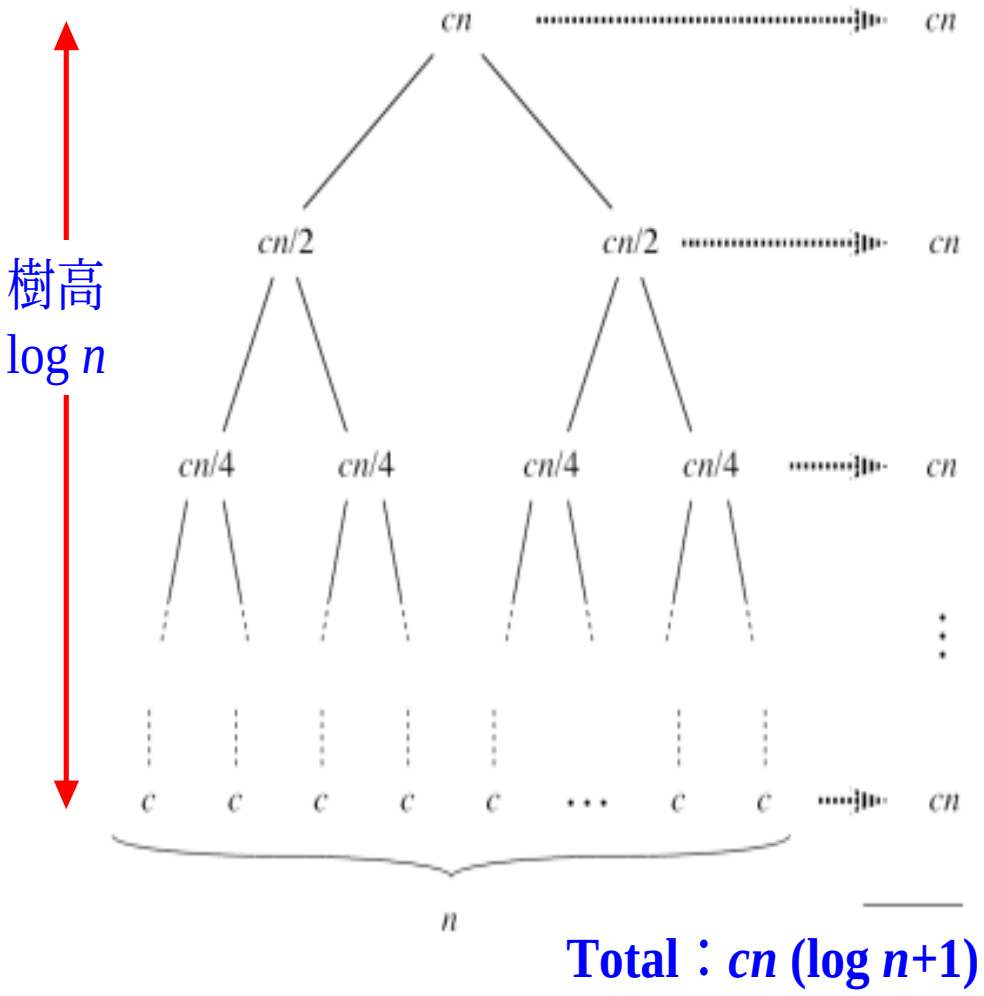


Best case 發生在 quicksort 每次都能平均地將 S 和 L 切成一半

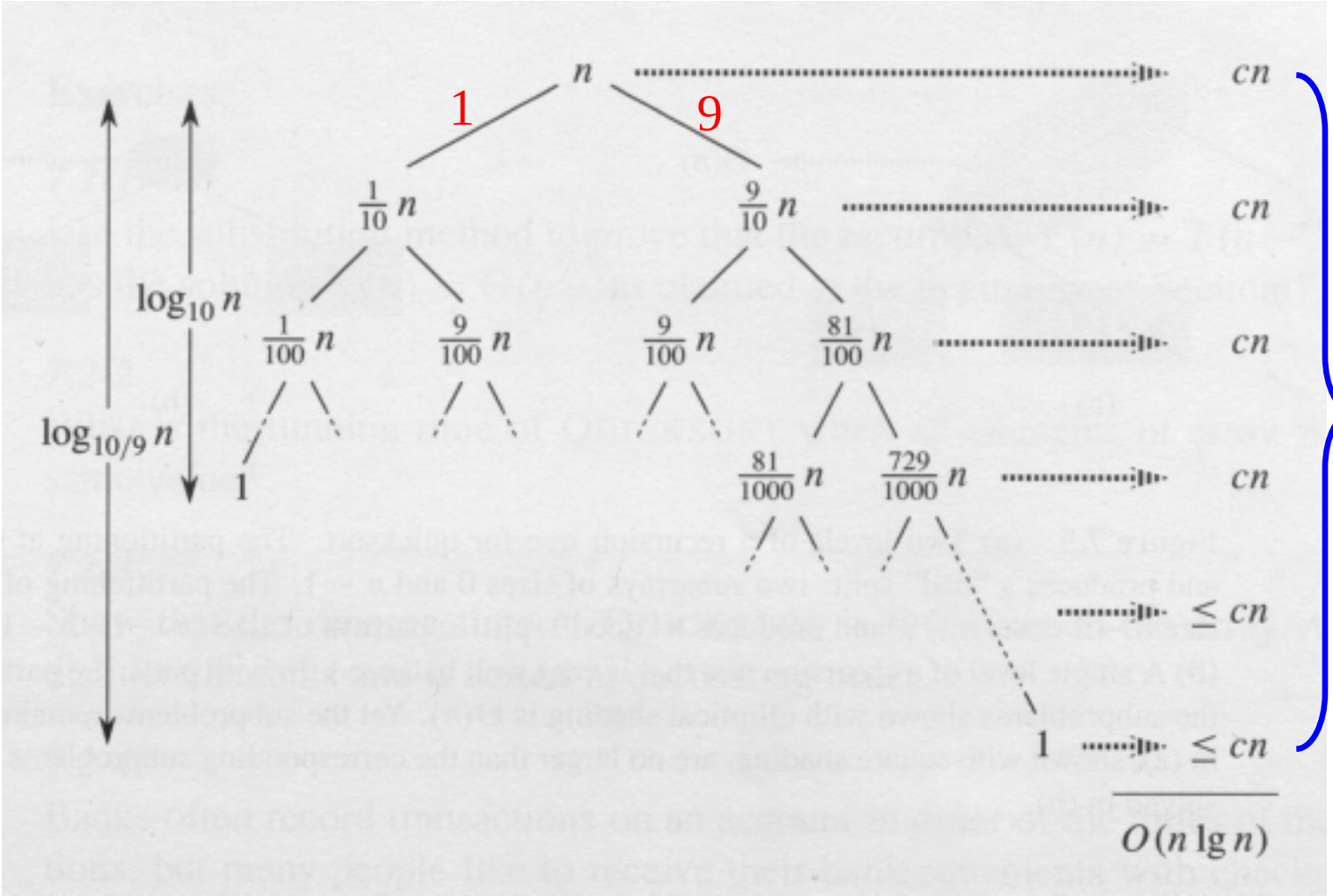
此時

$$T(n) = T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + cn$$

$$\leq 2T(n/2) + cn = O(n \log n)$$



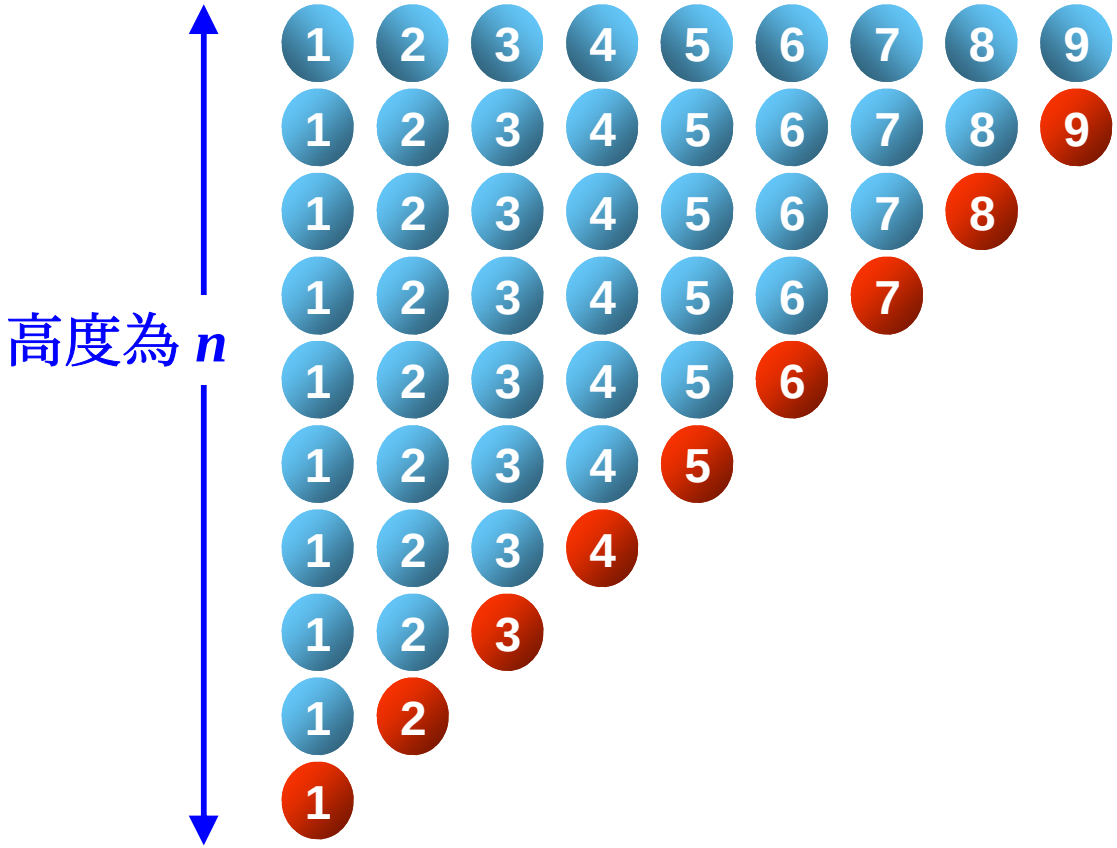
Running Time When Partition Is Unbalanced



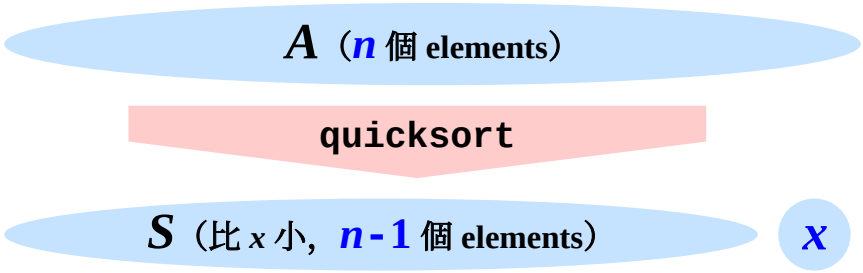
$$T(n) \le T(9n/10) + T(n/10) + cn$$
$$= O(n \log n)$$

即使沒有很平均的 **partition,**
quicksort 的 **performance** 還是很好

Worst Case Running Time




```
quicksort(A) {
  if (length[A] == 0)
    return;
  select an element x from A;
  S:={(y 屬於 A) and (y ≤ x)};
  L:={(z 屬於 A) and (z > x)};
  quicksort(S);
  print x;
  quicksort(L);
}
```



在 worst case 之下，L （或 S）的 size 永遠為 0，此時

$$T(n)=T(n-1)+cn=T(n-2)+c\left[(n-1)+n\right]=T(n-3)+c\left[(n-2)+(n-1)+n\right]$$

$$=T(1)+c\left[1+2+\cdots+(n-1)+n\right]=O(n^2)$$

Key: Select the Median as Pviot

若要使得 **quick sort** 的執行時間達到 **optimal**，就必須謹慎選擇選擇 **pivot** x，使得每次都能剛好將 **array A** 平均地 **partition** 成兩半，而且在 $O(n)$ 之內完成，以確保 $T(n) = 2T(n/2) + O(n) = O(n \log n)$

超級任務：必須在 $O(n)$ 之內找出「中位數」！

```
optimal_quicksort(A) {
  if (length[A] == 0)
    return;
  select the median x from A;
  // such that |length[S]-length[L]| ≤ 1;
  S:={(y 屬於 A) and (y ≤ x)};
  L:={(z 屬於 A) and (z > x)};
  optimal_quicksort(S);
  print x;
  optimal_quicksort(L);
}
```

