Algorithms
Chapter 2 Getting Started

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Insertion sort** 插入性排序
- Analyzing algorithms 分析演算法(1)正確性(2)時間複雜度
- Designing algorithms 設計演算法(1)遞增法 (2)分別擊破法

The purpose of this chapter

- Start using frameworks for describing and analyzing algorithms. (1)描述(2)分析演算法
- Examine two algorithms for sorting: insertion sort and merge sort. 以兩個演算法為例：insertion sort , merge sort
- Learn how to prove the correctness of an algorithm. 證明演算法的正確性
- Begin using asymptotic notation to express running-time analysis. 使用漸進式符號描述時間複雜度(O, θ, Ω)
- Learn the technique of "divide and conquer" in the context of merge sort. 學習 divide and conquer 的技巧

Algorithm 精確的計算過程 some value input $\xrightarrow{\text{Algorithm}}$ some value output

- **Algorithm**: a well-defined computational procedure that takes some value as input and produces some value as output.
- Major concerns: 兩個重點
  - Correctness 正確性
  - Time complexity 時間複雜度
- For example: The sorting problem
  - **Input**: A sequence of *n* numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
  - **Output**: A permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \ldots \le a'_n$. 由小到大排序
  - Given the input sequence 31, 41, 59, 26, 41, 58, a sorting algorithm returns as output the sequence 26, 31, 41, 41, 58, 59.

Insertion sort 插入性排序

- Insertion sort: an efficient algorithm for sorting a small number of elements. 在個數不多時是個有效率的演算法

INSERTION-SORT(A)

