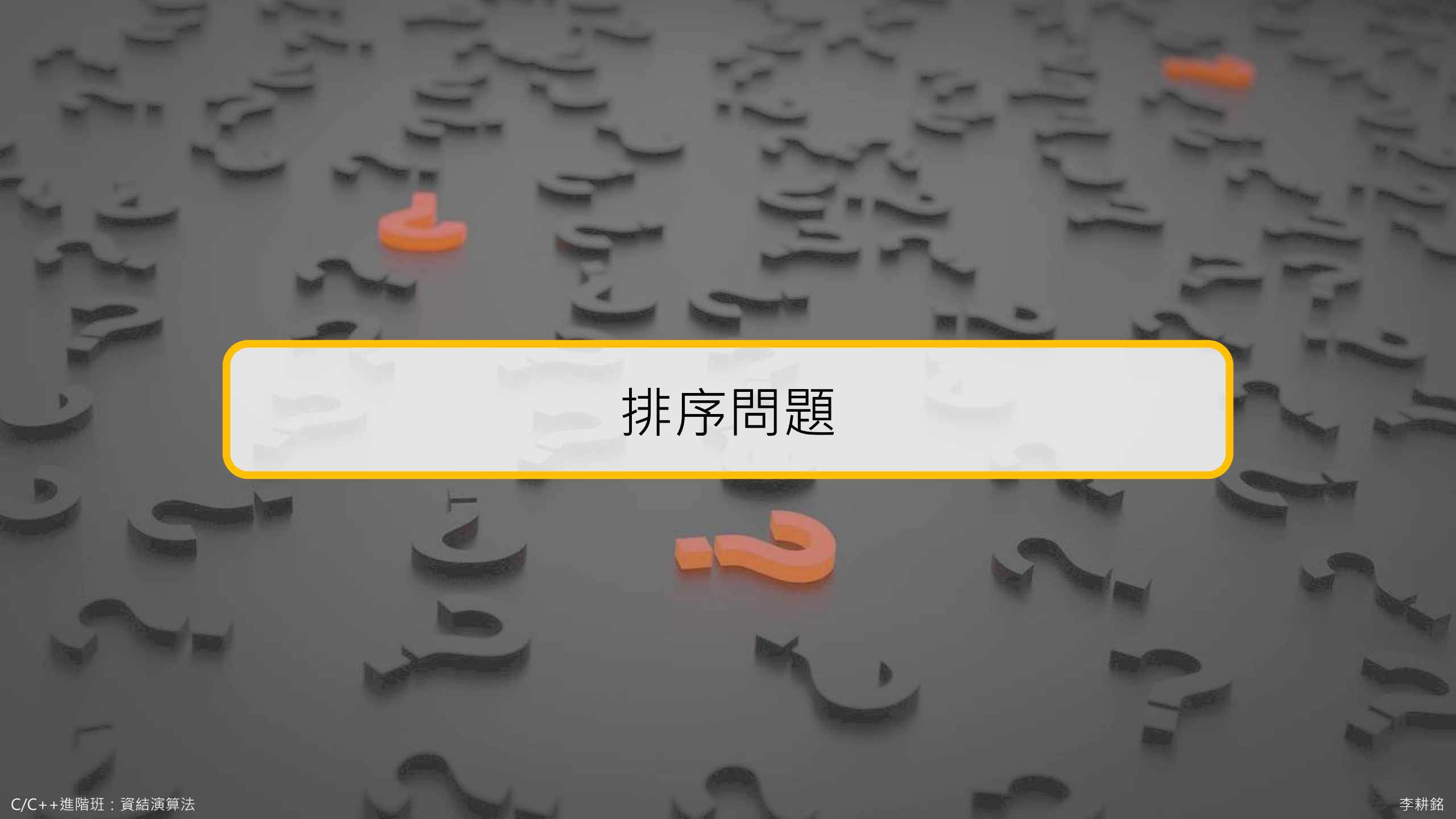
C/C++ 進階班
演算法

排序
(Sort)

李耕銘

課程大綱

- 排序問題
- 排序方式介紹
  1. 插入排序法 (insertion sort)
  2. 謝爾排序法 (shell sort)
  3. 選擇排序法 (selection sort)
  4. 泡沫排序法 (bubble sort)
  5. 合併排序法 (merge sort)
  6. 堆積排序法 (heap sort)
  7. 快速排序法 (quick sort)
  8. 桶排序法 (bucket sort)
  9. 基數排序法 (radix sort)
- 排序方式總結
- C++ STL 中的排序
- 實戰練習



排序問題

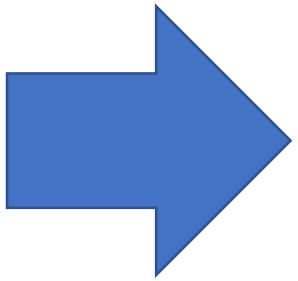
排序問題的基本原理

如果資料未事先經過排序或整理，日後要進行操作會非常麻煩

排序問題的基本原理

如果資料未事先經過排序或整理，日後要進行操作會非常麻煩

Name	Phone
Mick	34275229
David	43245832
Alexis	63498433
John	12312498
Rallod	54389232
Leo	23498534
Andrew	56908433



Name	Phone
Alexis	63498433
Andrew	56908433
David	43245832
John	12312498
Leo	23498534
Mick	34275229
Rallod	54389232

排序問題的基本原理

- 排序方法區分
  1. 內部排序 (internal sort)
     - ➢ 排序時的資料可以完全儲存在記憶體內
  2. 外部排序 (external sort)
     - ➢ 排序時的資料會用到外部的儲存器 (Ex: 硬碟)

排序問題的基本原理

- 排序方法區分
  1. 穩定排序 (stable sort)
     - ➢ 排序後，同樣的 key 的順序可以被保留
  2. 不穩定排序 (unstable sort)
     - ➢ 排序後，同樣的 key 的順序不能被保留

Key	Value
1	Mick
2	John
2	David
1	William
2	Rallod

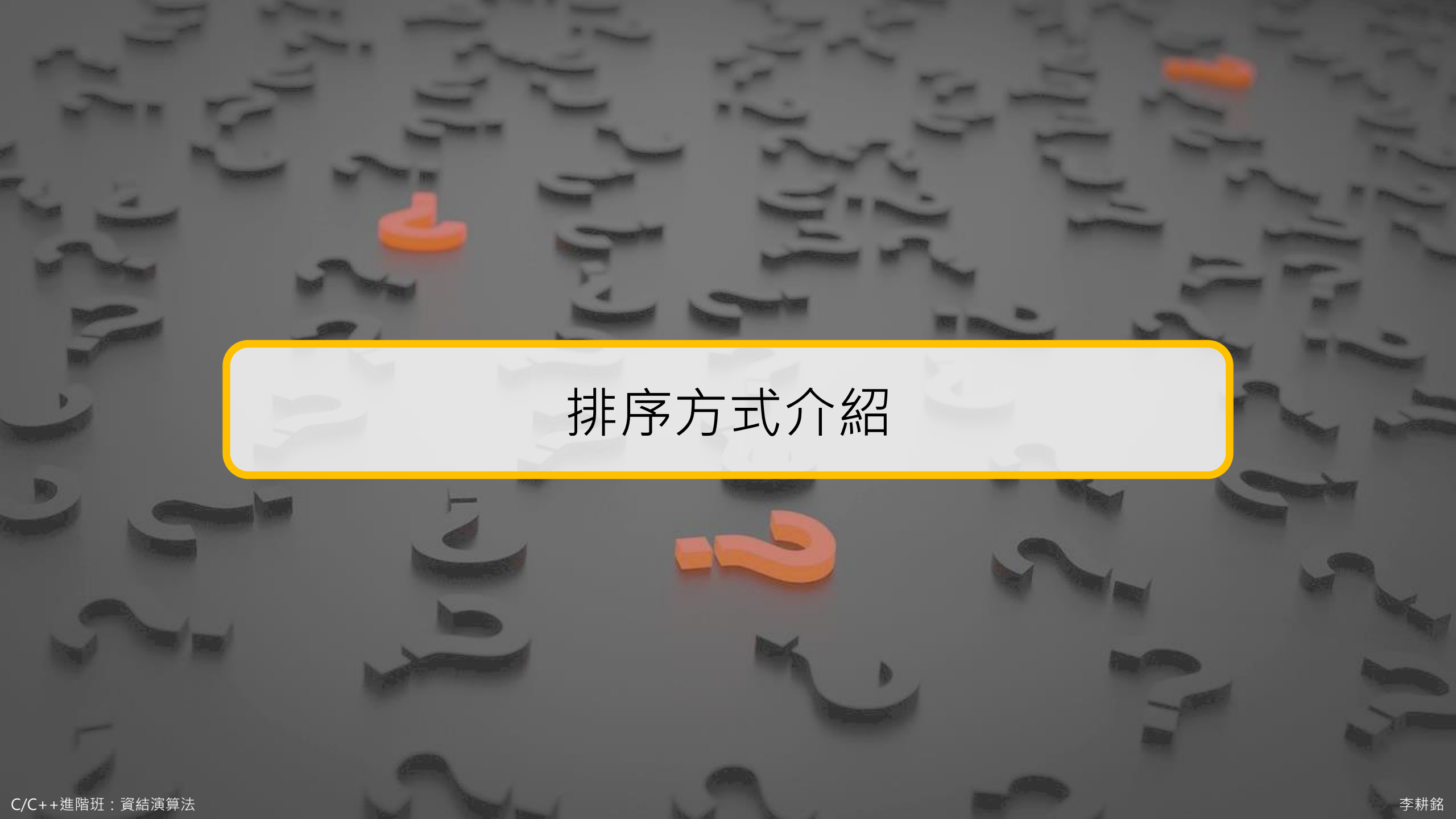
Origin Data

Key	Value
1	Mick
1	William
2	John
2	David
2	Rallod

Stable Sort

Key	Value
1	William
1	Mick
2	Rallod
2	John
2	David

Unstable Sort



排序方式介紹

Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

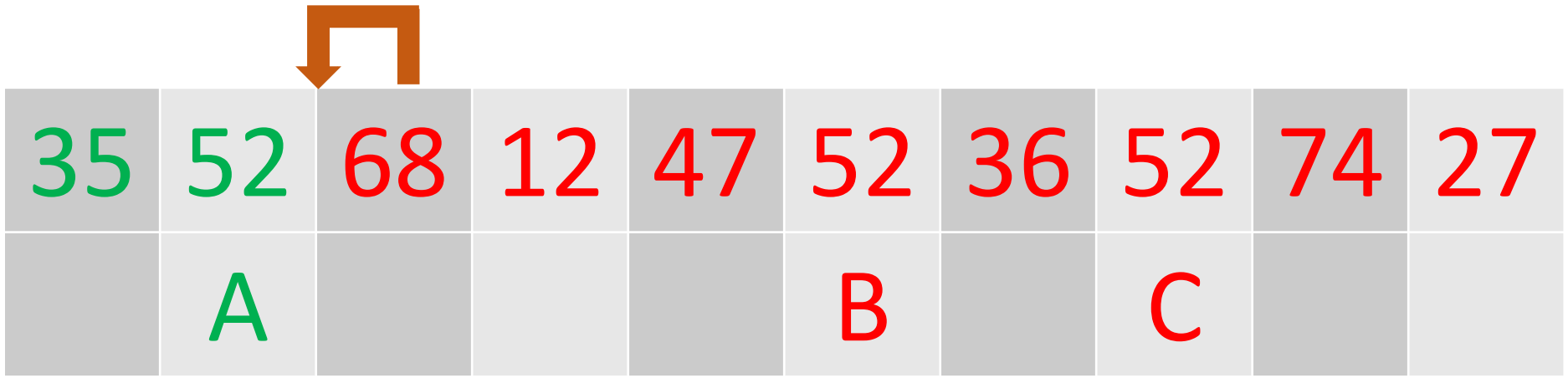
Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序



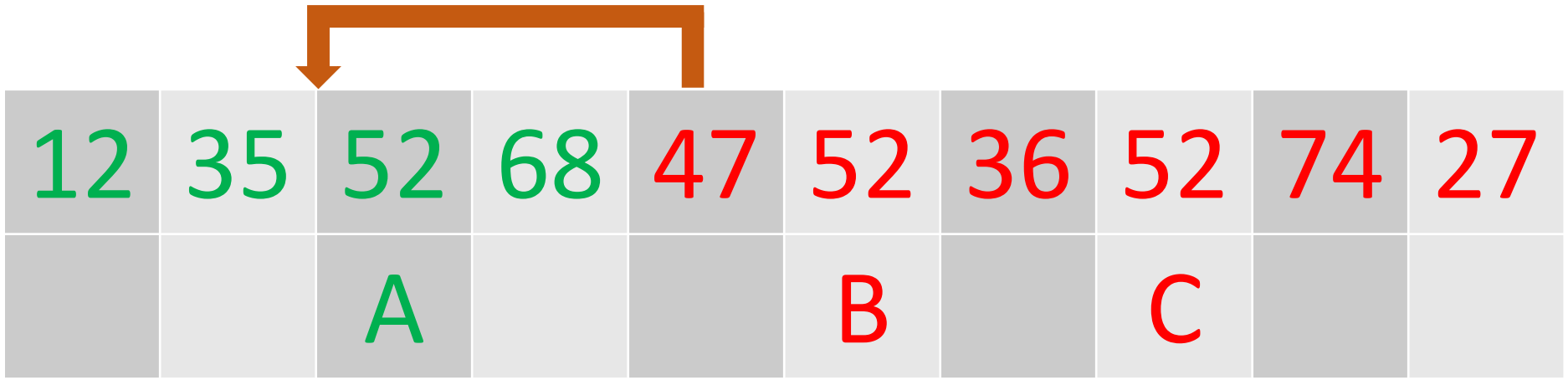
Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

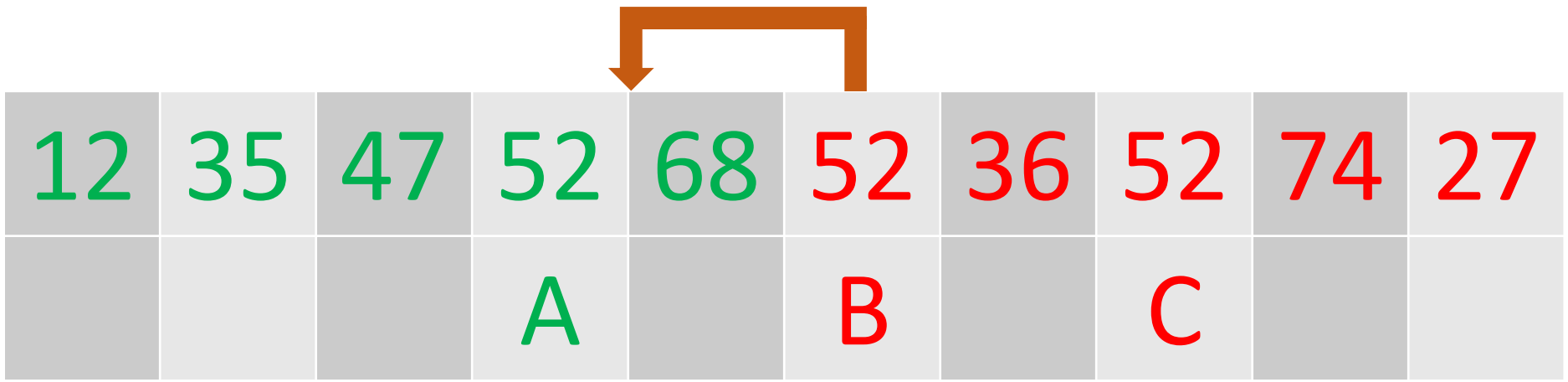
Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序



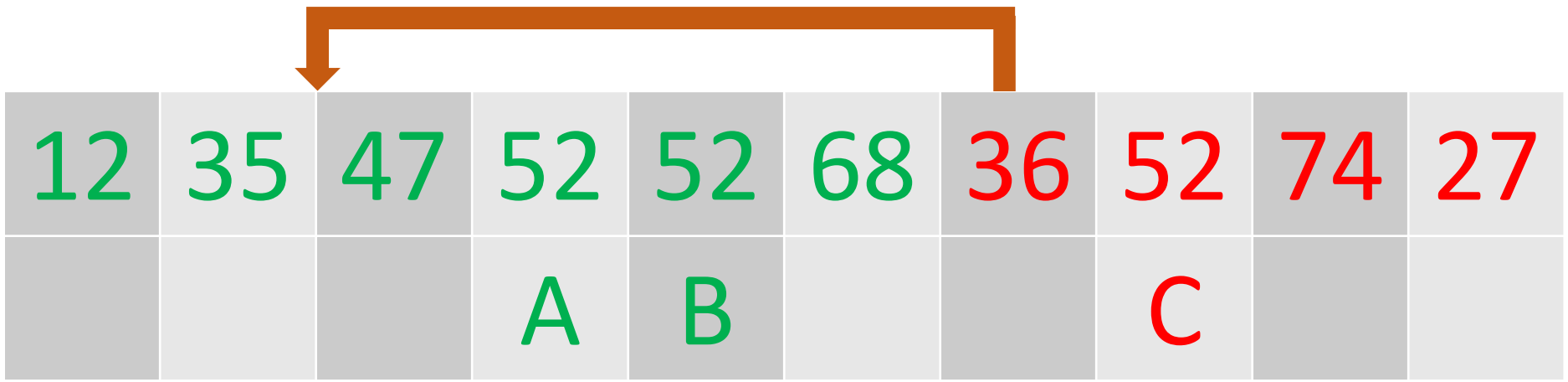
Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序



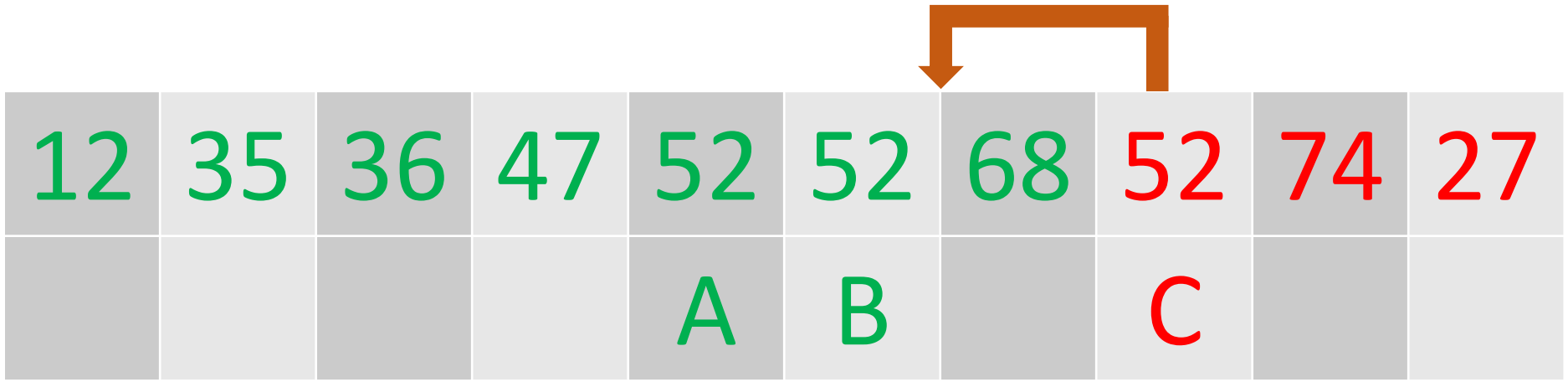
Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序



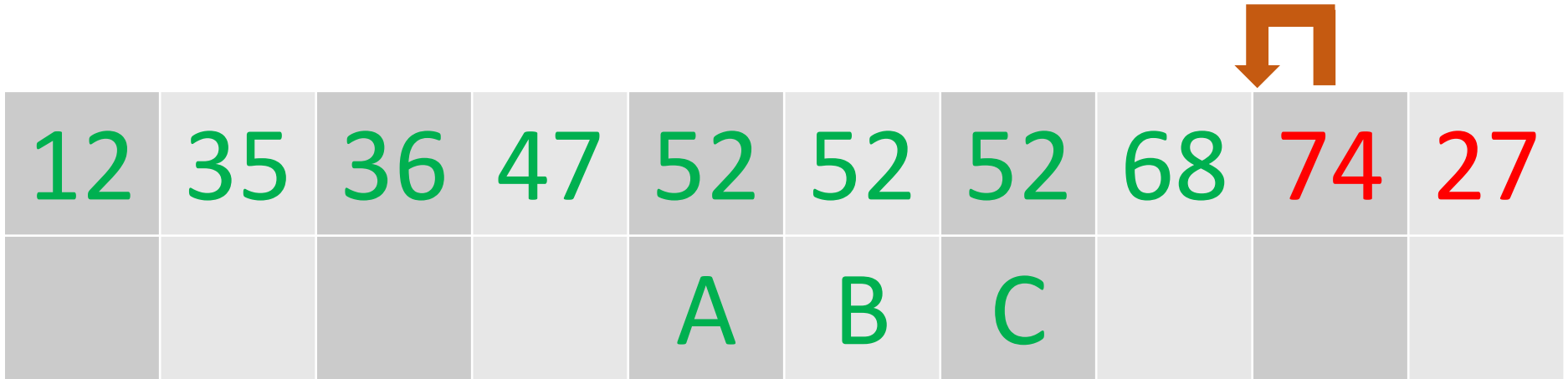
Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序



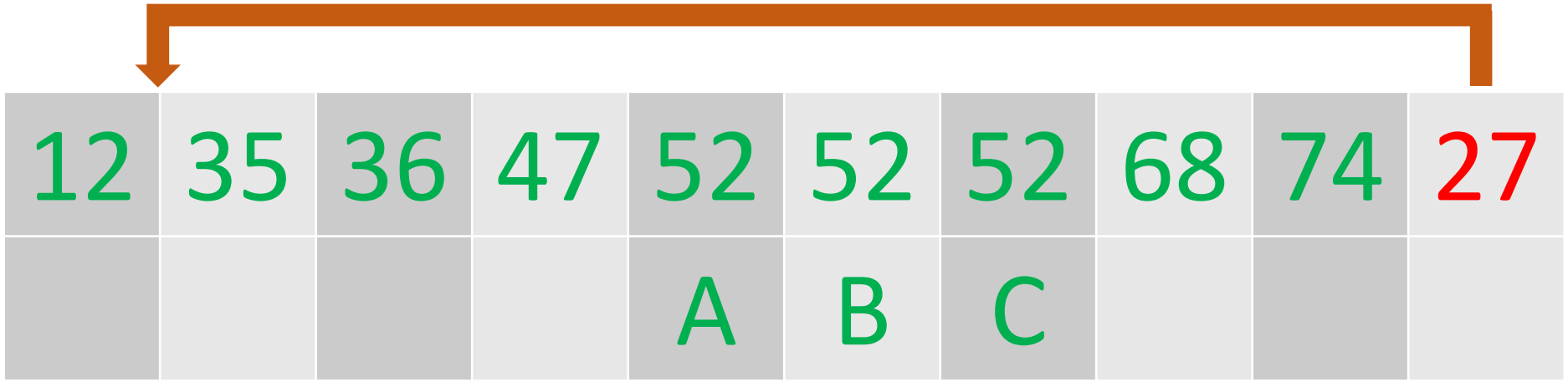
Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序



Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序



Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料插到對的位置
  - ➢ 屬於穩定排序、內部排序

12	27	35	36	47	52	52	52	68	74
					A	B	C		

Insertion Sort

- 不斷把資料插入到對的位置
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆資料
    - ✓ 插到對的位置
  - ➢ 屬於穩定排序、內部排序

	Complexity
Best	$O(n)$，comparsion $O(1)$，swap
Average	$O(n)$，comparsion $O(1)$，swap
Worst	$O(n^2)$，comparsion $O(n^2)$，swap
Memory	$O(n)$
Memory (Auxiliary)	$O(1)$
Stable	True

Example Code

Mission

實作 Insertion Sort

Shell Sort

- Insertion sort 的改進
  - Insertion Sort 對已經排好順序的資料效率高：O(n)
    - ✓ 但每次只能將移動一個數字到正確位置
  - Shell sort 把陣列分成 n/k 組，每組陣列有 k 筆資料
    - ✓ 間距(Gap) = n/k，間距通常從 $\frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8} \rightarrow \frac{n}{16} \ldots \rightarrow 1$

35	52	68	12	47	52	36	52	74	27
	A				B		C		

Shell Sort

35	52	68	12	47	52	36	52	74	27
	A				B		C		

間距(Gap) = 10/2 = 5
分成 5 組、每組 2 筆資料

組內排序

35	52	68	12	47
52	36	52	74	27

Shell Sort

35	36	52	12	27	52	52	68	74	47
		C			B	A			

間距(Gap) = 10/2 = 5
分成 5 組、每組 2 筆資料

組內排序

35	36	52	12	27
52	52	68	74	47

Shell Sort

35	36	52	12	27	52	52	68	74	47
		C			B	A			

間距(Gap) = 10/4 = 2
分成 2 組、每組 5 筆資料

35
52
27
52
74

36
12
52
68
47

Shell Sort

27	12	35	36	52	47	52	52	74	68
				C		A	B		

間距(Gap) = 10/4 = 2
分成 2 組、每組 5 筆資料

27
35
52
52
74

12
36
47
52
68

Shell Sort

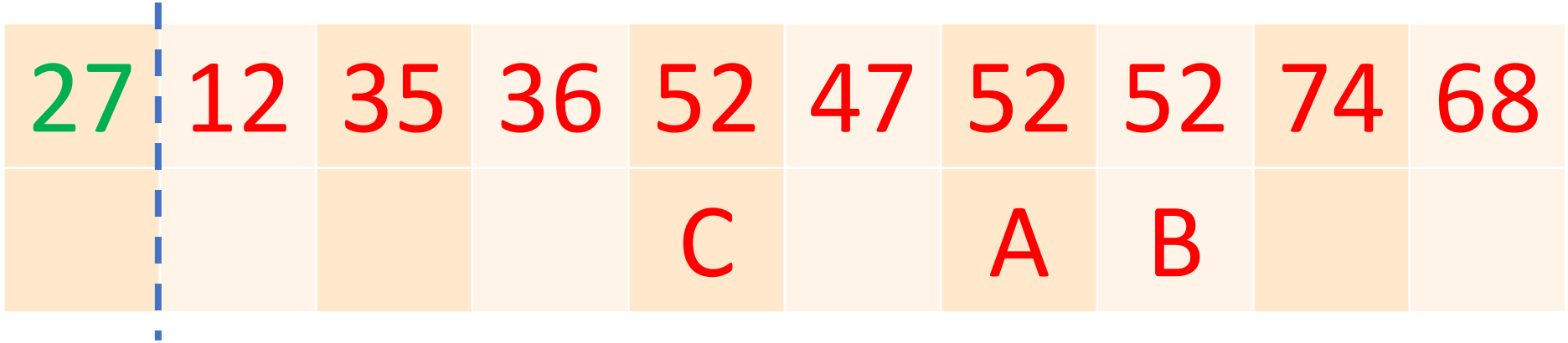
- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ n/k = 1

27	12	35	36	52	47	52	52	74	68
				C		A	B		

間距(Gap) = 10/10 = 1
分成 1 組、每組 10 筆資料

Shell Sort

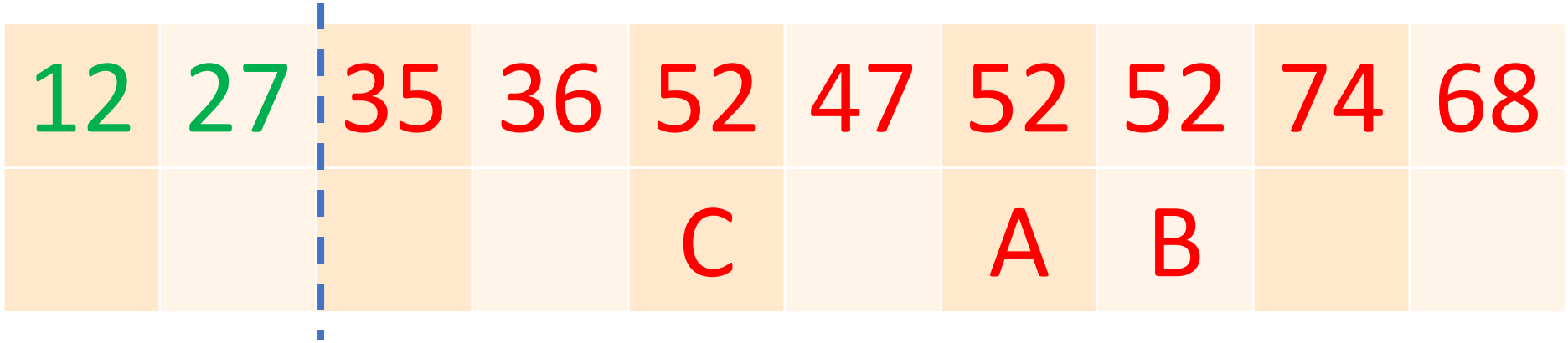
- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ K = 1



間距(Gap) = 10/10 = 1
分成 10 組、每組 1 筆資料

Shell Sort

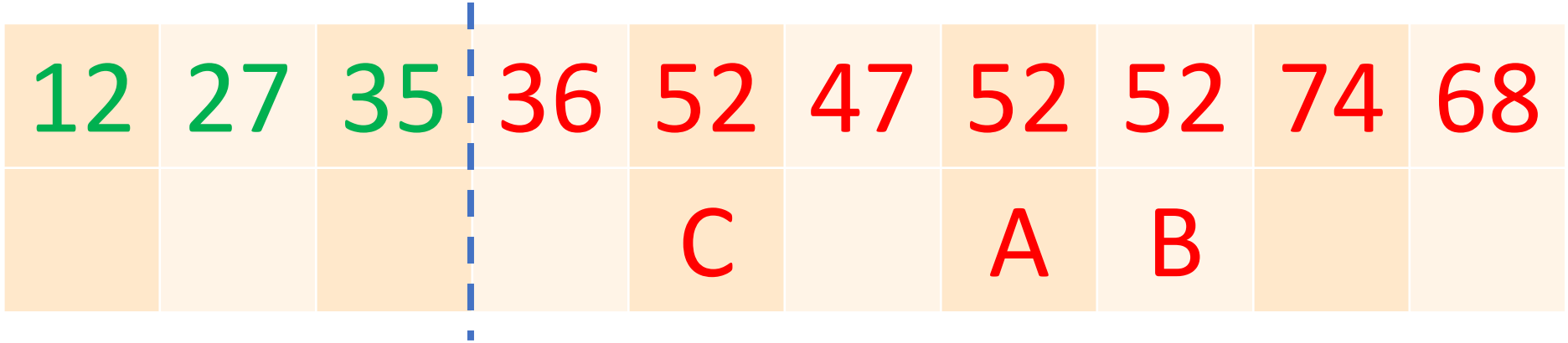
- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ K = 1



間距(Gap) = 10/10 = 1
分成 10 組、每組 1 筆資料

Shell Sort

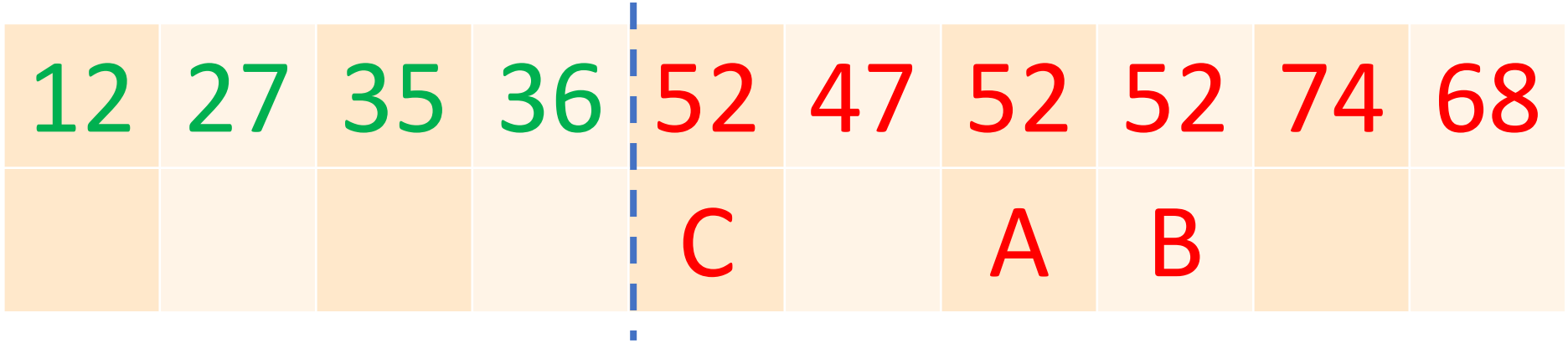
- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ K = 1



間距(Gap) = 10/10 = 1
分成 10 組、每組 1 筆資料

Shell Sort

- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ K = 1



間距(Gap) = 10/10 = 1
分成 10 組、每組 1 筆資料

Shell Sort

- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ K = 1

12	27	35	36	52	47	52	52	74	68
				C		A	B		

間距(Gap) = 10/10 = 1
分成 10 組、每組 1 筆資料

Shell Sort

- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ K = 1

12	27	35	36	47	52	52	52	74	68
					C	A	B		

間距(Gap) = 10/10 = 1
分成 10 組、每組 1 筆資料

Shell Sort

- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ K = 1

12	27	35	36	47	52	52	52	74	68
					C	A	B		

間距(Gap) = 10/10 = 1
分成 10 組、每組 1 筆資料

Shell Sort

- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ K = 1

12	27	35	36	47	52	52	52	74	68
					C	A	B		

間距(Gap) = 10/10 = 1
分成 10 組、每組 1 筆資料

Shell Sort

- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ K = 1

12	27	35	36	47	52	52	52	74	68
					C	A	B		

間距(Gap) = 10/10 = 1
分成 10 組、每組 1 筆資料

Shell Sort

- Shell Sort = Insertion sort 的改進
  - ➢ 最後一輪就是典型的 Insertion sort
    - ✓ K = 1

12	27	35	36	47	52	52	52	68	74
					C	A	B		

間距(Gap) = 10/10 = 1
分成 10 組、每組 1 筆資料

不穩定排序！

Shell Sort

- Insertion sort 的改進
  - 把陣列分成 n/k 組，從 len/2~1
    - ✓ 有 n/k 個子陣列要排序
    - ✓ 每個子陣列有 k 個資料
  - 經驗上，平均需要 $O(n^{1.25})$
  - 最壞狀況仍需 $O(n^2)$
    - ✓ 透過調整 K 可以壓到 $O(n^{1.5})$

	Complexity
Best	$O(n)$
Average	$O(n^{1.25})$，empirically
Worst	$O(n^2)$ can reduce to $O(n^{1.5})$
Memory	$O(n)$
Memory (Auxiliary)	$O(1)$
Stable	False

Example Code

Mission

實作 Shell Sort

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	52	68	35	47	52	36	52	74	27
	A				B		C		

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	52	68	35	47	52	36	52	74	27
	A				B		C		

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	68	35	47	52	36	52	74	52
					B		C		A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	68	35	47	52	36	52	74	52
					B		C		A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	68	47	52	36	52	74	52
					B		C		A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	68	47	52	36	52	74	52
					B		C		A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	68	52	74	52
					B		C		A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	68	52	74	52
					B		C		A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	68	52	74	52
					B		C		A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	68	52	74	52
					B		C		A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	68	52	74	52
					B		C		A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	68	52	74	52
					B		C		A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	52	68	74	52
					B	C			A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	52	68	74	52
					B	C			A

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	52	52	74	68
					B	C	A		

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	52	52	74	68
					B	C	A		

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	52	52	68	74
					B	C	A		

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	52	52	68	74
					B	C	A		

Selection Sort

- 不斷挑出未排列中，最大/最小的那個數字
  - ➢ 把陣列分成已排序、未排序兩組
    - ✓ 依序從未排序那組抓一筆最大/最小資料插到已排列組
  - ➢ 屬於不穩定排序、內部排序

12	27	35	36	47	52	52	52	68	74
					B	C	A		

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	27	35	36	47	52	52	52	68	74
					B	C	A		

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	35	52	68	47	52	36	52	74	27
		A			B		C		

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	27	35	52	68	47	52	36	52	74
			A			B		C	

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	27	35	52	68	47	52	36	52	74
			A			B		C	

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	27	35	36	52	68	47	52	52	74
				A			B	C	

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	27	35	36	47	52	68	52	52	74
					A		B	C	

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	27	35	36	47	52	68	52	52	74
					A		B	C	

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	27	35	36	47	52	52	68	52	74
					A	B		C	

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	27	35	36	47	52	52	52	68	74
					A	B	C		

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	27	35	36	47	52	52	52	68	74
					A	B	C		

Selection Sort

- Selection Sort 可以穩定嗎？
  - 可以，把 swap 改成 insert
  - 通常用 linked list 來讓 selection sort 達成穩定排序

12	27	35	36	47	52	52	52	68	74
					A	B	C		

Selection Sort

- 不斷選一筆最大/最小值出來
  - 把陣列分成已排序、未排序兩組
    - ✓ 從未排序那組抓最大/最小資料
    - ✓ 插到最左/最右的位置
  - (可以是)穩定排序、內部排序

	Complexity
Best	$O(n^2)$，comparsion $O(1)$，swap
Average	$O(n^2)$，comparsion $O(n)$，swap
Worst	$O(n^2)$，comparsion $O(n)$，swap
Memory	$O(n)$
Memory (Auxiliary)	$O(1)$
Stable	False (Array) True (Linked List)