$T(n/2)$

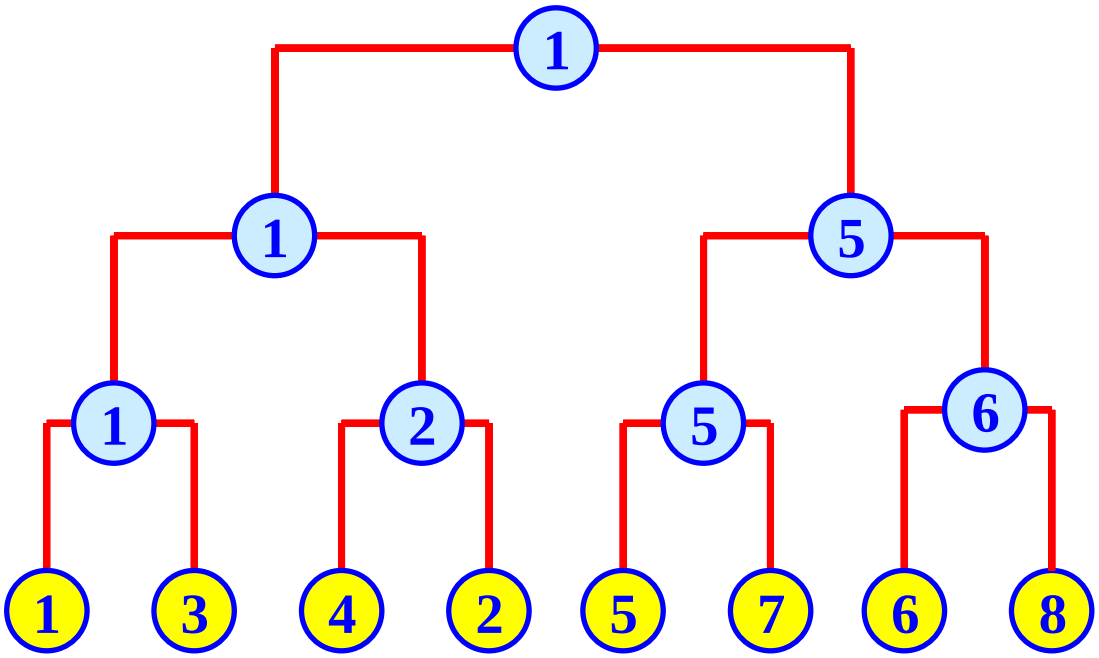
$T(n/2)$

Mathematical Definitions

- The *i*-th *order statistic* in a set of *n* elements is the *i*-th smallest element
  - The *minimum* is thus the 1st order statistic
  - The *maximum* is the *n*-th order statistic
- For simplicity, we define the *median* as the $\lceil n/2 \rceil$ order statistic.

Lower Bound for Finding the Champion

讓我們考慮一個問題：有 n 個參賽者，至少要比賽幾場才能產生「冠軍」？



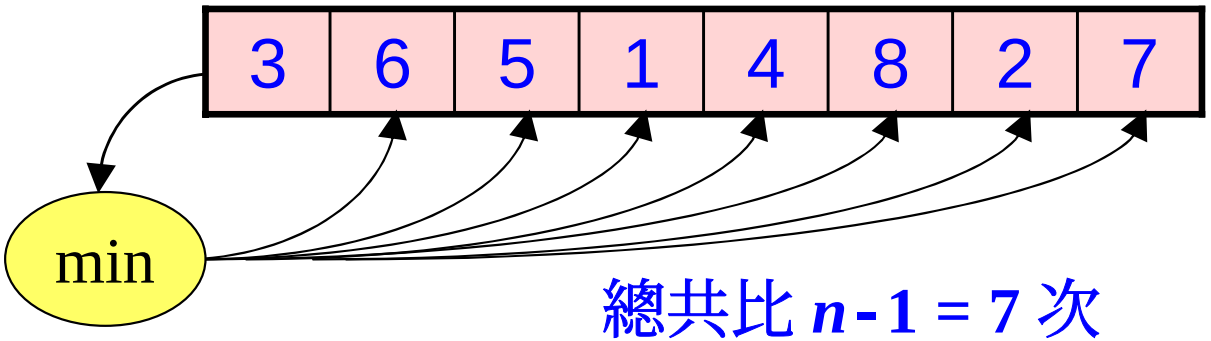
要產生冠軍，必須除了冠軍之外，其餘所有參賽者都必須至少「直接」或「間接」敗給冠軍一場。想想看，除了冠軍之外，如果還有一位參賽者從未失敗過，那我們怎麼有辦法決定出冠軍呢？因此**要讓冠軍產生，至少需比賽 $n-1$ 場**（意即：**除了冠軍之外，其餘每位參賽者都至少輸過一場**）

Algorithms for Finding the Minimum

- How many comparisons are needed to find the minimum or the maximum element in the set A of n distinct elements ?

```
MINIMUM(A, n)
min ← A[1]
for i ← 2 to n
    do if min > A[i]
        then min ← A[i]
return min
```

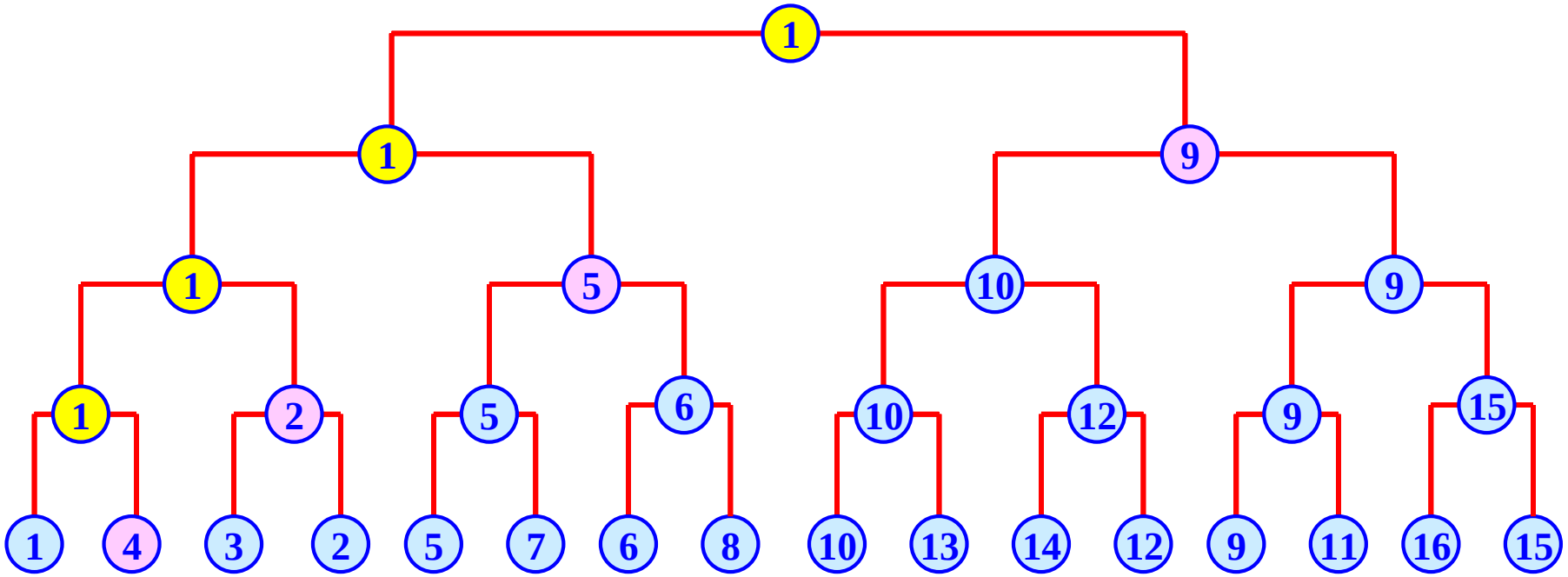
問：這個是 **optimal algorithm** 嗎？



總共比 $n-1 = 7$ 次

Lower Bound for Finding the Runner-Up (1/2)