把第j張放到適當位置

for $j \leftarrow 2$ **to** *length*[*A*] 從第2張開始排序

    do $key \leftarrow A[j]$ 將第j張存起來→第j張是將要排序者

        /* Insert $A[j]$ into the sorted sequence $A[1\ldots j-1]$*/

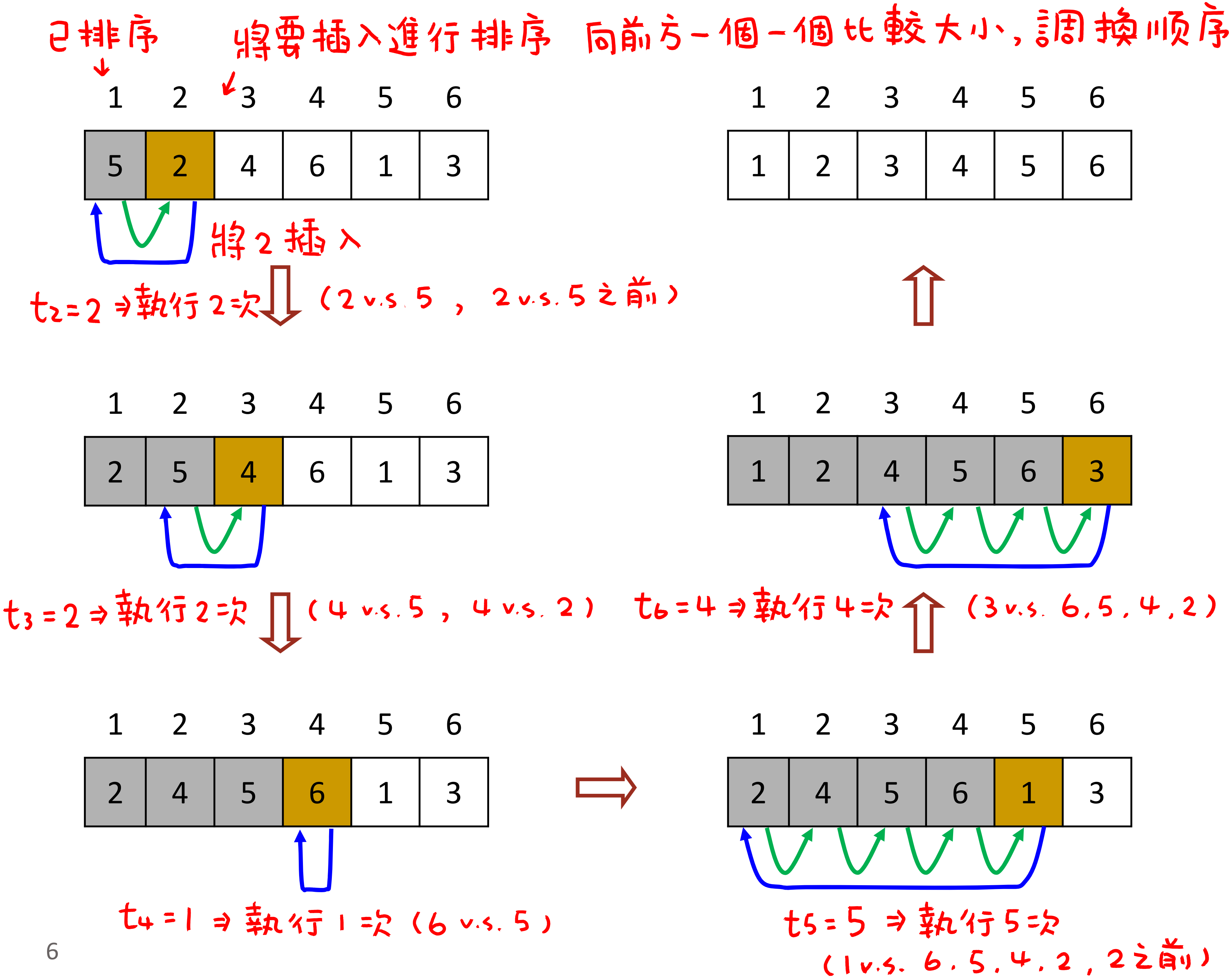
        $i \leftarrow j-1$ 從前一張開始問

        while $i > 0$ and $A[i] > key$ 當前方還有人，而且比key大

            do $A[i+1] \leftarrow A[i]$ 往右移一個位置

                $i \leftarrow i-1$ 繼續向前一個比較

        $A[i+1] \leftarrow key$ 前方沒人或是沒有比較大


已排序
將要插入進行排序
向前方一個一個比較大小，調換順序
1 2 3 4 5 6
5 2 4 6 1 3
將2插入
$t_2=2$ ⇒ 執行2次 (2 v.s. 5 , 2 v.s. 5之前)
1 2 3 4 5 6
2 5 4 6 1 3
$t_3=2$ ⇒ 執行2次 (4 v.s. 5 , 4 v.s. 2)
1 2 3 4 5 6
2 4 5 6 1 3
$t_4=1$ ⇒ 執行1次 (6 v.s. 5)
1 2 3 4 5 6
2 4 5 6 1 3
$t_5=5$ ⇒ 執行5次
(1 v.s. 6, 5, 4, 2, 2之前)
1 2 3 4 5 6
1 2 4 5 6 3
$t_6=4$ ⇒ 執行4次 (3 v.s. 6, 5, 4, 2)
1 2 3 4 5 6
1 2 3 4 5 6

Loop invariant for proving correctness

- We may use **loop invariants** to prove the correctness.

  用 loop invariants 去證明正確性(共3個步驟)

(I) - **Initialization**: It is true before the first iteration of the loop.

  在第一個 loop 前正確(在執行 loop 前,此演算法是正確的)

(II) - **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.

  如果在第i個之前正確,則在經過第i個 loop 後依然正確

(III) - **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

  當 loop 結束之後,可用 invariant 的性質證明正確性

- Using loop invariants is like mathematical induction.

用 loop invariant 證明 insertion sort 是正確的

Correctness of INSERTION-SORT

經過 loop 後仍保持的性質

- **Loop invariant**: At the start of each iteration of the **for** loop of lines 1-8, the subarray $A[1\ldots j-1]$ consists of the elements originally in $A[1\ldots j-1]$ but in sorted order. (執行第 j 個 for loop 前，前 j-1 個已排序好)

(I) - **Initialization**: Before the first iteration, $j = 2$. $A[1]$ is trivially sorted.
  j=2，只有一個，本身已排序好 Ex: P.6 圖一的 5

(II) - **Maintenance**: Note that the body of the outer **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, ..., and so on by one position to the right until the proper position for $A[j]$ is found.
  在 for loop 中，我們將 A[j-1] A[j-2] A[j-3]… 往右移，且為A[j]找到適當位置

(III) - **Termination**: The outer **for** loop ends when j exceeds n, i.e., when $j = n + 1$. Then, $A[1\ldots n]$ is sorted.
  當 loop 結束後 j=n+1，所以根據 loop invariant，A[1…n] 是排序好的
  (在執行 j 個 for loop，前 j-1 個已排序)

Outline

- Insertion sort
- **Analyzing algorithms**
- Designing algorithms

Time complexity of INSERTION-SORT

- Let t_j be the number of times the while loop test for value j.