Example Code

Mission

實作 Selection Sort

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	68	12	47	52	36	52	74	27
	A				B		C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	68	47	52	36	52	74	27
	A				B		C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	68	47	52	36	52	74	27
	A				B		C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	47	68	52	36	52	74	27
	A				B		C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	47	68	52	36	52	74	27
	A				B		C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	47	52	68	36	52	74	27
	A			B			C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	47	52	68	36	52	74	27
	A			B			C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	47	52	36	68	52	74	27
	A			B			C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	47	52	36	68	52	74	27
	A			B			C		

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	47	52	36	52	68	74	27
	A			B		C			

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	47	52	36	52	68	74	27
	A			B		C			

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序

35	52	12	47	52	36	52	68	74	27
	A			B		C			

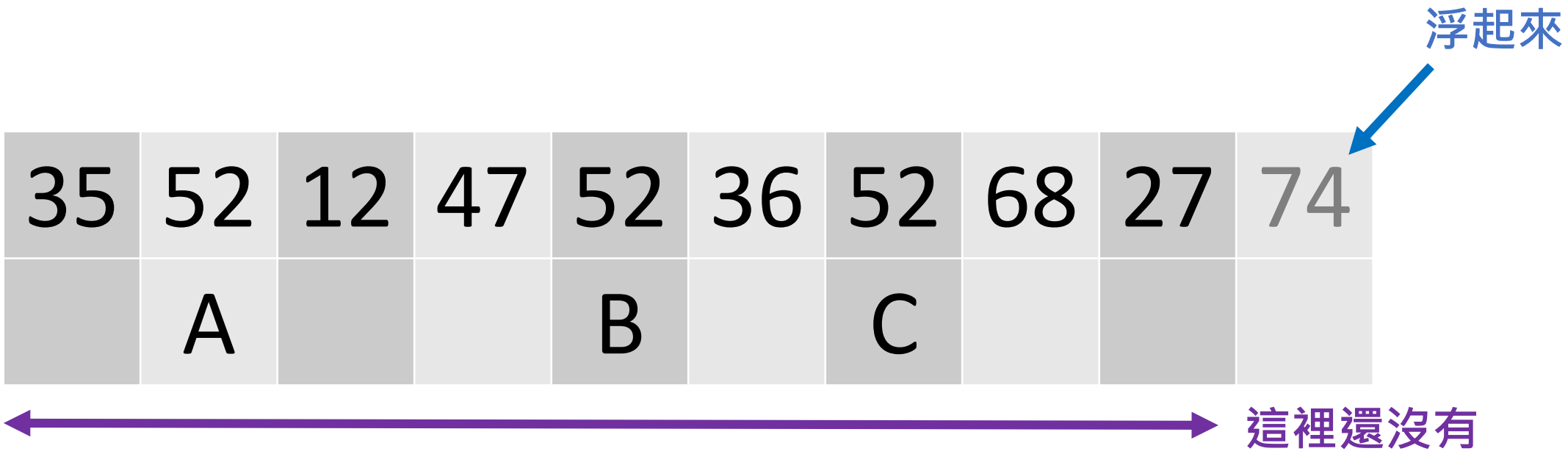
Bubble Sort

- 依序倆倆比較
  - 不符合規範的時候就進行對調 (swap)
  - 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - 屬於穩定排序、內部排序

35	52	12	47	52	36	52	68	27	74
	A			B		C			

Bubble Sort

- 依序倆倆比較
  - ➢ 每輪比較完至少有一筆最大/小資料可以排列至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 需重複 N-1 輪



Bubble Sort

```
bubble_sort(int data[],int len)
1  for i=0 ~ len-2                    } O(n^2)
2      for j= 0 ~ len-i-2:
3          if data[j] > data[j+1]:
4              tmp = data[j];
5              data[j] = data[j+1];
6              data[j+1] = tmp;
```

	Complexity
Best	$O(n^2)$
Average	$O(n^2)$
Worst	$O(n^2)$
Memory	$O(n)$
Memory (Auxiliary)	$O(1)$
Stable	True

Bubble Sort

- Bubble sort 的改進
  - 假設給定已排序好的陣列
  - 若該輪沒有進行任何 swap，則可以立刻終止迴圈

Bubble Sort

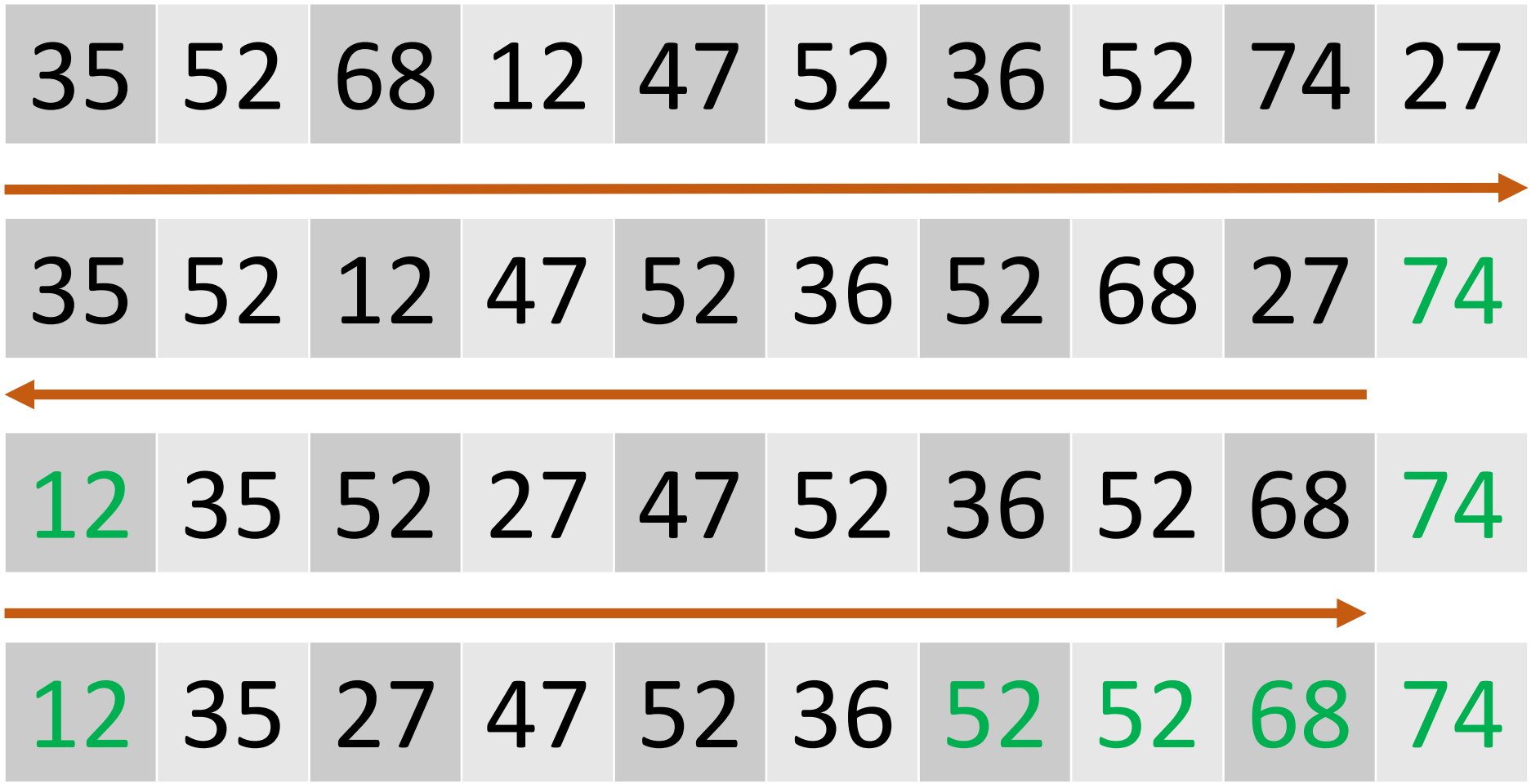
- Bubble sort 的改進
  - 假設給定一起始陣列內容如下
  - 右邊五個已排序完成，sort 過程中不會發生任何變化
  - 記錄下最後改變的位置即可

Shaker Sort

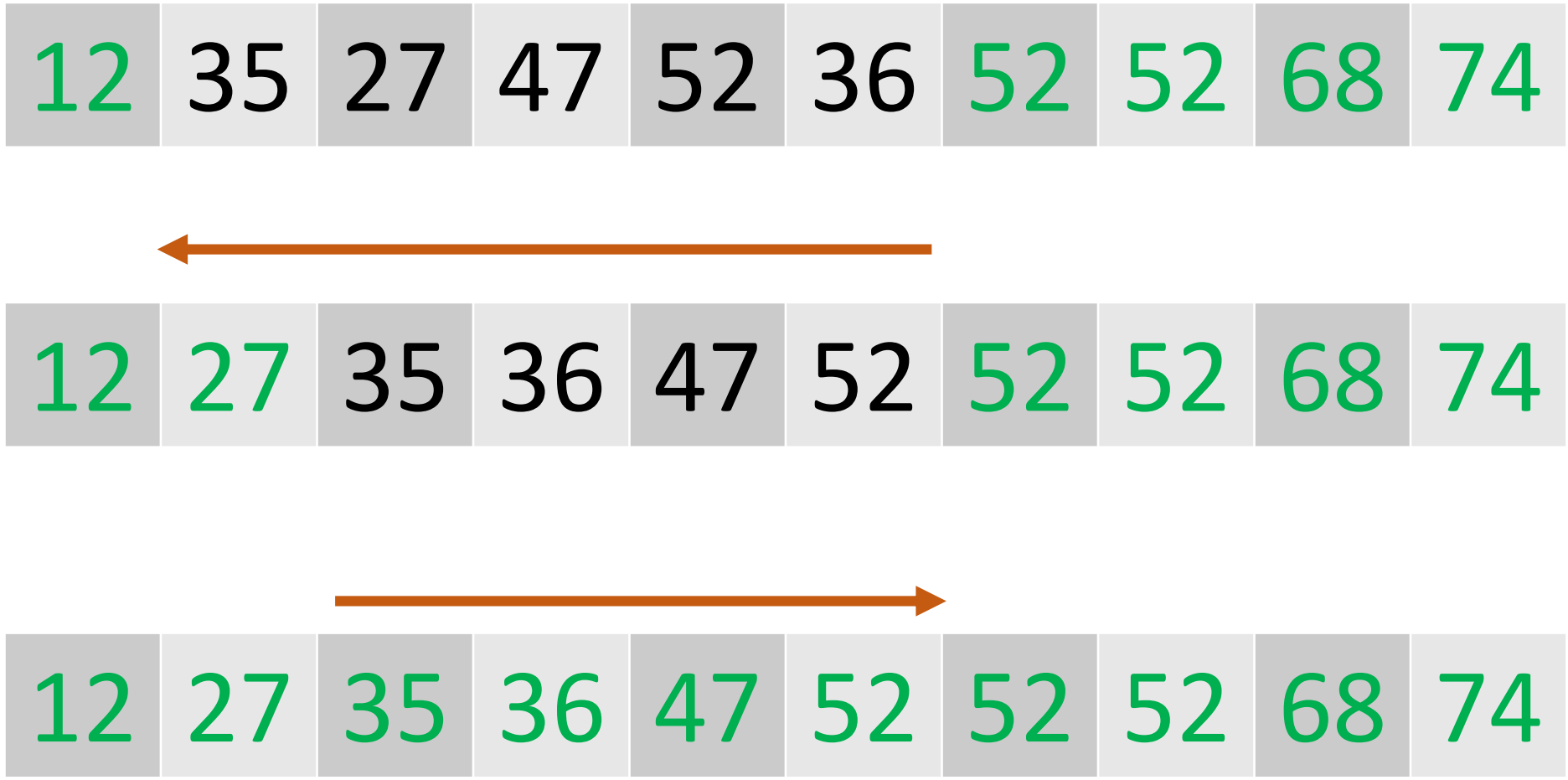
- Bubble sort 的改進
  - 但有可能已排序好的部分是在陣列的左邊
  - 解決方式：雙向的 bubble sort
  - 把陣列分成已排序好、未排序好兩組，未排序好在中間
    - 由 left、right 紀錄未排序好的區間
  - 每一輪做完就記錄下最後改變的位置

35	52	68	12	47	52	36	52	74	27

Shaker Sort


35 52 68 12 47 52 36 52 74 27
35 52 12 47 52 36 52 68 27 74
12 35 52 27 47 52 36 52 68 74
12 35 27 47 52 36 52 52 68 74

Shaker Sort


12 35 27 47 52 36 52 52 68 74
12 27 35 36 47 52 52 52 68 74
12 27 35 36 47 52 52 52 68 74

Bubble Sort

- 依序倆倆比較
  - ➢ 不符合規範的時候就進行對調 (swap)
  - ➢ 每輪比較完至少有一筆資料可以排至正確位置
    - ✓ 像泡泡一樣浮起來
  - ➢ 屬於穩定排序、內部排序
  - ➢ 有許多改良方式，都在 Bubble sort 大家族裡

	Complexity
Best	$\underline{\boldsymbol{O(n)}}$
Average	$O(n^2)$
Worst	$O(n^2)$
Memory	$O(n)$
Memory (Auxiliary)	$O(1)$
Stable	True

Example Code

Mission

實作 Bubble Sort

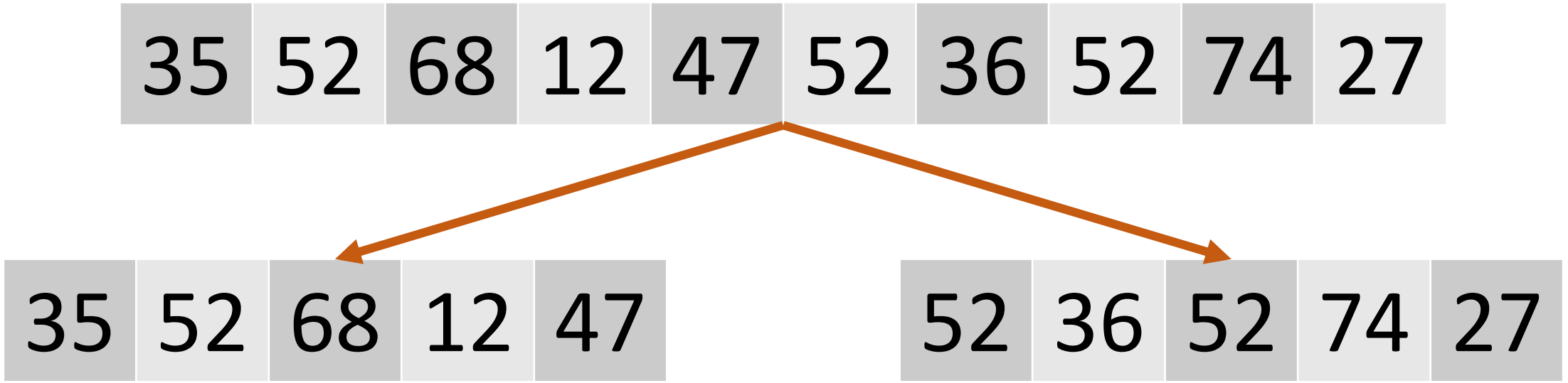
Practice 1

Mission

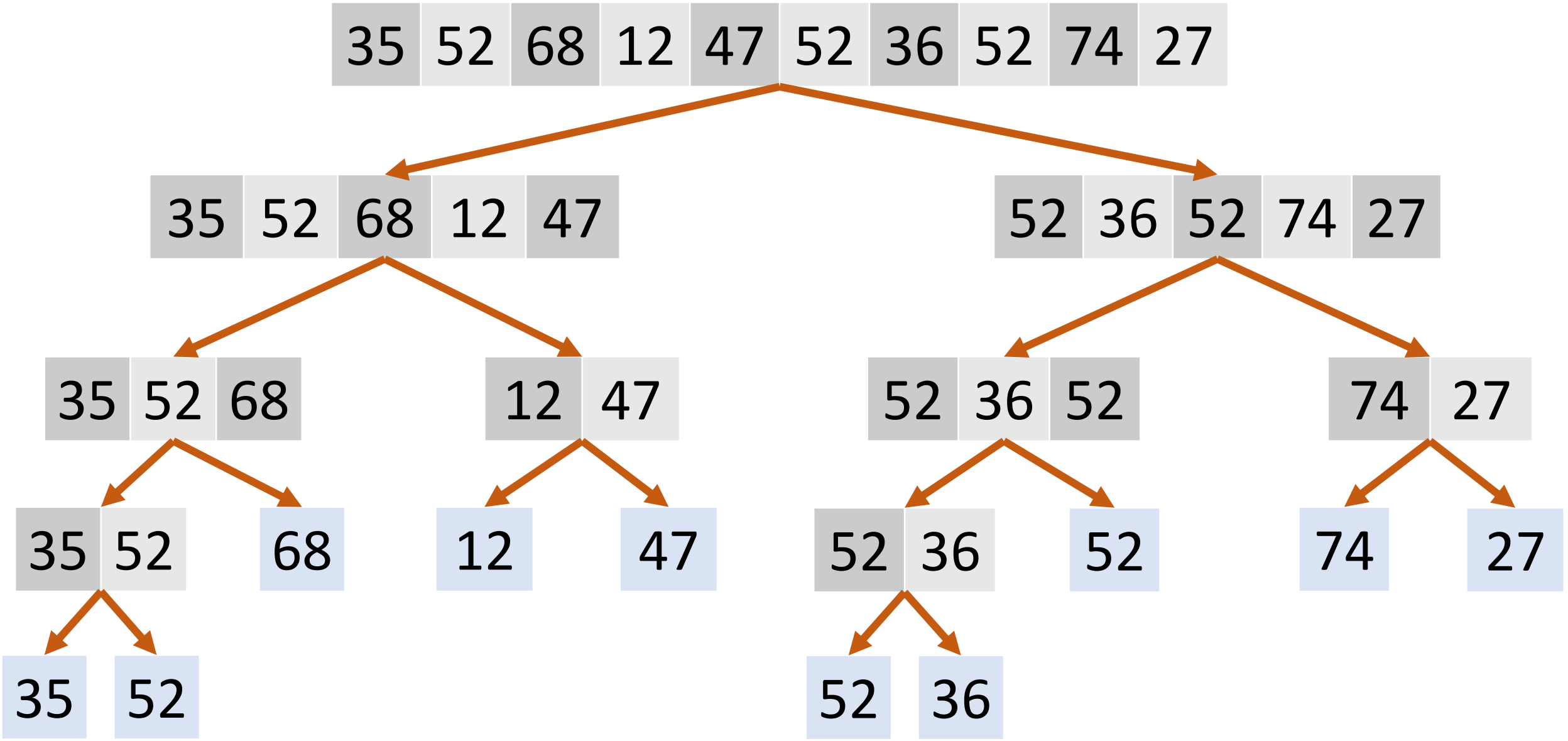
請實作 Shaker Sort

Merge Sort

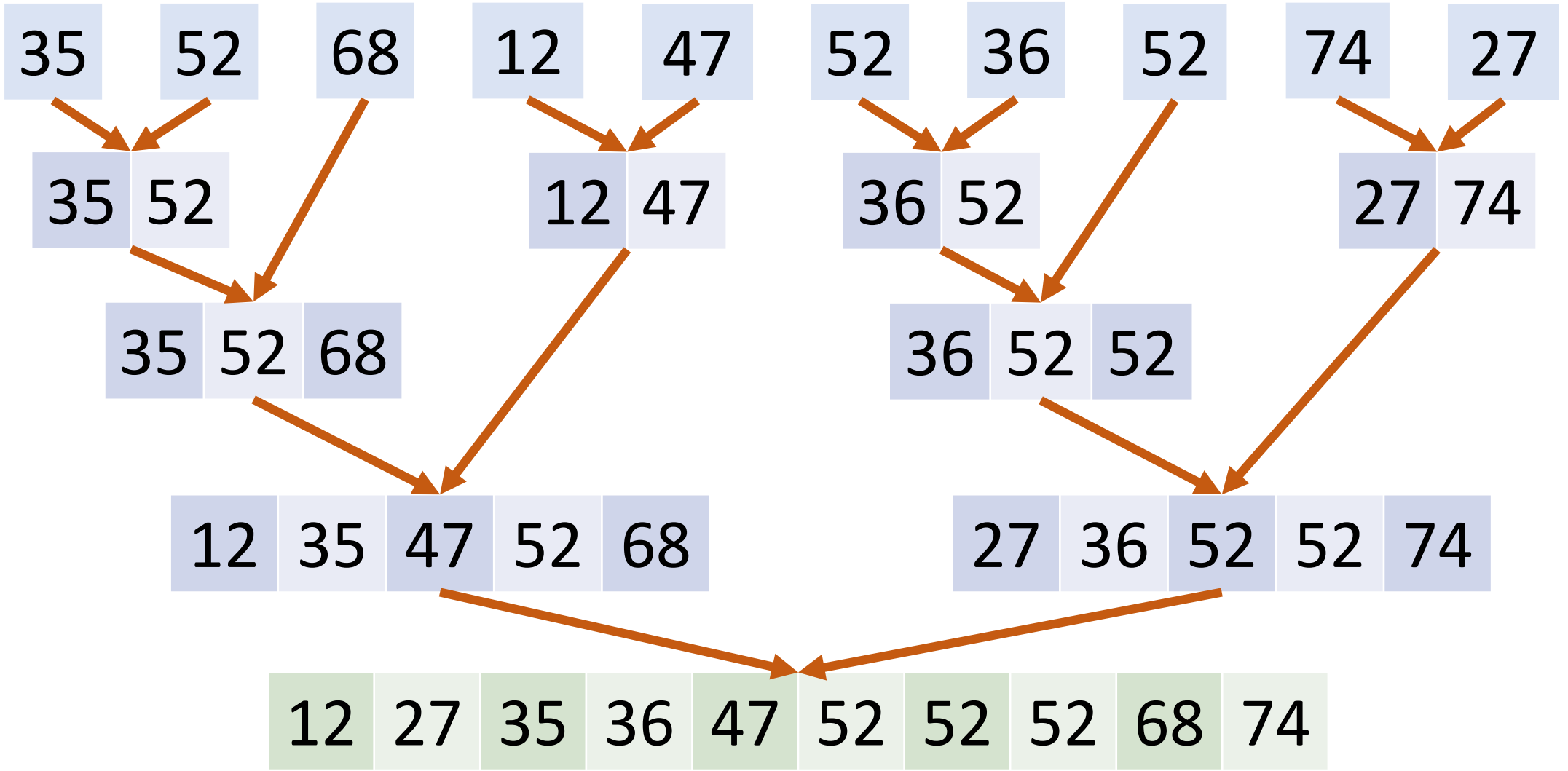
- 切割資料後再融合
  - 把資料切成兩組，分別排序
  - 再把已排序好的兩組資料融合在一起
  - 分治法 (Divide and Conquer) 的應用



Merge Sort

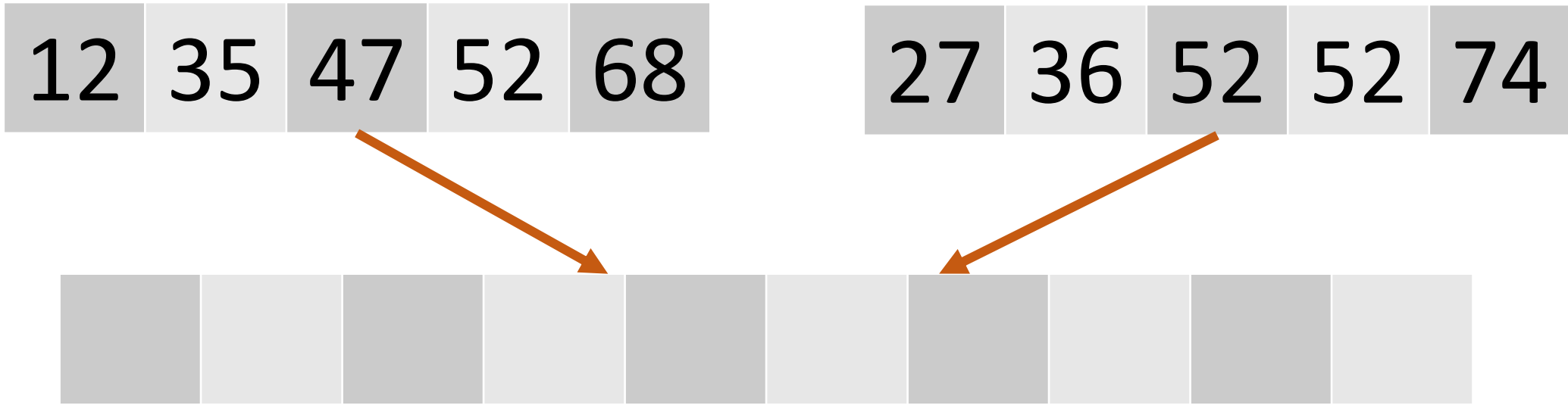

35 52 68 12 47 52 36 52 74 27
35 52 68 12 47
52 36 52 74 27
35 52 68
12 47
52 36 52
74 27
35 52
68
12
47
52 36
52
74
27
35
52
52
36

Merge Sort


35 52 68 12 47 52 36 52 74 27
35 52 12 47 36 52 27 74
35 52 68 36 52 52
12 35 47 52 68 27 36 52 52 74
12 27 35 36 47 52 52 52 68 74

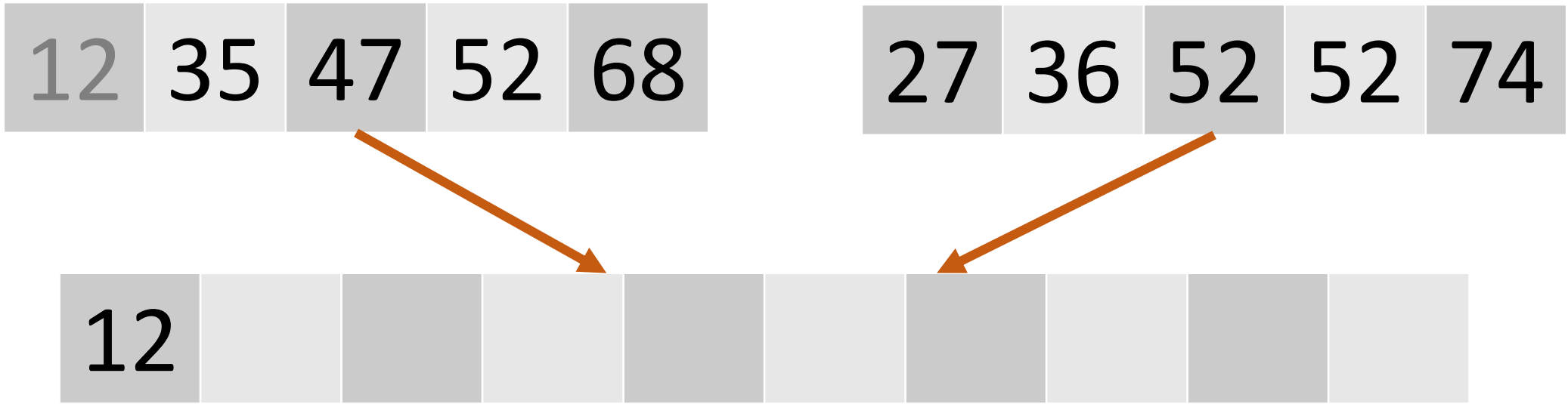
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況



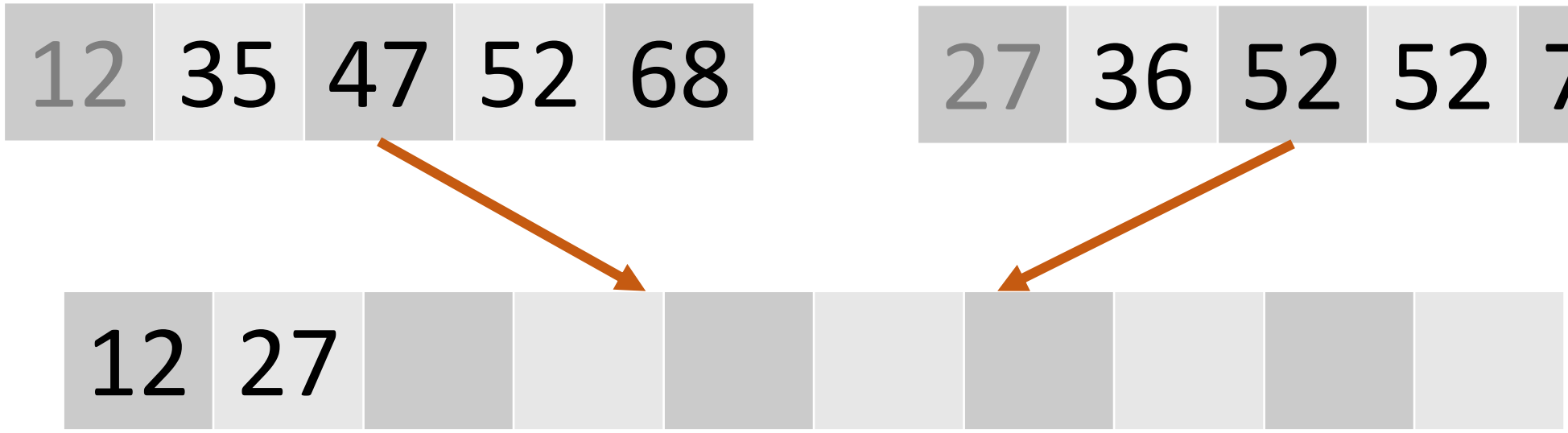
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況



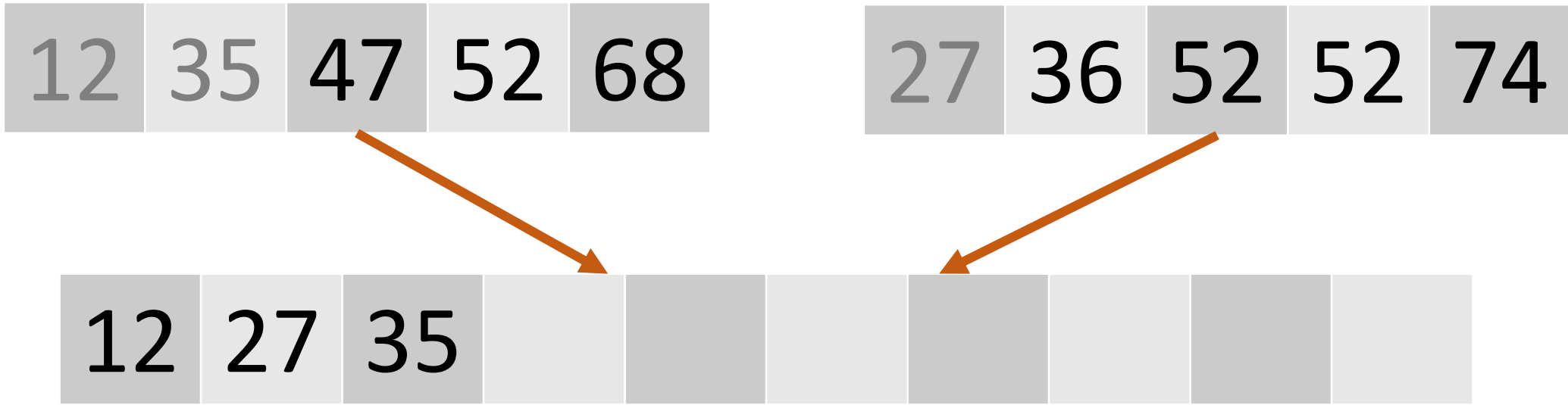
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況



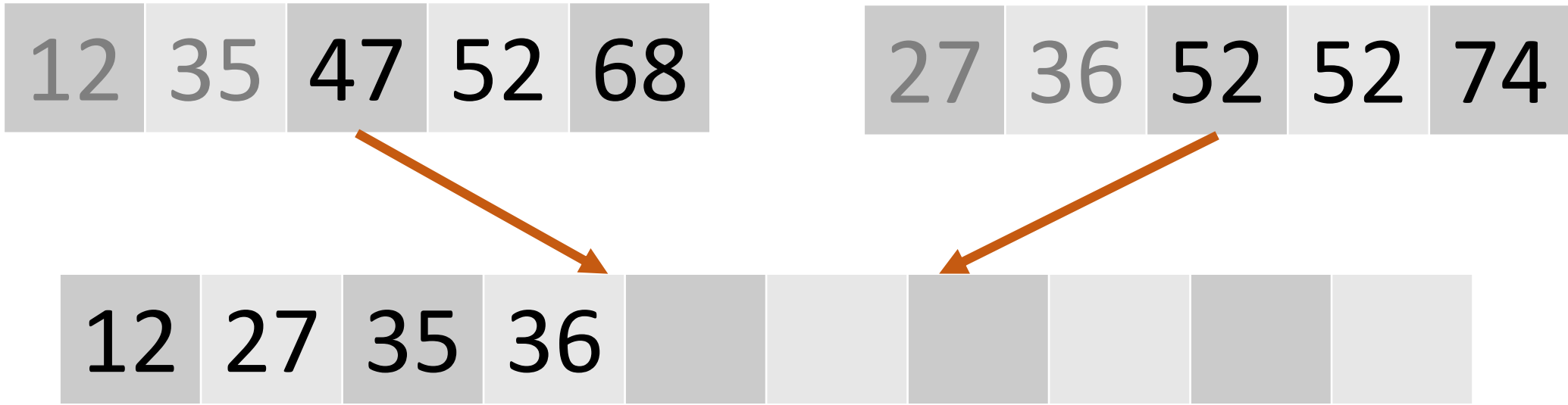
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況



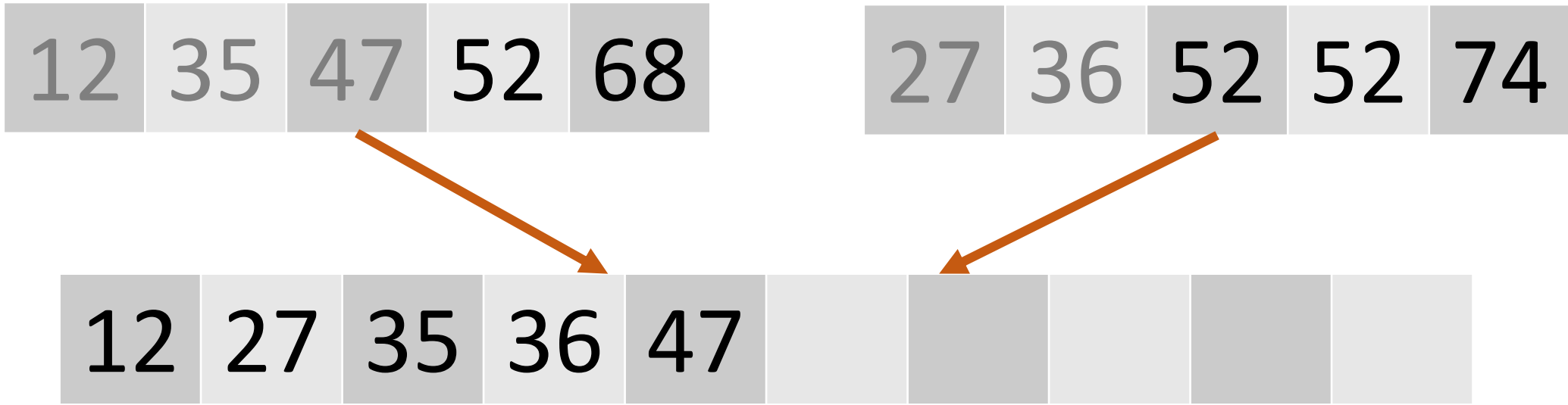
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況



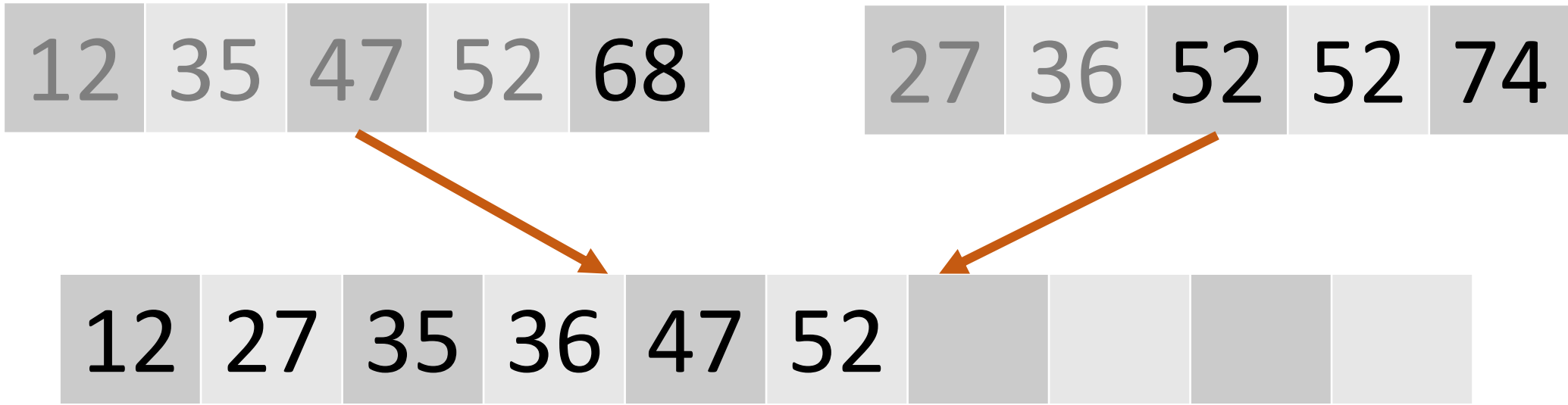
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況