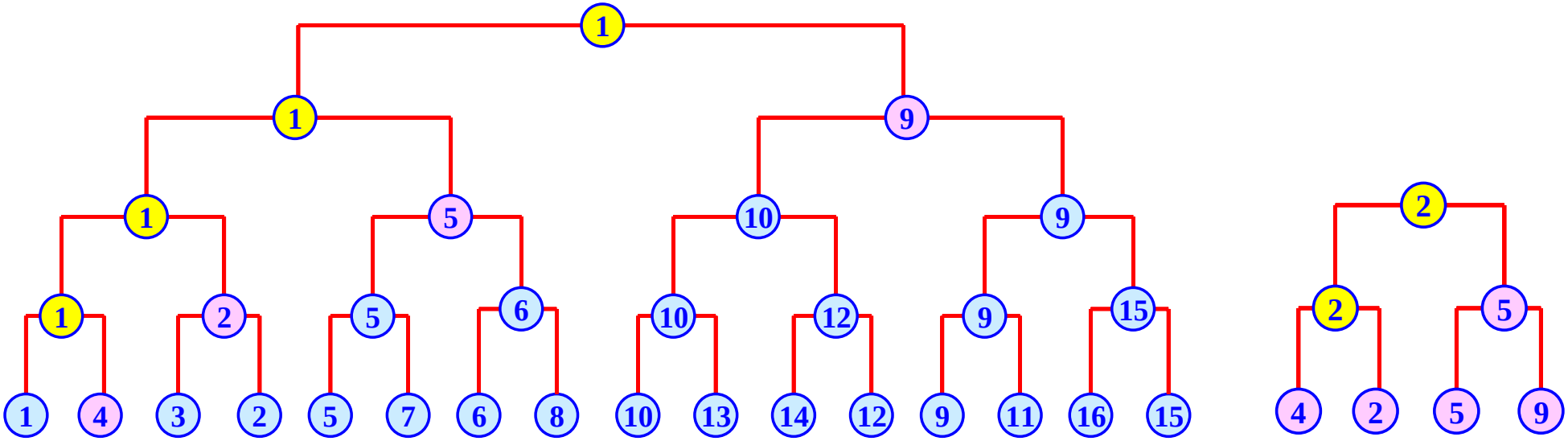
Claim：給定 n 個元素，要決定出亞軍，需比較 $n-2+\lceil \lg n \rceil$ 次



要產生亞軍，就必須先要產生冠軍（否則沒辦法決定出誰是亞軍）

所以至少要先有 $n-1$ 次的比賽（或 比較），以便決定出冠軍

Lower Bound for Finding the Runner-Up (2/2)



由於冠軍（如左圖黃色點）只出賽 $p=\lceil \lg n \rceil$ 場，且出賽的對象都不一樣。

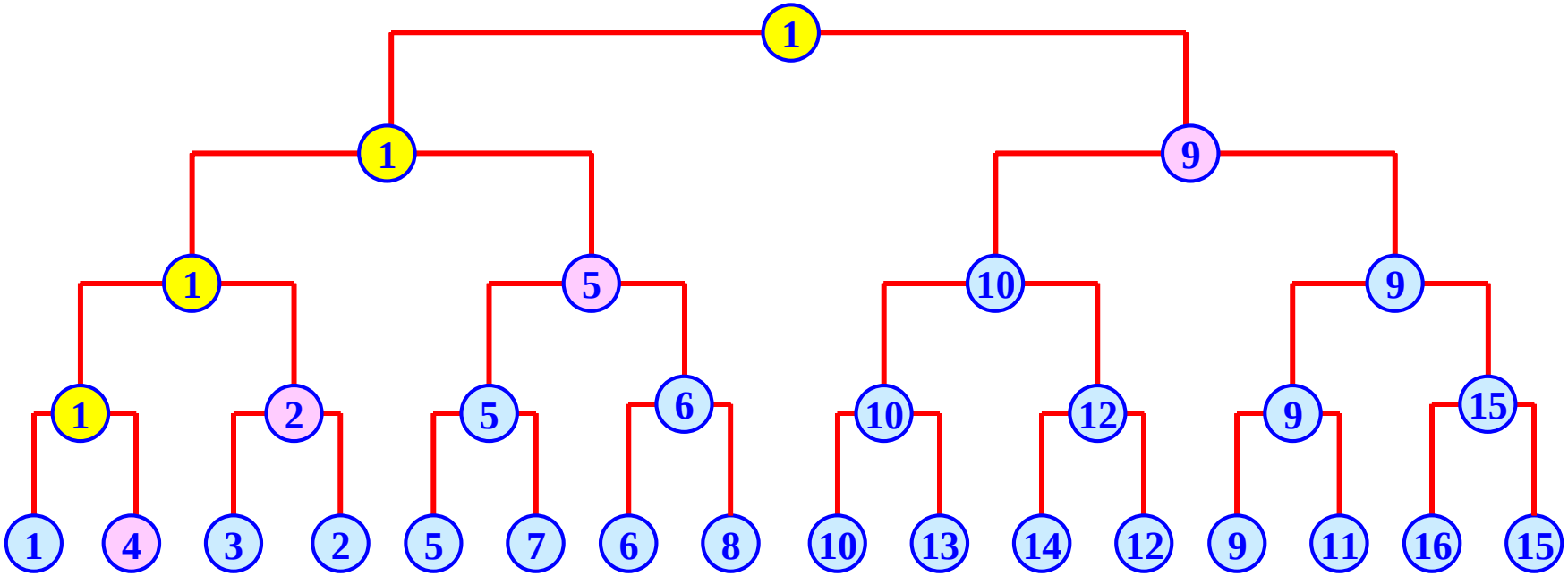
這意味著這 p 位參賽者（如左圖紅色點），除了敗給冠軍之外，未曾輸過。

因此亞軍就落在這 p 位參賽者之中。

而要在這 p 位參賽者之中決定出最優秀者（亞軍）至少需 $p-1$ 次的比較，

因此要決定出亞軍，需比較 $(n-1)+(p-1)=n-2+\lceil \lg n \rceil$ 次

How to Find the *i*-th Order Statistic ?



找出「中位數」，顯然比找出「冠軍」和「亞軍」更困難。

那我們還有辦法在 $O(n)$ 的時間之內找出第 i 小的元素嗎？

其中 $1 \le i \le n$。如果有辦法的話，我們就能在 $O(n)$ 時間之內找出中位數

Selection in Linear Time

Now we show an amazing deterministic algorithm which solves the selection problem in $O(n)$ time even in the worst case. We will learn the following techniques.