INSERTION-SORT(A)	花費時間 cost	執行次數 times	
for $j \leftarrow 2$ **to** *length*[A]	c_1	n	$j=2 \sim n$ 成功, $j=n+1$ 失敗 $(n-1)+1$
do *key* $\leftarrow A[j]$	c_2	$n-1$	for loop 執行 $n-1$ 次
/* Insert $A[j]$ into the sorted sequence $A[1\ldots j-1]$. */	0	$n-1$	註解不需花費
$i \leftarrow j-1$	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	$\sum_{j=2}^{n} (t_j - 1)$	移動次數比測試少1
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$	
$A[i+1] \leftarrow key$	c_8	$n-1$	將第j個放好

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

* t_j : 放入第j個所要測試的次數 (P.6)

Time complexity of INSERTION-SORT

(I) ▸ **Best-case**: The array is already sorted.最好的情況：已排序 Ex: 1, 2, 3, 4, 5, 6

- $t_2 = t_3 \ldots = t_n = 1$. 每人只測試一次
- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$

  $= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$
- A **linear function** of n. n的線性函數（一次式）

(II) ▸ **Worst-case**: The array is in reverse sorted order.最差的情況：倒序

Ex: 6, 5, 4, 3, 2, 1

- $t_2 = 2, t_3 = 3, \ldots, t_n = n$. 每個人都要問到盡頭
- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right)$

  $+ c_6 \frac{n(n-1)}{2} + c_7 \frac{n(n-1)}{2} + c_8(n-1)$

  $= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n$ ⇒ 成長速率相差很大

  $- (c_2 + c_4 + c_5 + c_8)$
- A **quadratic function** of n. n的二次函數（平方式）

	linear	quadratic
	$5n$	$5n^2$
10	50	500
100	500	50000
10000	50000	500000000

分析時有3種情形：(I) worse case (II) best case (III) average case
→通常只關心最差的

Worst-case and average-case analysis

- We shall usually concentrate on finding only the worst-case
  - The worst-case running time gives us a guarantee that the algorithm will never take any longer. 給我們一種保證
  - For some algorithms, the worst case occurs fairly often. 最差情形常發生
  - The "average case" is often roughly as bad as the worst case.
    average case 與 worst case 差不多
- For example:
  - Consider the insertion sort, on average, we check half of the subarray $A[1\ldots j-1]$, so $t_j = j/2$. 如果每個人都只往前測試一半
  - The average-case running time is still a quadratic function of n.
    平均情形仍會是n的2次函數

Order of growth 成長速率為最要點

- Another abstraction to ease analysis and focus on the important features. 使用符號來幫助我們(1)更易分析(2)僅關心重點
- Look only at the leading term of the formula for running time.
  - (I) Drop lower-order terms. 只看最高項次，其餘項次不看
  - (II) Ignore the constant coefficient in the leading term. 最高項次係數也不要
- For example:
  - The worst-case running time of insertion sort is $an^2 + bn + c$. n的2次函數
  - Drop lower-order terms $\Rightarrow an^2$. 只看最高項
  - Ignore constant coefficient $\Rightarrow n^2$. 省去係數
  - We say that the running time is $\Theta(n^2)$ to capture the notion that the order of growth is n^2. 表示 insertion sort 的成長速度是 cn^2 (n^2的常數倍) c = 常數

Outline

- Insertion sort
- Analyzing algorithms
- **Designing algorithms**

Designing algorithms 設計演算法

- There are many ways to design algorithms. 設計演算法有很多技巧
- **Incremental**: 遞增法
  - For example of insertion sort, having sorted subarray $A[1...j-1]$ and then yielding the sorted array $A[1...j]$.
- **Divide and conquer** 分別擊破法 (3 steps)
  - (I) **Divide** the problem into a number of subproblems. 分成性質相同的子問題
  - (II) **Conquer** the subproblems by solving them recursively.
    - If the subproblems sizes are small enough, just solve them in a straightforward manner. 用遞迴方法解決子問題，如果問題夠小，用暴力法
  - (III) **Combine** the subproblem solutions to give a solution to the original problem. 將子問題的答案合併，形成原問題的答案

Merge sort 用 devide and conquer 解決排序

- **Divide** by splitting into two subarrays $A[p\ldots q]$ and $A[q+1\ldots r]$, where q is the halfway point of $A[p\ldots r]$.
  將陣列分成兩個大小相等的子陣列
- **Conquer** by recursively sorting the two subarrays $A[p\ldots q]$ and $A[q+1\ldots r]$. 分別排序兩個子陣列
- **Combine** by merging the two sorted subsequences to produce the sorted answer.
  將兩個已排序的子陣列合併，形成一排好的陣列

最左邊 最右邊

MERGE-SORT (A, p, r)

if $p < r$ 陣列個數大於1	//Check for base case	
then $q \leftarrow \lfloor (p+r)/2 \rfloor$	//Divide	中間那一個
MERGE-SORT(A, p, q)	//Conquer	排左邊
MERGE-SORT$(A, q+1, r)$	//Conquer	排右邊
MERGE(A, p, q, r)	//Combine	合併