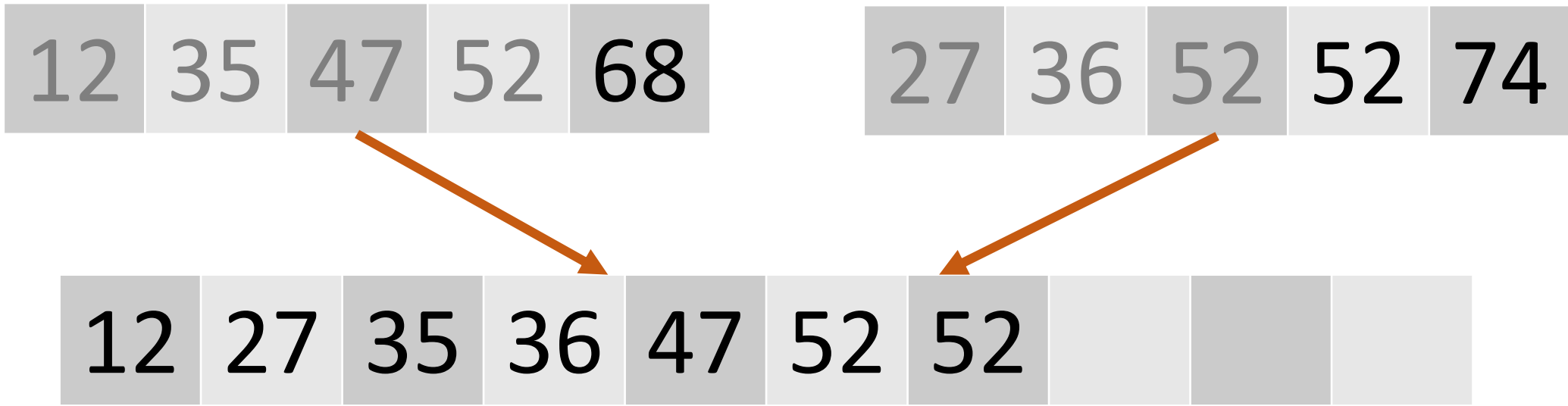
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況



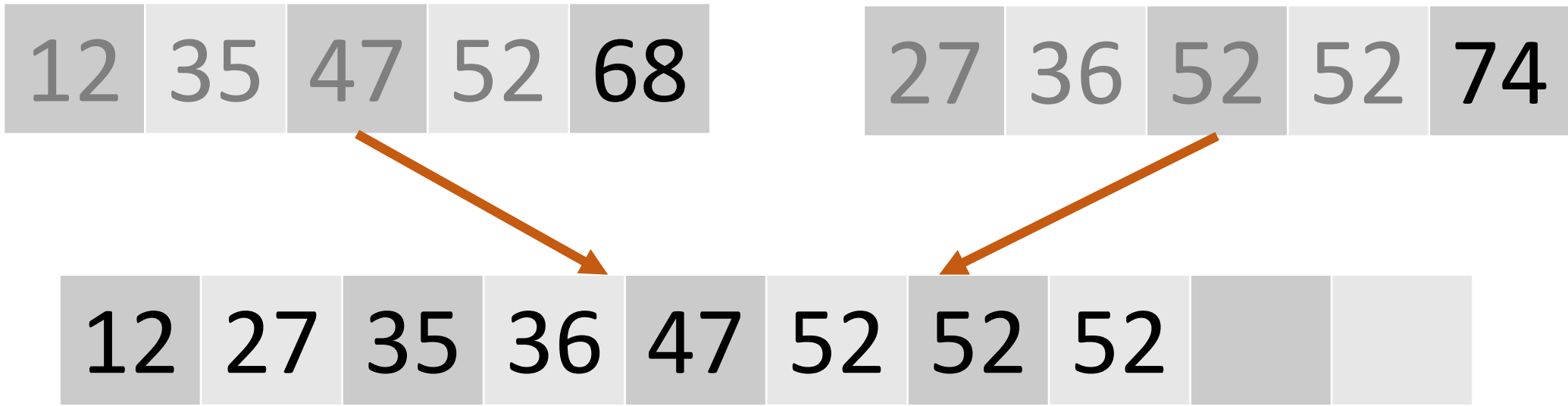
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況



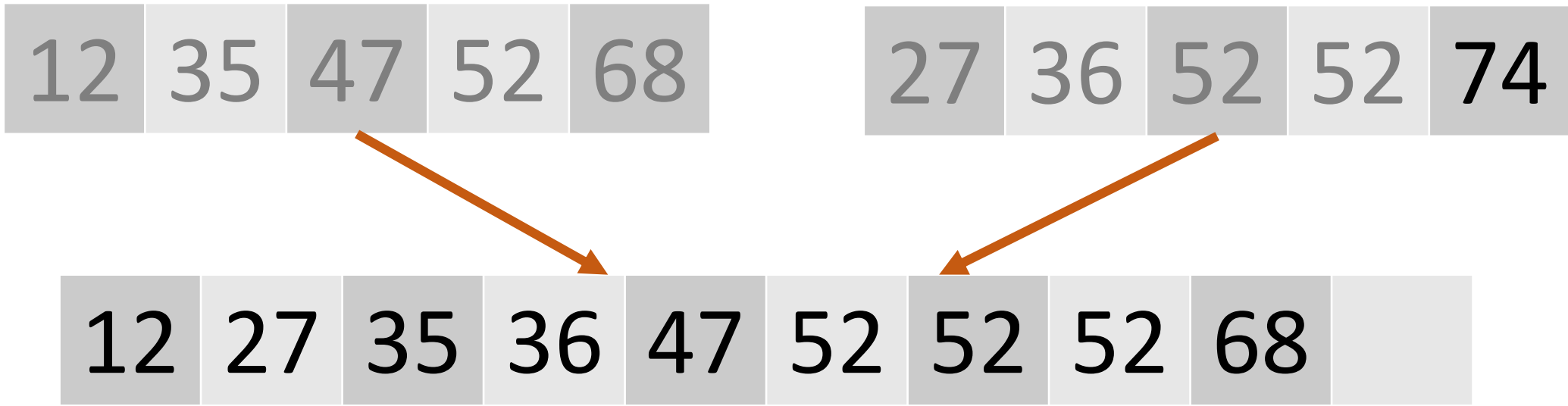
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況



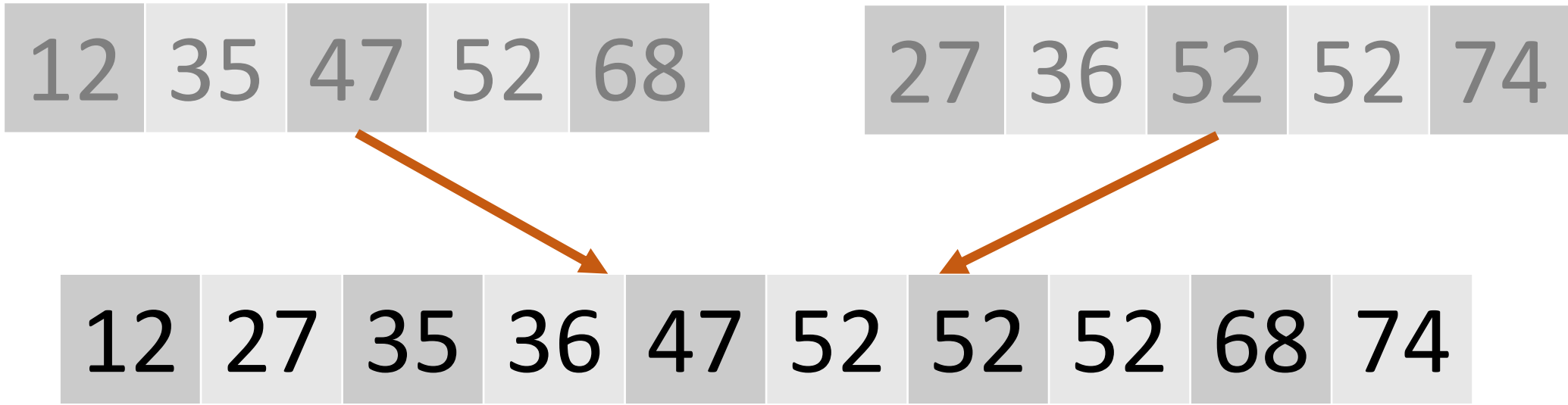
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況



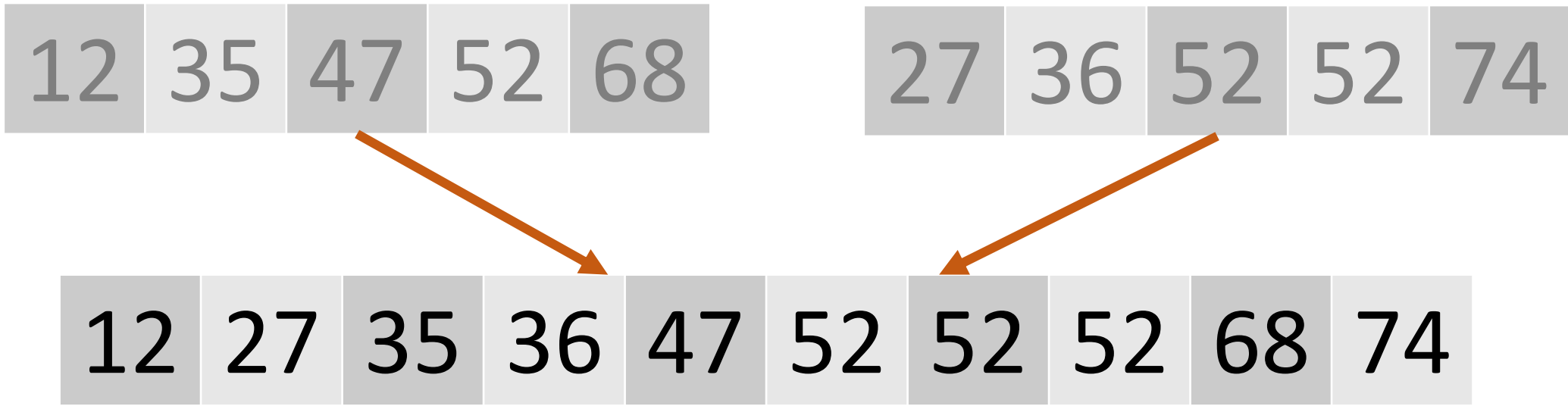
Merge Sort

- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況

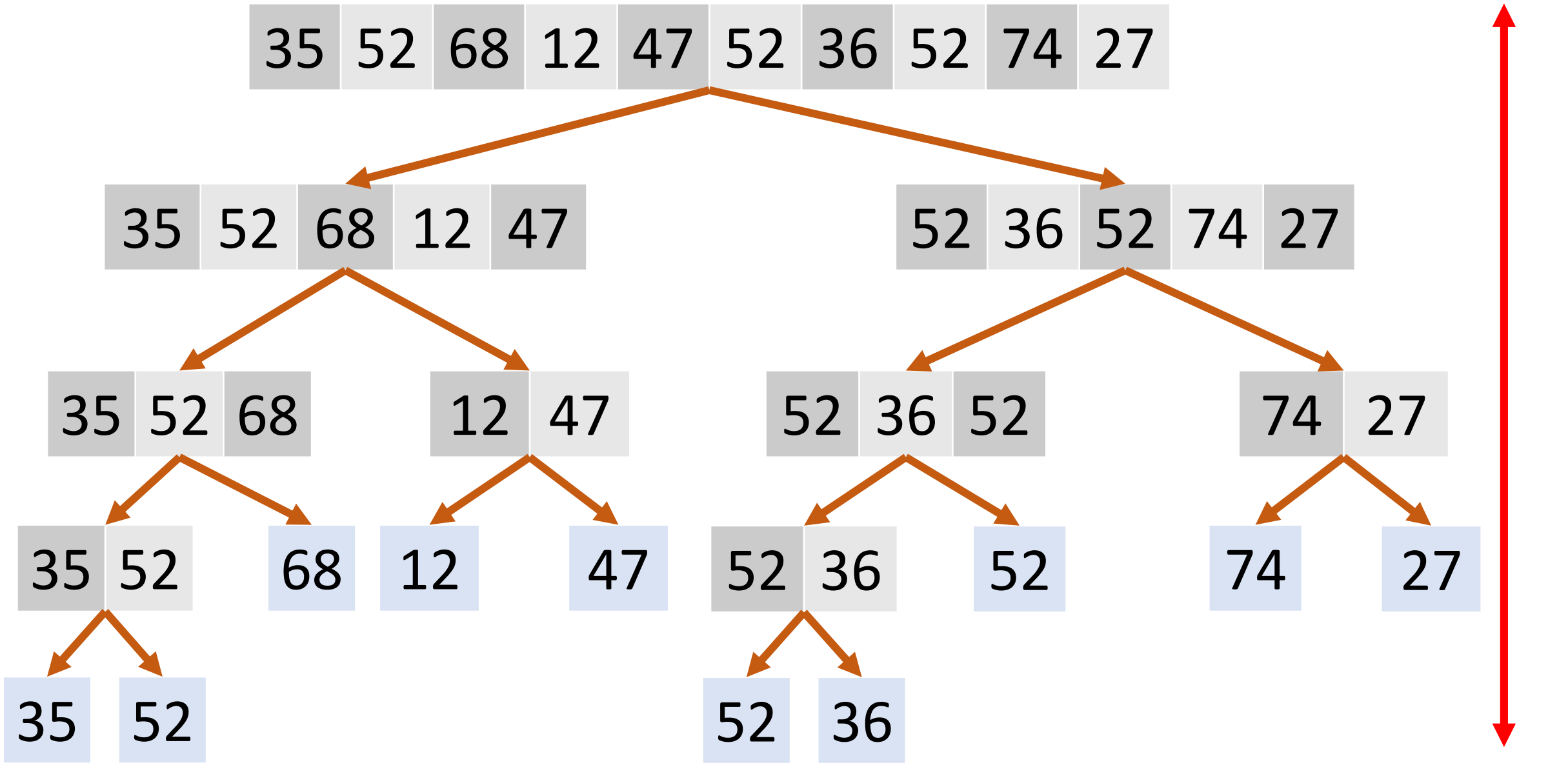


Merge Sort

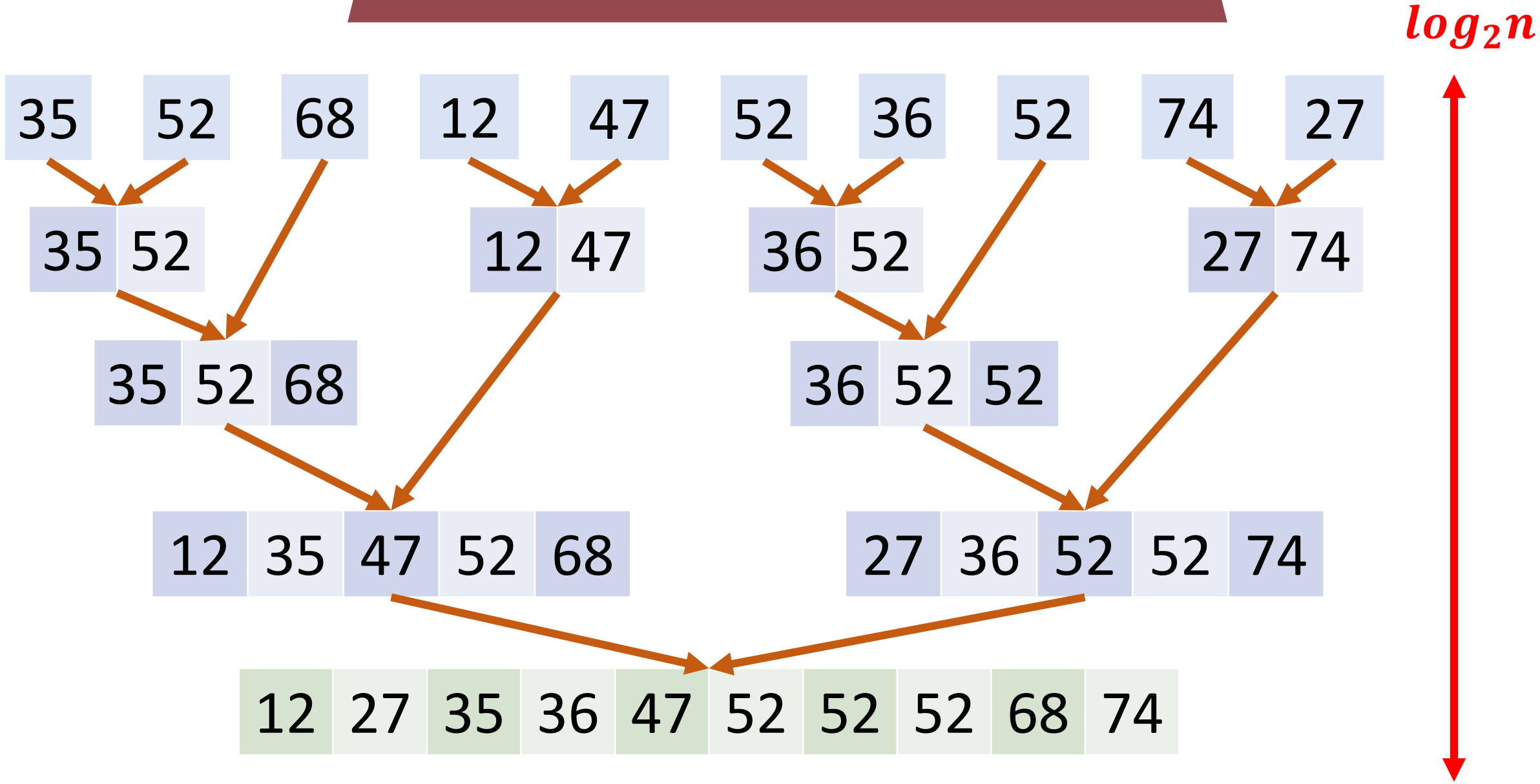
- Q：如何融合？複雜度多少？
  - 因這兩組資料已經排序過，依序比較兩陣列中的最小值
  - 注意有可能發生某一陣列中資料已經拿完的狀況
  - **融合的複雜度為 O(n)**



合併排序

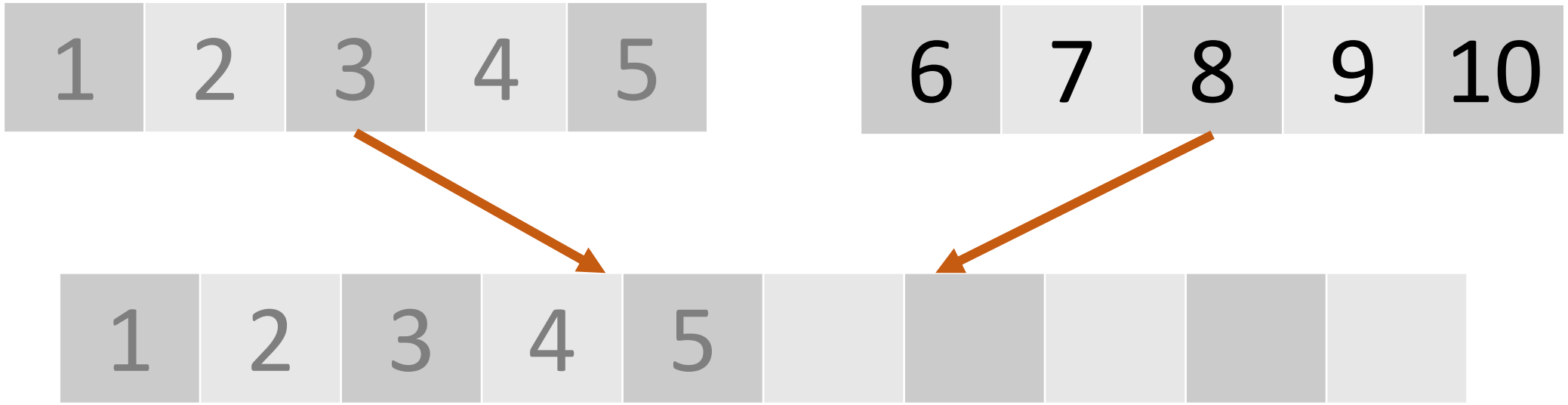

35 52 68 12 47 52 36 52 74 27
35 52 68 12 47
52 36 52 74 27
35 52 68
12 47
52 36 52
74 27
35 52
68
12
47
52 36
52
74
27
35
52
52
36
$log_2 n$

合併排序


35 52 68 12 47 52 36 52 74 27
35 52 12 47 36 52 27 74
35 52 68 36 52 52
12 35 47 52 68 27 36 52 52 74
12 27 35 36 47 52 52 52 68 74
$log_2 n$

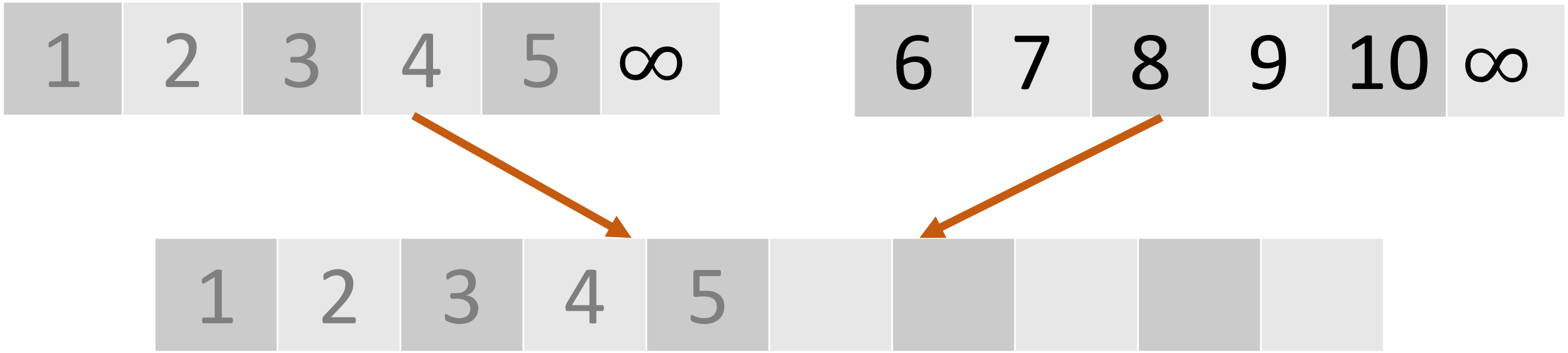
Merge Sort

- 合併時有可能發生一組已經耗盡的狀況
  - ➢ 比較時會出問題！



Merge Sort

- 合併時有可能發生一組已經耗盡的狀況
  - ➢ 解決方式：在兩組結尾加無限大



Merge Sort

- 切割資料後再融合
  - 把資料切成兩組，分別排序
  - 再把已排序好的兩組資料融合在一起
  - 分治法 (Divide and Conquer) 的應用

	Complexity
Best	$O(nlog_2n)$
Average	$O(nlog_2n)$
Worst	$O(nlog_2n)$
Memory	$O(n)$
Memory (Auxiliary)	$O(n)$ $O(1), with\ linked\ list$
Stable	True

Example Code

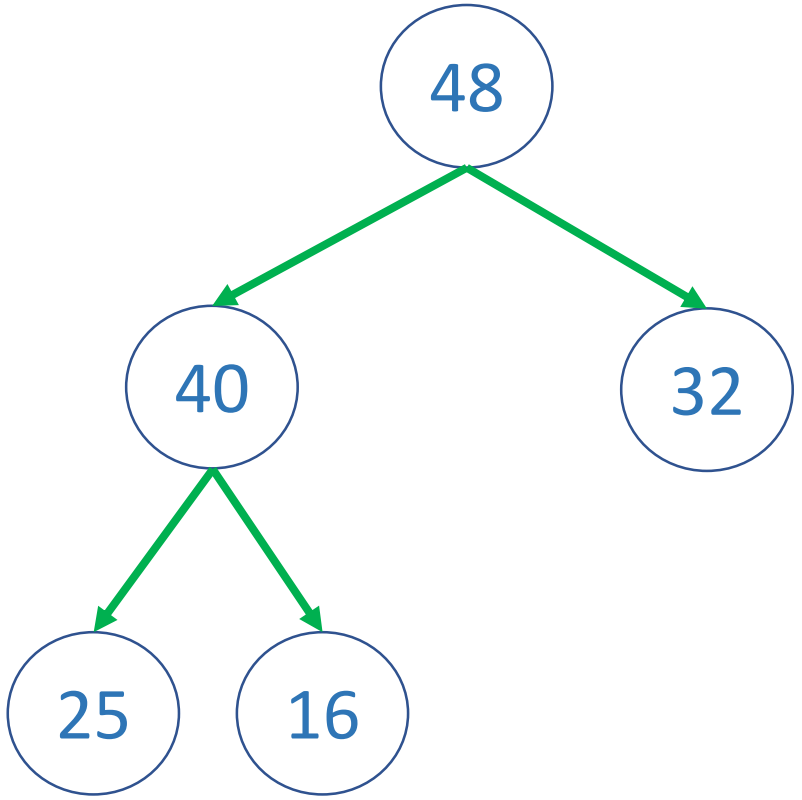
Mission

實作 Merge Sort

二元堆疊

二元堆疊 (Binary Heap)

- 每個節點最多有兩個子節點
- 同一階層 (level) 內要從左到右排列
  - ➢ 完整二元樹可以直接用陣列表達
- Min-Heap 與 Max-Heap
  1. Min-Heap：根節點為最小值
  2. Max-Heap：根節點為最大值

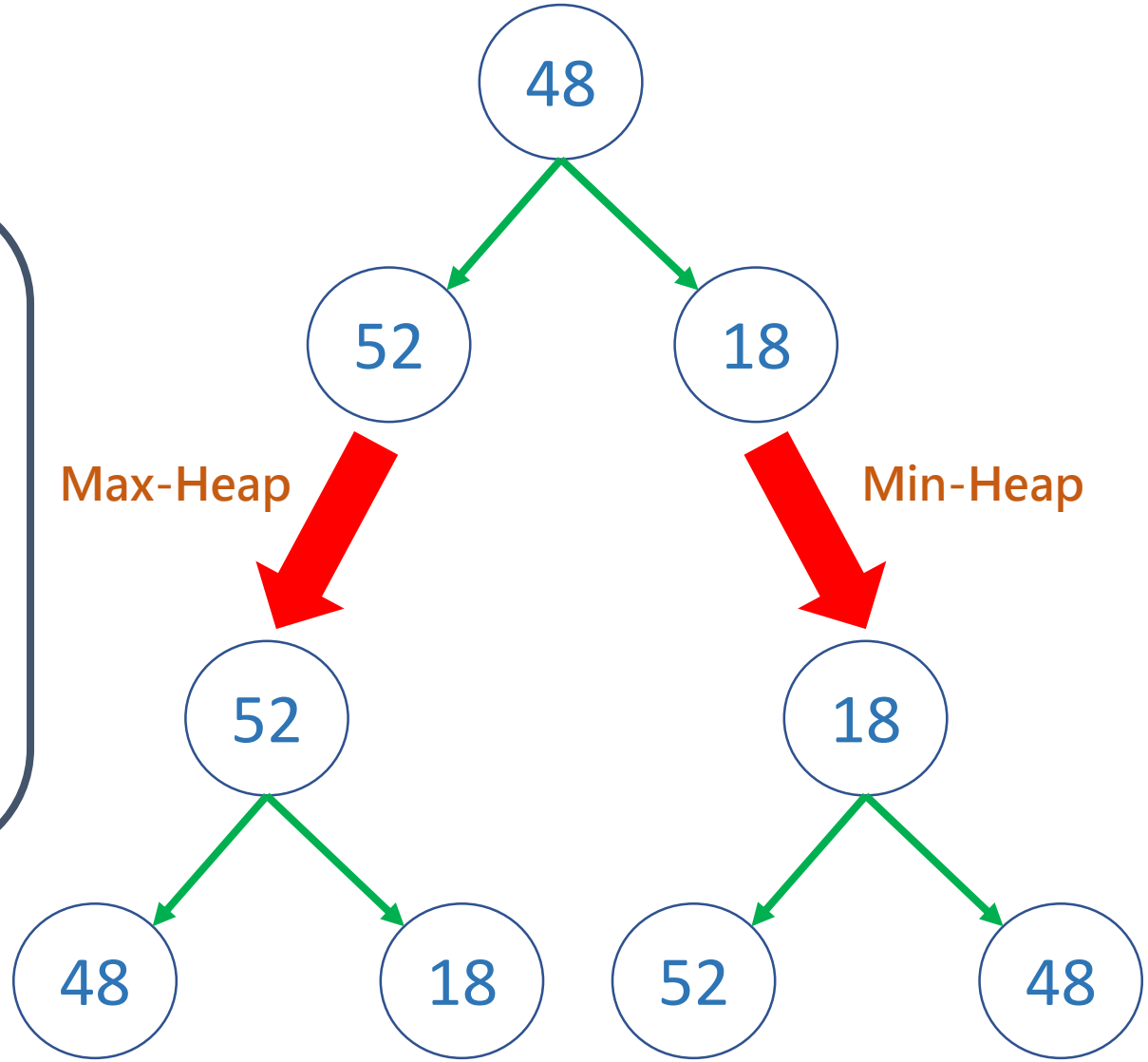


48	40	32	25	16

二元堆疊

Heapify

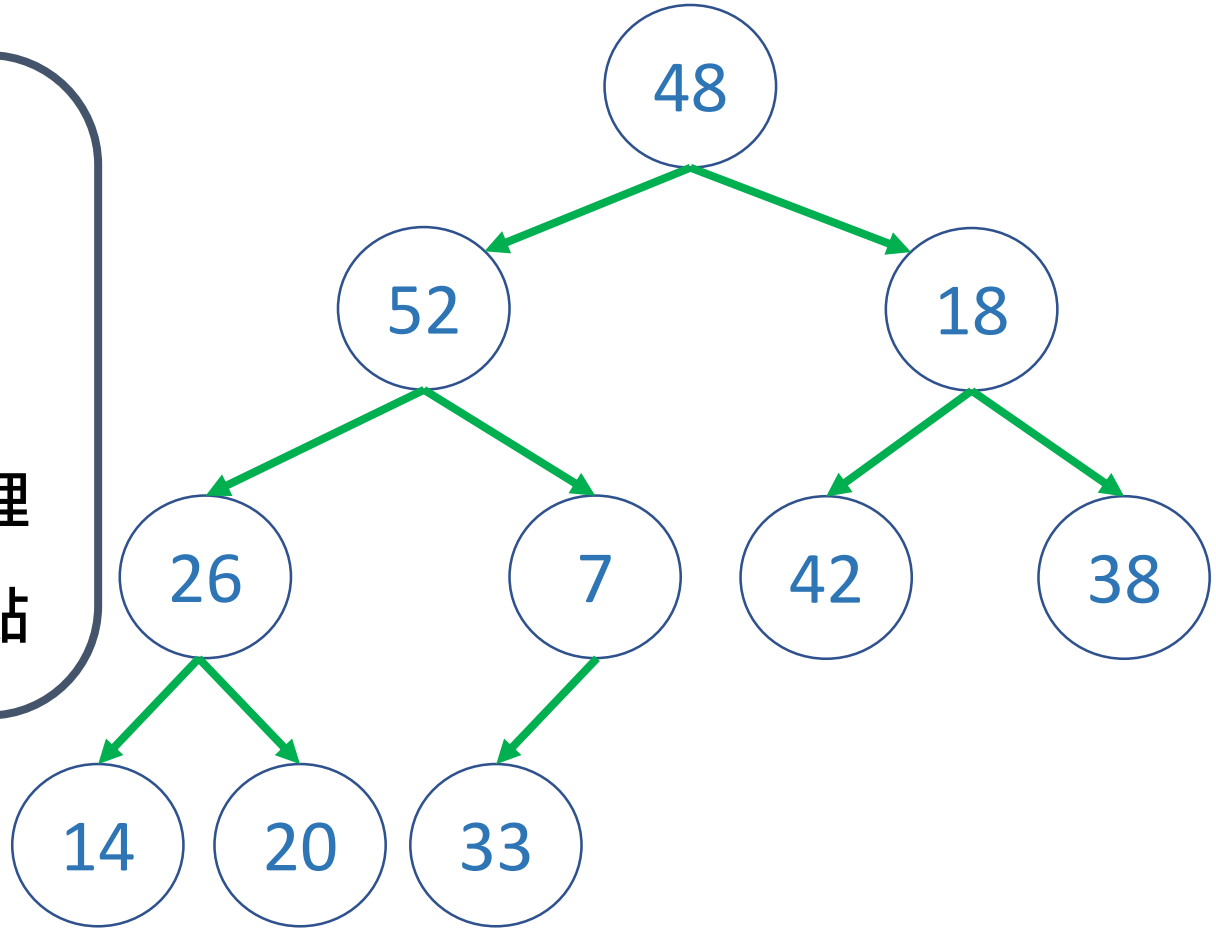
- Min-Heap：根節點為最小值
  - ➢ 取左、中、右最小的節點當根節點
- Max-Heap：根節點為最大值
  - ➢ 取左、中、右最小的節點當根節點