- Divide-and-conquer approach
- Analysis of recurrence relation

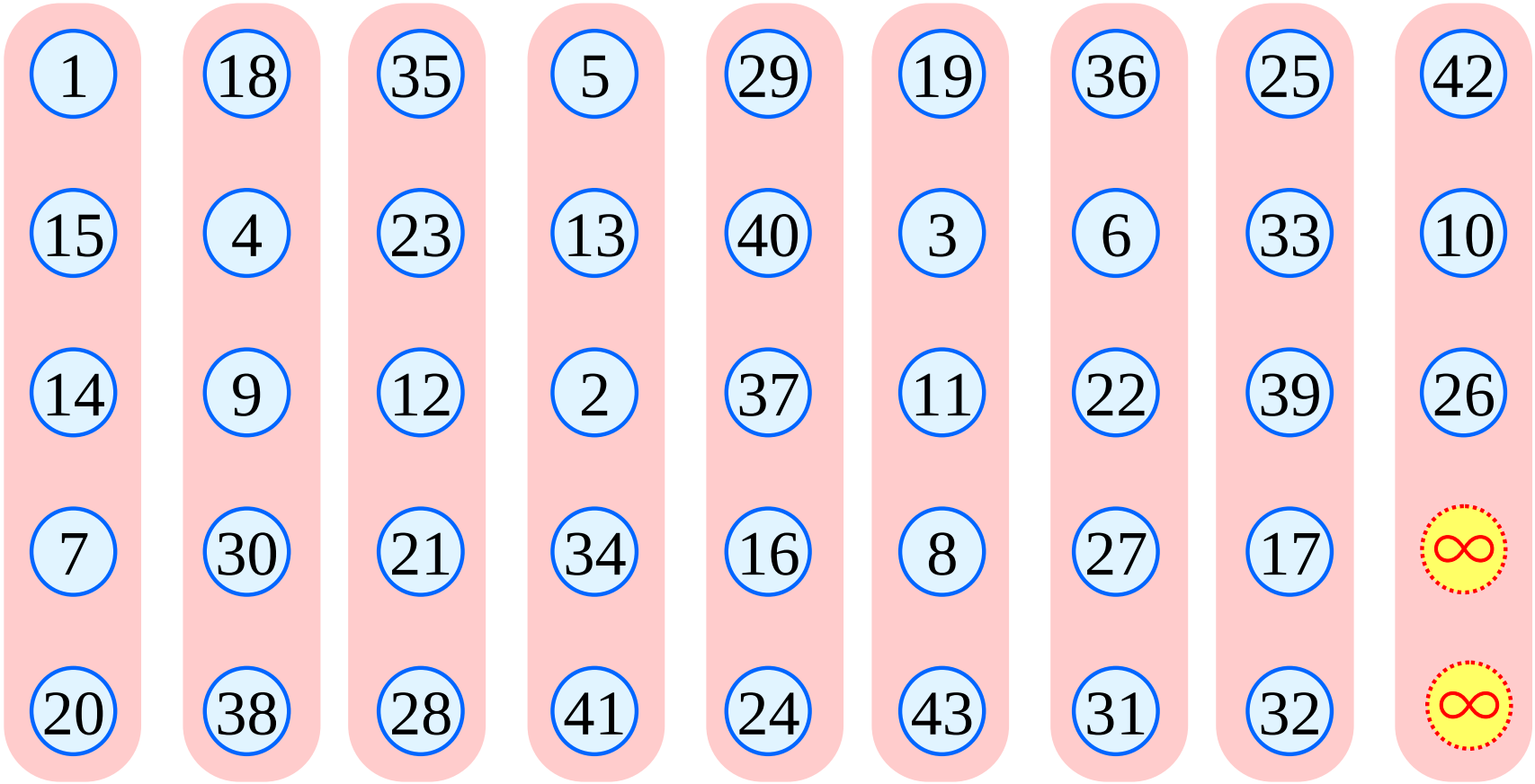
如果你想不到,
不要哭喔,
那是正常的喔

- Manuel Blum (Turing Award 1995)
- Robert W. Floyd (Turing Award 1978)
- Vaughan R. Pratt
- Ronald L. Rivest (Turing Award 2002)
- Robert E. Tarjan (Turing Award 1986)