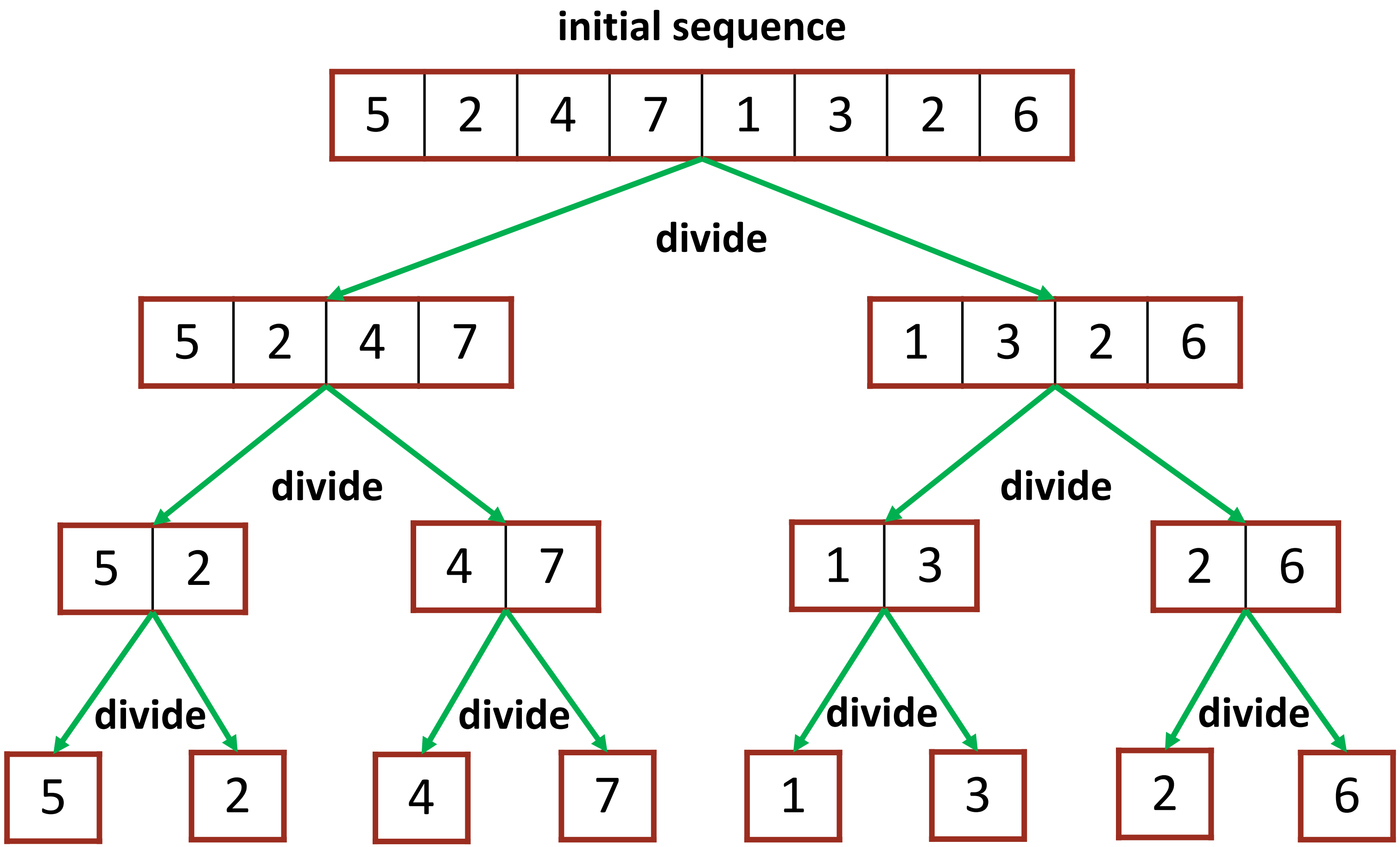
用遞迴的方式（直接遞迴）

輸入陣列 最右方為n

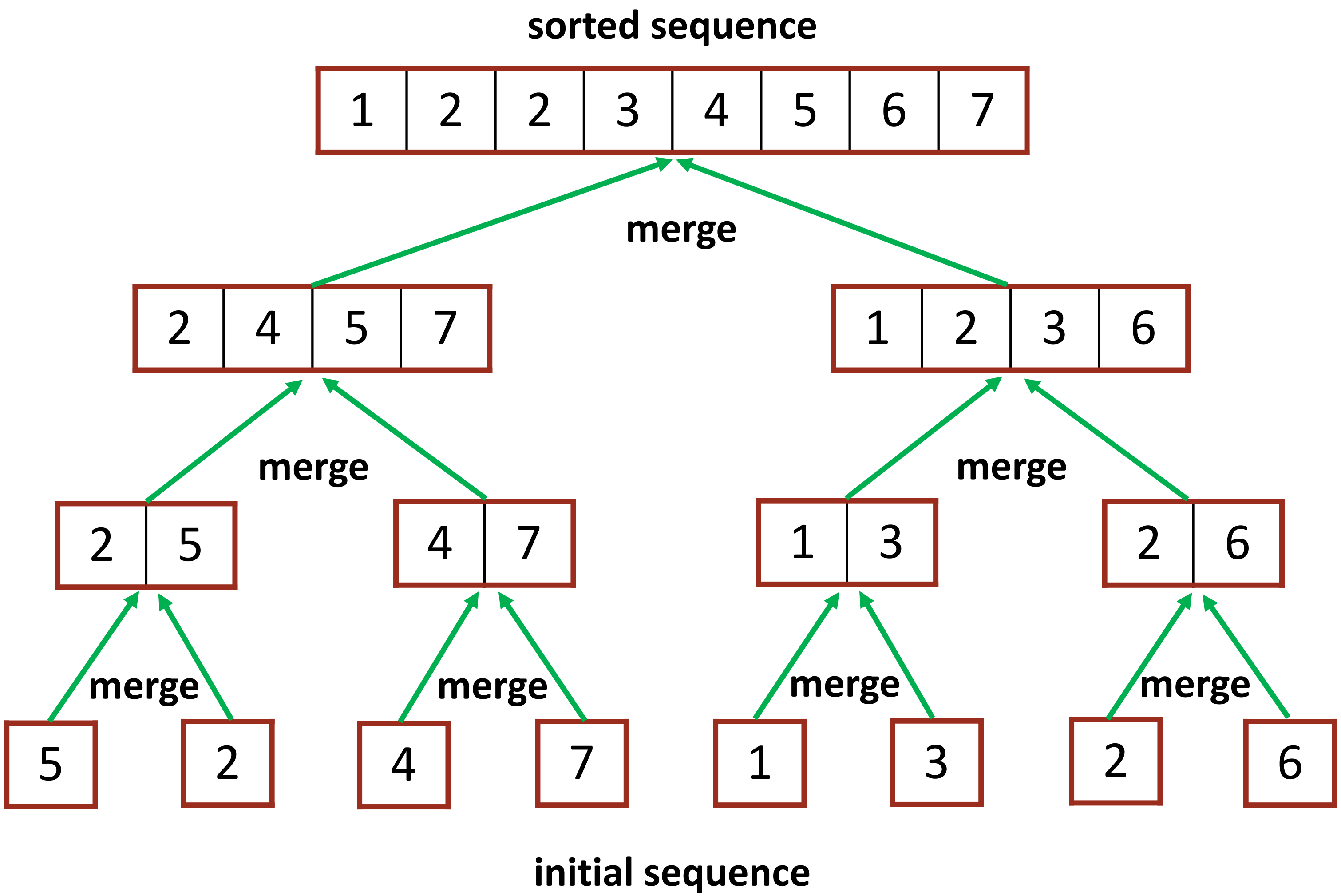
- **Initial call**: MERGE-SORT$(A, 1, n)$

最左方為1

An example for MERGE-SORT



An example for MERGE-SORT



Linear-time merging

MERGE (*A*, *p*, *q*, *r*)

```
n1 ← q – p + 1
n2 ← r – q
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
```