二元堆疊

建立最大二元堆疊 (Binary Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

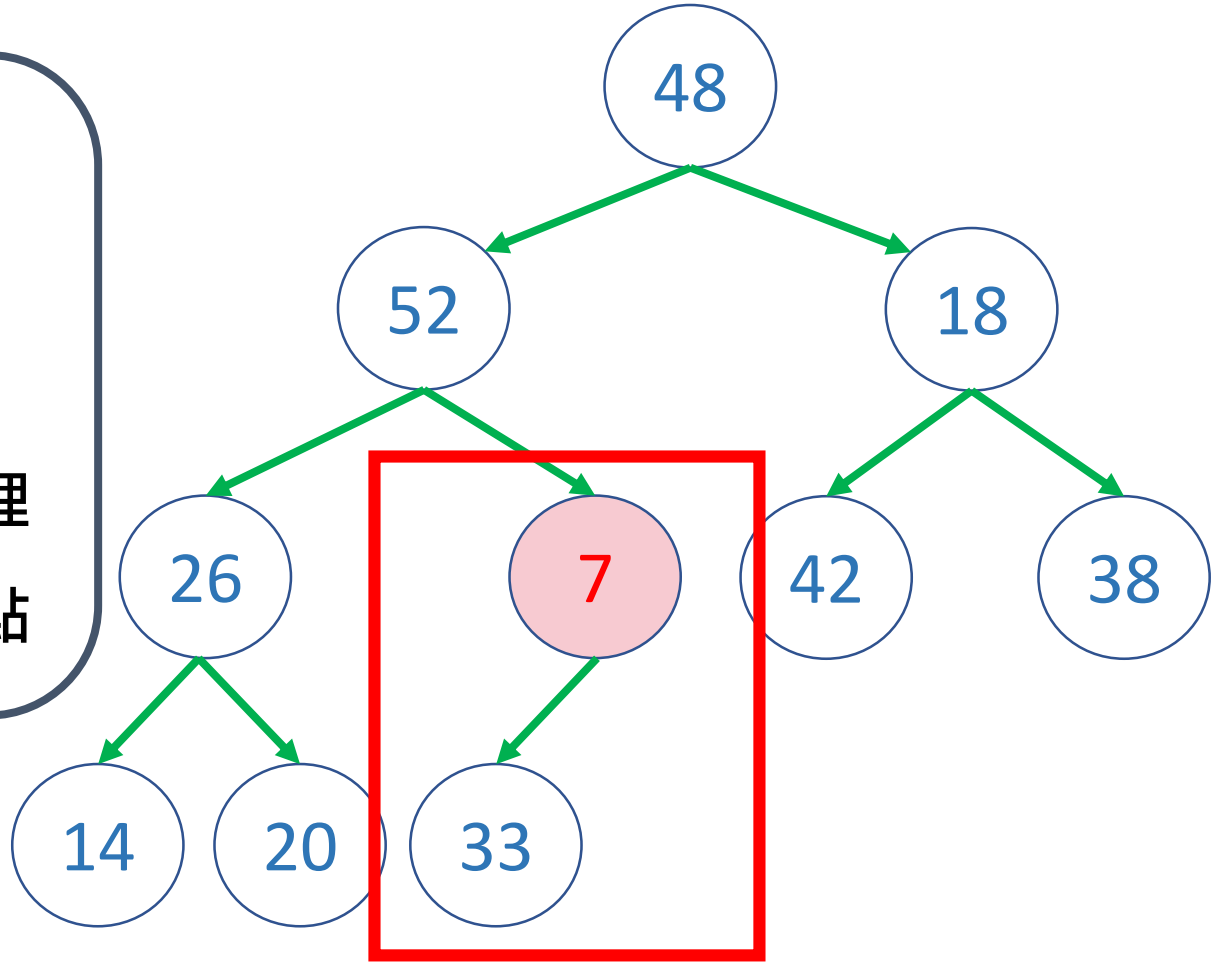


48	52	18	26	7	42	38	14	20	33

二元堆疊

建立最大二元堆疊 (Binary Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最小的為根節點

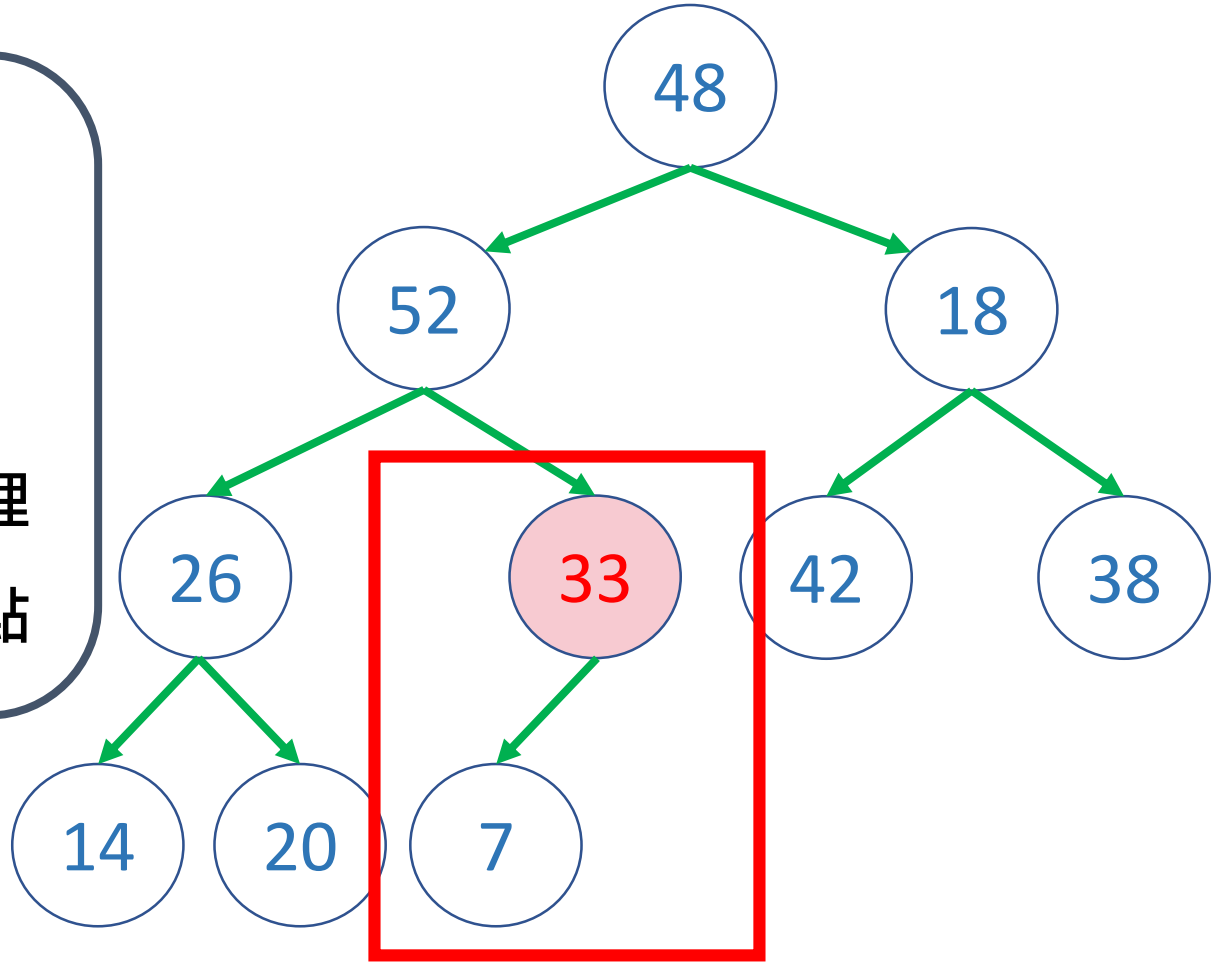


48	52	18	26	7	42	38	14	20	33

二元堆疊

建立最大二元堆疊 (Binary Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最小的為根節點

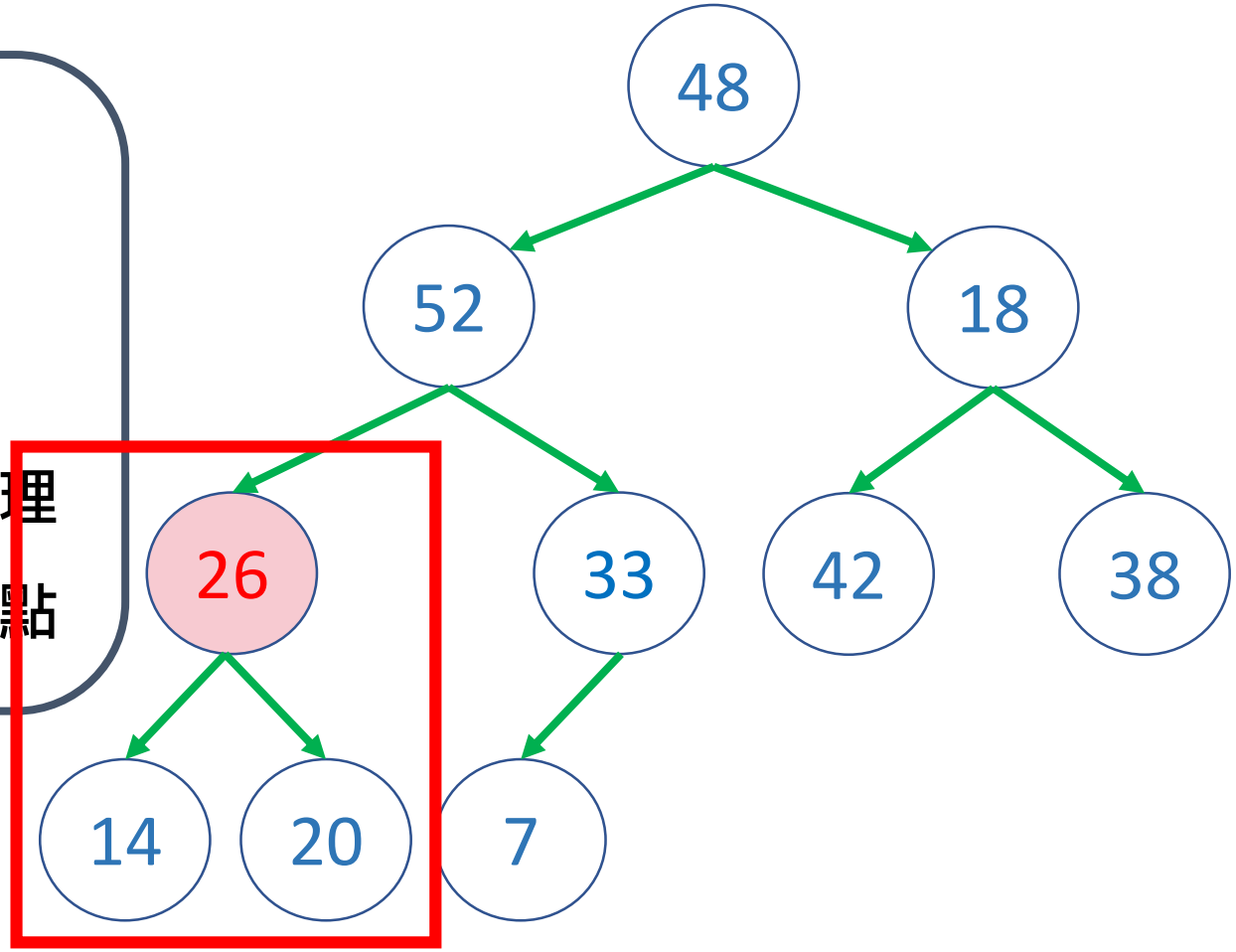


48	52	18	26	33	42	38	14	20	7

二元堆疊

建立最大二元堆疊 (Binary Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最小的為根節點

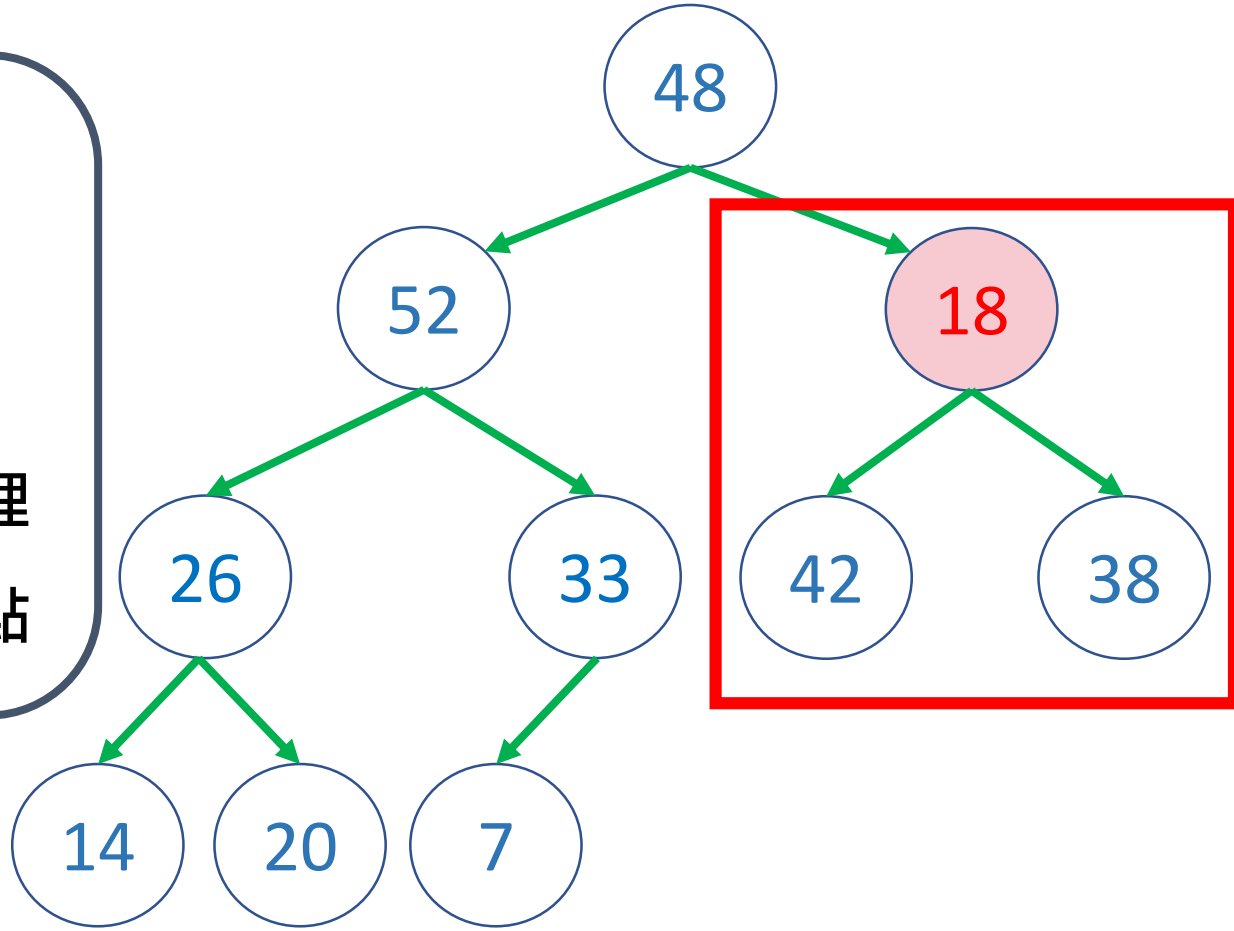


48	52	18	26	33	42	38	14	20	7

二元堆疊

建立最大二元堆疊 (Binary Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最小的為根節點

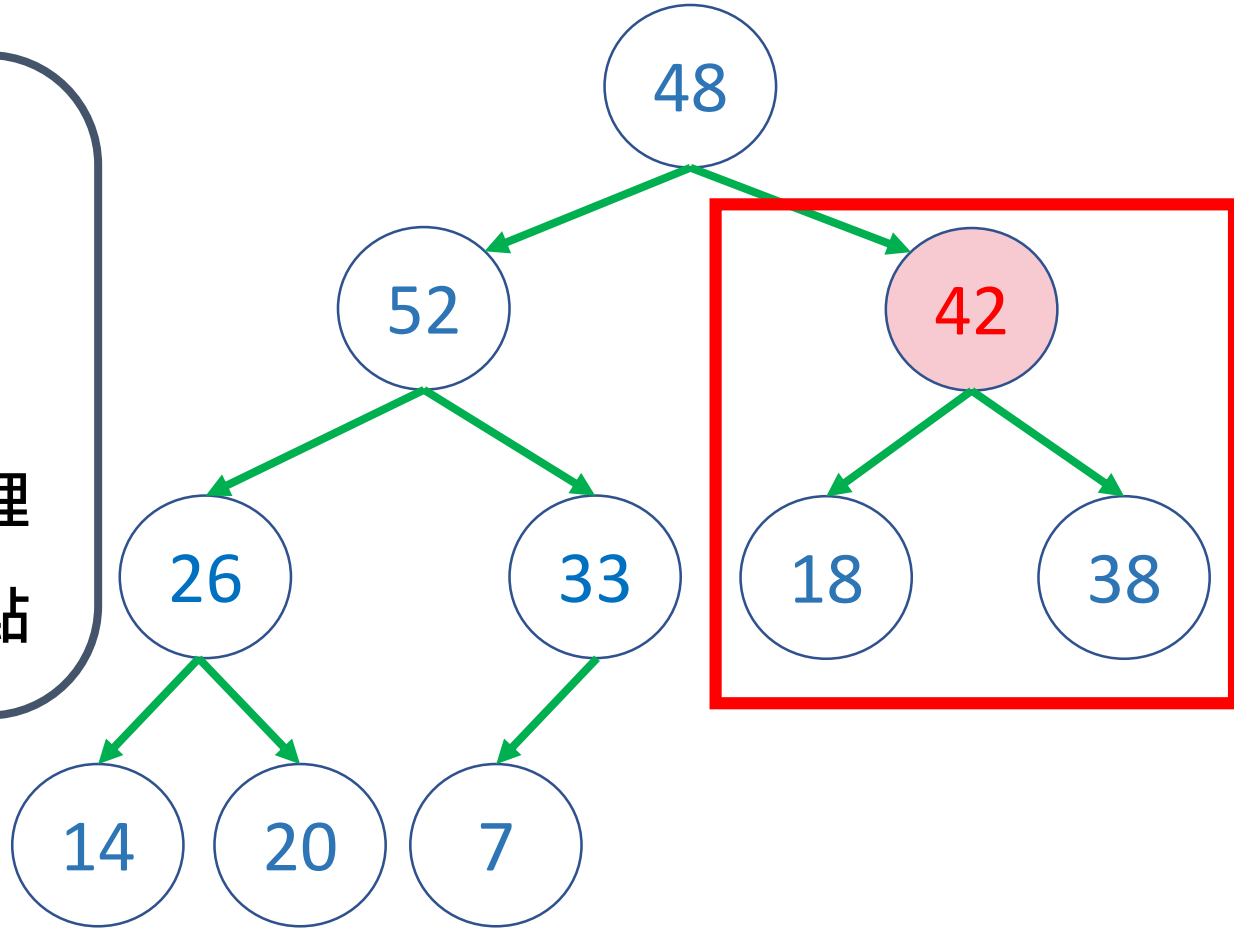


48	52	18	26	33	42	38	14	20	7

二元堆疊

建立最大二元堆疊 (Binary Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最小的為根節點

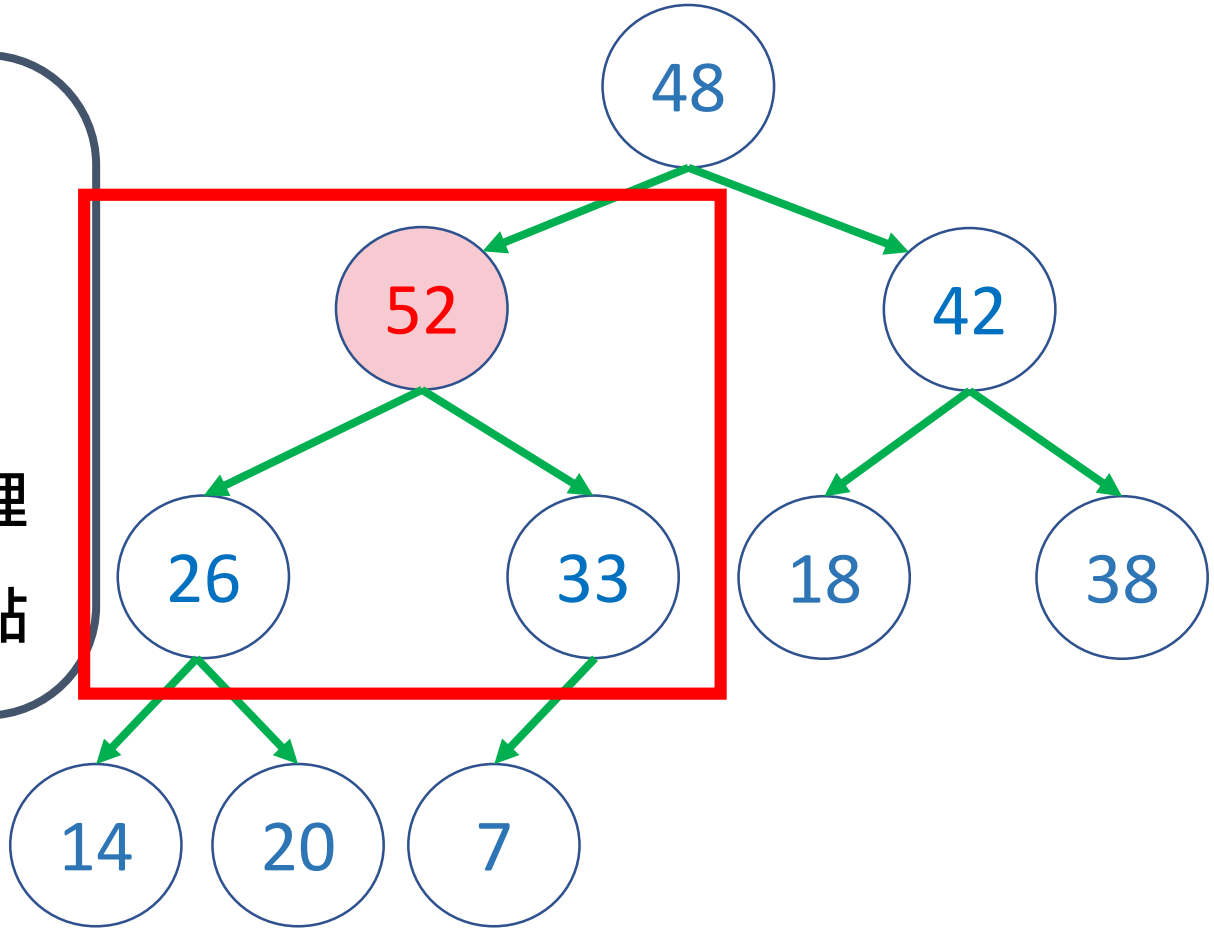


48	52	42	26	33	18	38	14	20	7

二元堆疊

建立最大二元堆疊 (Binary Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最小的為根節點

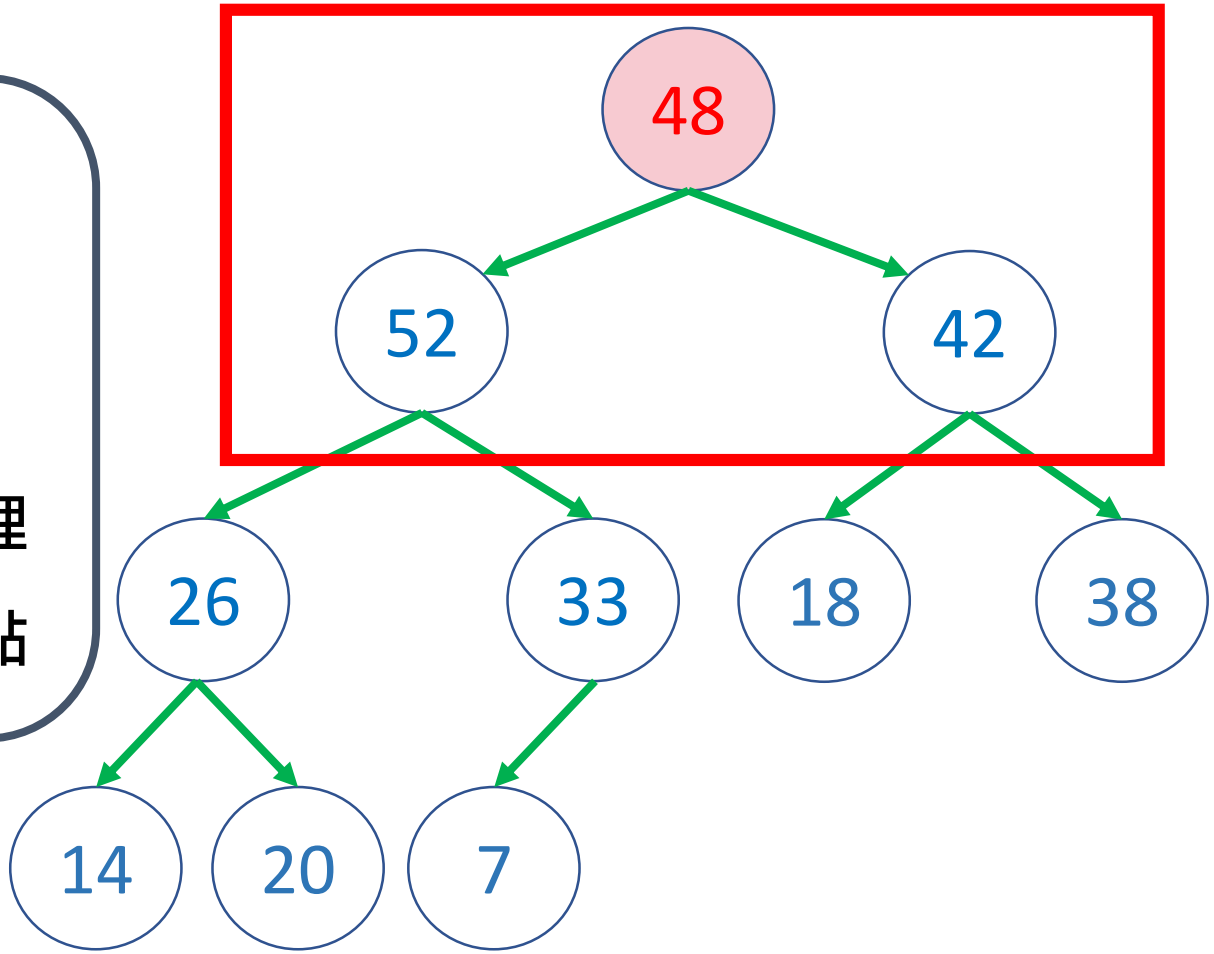


48	52	42	26	33	18	38	14	20	7

二元堆疊

建立最大二元堆疊 (Binary Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最小的為根節點

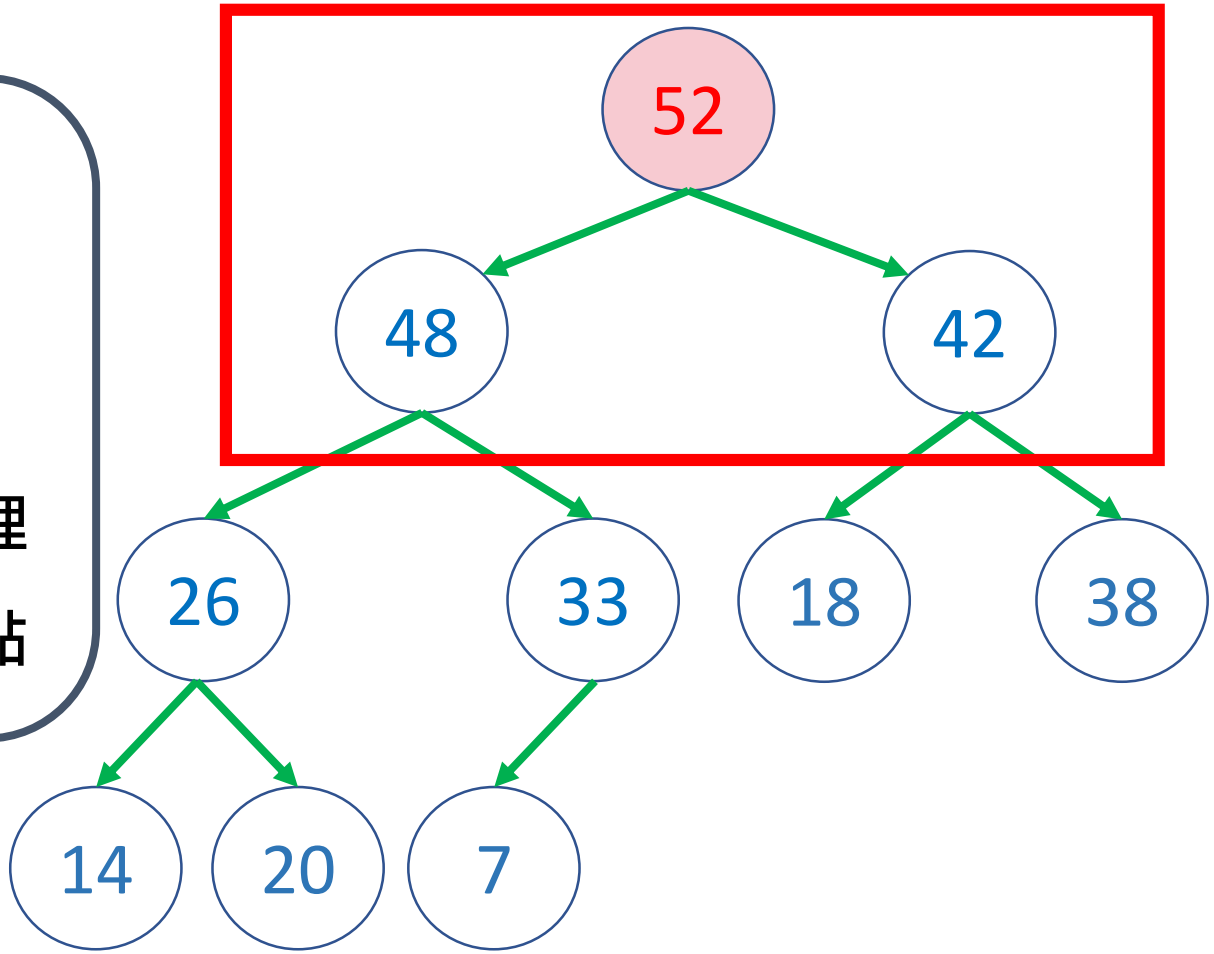


48	52	42	26	33	18	38	14	20	7

二元堆疊

建立最大二元堆疊 (Binary Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點



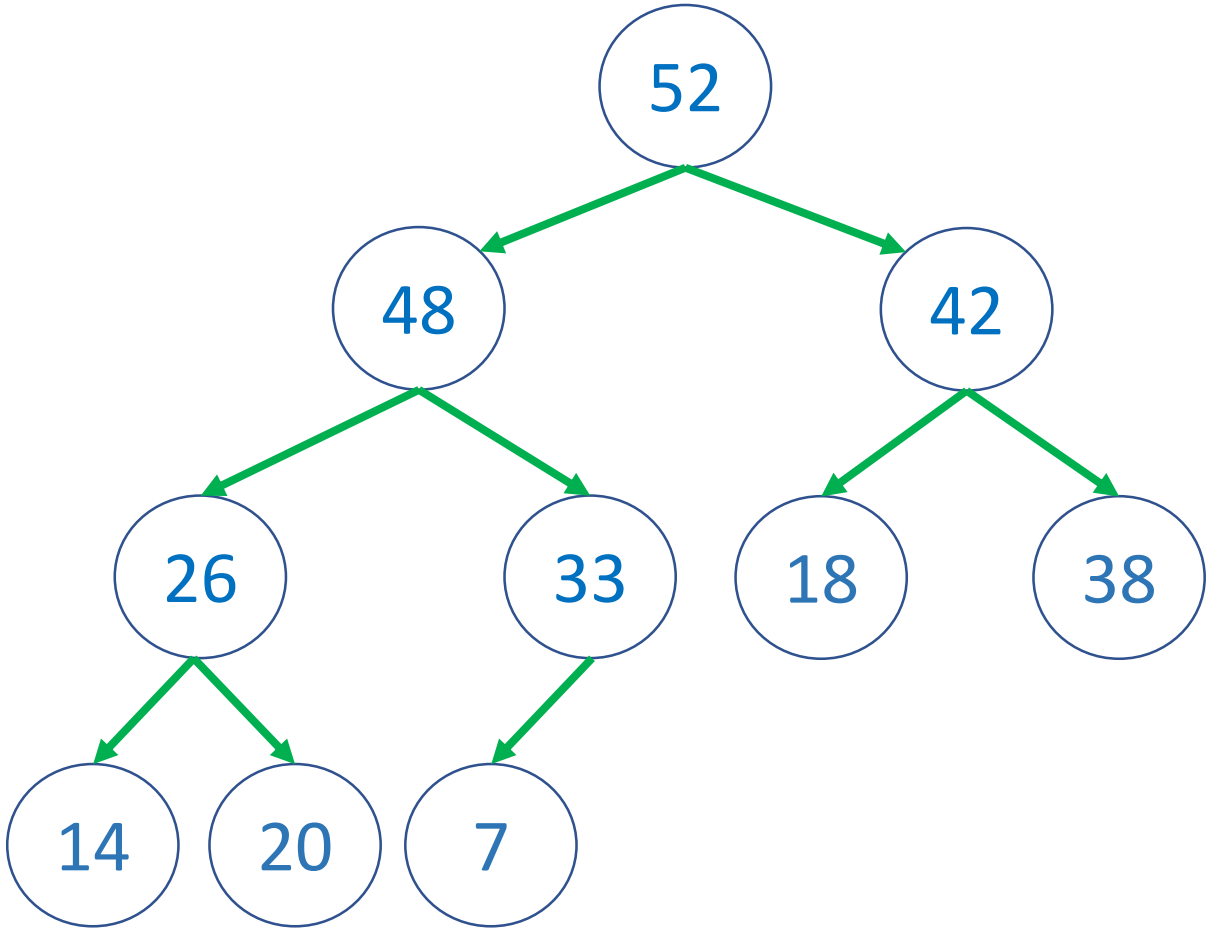
52	48	42	26	33	18	38	14	20	7

二元堆疊

建立二元堆疊 (Binary Heap)

- 給定一陣列長度為 len
  - 末端元素索引值：$len - 1$
  - 該元素的根節點索引值：$\left\lfloor \frac{len-2}{2} \right\rfloor$

```
for(int i = len/2-1; i>=0 ; i--){
        Heapify(i);
}
```