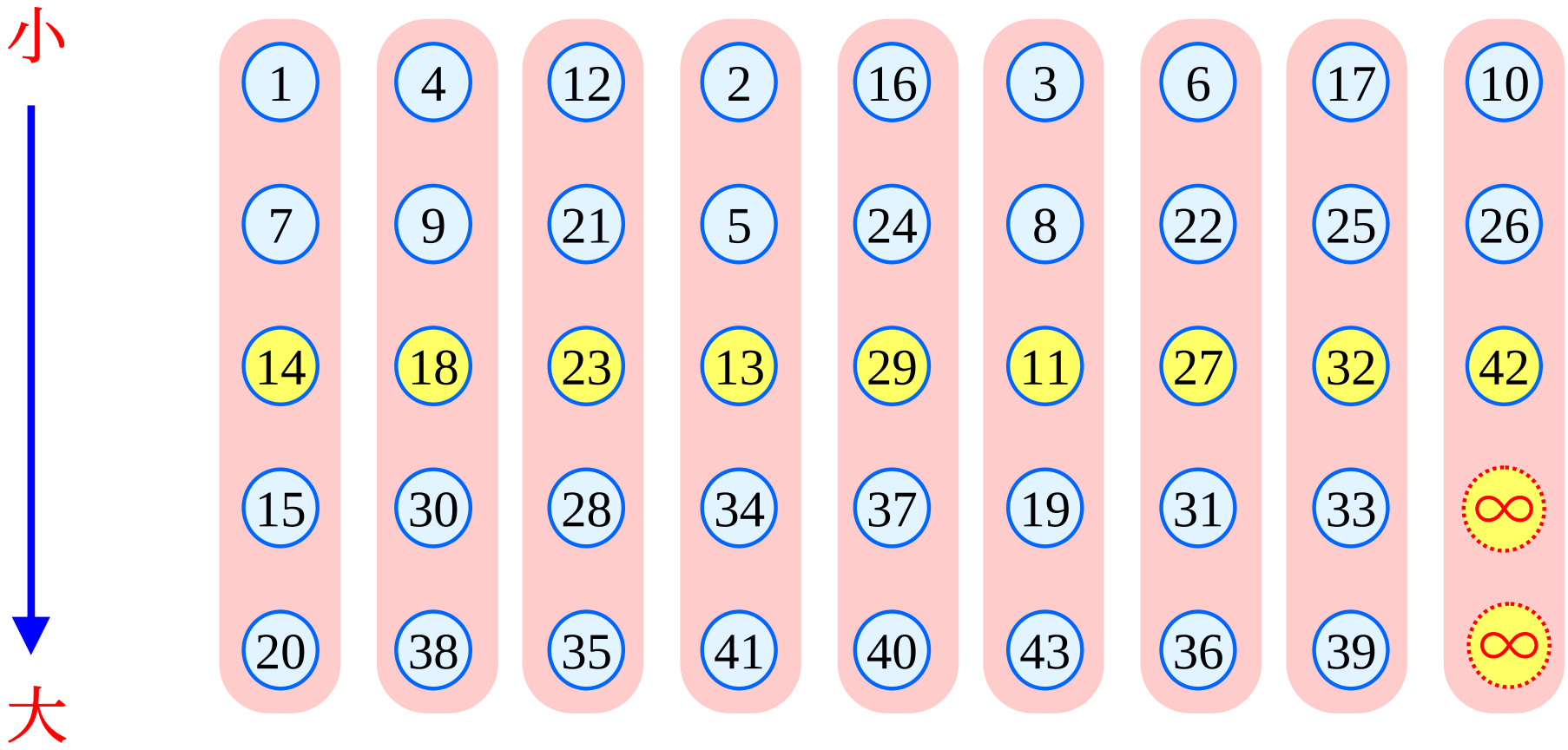


Step 1：5 Elements Per Group



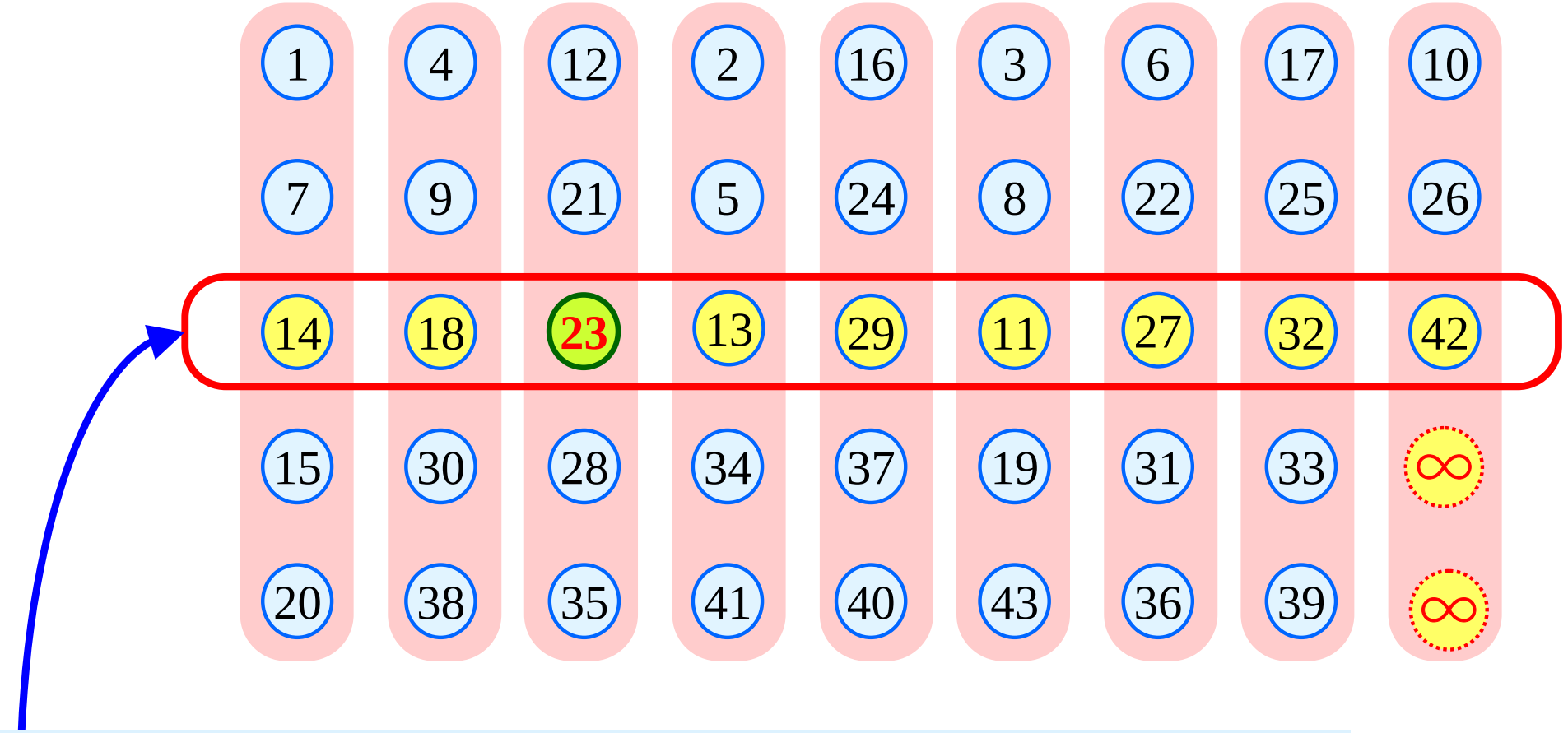
如上圖，共有 43 個 elements。如果最後一個 group 不滿 5 個，就補填∞，讓陣列元素個數為 5 的倍數。

Step 2：Sort Each Group and Pick the Median



使用 insertion sort 排序每個 group。由於每個 group 只有 5 個元素，所以 sorting 時間為 constant time，也就是 $O(1)$。總共有 $n/5$ 個 groups，所以總共執行時間需 $O(1) \times (n/5) = O(n)$。

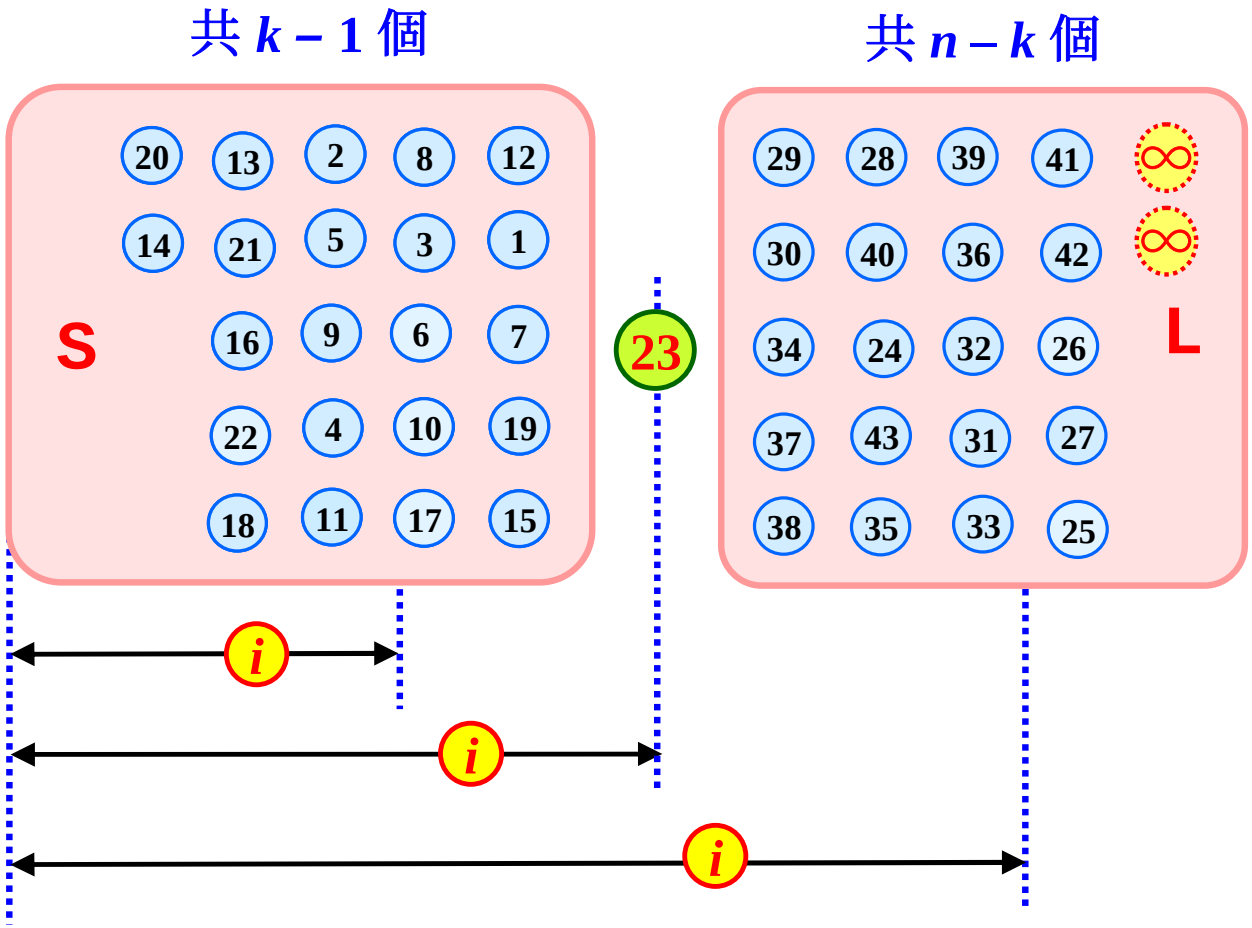
Step 3：Recursively Find the Median of Medians