Lines 1–3: $\Theta(1)$

n_1：陣列1的個數

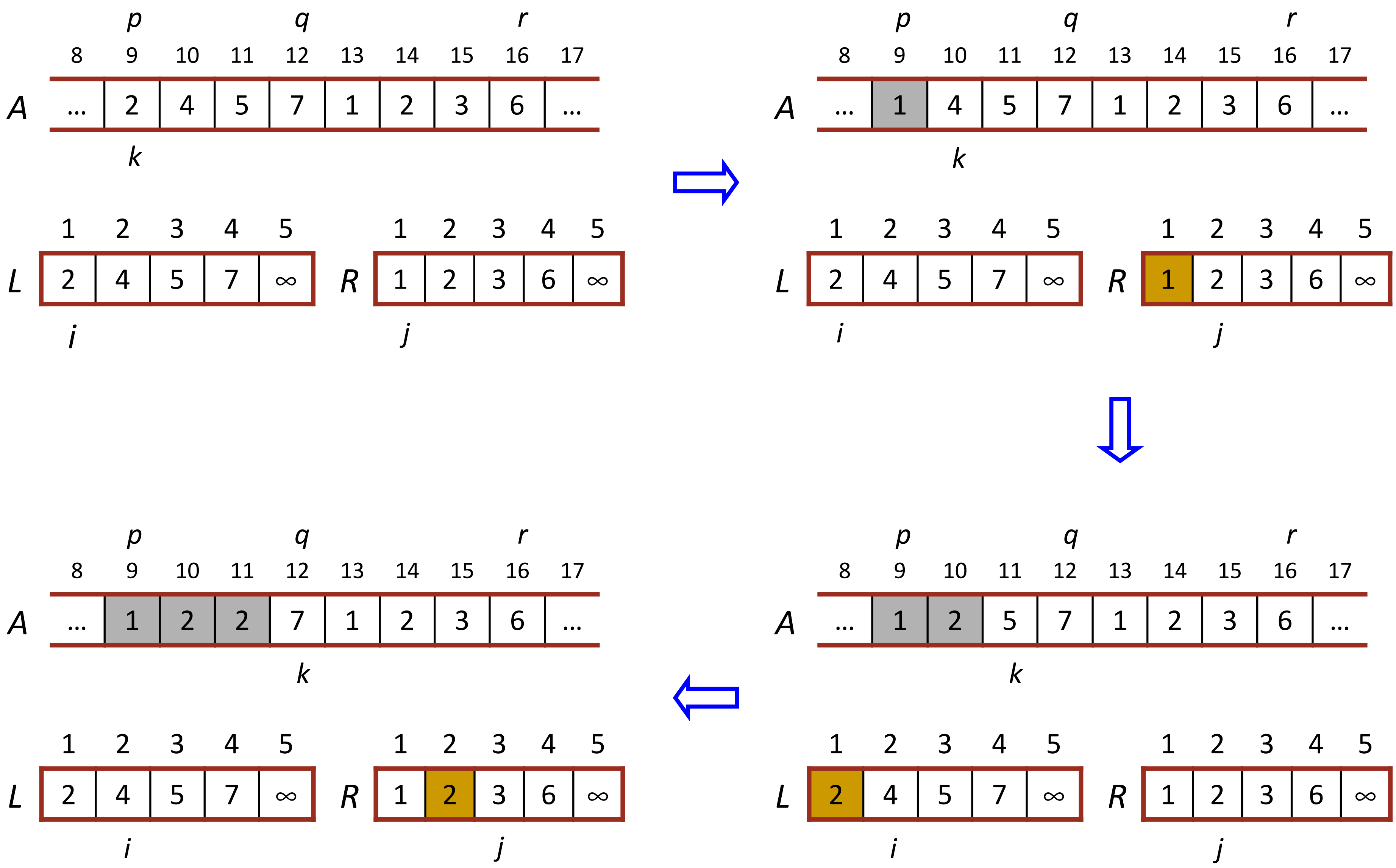
n_2：陣列2的個數

配製2個子陣列 $\underset{L}{\boxed{n_1+1}}$ $\underset{R}{\boxed{n_2+1}}$

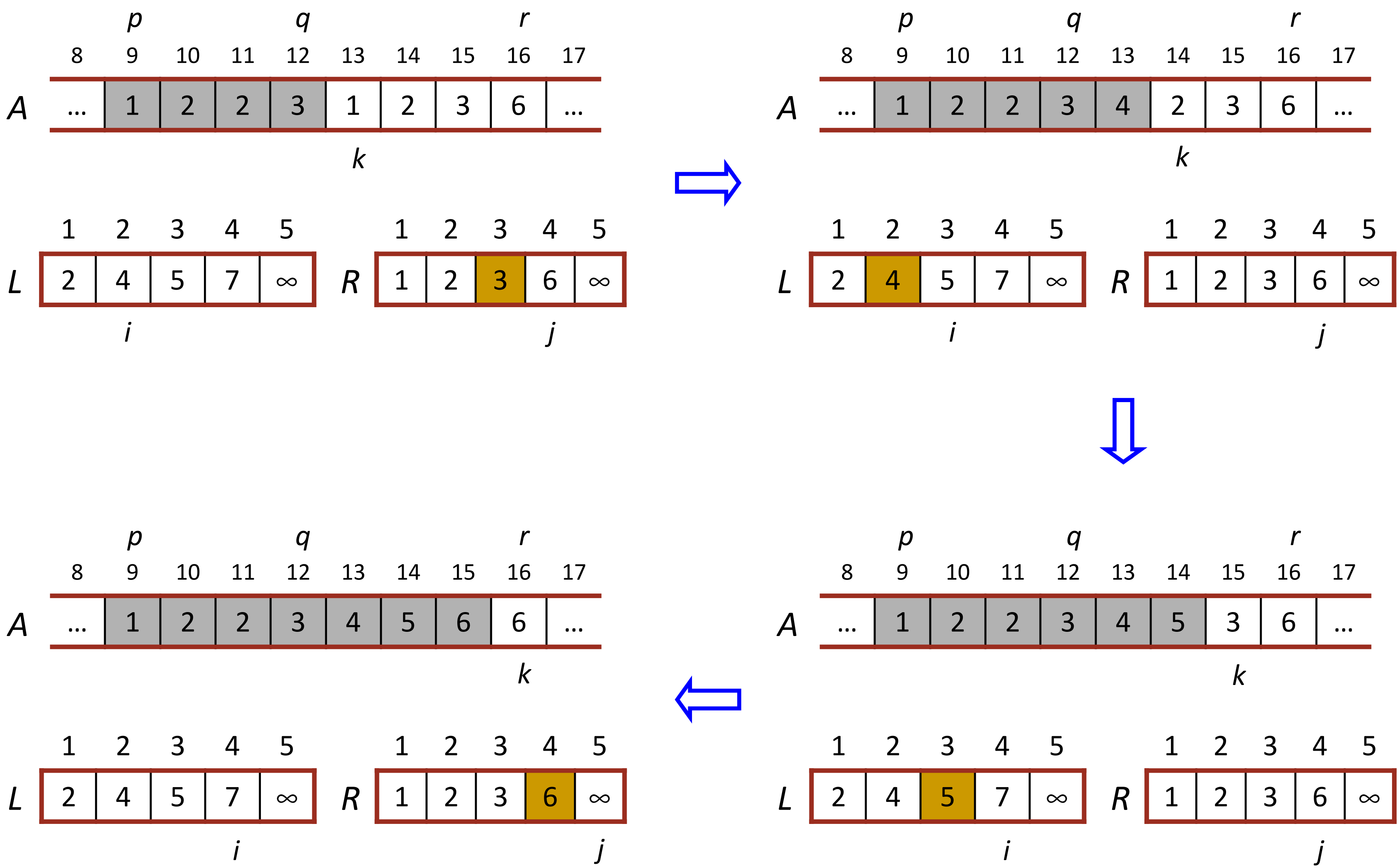
Lines 4–7: 將陣列1複製到L，陣列2複製到R $\Theta(n_1+n_2)$

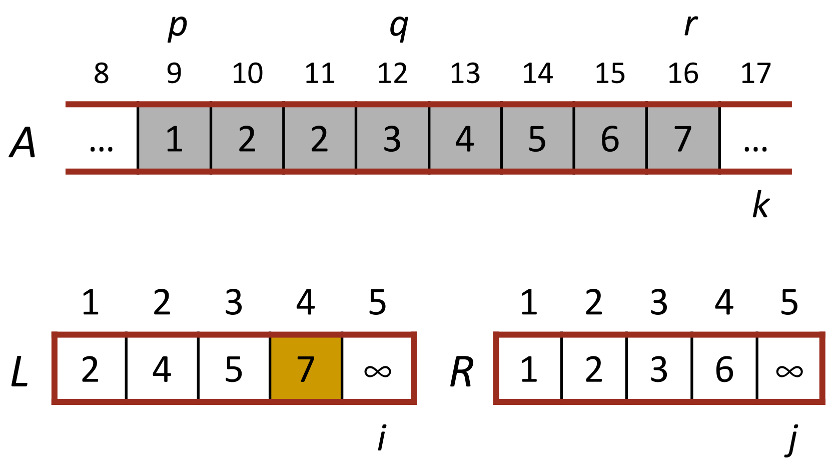
Lines 8–9: $\Theta(1)$ 將L的第 n_1+1 放 ∞，將R的第 n_2+1 放 ∞（作為邊界條件）

Lines 10–15: $\Theta(n_1+n_2)$ 比大小，將小的放入k的位置



p q r
8 9 10 11 12 13 14 15 16 17
A ... 2 4 5 7 1 2 3 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
p q r
8 9 10 11 12 13 14 15 16 17
A ... 1 4 5 7 1 2 3 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
p q r
8 9 10 11 12 13 14 15 16 17
A ... 1 2 5 7 1 2 3 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
p q r
8 9 10 11 12 13 14 15 16 17
A ... 1 2 2 7 1 2 3 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


p q r
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
p q r
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
p q r
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
p q r
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


p
q
r
8 9 10 11 12 13 14 15 16 17
A
… 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L
2 4 5 7 ∞
i
1 2 3 4 5
R
1 2 3 6 ∞
j

分析divid and conquer的時間複雜度

Analyzing divide-and-conquer algorithms

- Use a **recurrence equation** to describe the running time of a divide-and-conquer algorithm.