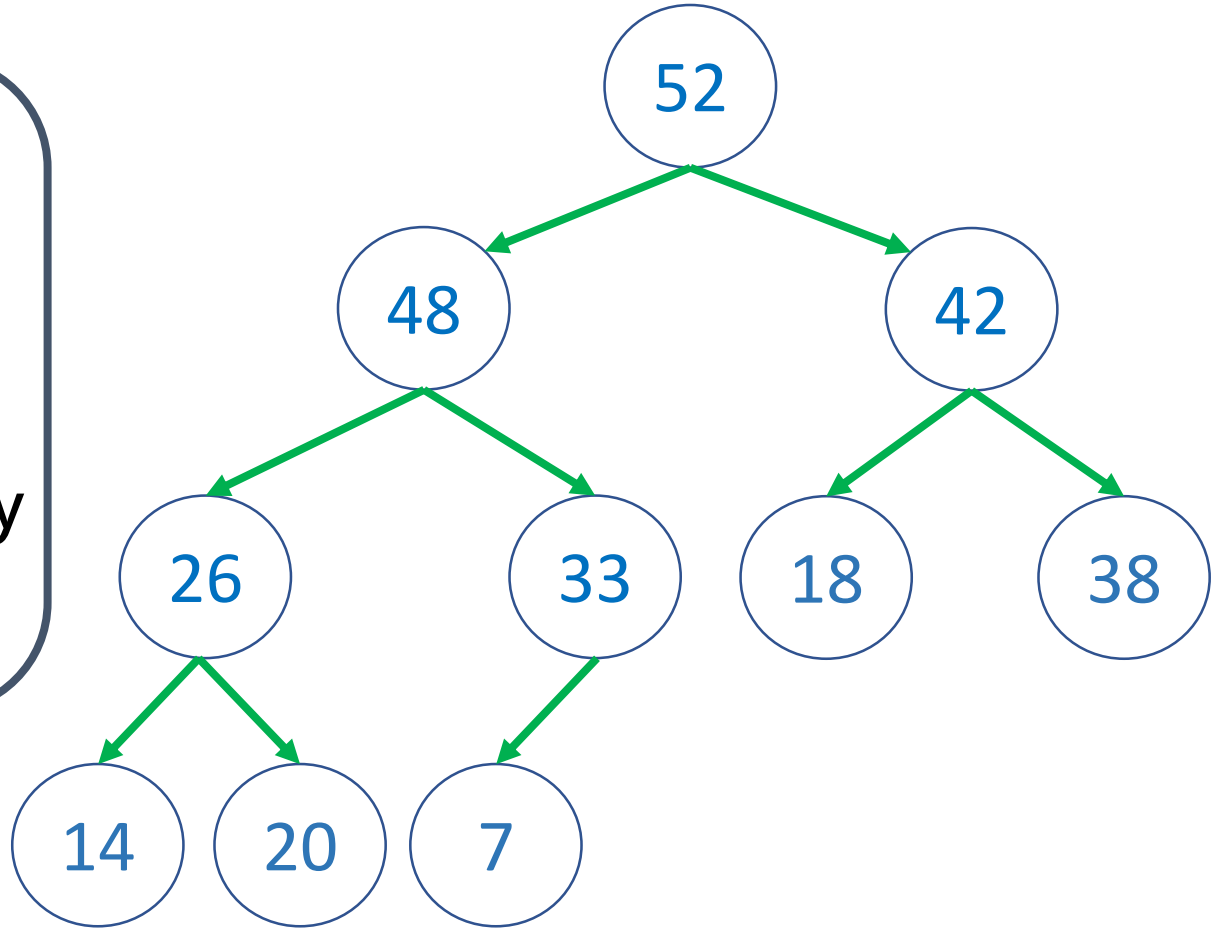


52	48	42	26	33	18	38	14	20	7

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

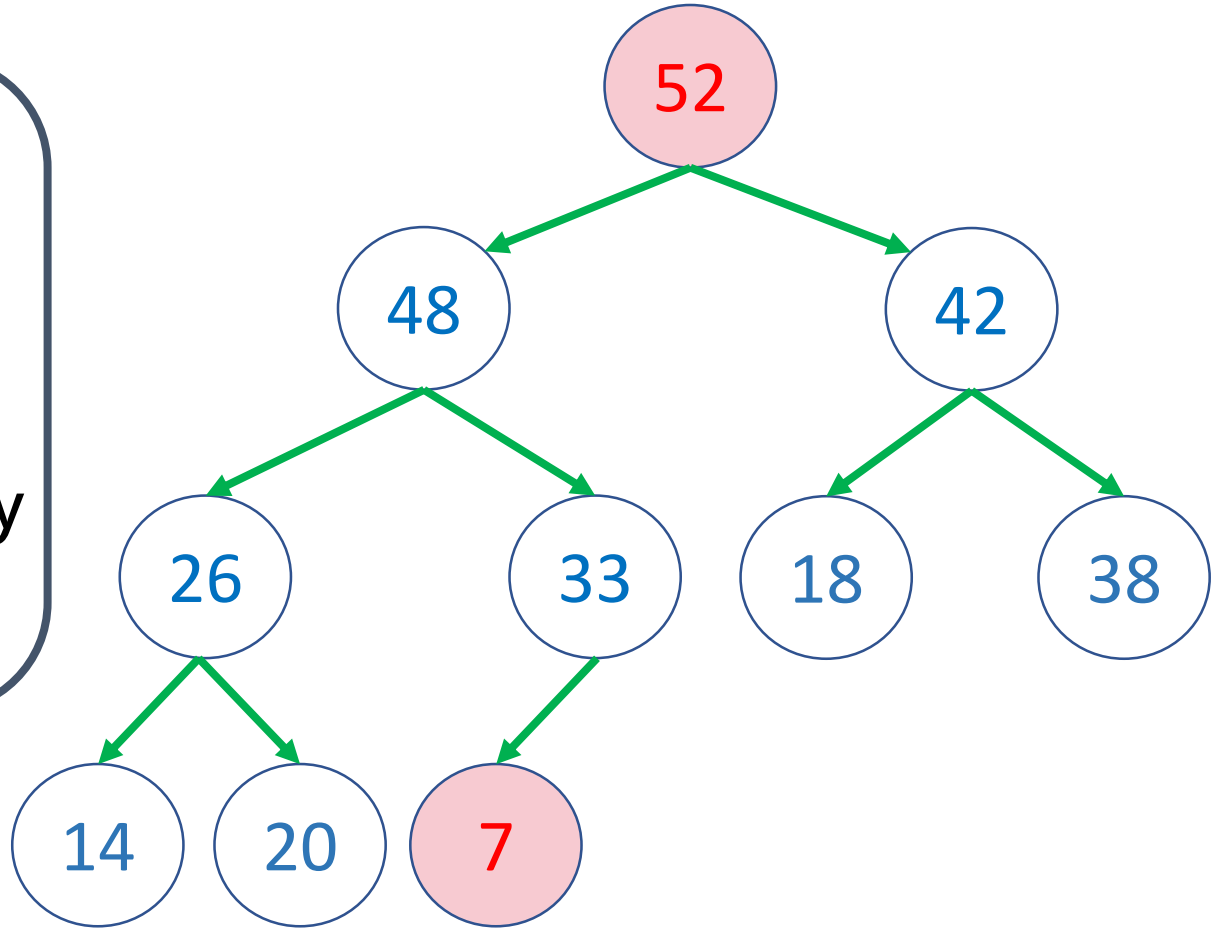


52	48	42	26	33	18	38	14	20	7

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

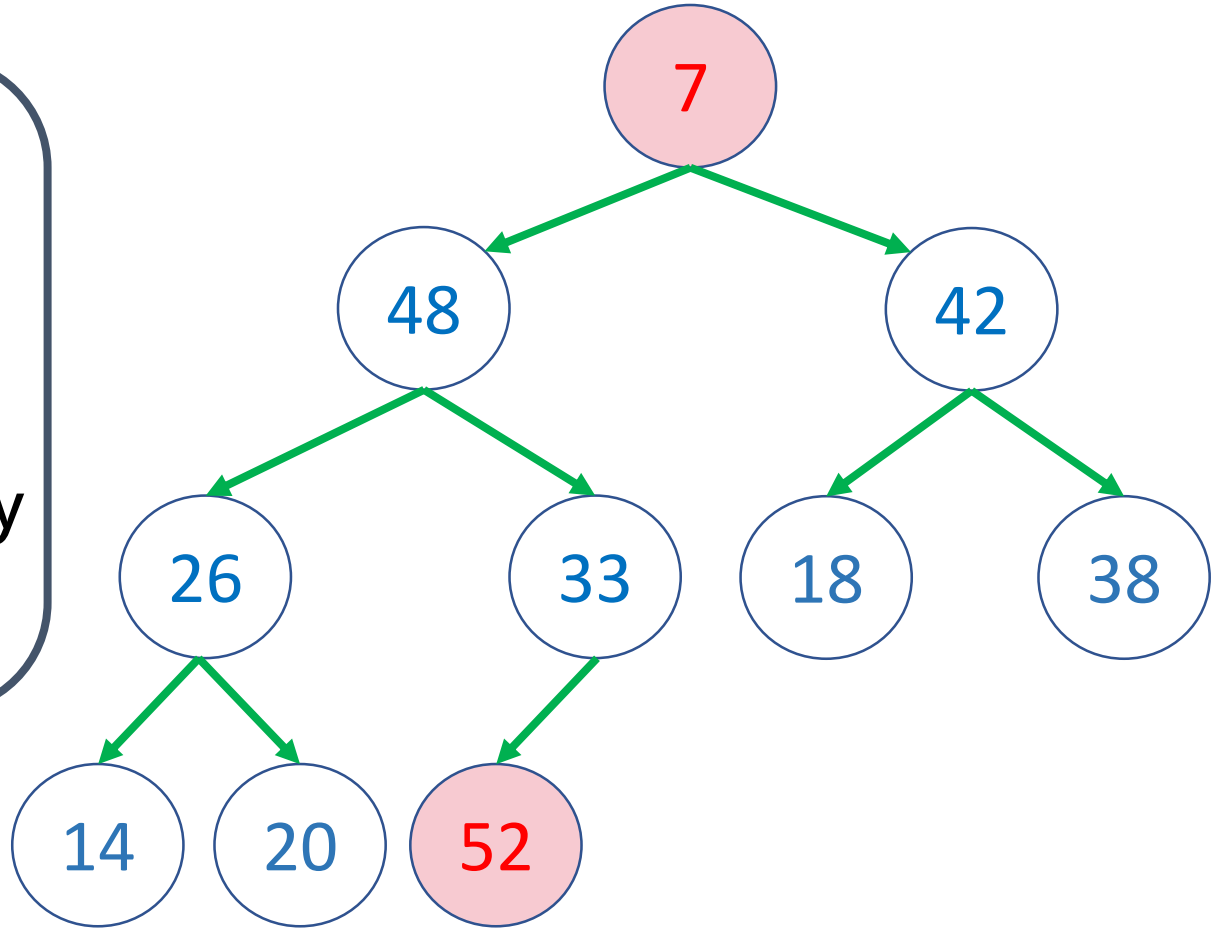


52	48	42	26	33	18	38	14	20	7

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

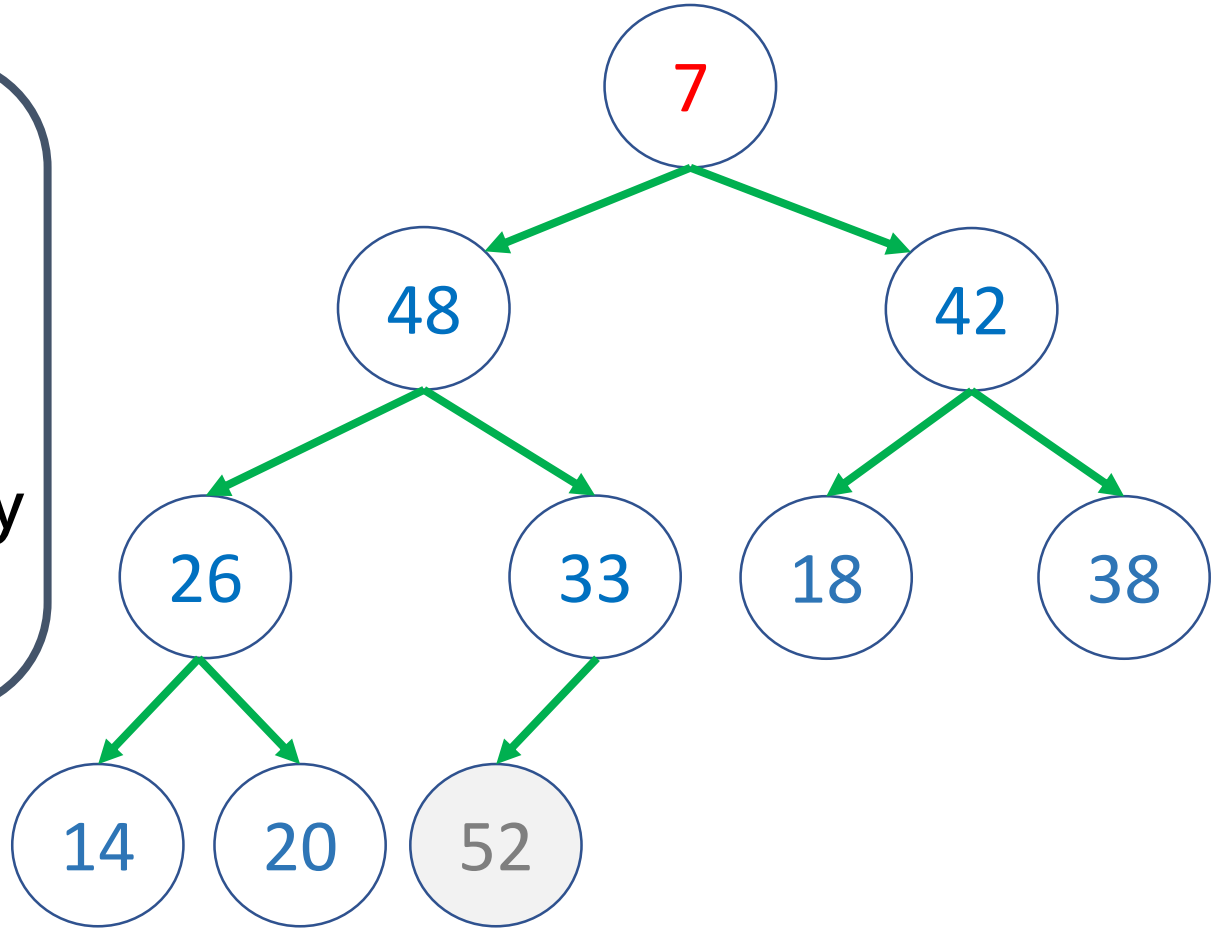


7	48	42	26	33	18	38	14	20	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

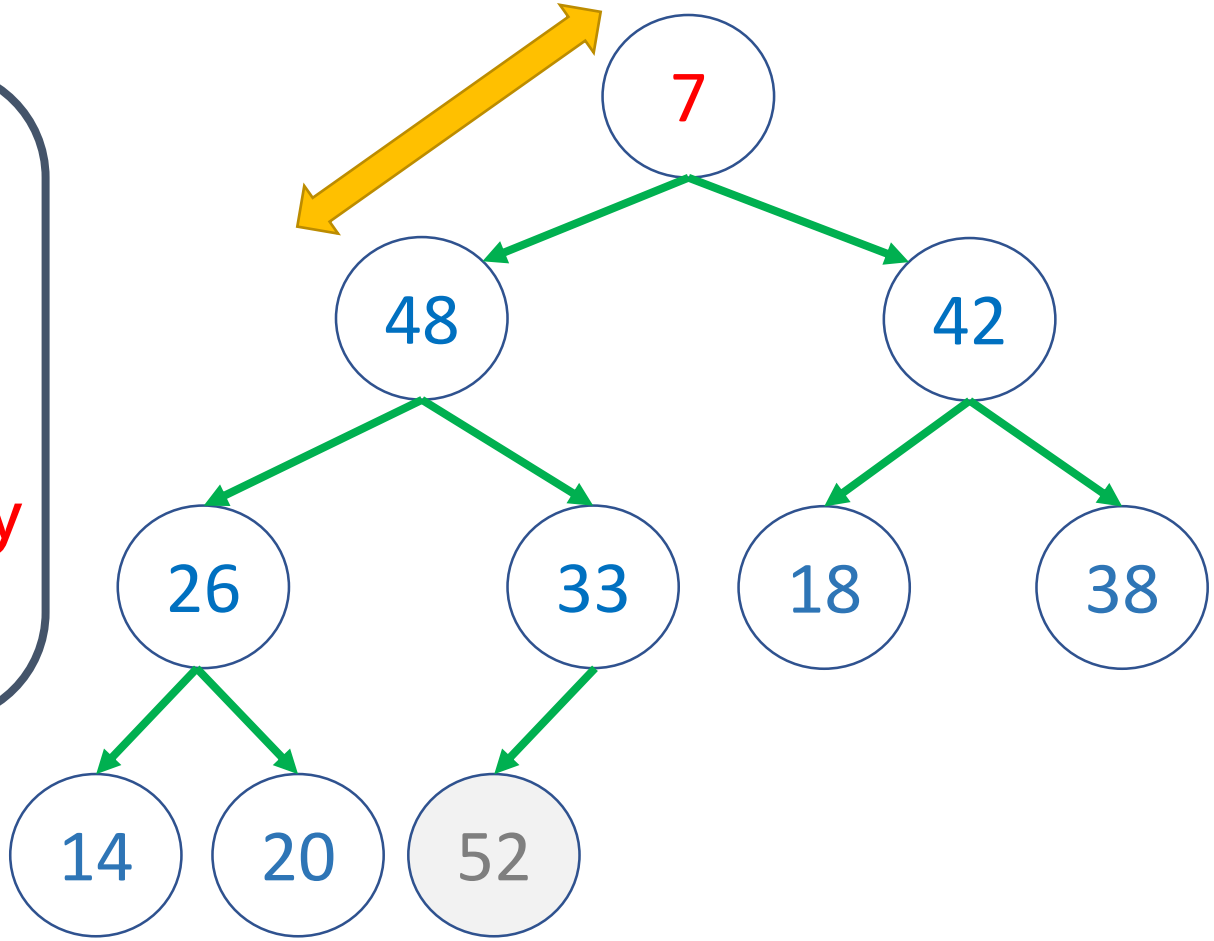


7	48	42	26	33	18	38	14	20	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

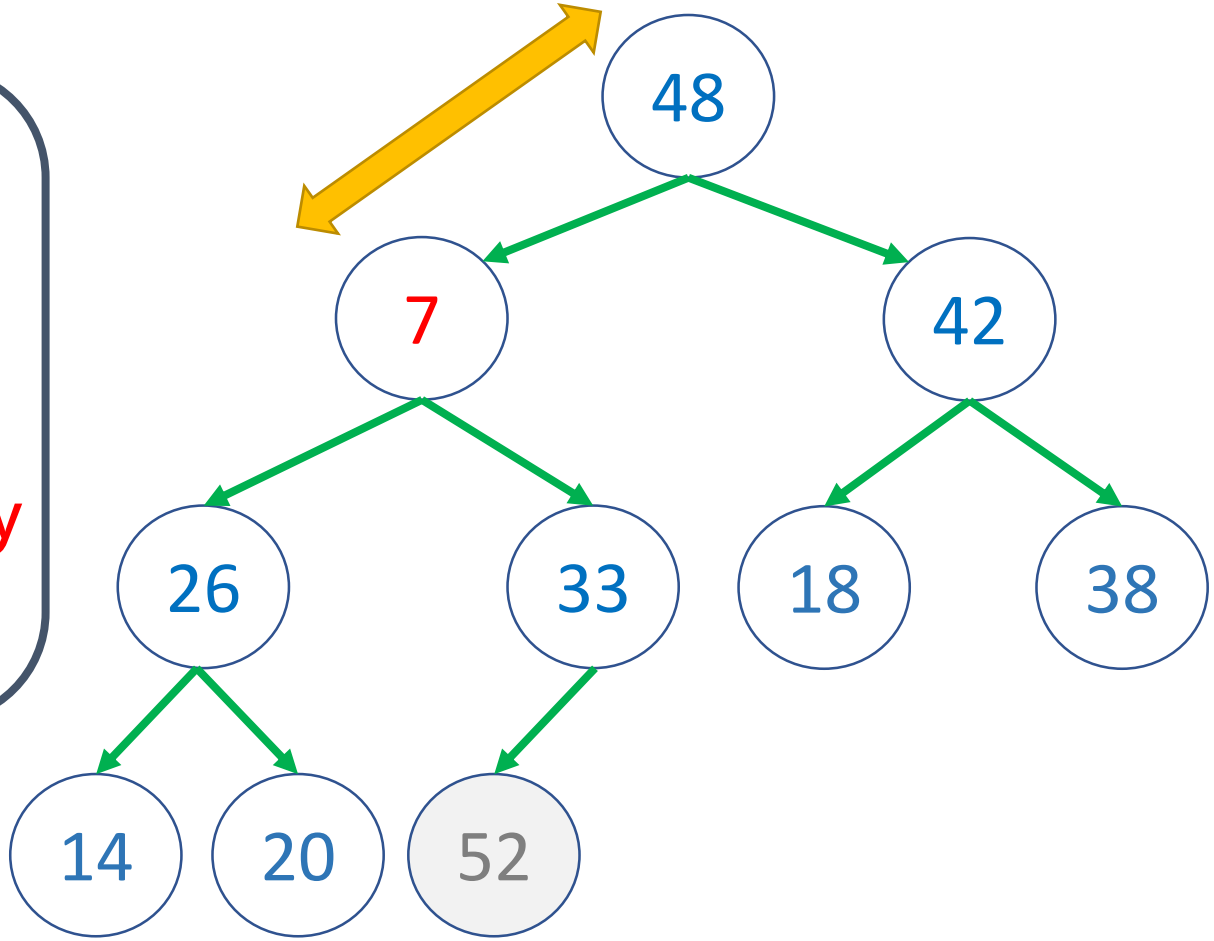


7	48	42	26	33	18	38	14	20	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

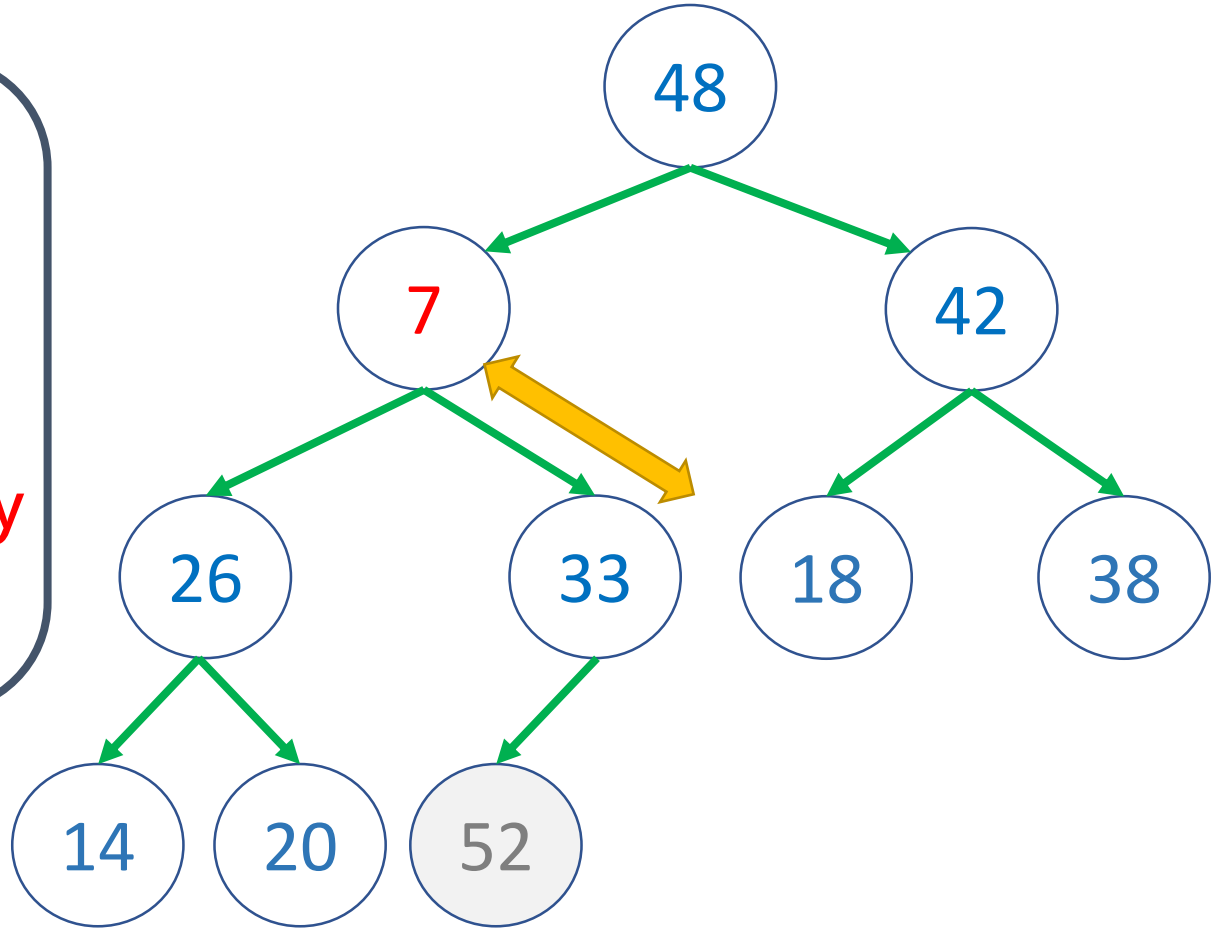


48	7	42	26	33	18	38	14	20	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

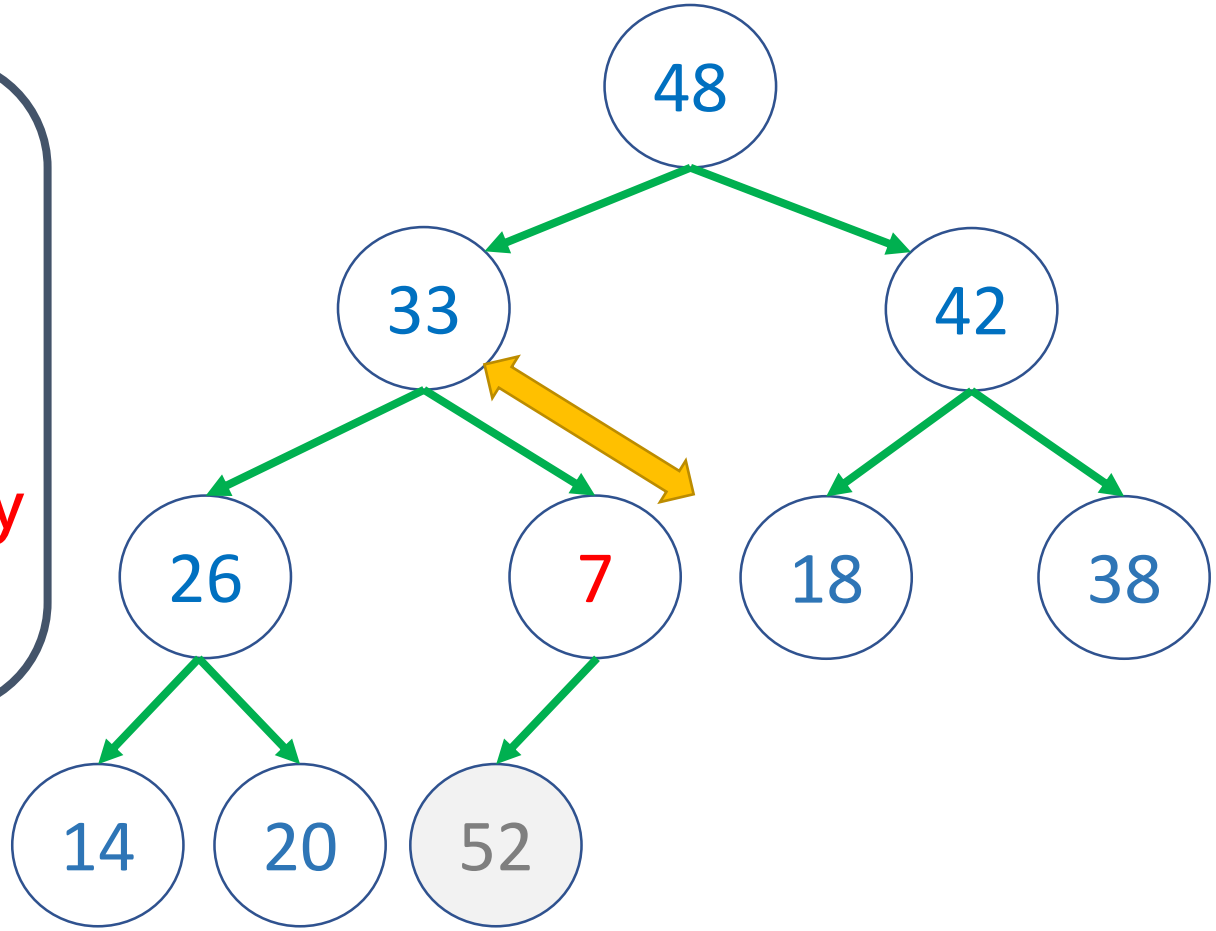


48	7	42	26	33	18	38	14	20	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

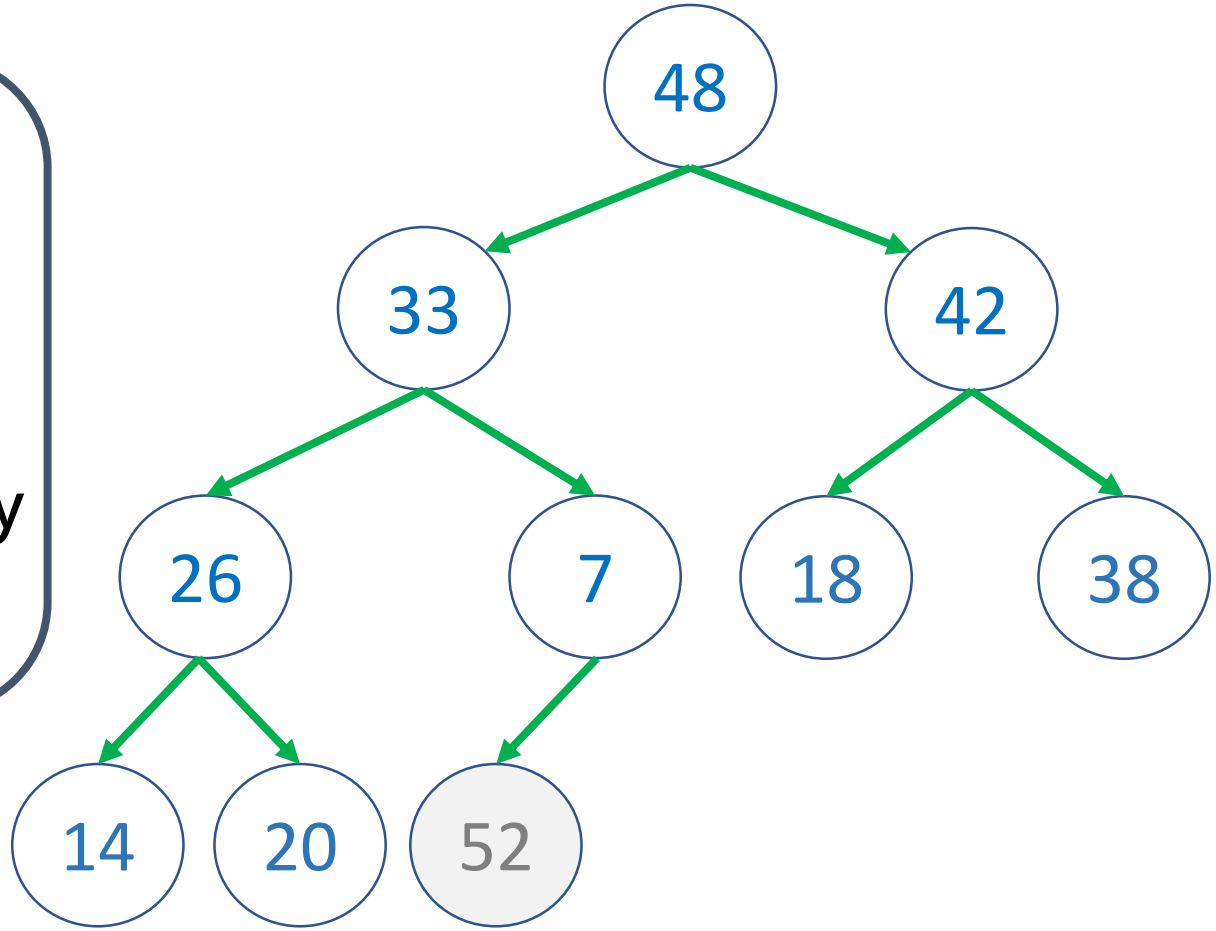


48	33	42	26	7	18	38	14	20	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

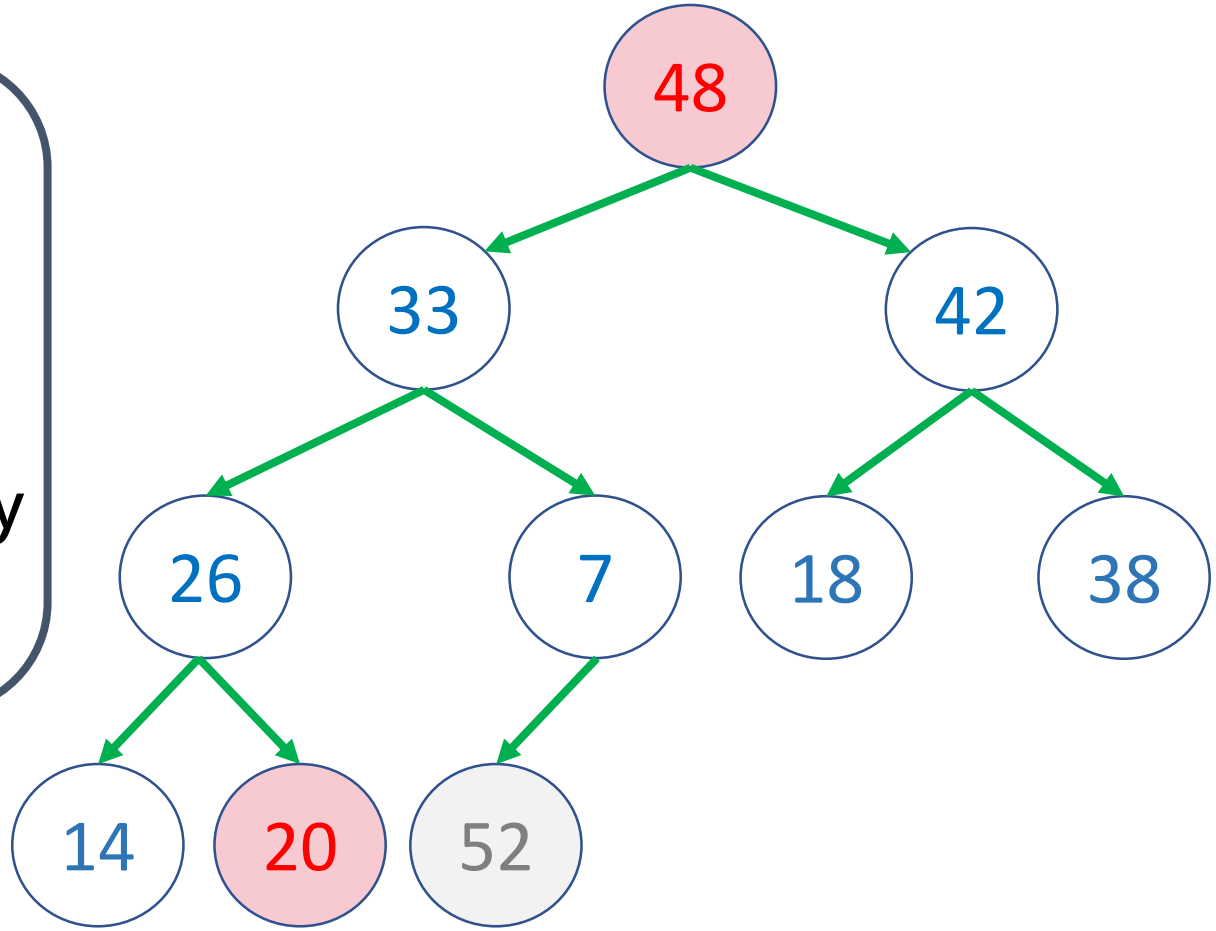


48	33	42	26	7	18	38	14	20	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

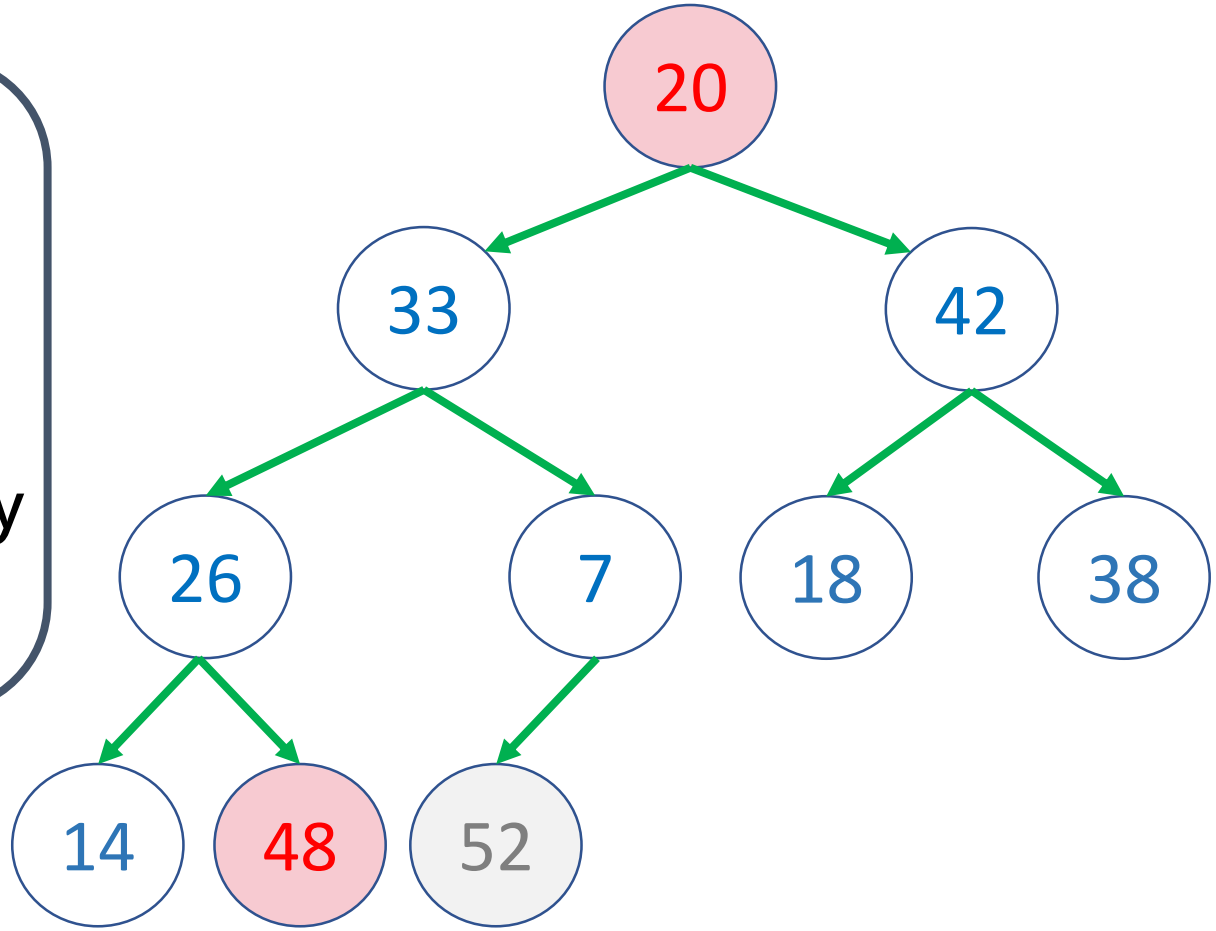


48	33	42	26	7	18	38	14	20	52

二元堆疊

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

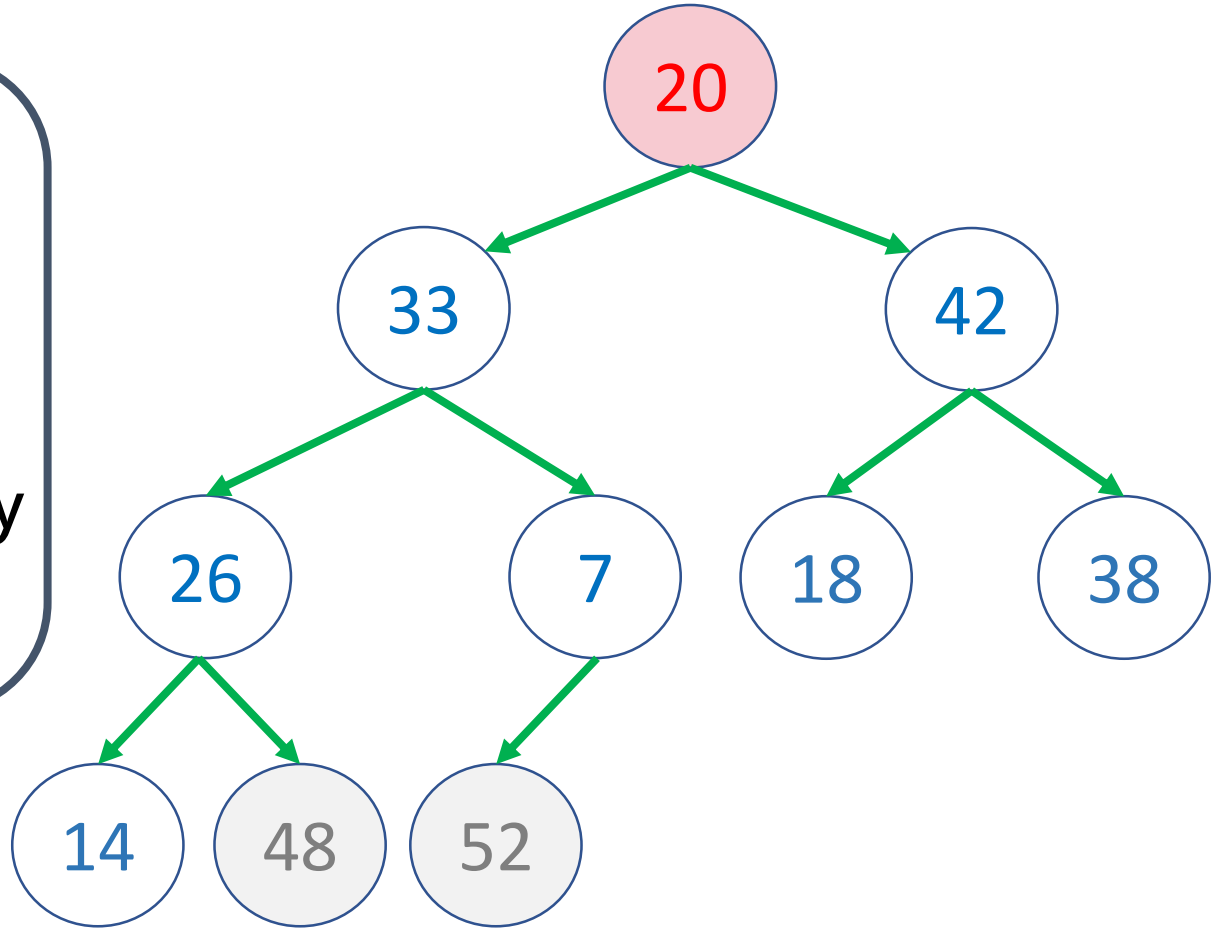


20	33	42	26	7	18	38	14	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

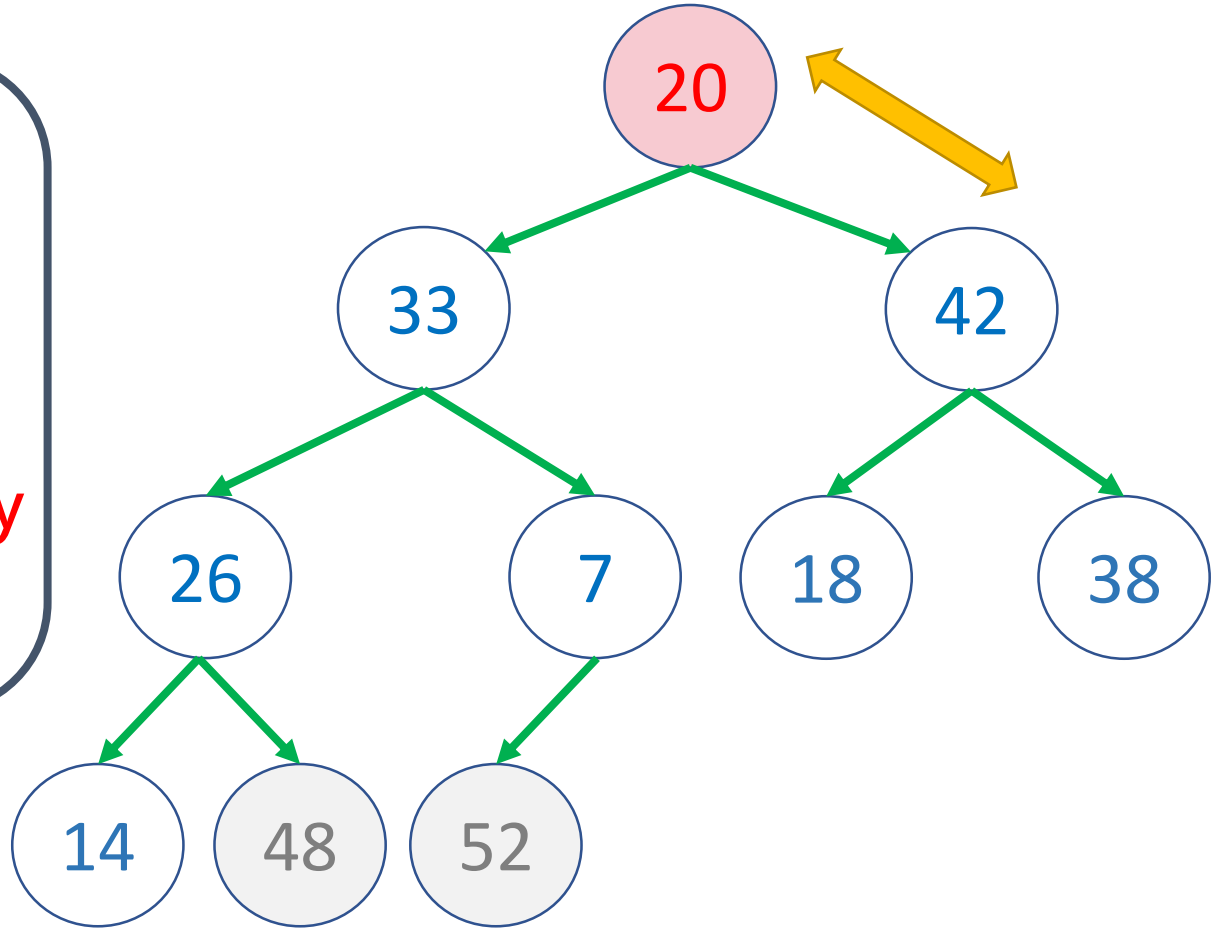