遞迴呼叫 **`select`**，找出這裡頭的 median（稱之為 median-of-medians，用符號 MoM 表示）。假設程式 **`select`** 所需時間為 $T(n)$，那麼這個 step 需時 $T(n/5)$

Step 4：Partition the Array around MoM

```
k = length[S]+1;
if (i == k)
  return MoM;
if (i < k)
  select(S, i);
else if (i > k)
  select(L, i-k);
```



以 MoM 為 pivot，將 array A 的內容 copy 到二個陣列，一個為 S（小於MoM），另一個為 L（大於 MoM），總共需時 $O(n)$

The Selection Algorithm

```
int select(A, i) {
  for (j = 1; j<= (length[A])/5; j++) {
    insertion_sort( A[5(j-1)+1], A[5(j-1)+2],
       A[5(j-1)+3], A[5(j-1)+4], A[5(j-1)+5] );
    M[j] = A[5(j-1)+3];   // 儲存中位數
  }
  MoM = select(M, (length[M])/2 );

  S:={(y 屬於 A) and (y ≤ MoM)};
  L:={(z 屬於 A) and (z > MoM)};
  k = length[S]+1;

  if ( i == k )
    return MoM;
  if ( i < k )
    select(S, i);
  else if ( i > k )
    select(L, i - k );
}
```

Running Time Analysis

```
int select(A, i) {
  for (j = 1; j<= (length[A])/5; j++) {
    insertion_sort( A[5(j-1)+1], A[5(j-1)+2],
       A[5(j-1)+3], A[5(j-1)+4], A[5(j-1)+5] );
    M[j] = A[5(j-1)+3];   // 儲存中位數
  }
  MoM = select(M, (length[M])/2 );

  S:={(y 屬於 A) and (y ≤ MoM)};
  L:={(z 屬於 A) and (z > MoM)};
  k = length[S]+1;

  if ( i == k )
    return MoM;
  if ( i < k )
    select(S, i);
  else if ( i > k )
    select(L, i - k );
}
```