常數時間

$$T(n)=\begin{cases}\theta(1) & \text{if } n\le c,\\ aT(n/b)+D(n)+C(n) & \text{otherwise.}\end{cases}$$

大小為n個所需時間

a個大小為n/b的子問題(解決子問題所需時間)

  - $T(n)$ = the running time on a problem of size n.
  - If $n \le c$ for some constant c, the solution takes $\Theta(1)$ time.
  - We divide into a subproblems, each $1/b$ the size of the original.
  - $D(n)$ = the time to divide a size-n problem. 分割所需時間
  - $C(n)$ = the time to combine solutions. 合併所需時間

Analyzing merge sort$_{1/2}$

- For simplicity, assume that *n* is a power of 2.

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1, \\ 2T(n/2) + \theta(n) & \text{otherwise.} \end{cases}$$

  - The base case occurs when *n* = 1.
  - **Divide**: compute the middle of the subarray, $D(n) = \Theta(1)$.
  - **Conquer**: Recursively solve 2 subproblems, each of size $n/2$
    $\Rightarrow a = 2$ and $b = 2$.
  - **Combine**: MERGE on an *n*-element subarray takes $\Theta(n)$ time
    $\Rightarrow C(n) = \Theta(n)$.

$D(n) + C(n) = \theta(1) + \theta(n) = \theta(n)$

divide + merge
$\theta(1)$ $\theta(n)$

分成2個子問題,每個子問題的大小是原來的一半

Analyzing merge sort$_{2/2}$

共h層

第i層 $0 \quad 1 \quad 2 \quad \cdots \quad h-1$

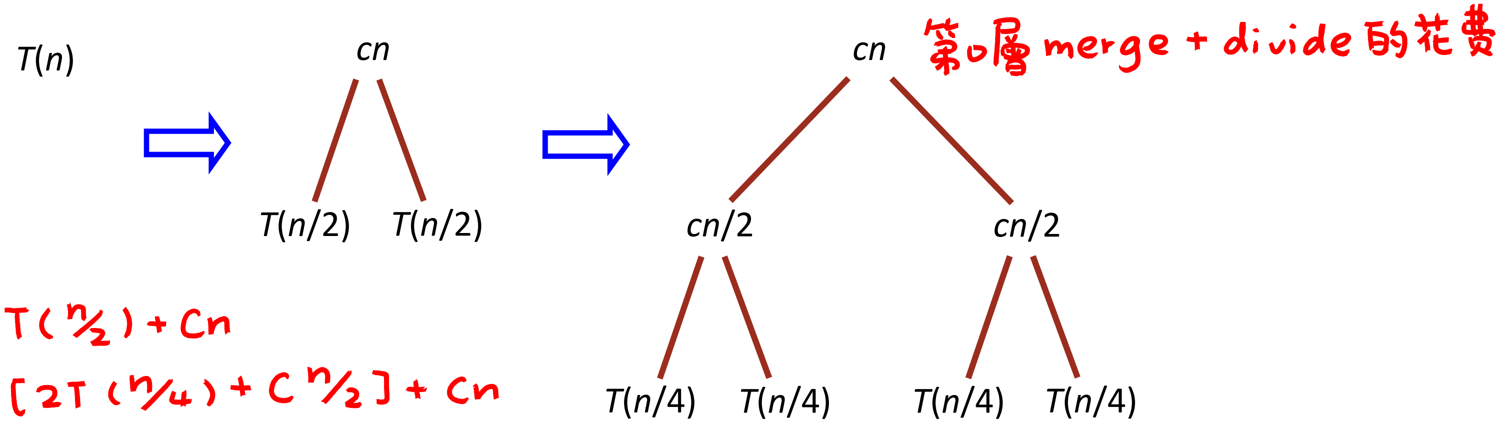
$n \quad \frac{n}{2} \quad \frac{n}{4} \quad \cdots \quad \frac{n}{2^{h-1}} = 1$

$2^{h-1} = n \Rightarrow h = \lg n + 1$

- Let *c* be a constant that describes
  - the running time for the base case
  - the time per array element for the divide and combine steps.
- Then, we can rewrite the recurrence as

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{otherwise.} \end{cases}$$

- The next slide shows successive expansions of the recurrence.
  - level i : 2^i nodes, each has a cost of $c(n/2^i)$.
    So, ith level has a cost of $2^i\, c(n/2^i) = cn$.
    node數 每個node的花费
  - At the bottom level, a tree with h levels has $2^{h-1} = n$ nodes.
    Therefore, $h = \lg n + 1$. (h: 層數)
  - The total cost is $cn(\lg n + 1) = cn\lg n + cn = \Theta(n\lg n)$.
    每一層所需花费



$$T(n) = 2T(n/2) + Cn$$
$$= 2[2T(n/4) + C\,n/2] + Cn$$
$$= 4T(n/4) + 2 \cdot \frac{cn}{2} + Cn$$