20	33	42	26	7	18	38	14	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

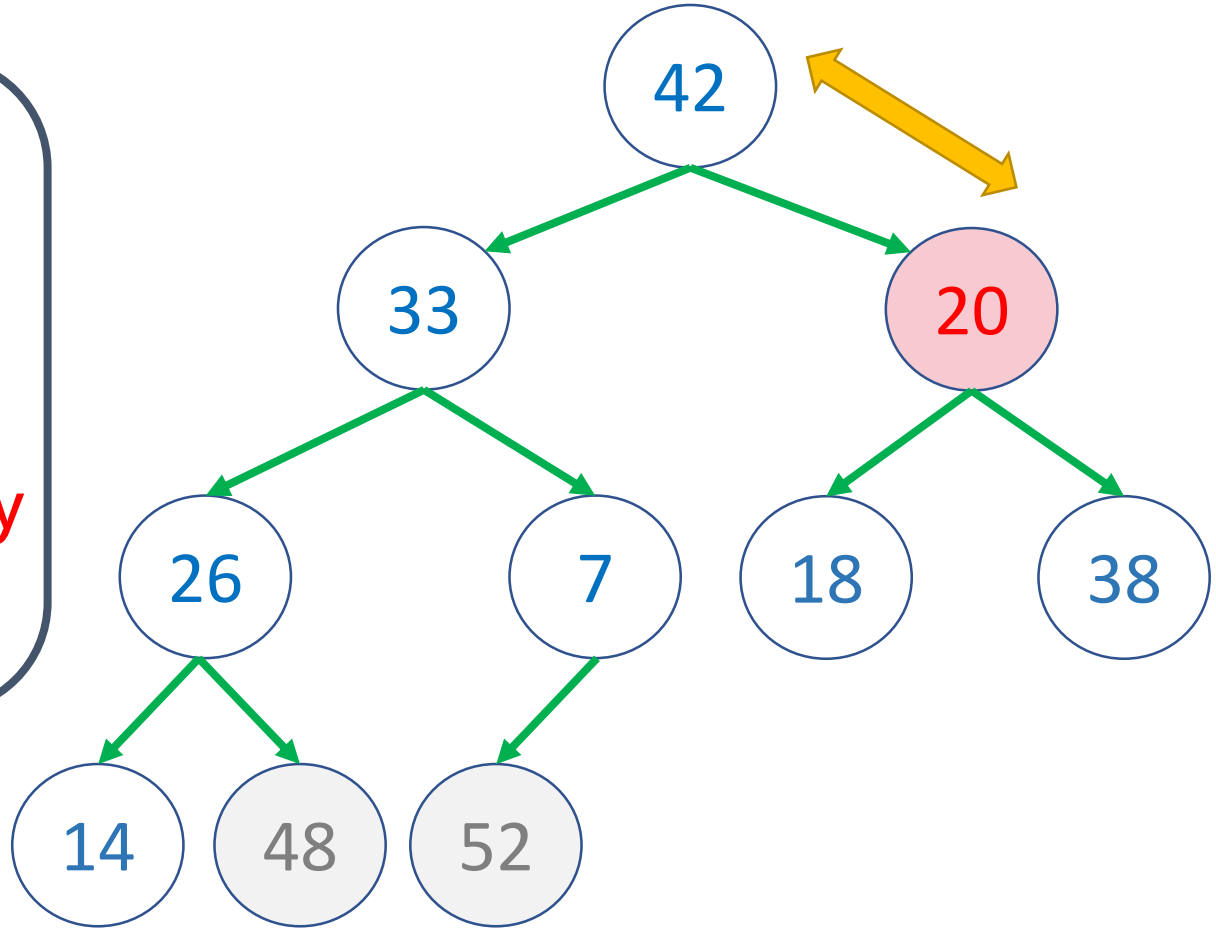


20	33	42	26	7	18	38	14	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

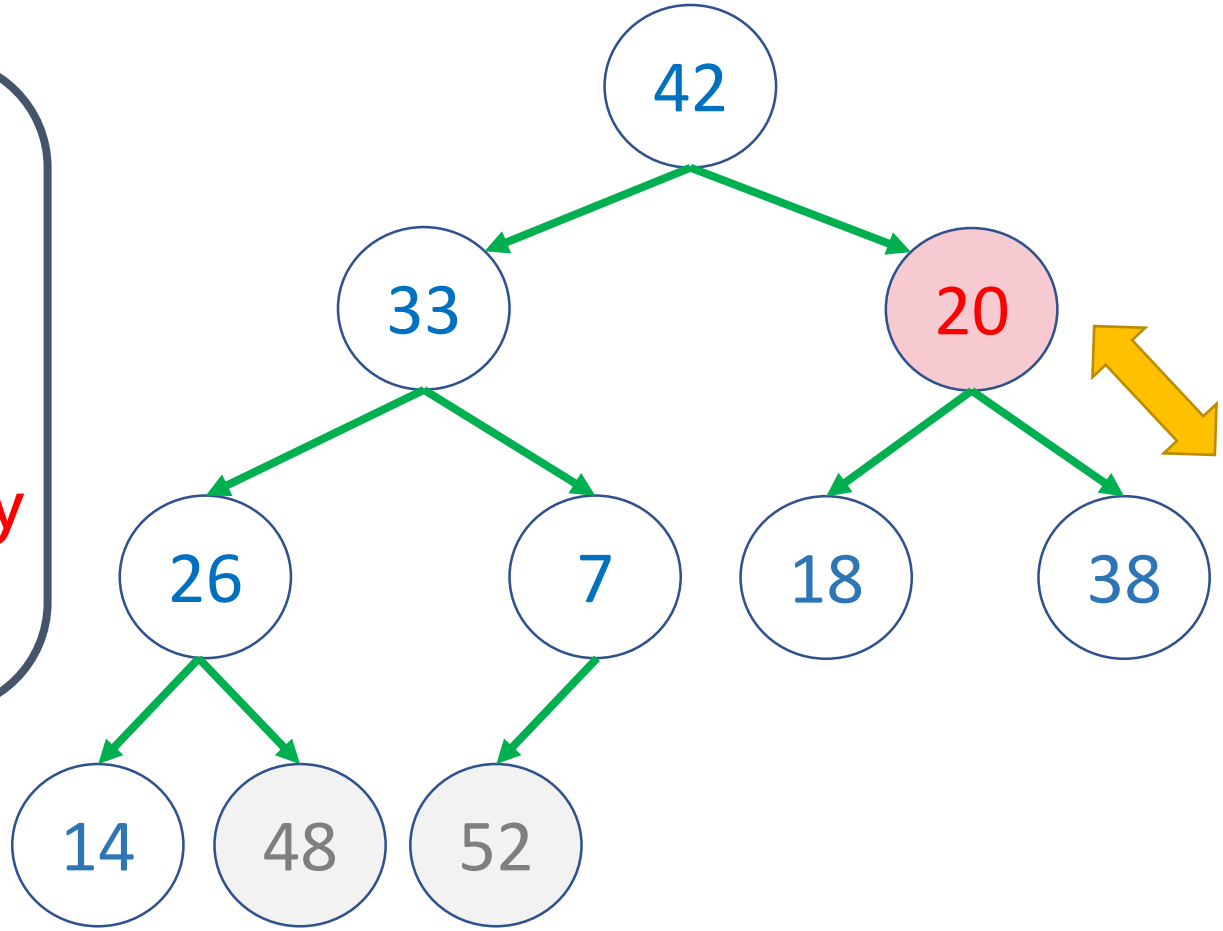


42	33	20	26	7	18	38	14	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

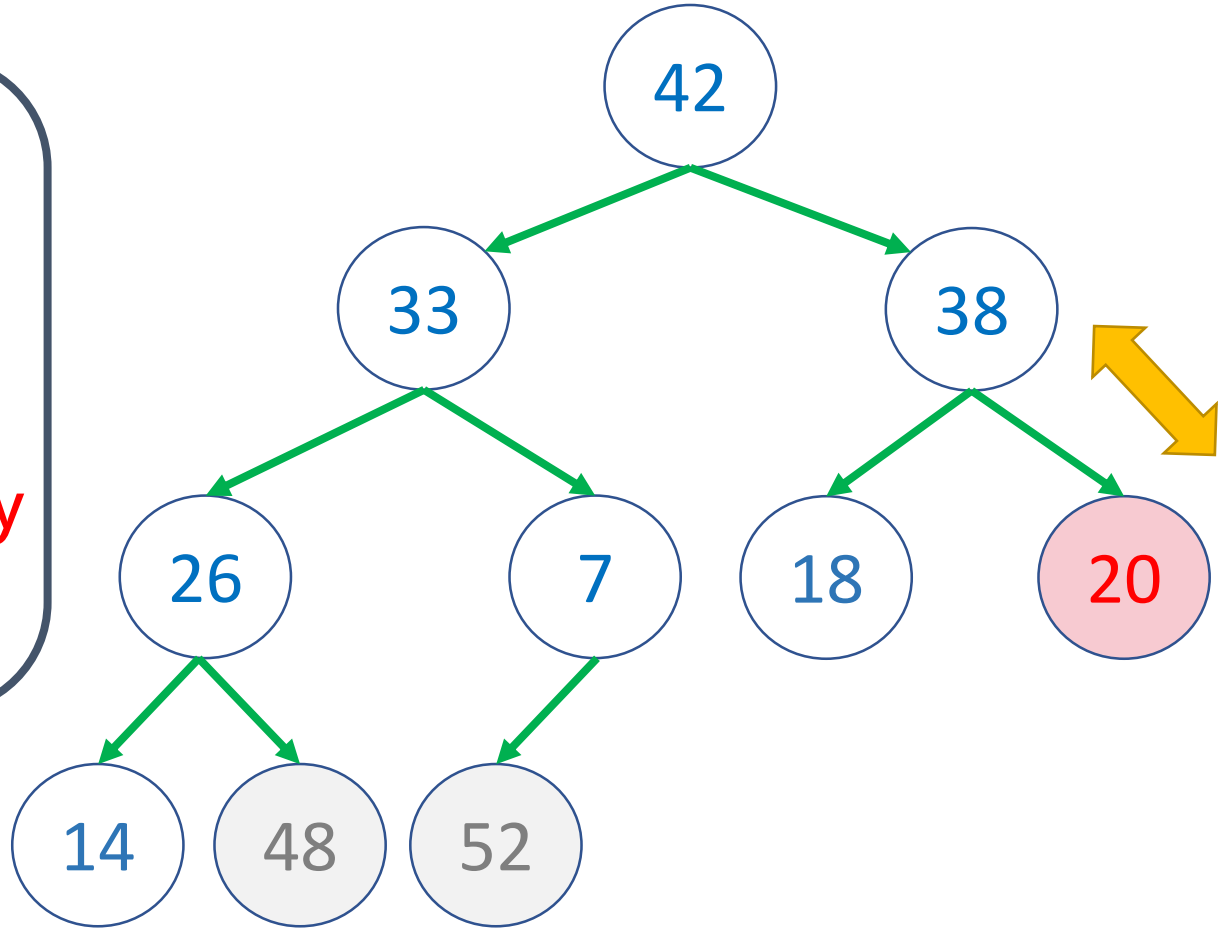


42	33	20	26	7	18	38	14	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

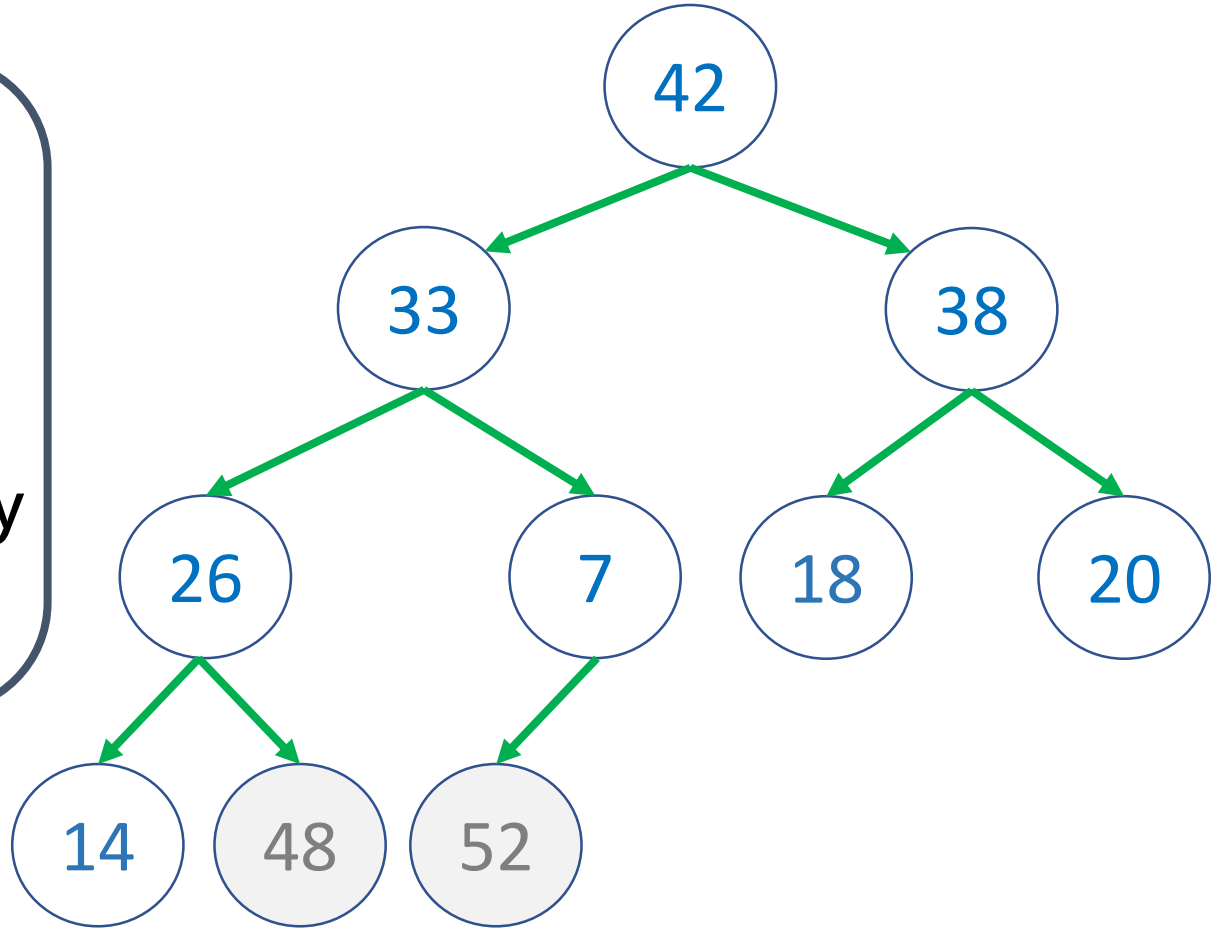


42	33	38	26	7	18	20	14	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

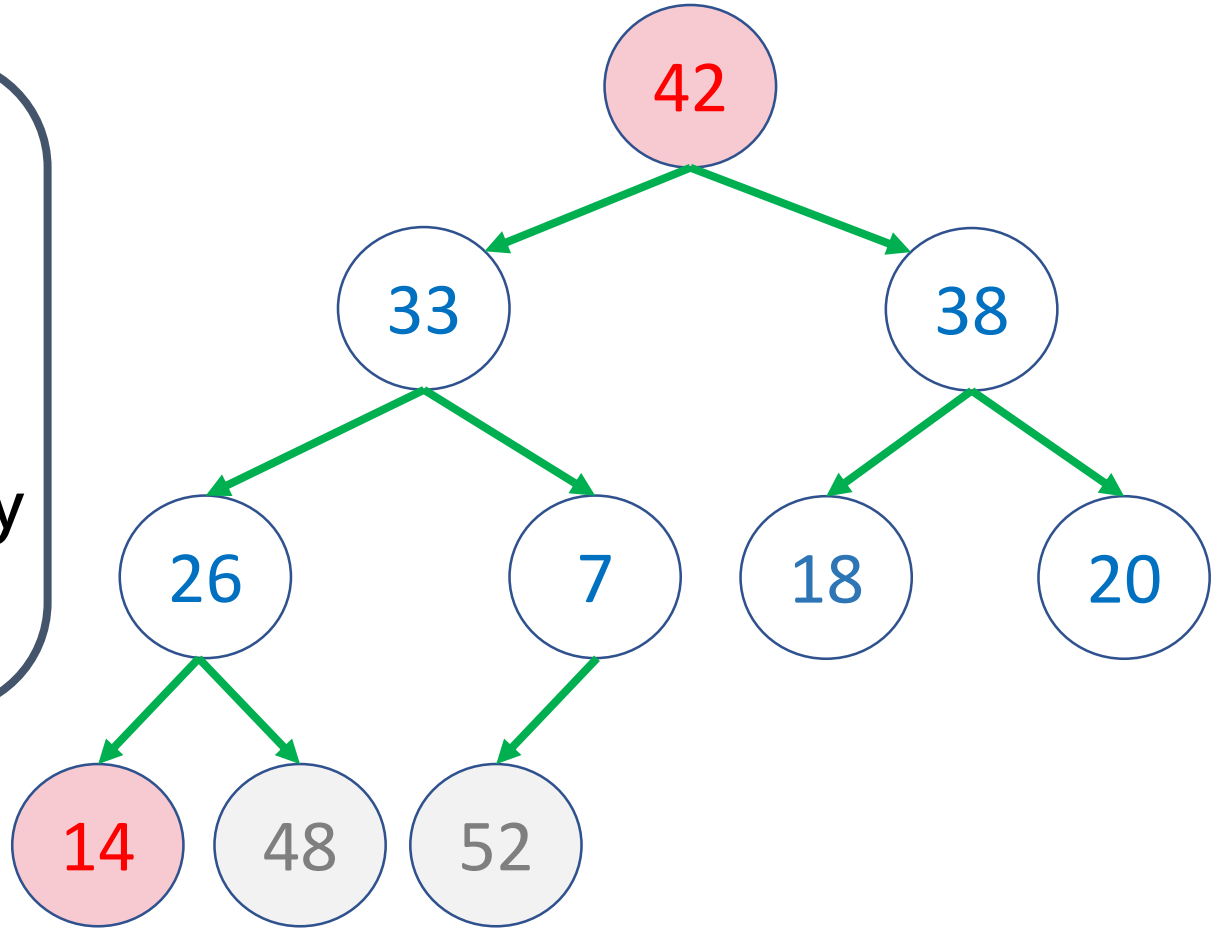


42	33	38	26	7	18	20	14	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

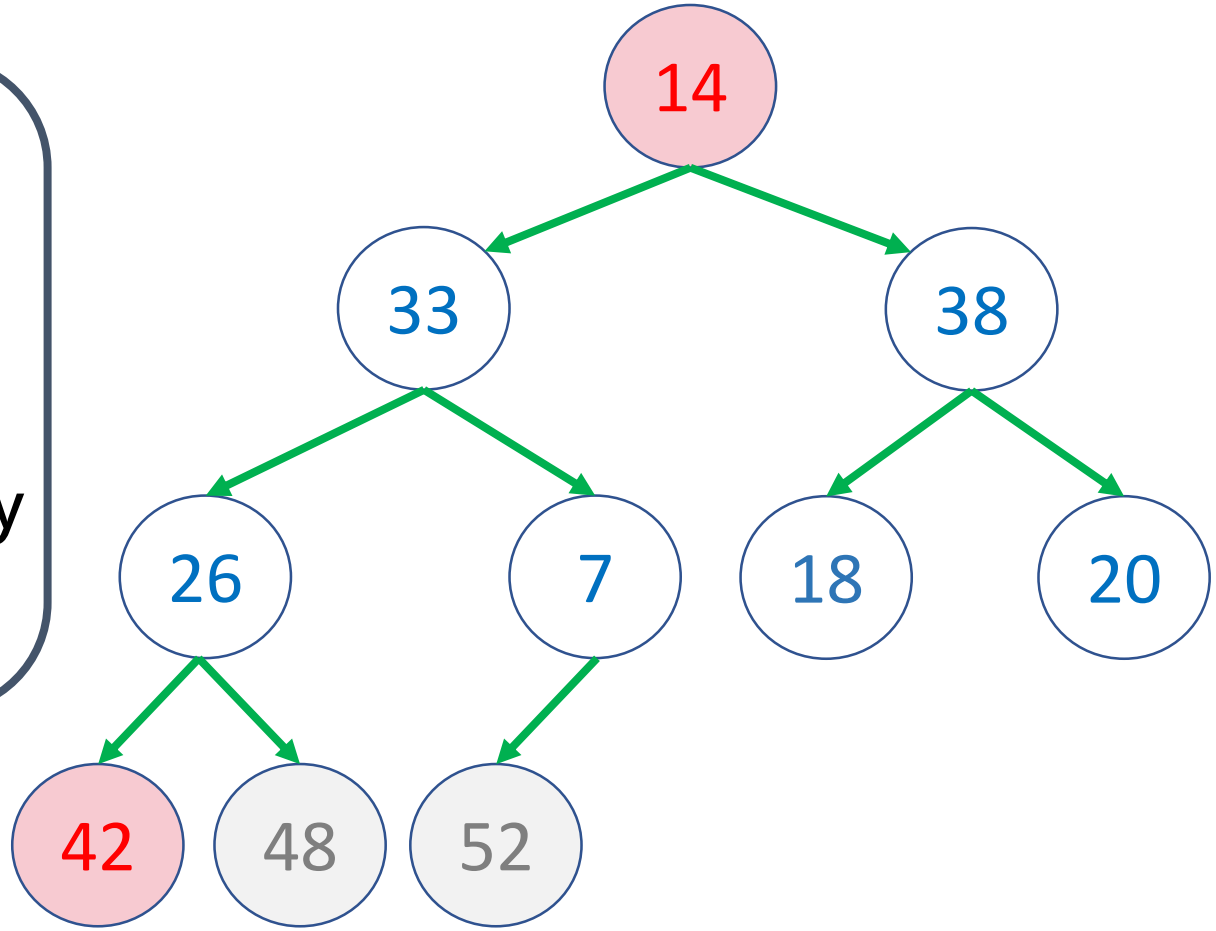


42	33	38	26	7	18	20	14	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

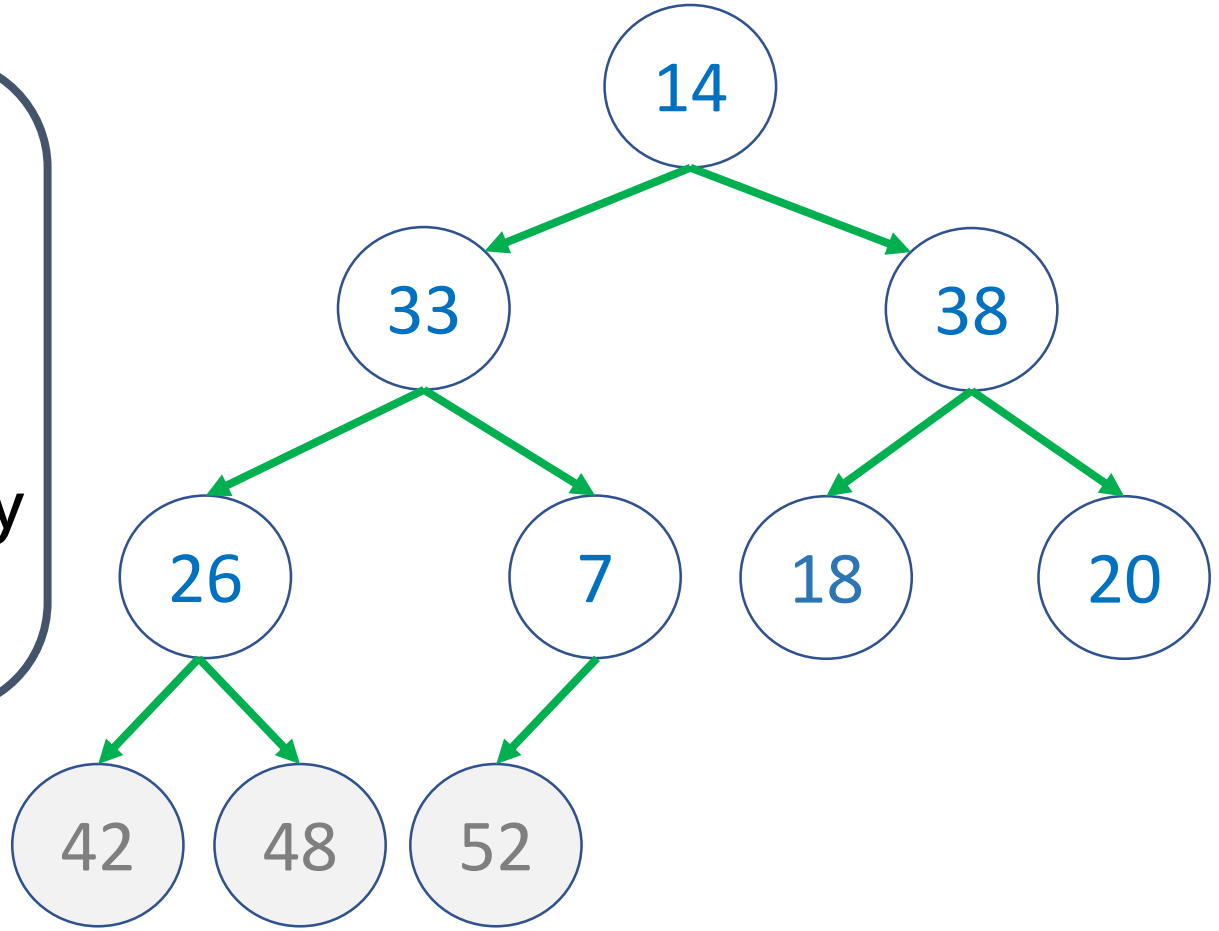


14	33	38	26	7	18	20	42	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

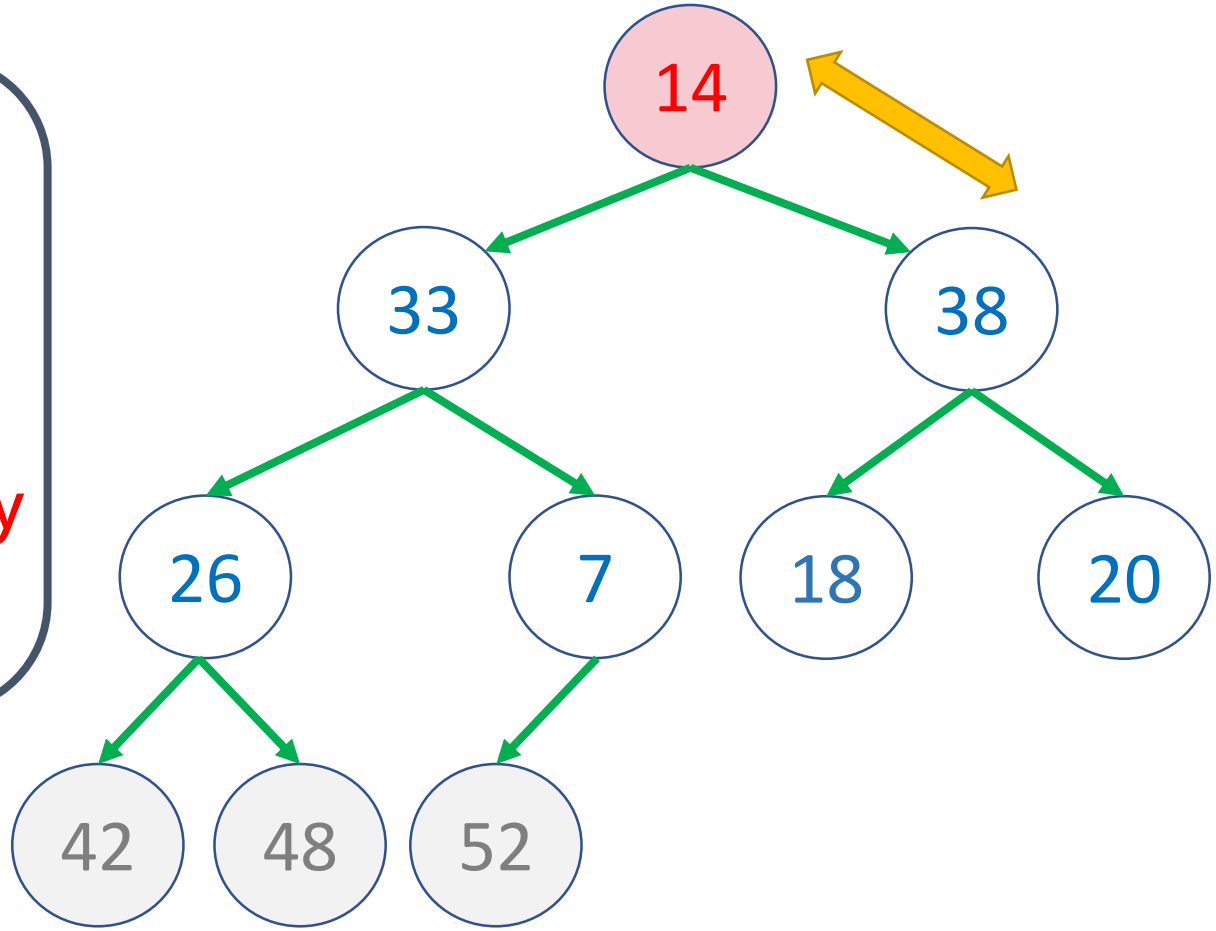


14	33	38	26	7	18	20	42	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

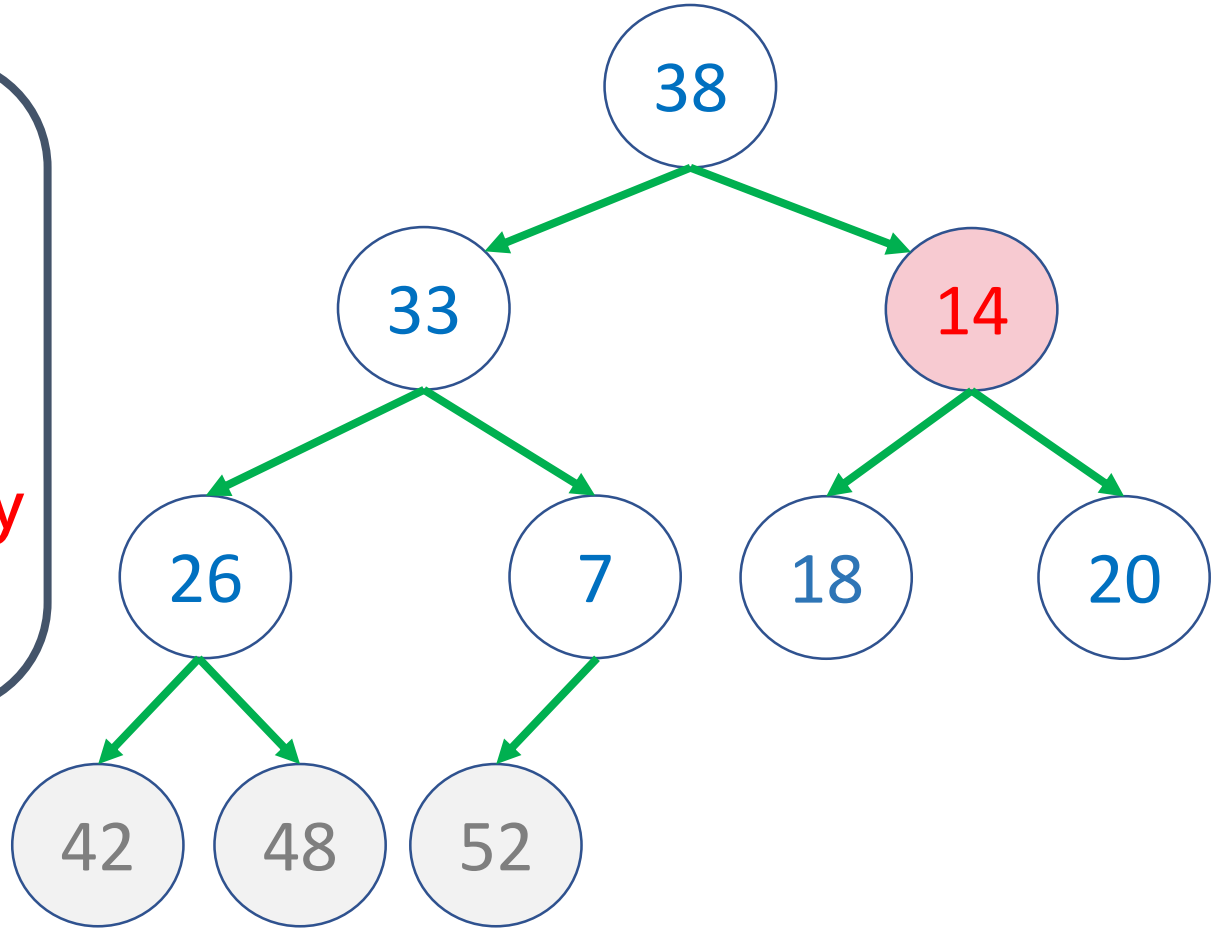


14	33	38	26	7	18	20	42	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

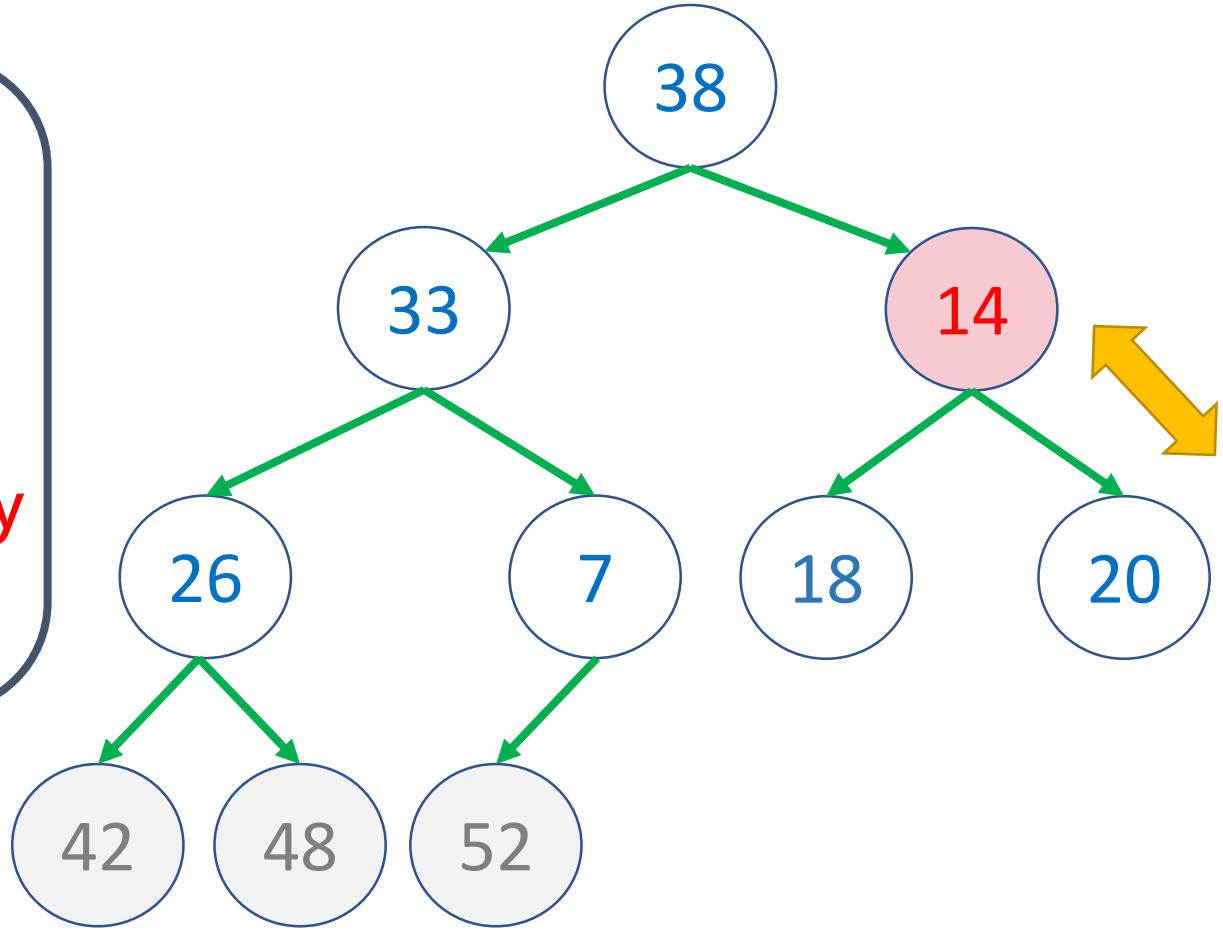


38	33	14	26	7	18	20	42	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

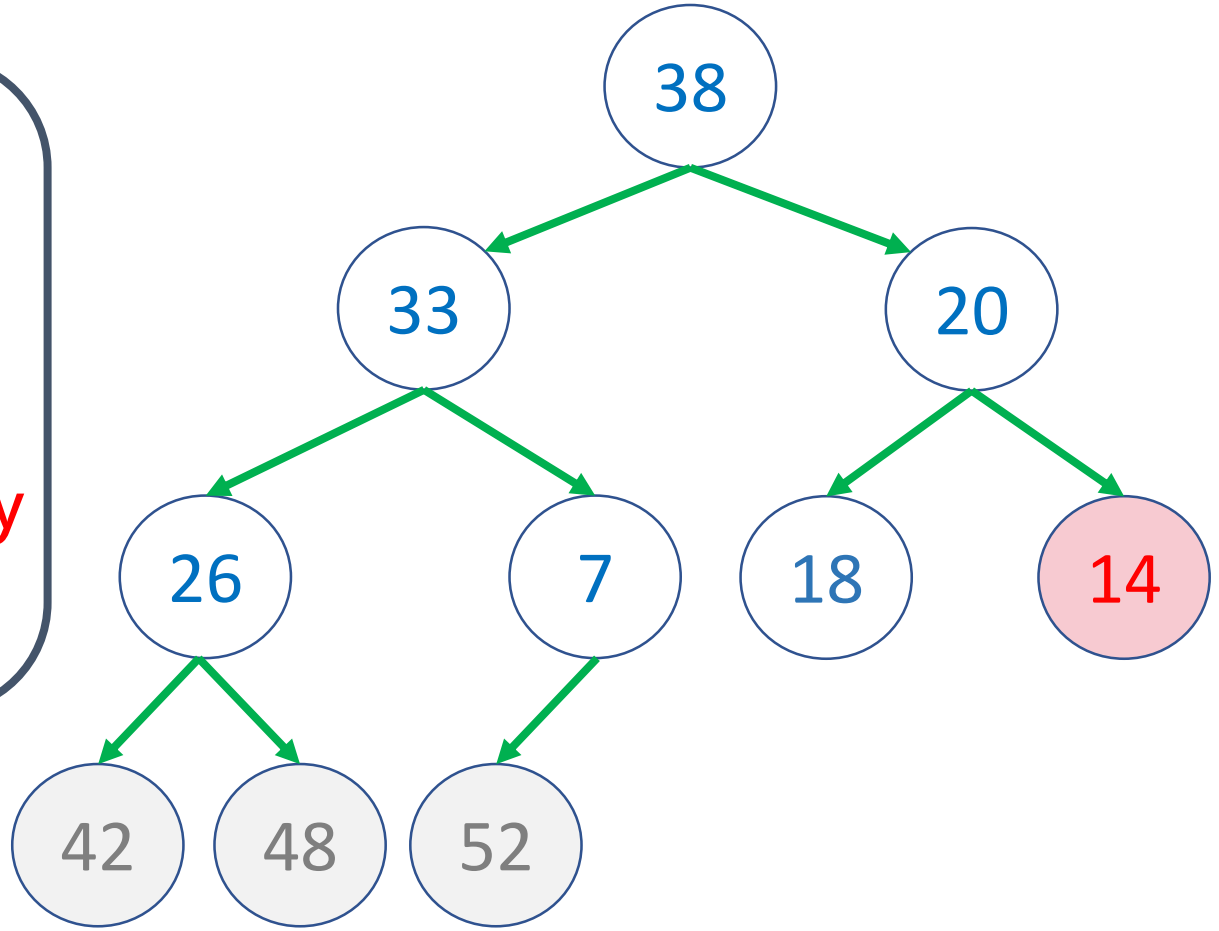


38	33	14	26	7	18	20	42	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

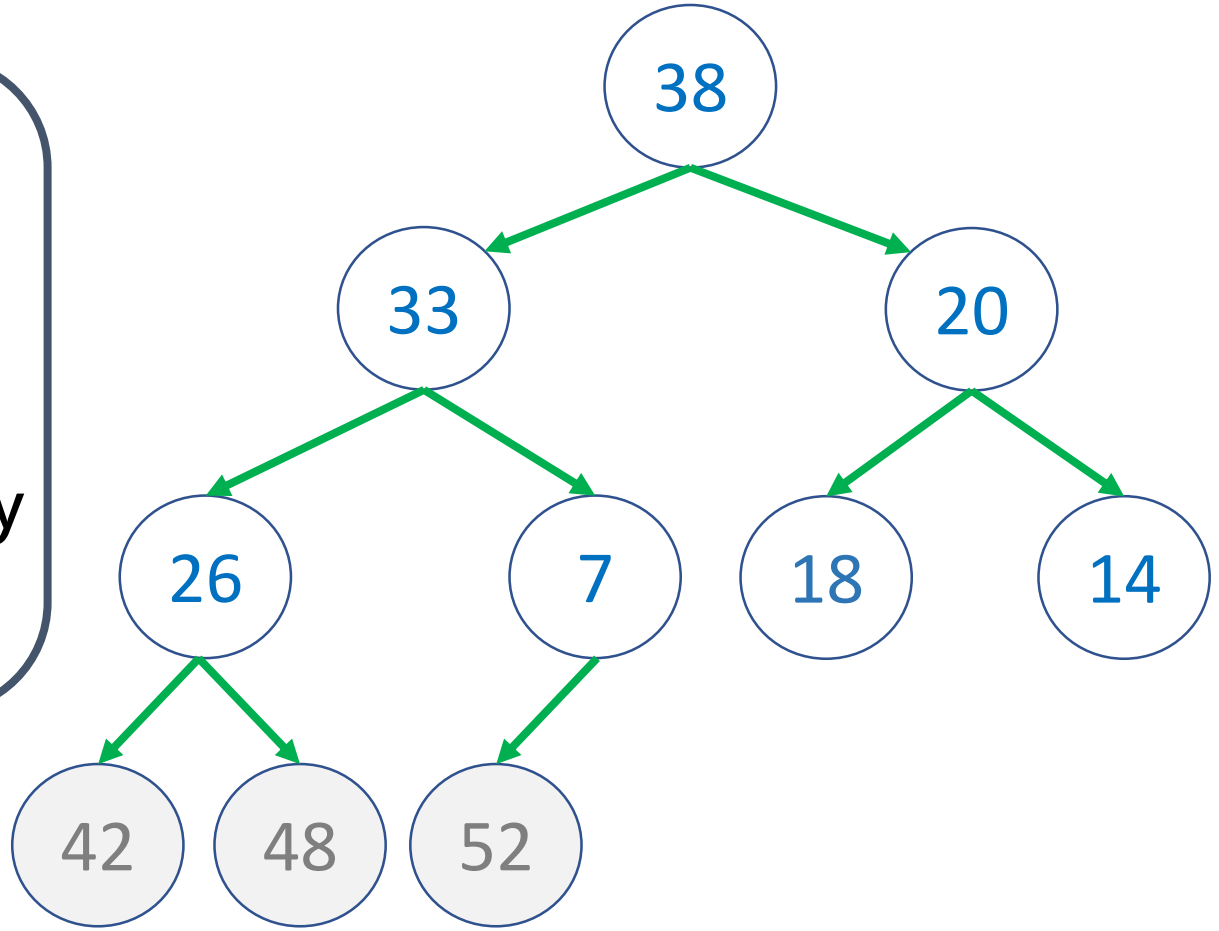


38	33	20	26	7	18	14	42	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

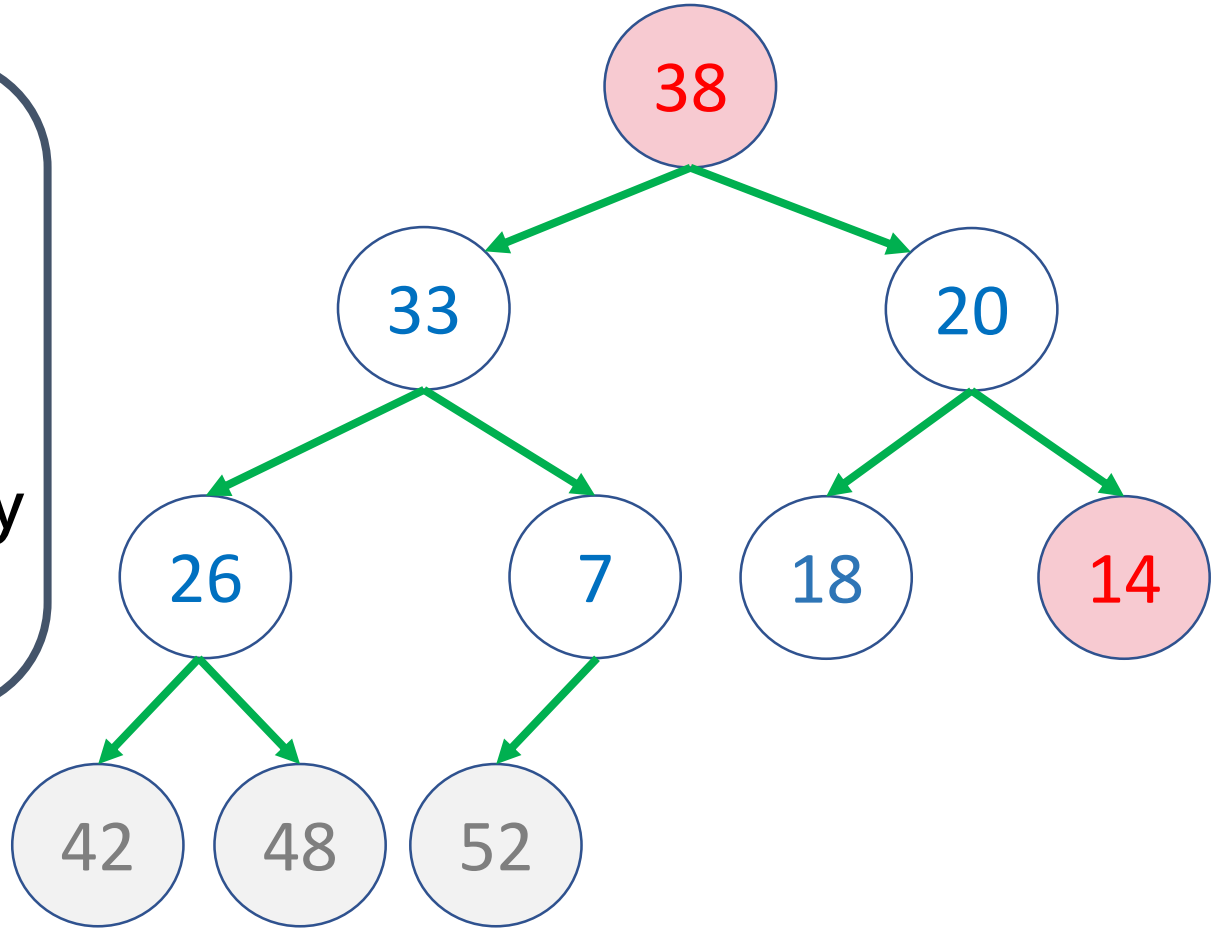


38	33	20	26	7	18	14	42	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