需時 $O(n)$

需時 $T(n/5)$

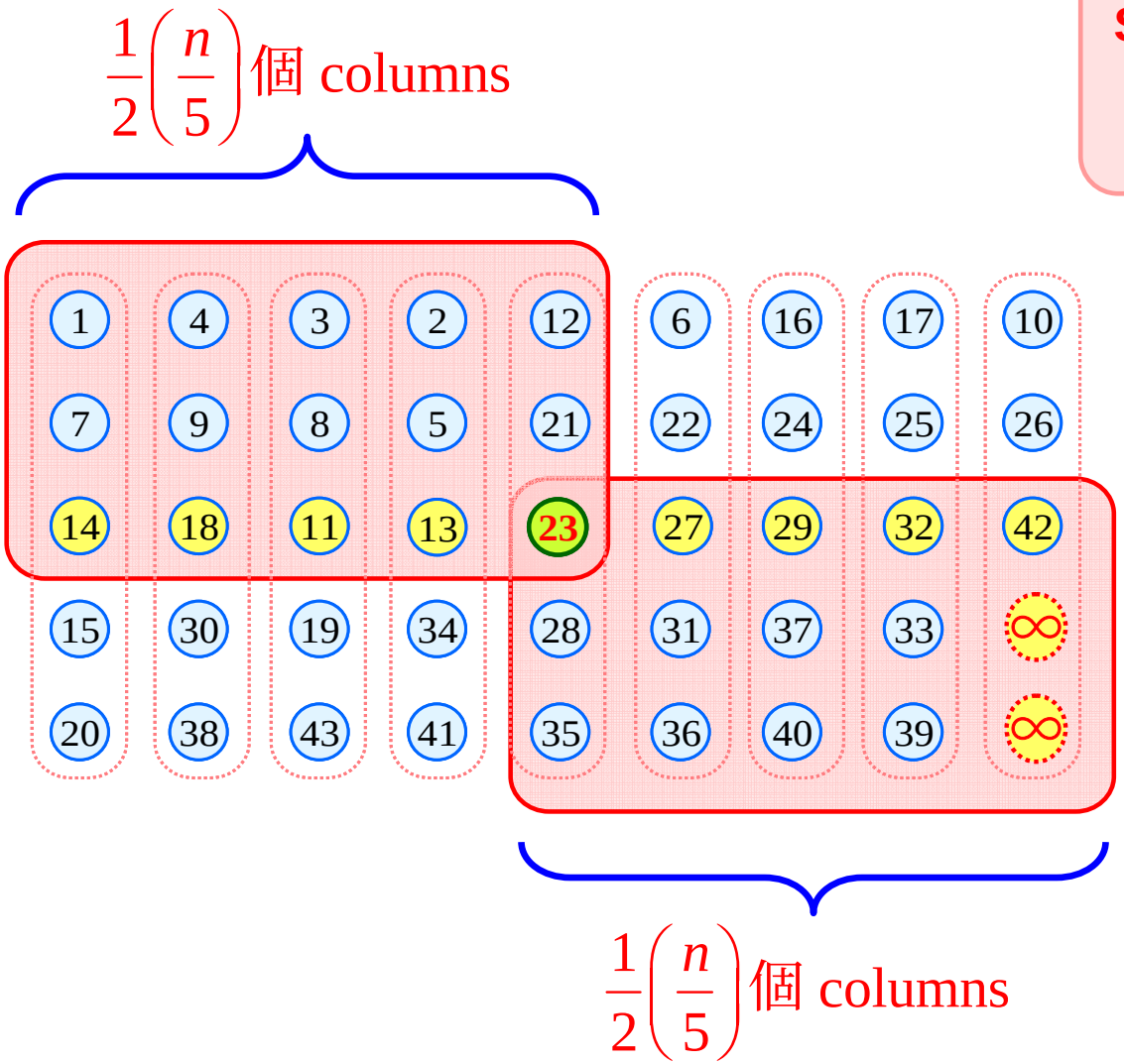
需時 $O(n)$

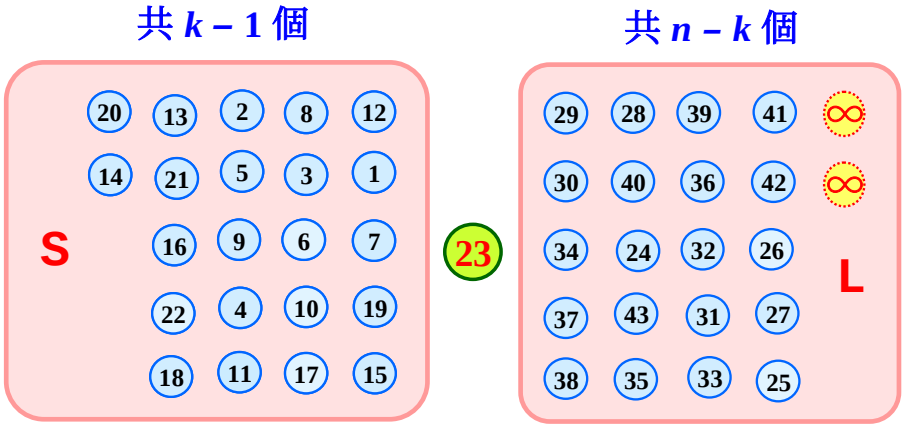
需時 $T(\max\{k-1, n-k\})$

所以執行時間為 $T(n) = T(n/5) + T\left(\max\{k-1, n-k\}\right) + O(n)$

$\max\{k-1, n-k\} \le 7n/10$

比 **MoM** 小 至少 **3** ✕ **(*n*/5)** ✕ **(1/2)** 個
→ 所以比 **MoM** 大 至多 **7*n*/10** 個



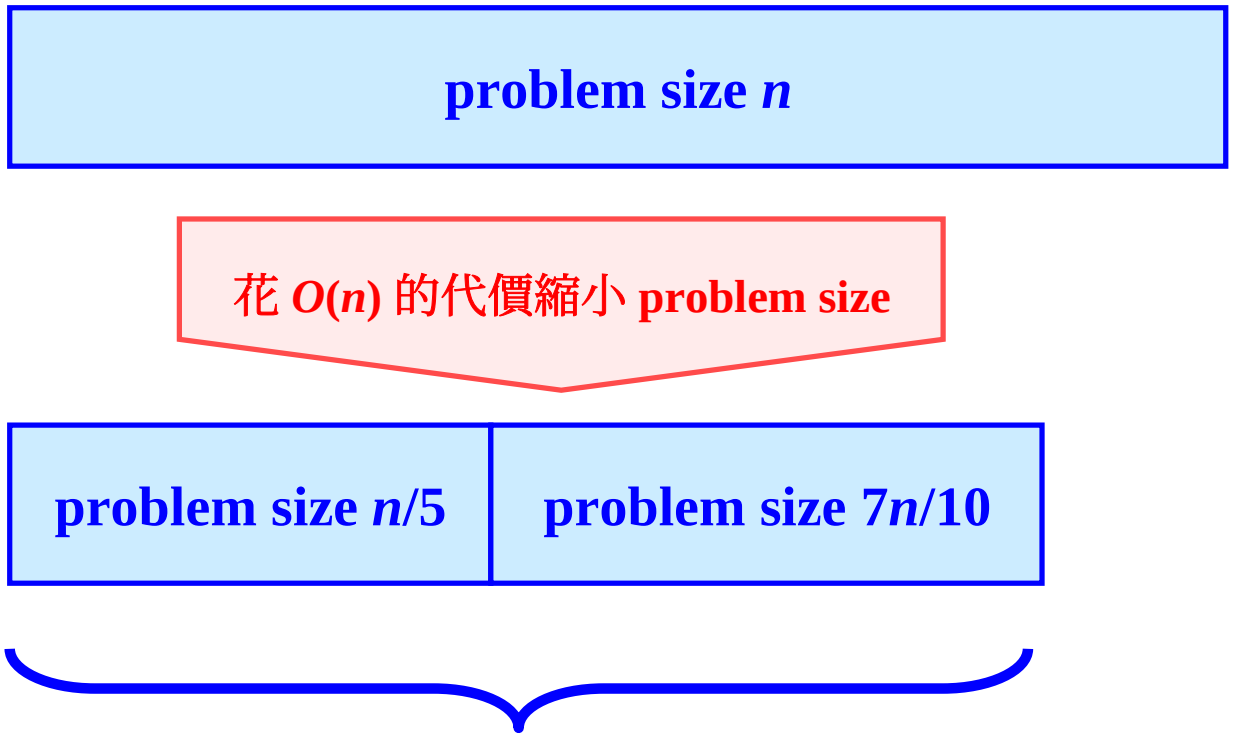


比 **MoM** 大 至少 **3** ✕ **(*n*/5)** ✕ **(1/2)** 個
→ 所以比 **MoM** 小至多 **7*n*/10** 個

Observations

所以執行時間為 $T(n) = T(n/5) + T\left(\max\{k-1, n-k\}\right) + O(n)$

$$\leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$



Running Time Analysis

已知 $T(n) \le T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$

這意味著：存在 $\hat{c}$，使得當 $n \ge n_0$ 時，$T(n) \le T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \hat{c}n$

我們猜測 $T(n) = O(n)$

$\Rightarrow$ 我們採用 substitution 的技巧來推導看看，

看是否存在常數 c，使得當 $n \ge n_1$ 時，我們有 $T(n) \le cn$。

假如 $T(n) \le cn$ 是正確的，那麼將此一不等式代入上述遞迴式

我們有 $T(n) \le \frac{cn}{5} + \frac{7cn}{10} + \hat{c}n = \frac{9cn}{10} + \hat{c}n = \left(\frac{9}{10}c + \hat{c}\right)n$

要讓 $T(n) = \left(\frac{9}{10}c + \hat{c}\right)n \le cn$，我們必須讓 $\frac{9}{10}c + \hat{c} \le c$

因此當 $c \ge 10\hat{c}$ 時，$T(n) \le cn$ 成立。 故得證。

Optimal Quick Sort

```
optimal_quicksort(A) {
  if (length[A] == 0)
    return;
  x = select(A, (length[A])/2 );
  // x 為 array A 的中位數;
  S:={(y 屬於 A) and (y ≤ x)};
  L:={(z 屬於 A) and (z > x)};
  optimal_quicksort(S);
  print x;
  optimal_quicksort(L);
}
```

使用 **linear-time `select` algorithm**
我們有 **optimal 的 quick sort**

所以 optimal quick sort 的執行時間為

$$T(n) = T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + O(n) \le 2T(n/2) + O(n) = O(n \log n)$$