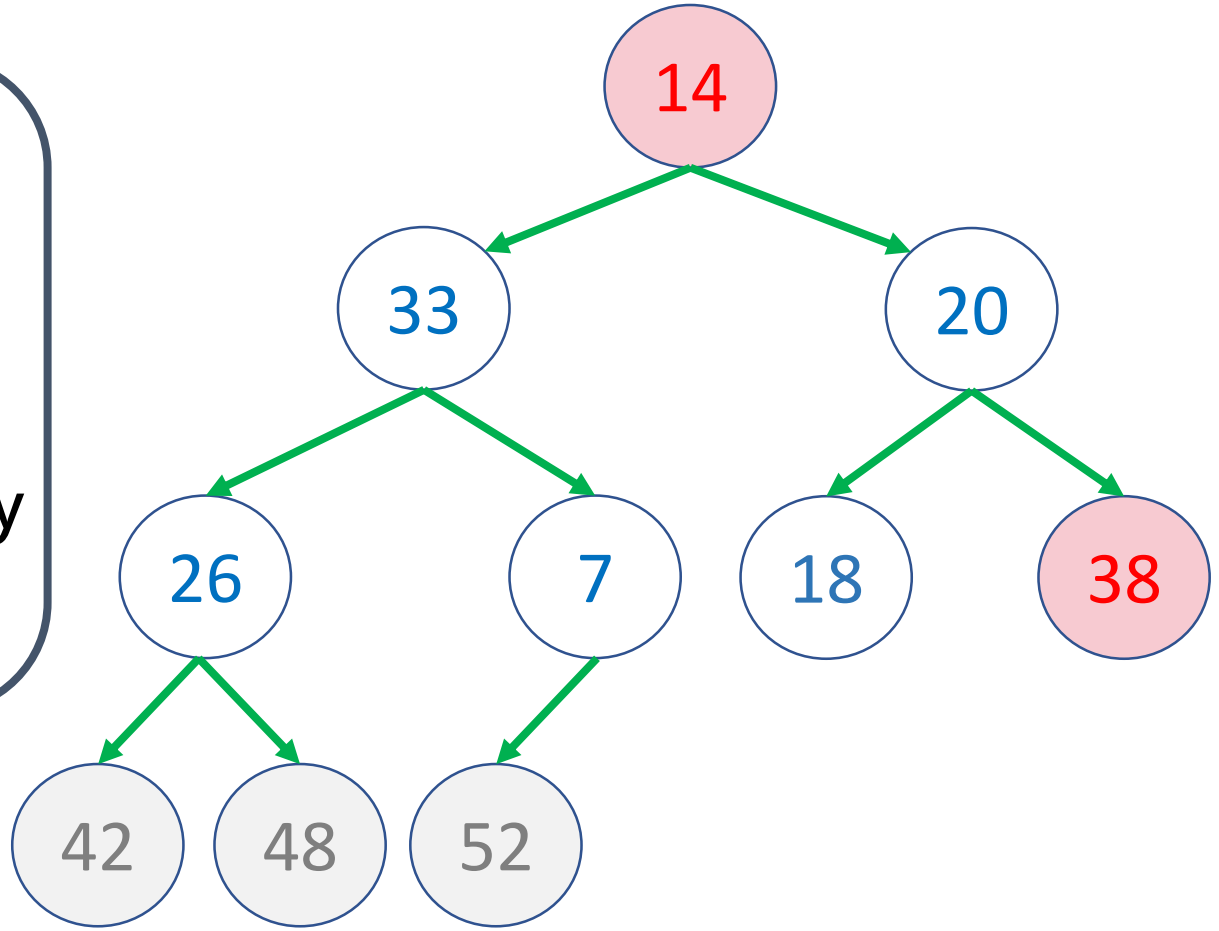


38	33	20	26	7	18	14	42	48	52

二元堆疊

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

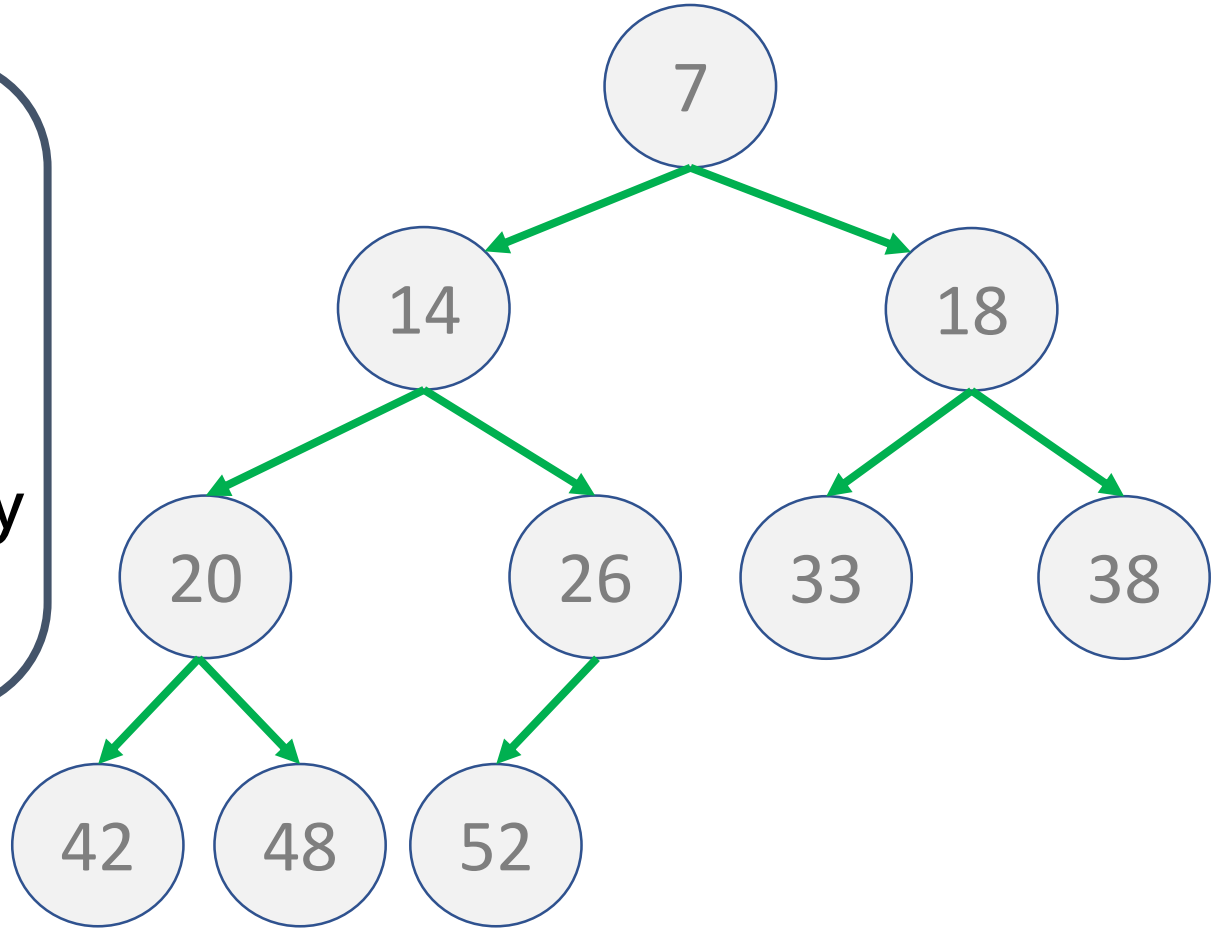


14	33	20	26	7	18	38	42	48	52

Heap Sort

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

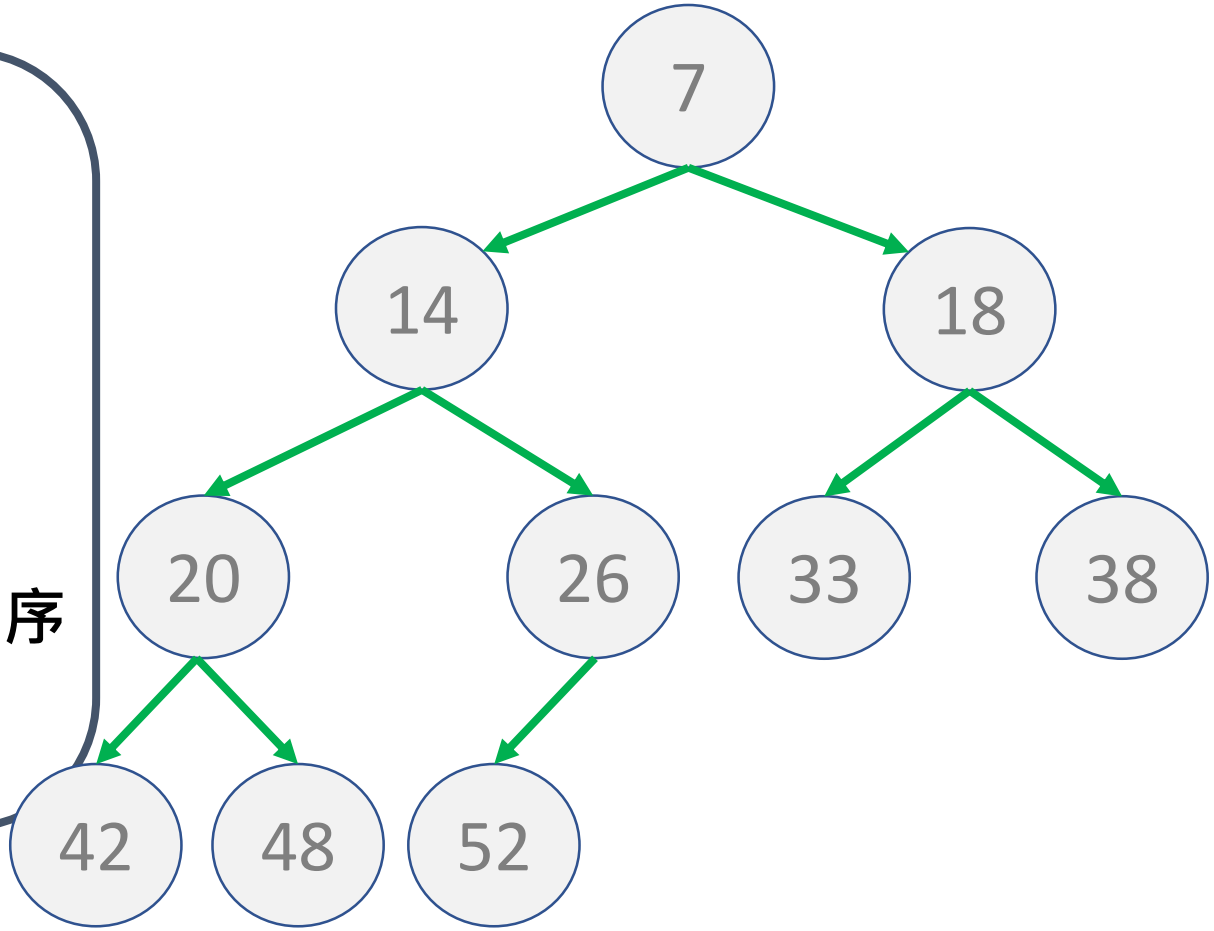


7	14	18	20	26	33	38	42	48	52

Heap Sort

Heap Sort

1. 每次形成 max/小-heap 時
   - ➢ 根節點就是該二元樹的最大/小值
2. 首項與末項對調後最大/小值便在最後
3. 依序取出剩餘數列中最大/最小即完成排序
4. 升冪用 max-heap、降冪用 min-heap

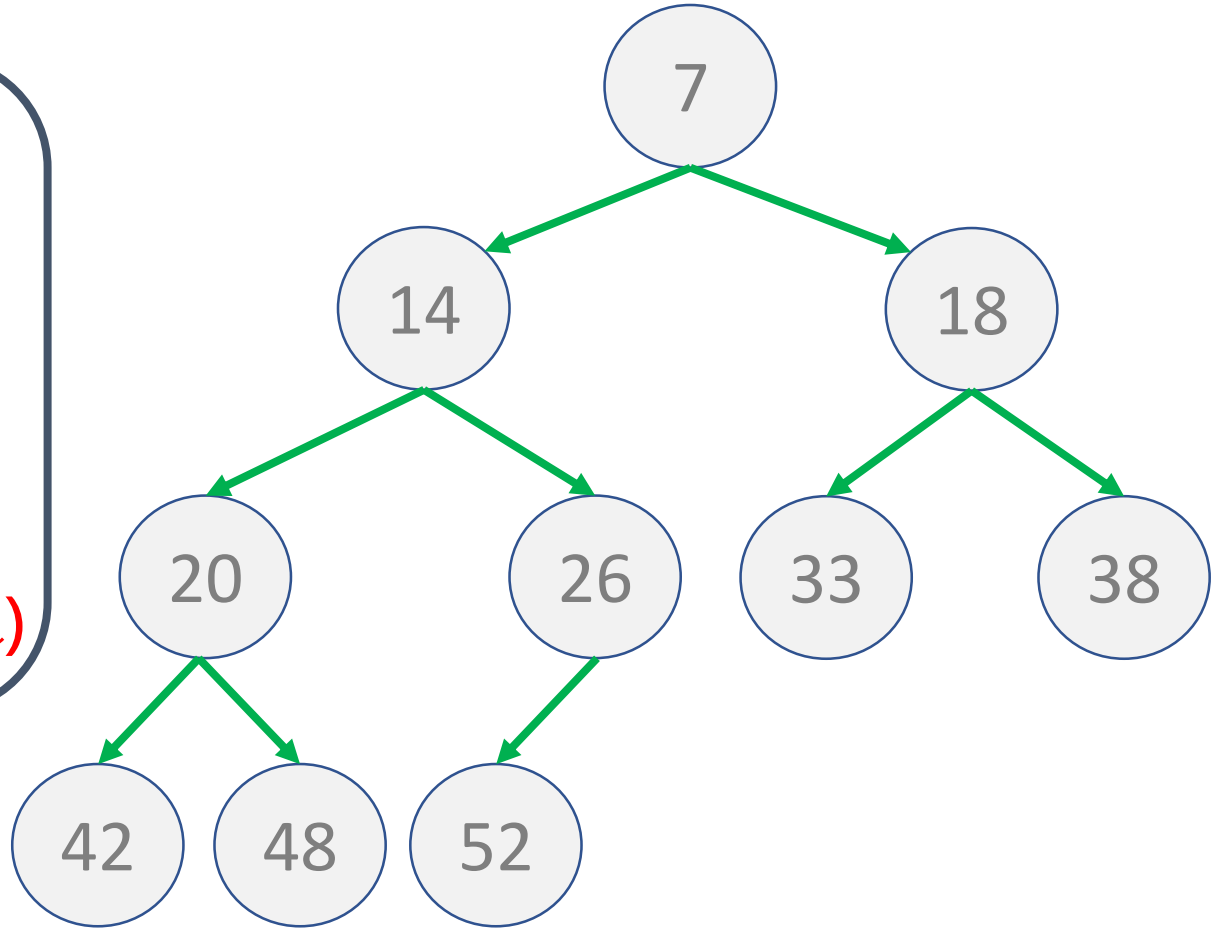


7	14	18	20	26	33	38	42	48	52

Heap Sort

Heap Sort 複雜度

1. 建立二元堆疊：$O(n/2) = O(n)$
2. 每輪 heapify： $O(\log_2 n)$
3. 幾輪 heapify：$O(n)$
4. 總複雜度：$O(n)+O(n \log_2 n) = O(n\log_2 n)$



7	14	18	20	26	33	38	42	48	52

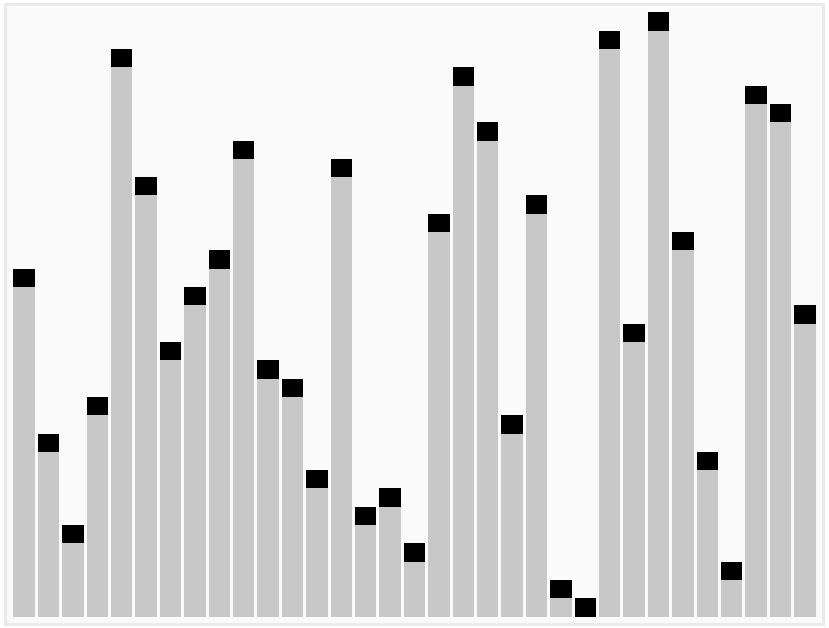
Example Code

Mission

實作 Heap Sort

Quick Sort

- 隨機選出一筆資料當基準點
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1



Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

35	52	68	12	47	52	36	52	74	27
	A				B		C		

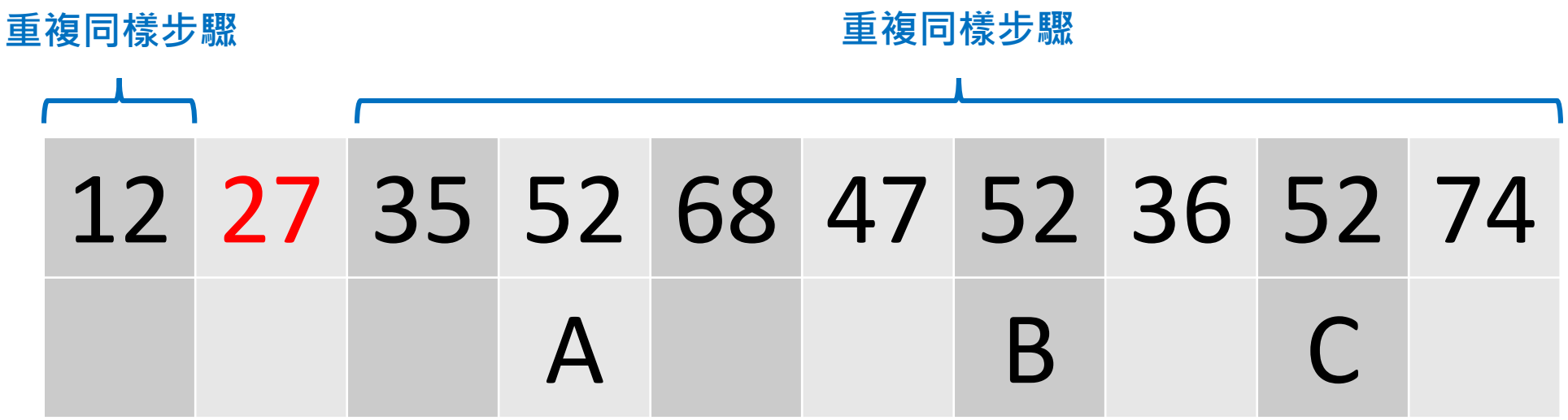
Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

12	27	35	52	68	47	52	36	52	74
			A			B		C	

Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1



Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

重複同樣步驟

12	27	35	52	68	47	52	36	52	74
			A			B		C	

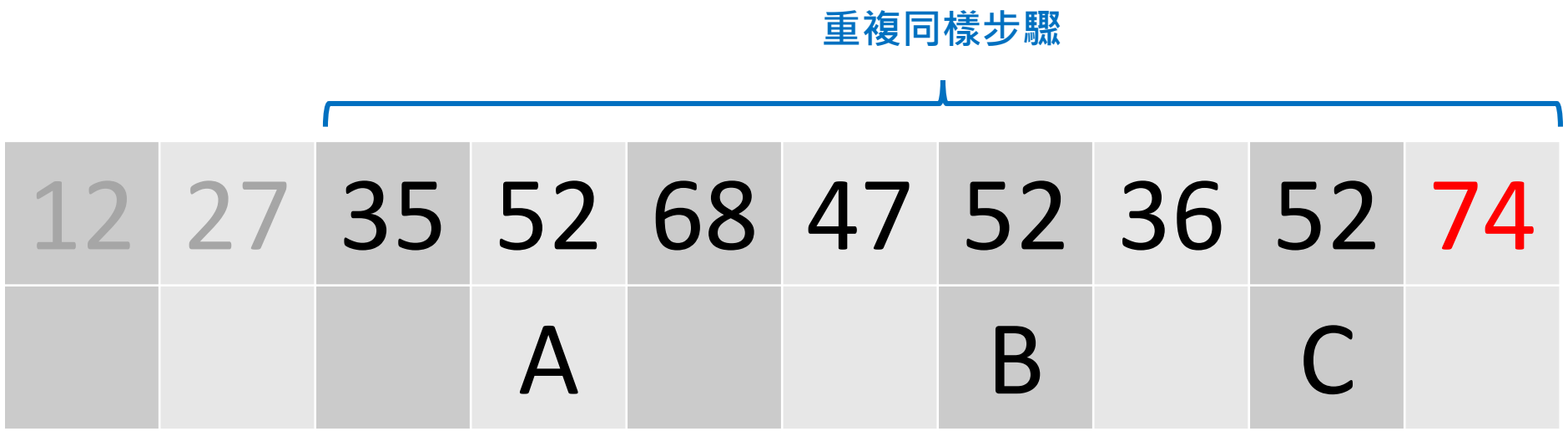
Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

12	27	35	52	68	47	52	36	52	74
			A			B		C	

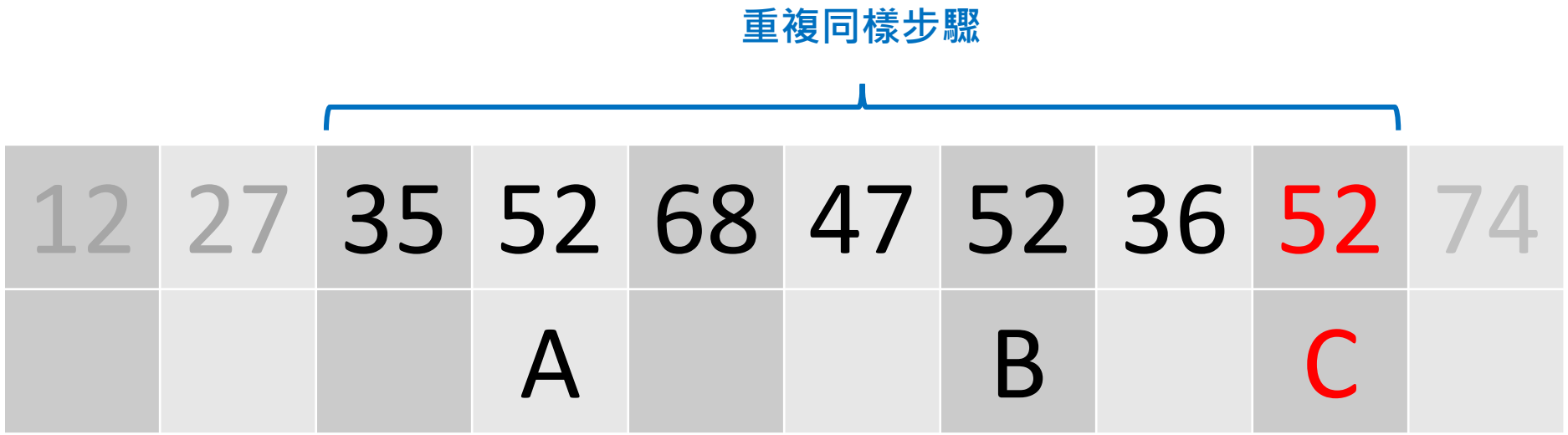
Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1



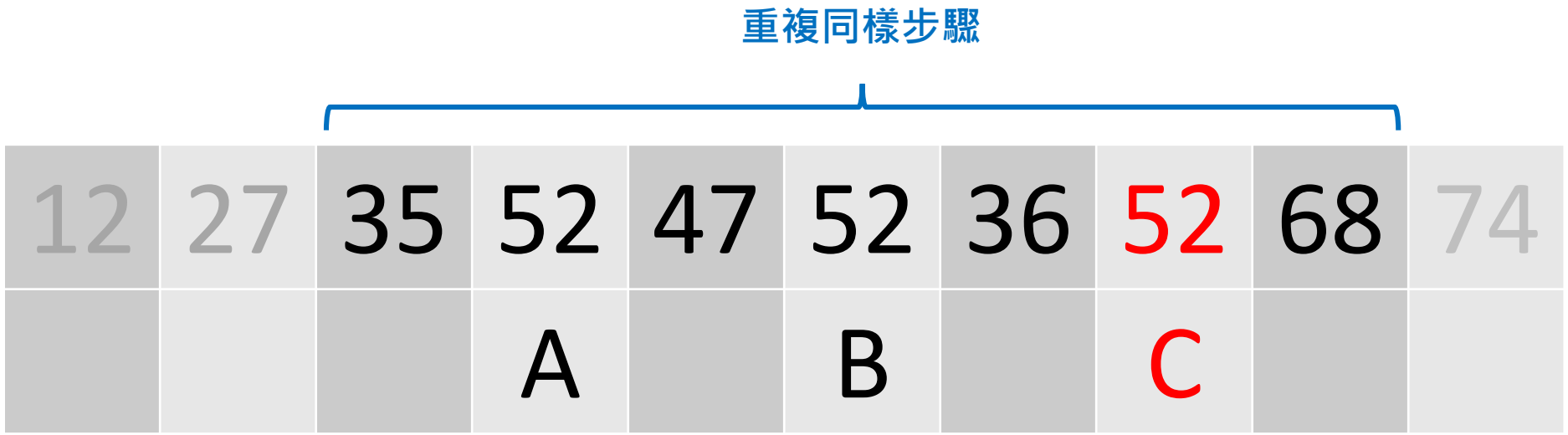
Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1



Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1



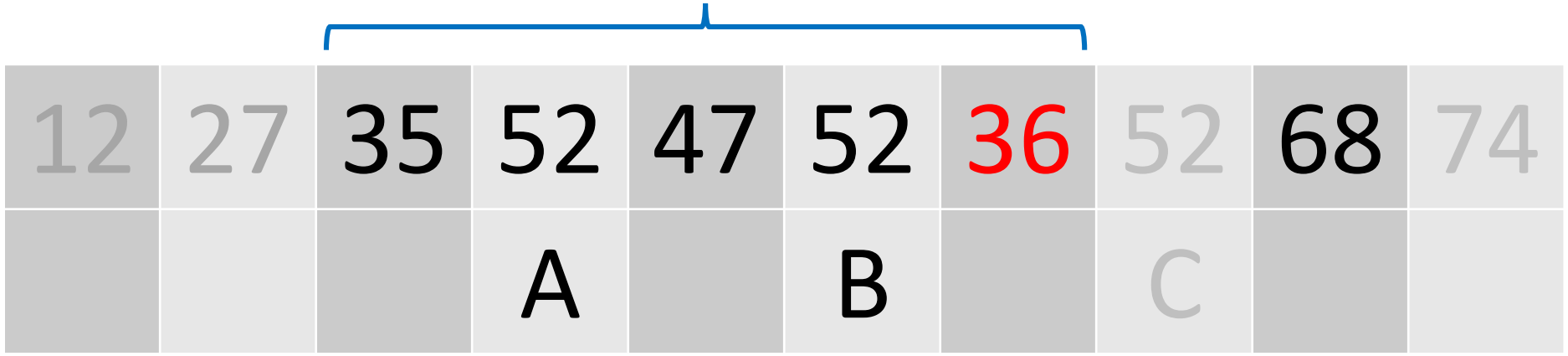
Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

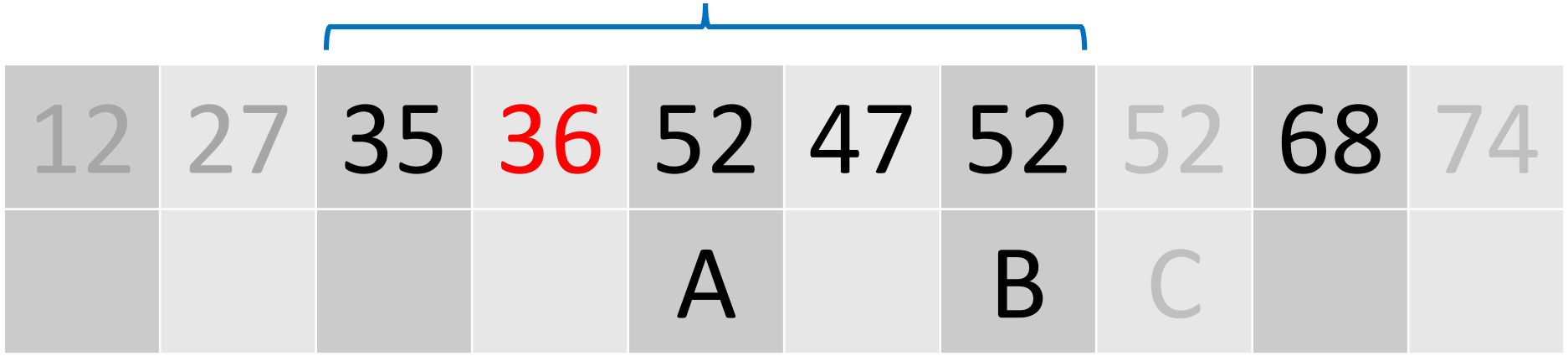
重複同樣步驟



Quick Sort

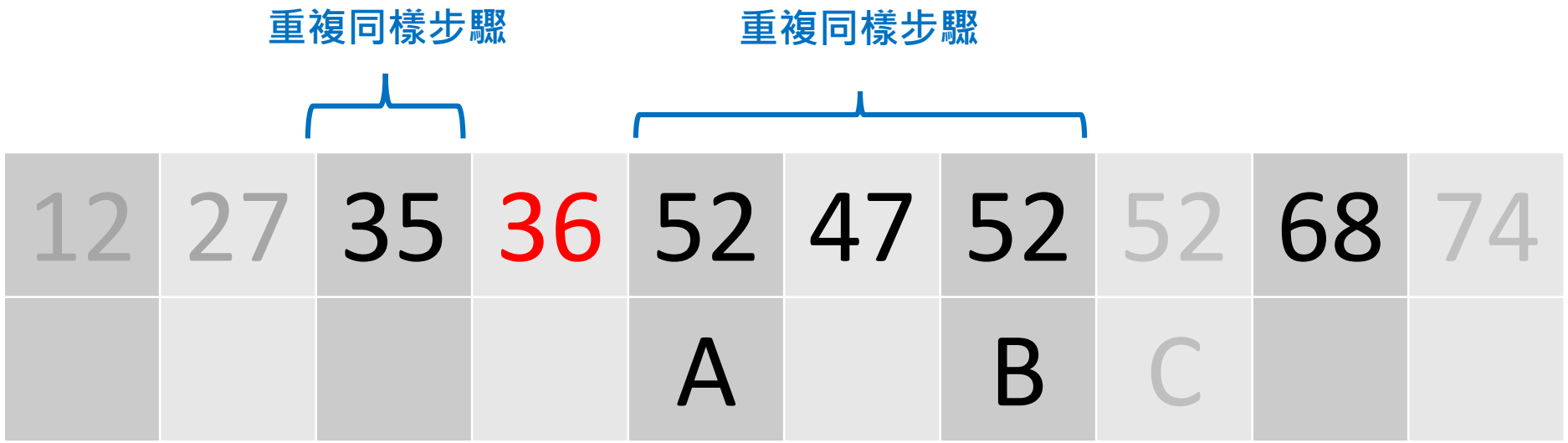
- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

重複同樣步驟



Quick Sort

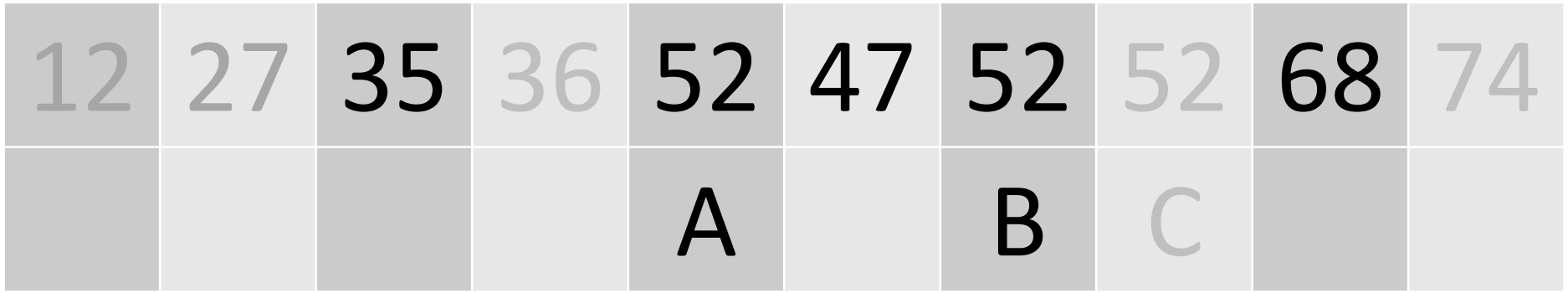
- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1



Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

重複同樣步驟



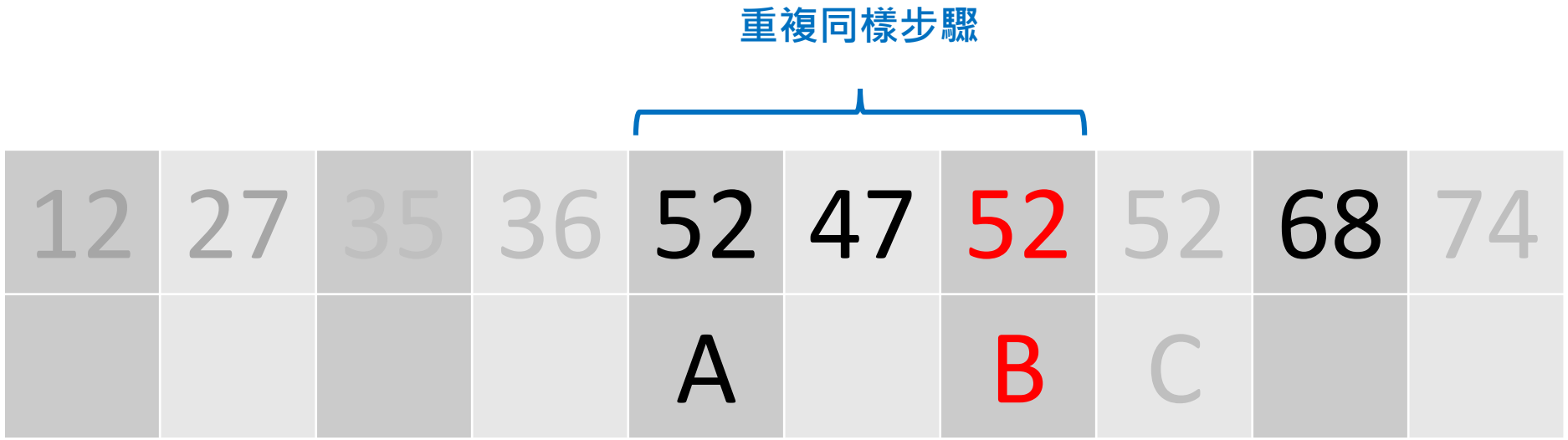
Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

12	27	35	36	52	47	52	52	68	74
				A		B	C		

Quick Sort

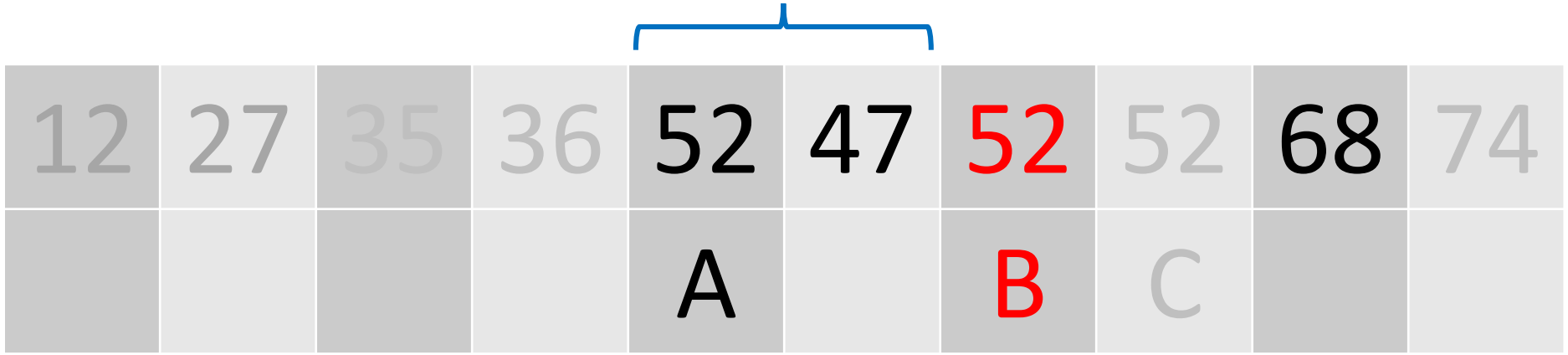
- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1



Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

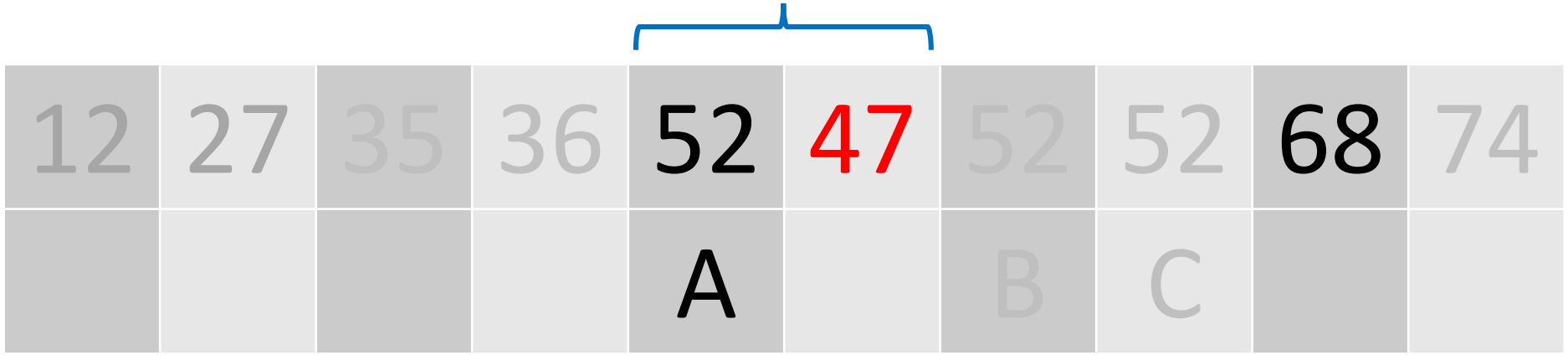
重複同樣步驟



Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

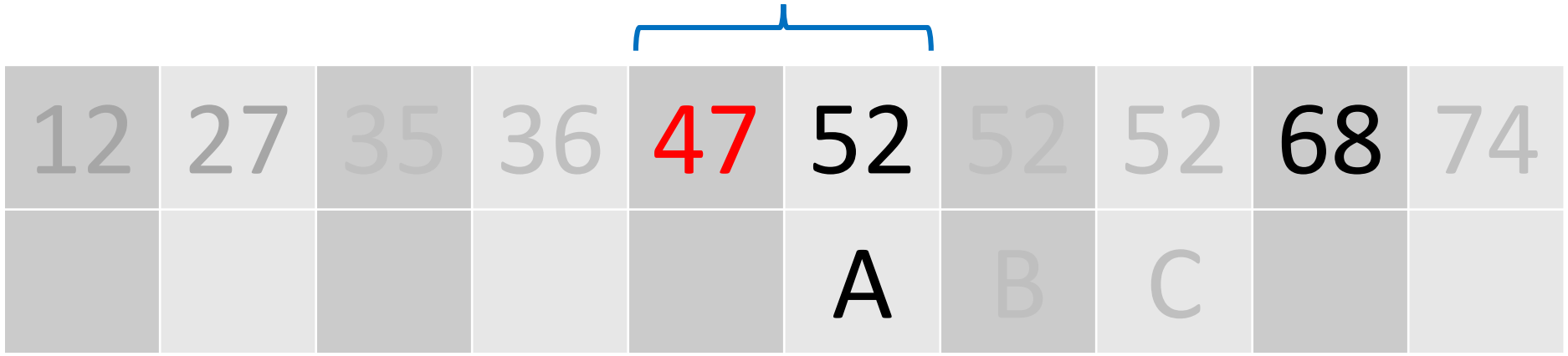
重複同樣步驟



Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

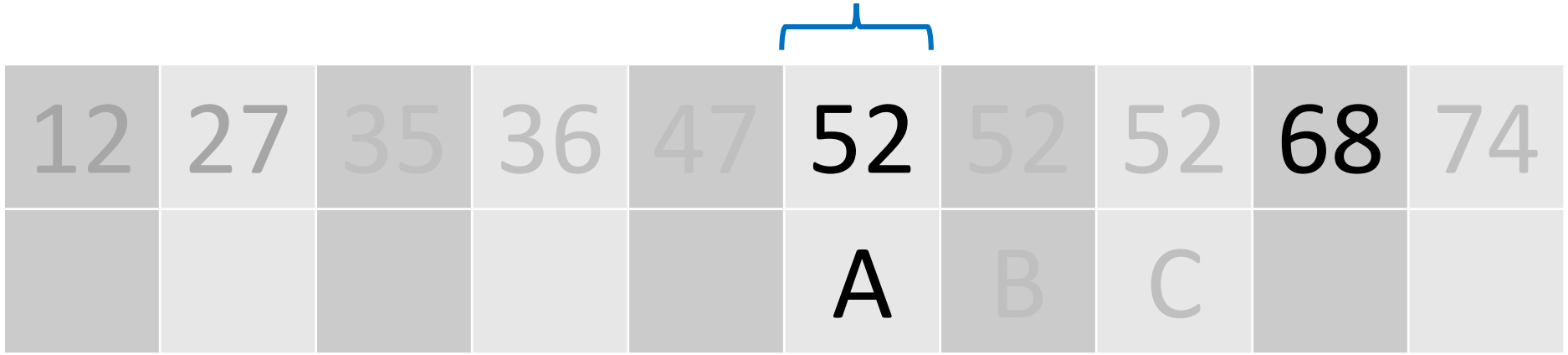
重複同樣步驟



Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

重複同樣步驟



Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

重複同樣步驟

Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

Quick Sort

- 隨機選出一筆資料當基準點→偷懶選最右邊的
  - 比該筆資料小的放左邊
  - 比該筆資料大的放右邊
  - 依序做至資料數目為1

12	27	35	36	47	52	52	52	68	74
					A	B	C		

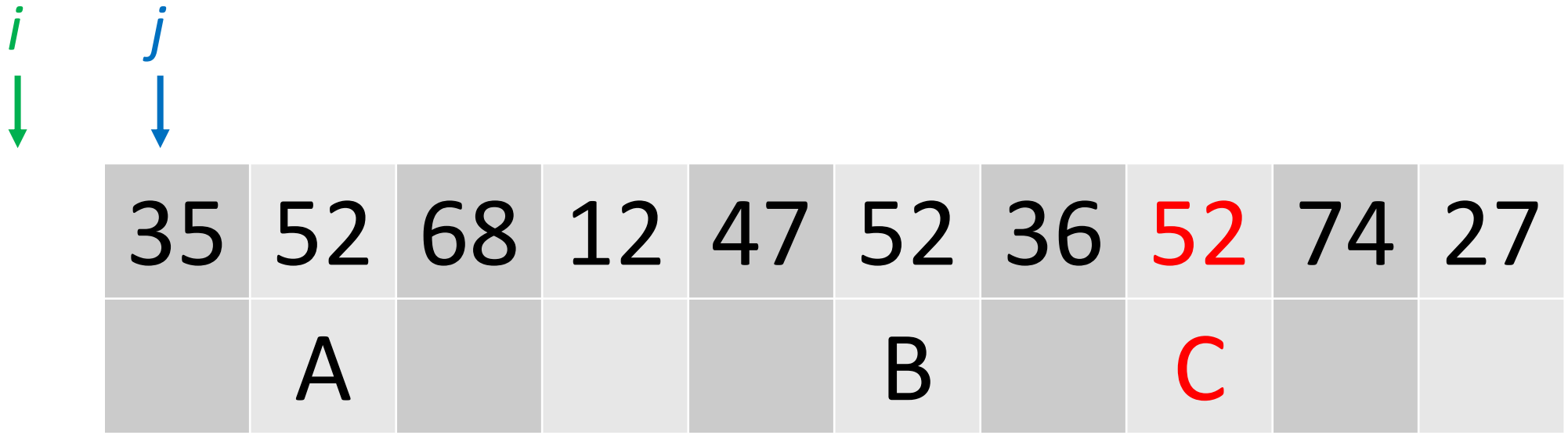
Quick Sort

- 依序挑出一個值 (Pivot)
  - 比其小的放左邊、比其大的放右邊
    - ✓ 該步驟名為：Partition
  - 不一定屬於穩定排序
  - 屬於分治法的一種

	Complexity
Best	$O(nlog_2n)$
Average	$O(nlog_2n)$
Worst	$O(n^2)$
Memory	$O(n)$
Memory (Auxiliary)	$O(1)$
Stable	Depends

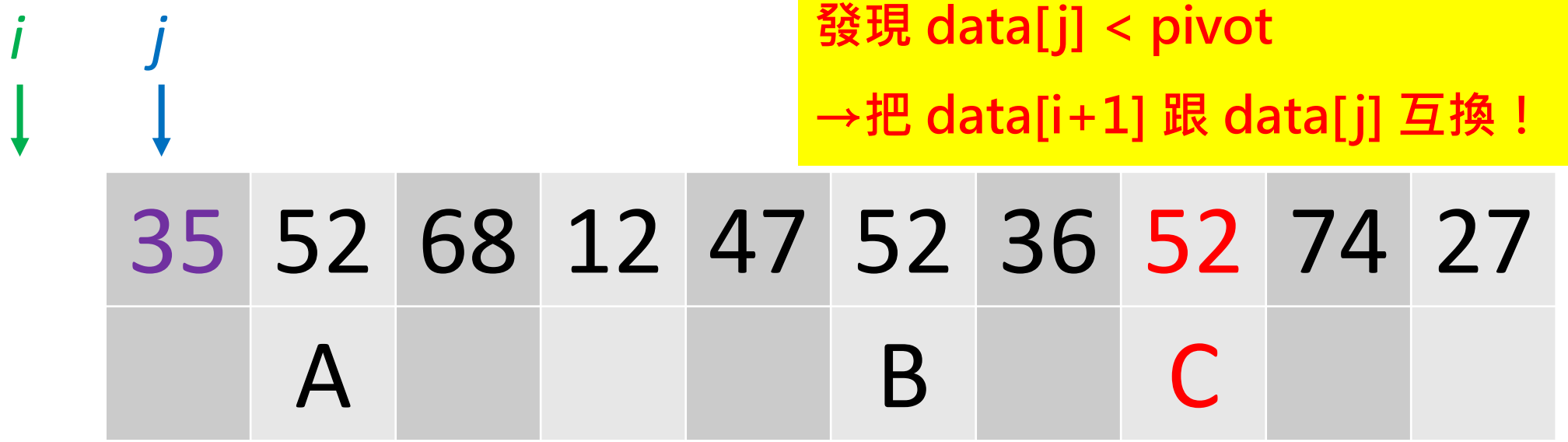
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



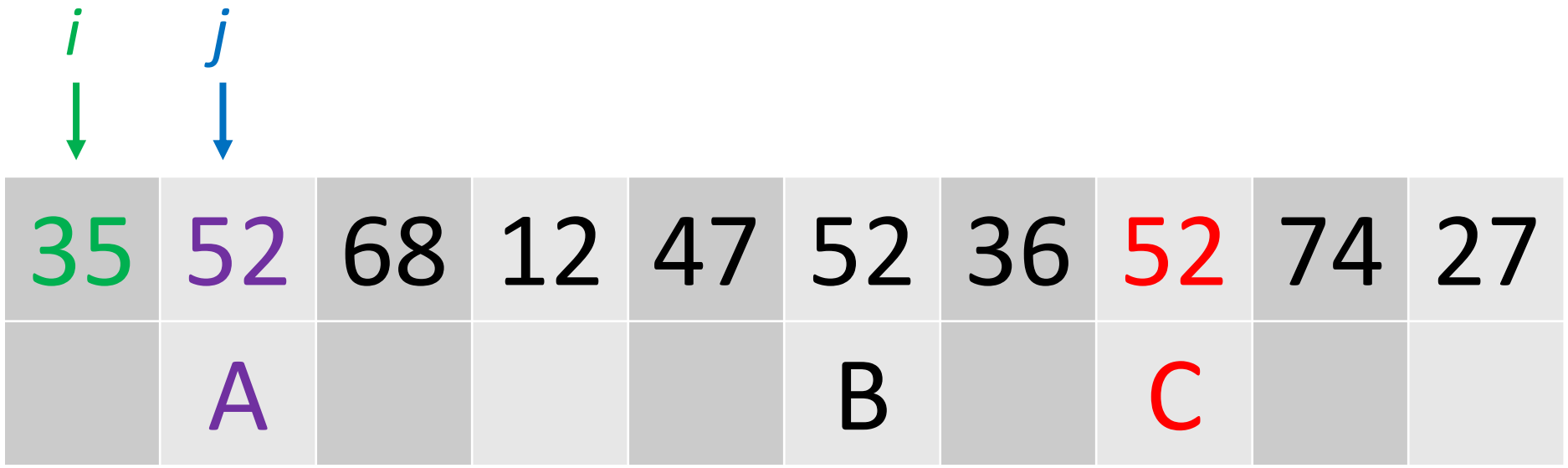
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



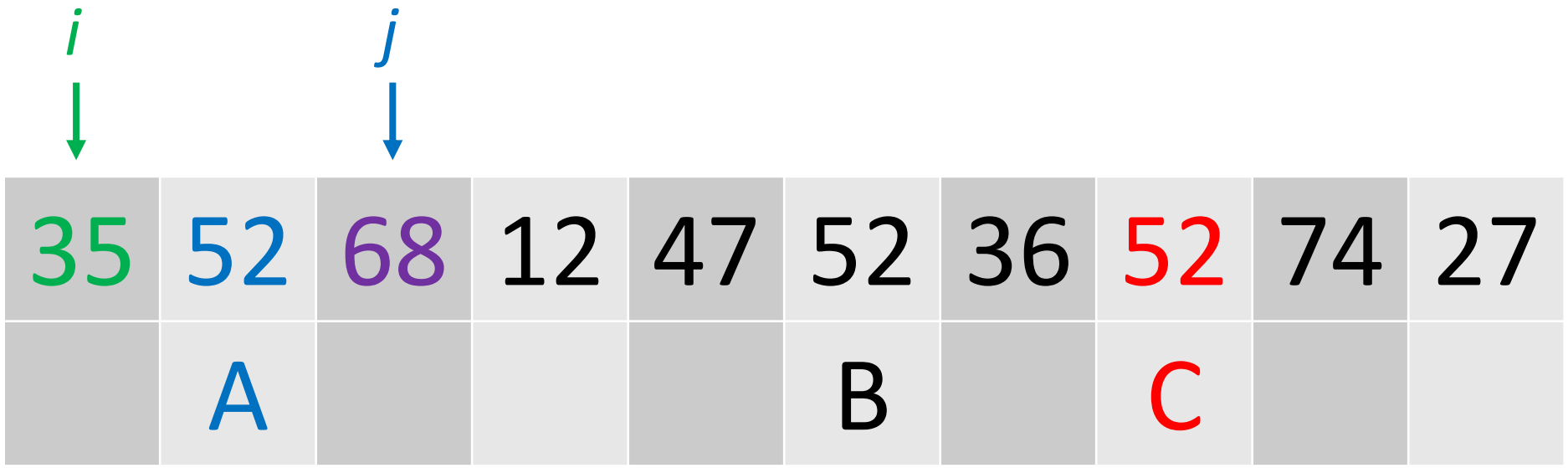
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



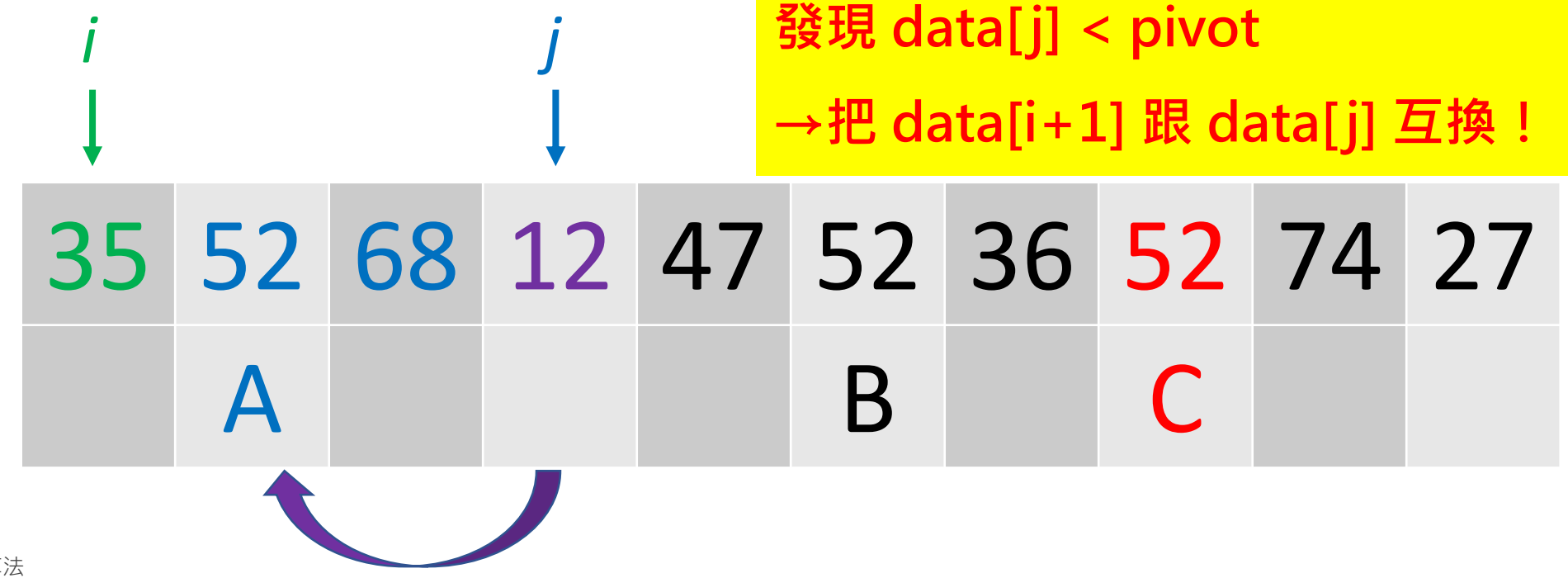
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



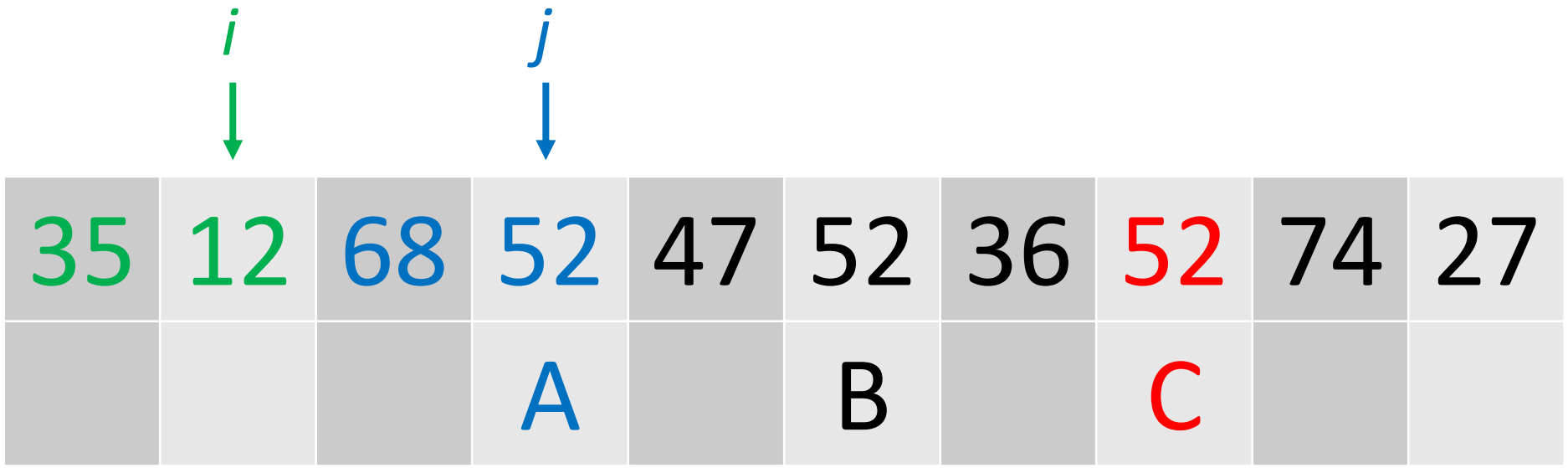
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



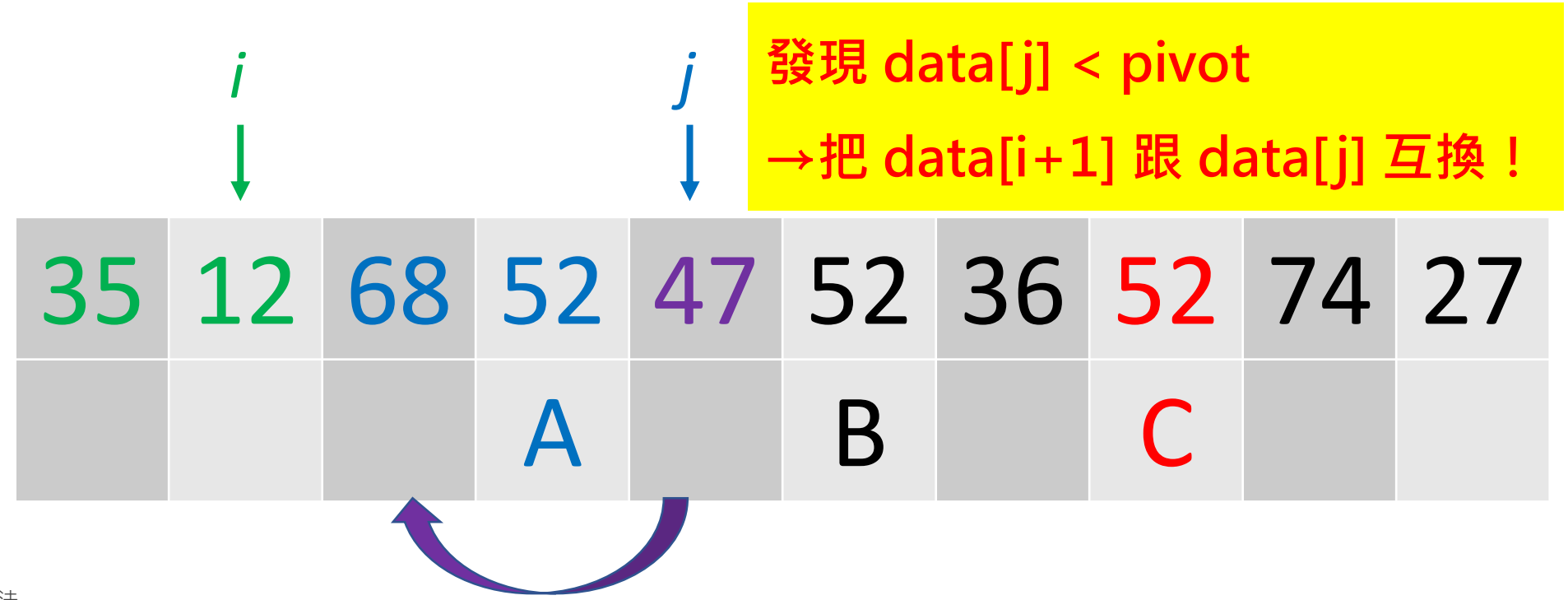
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



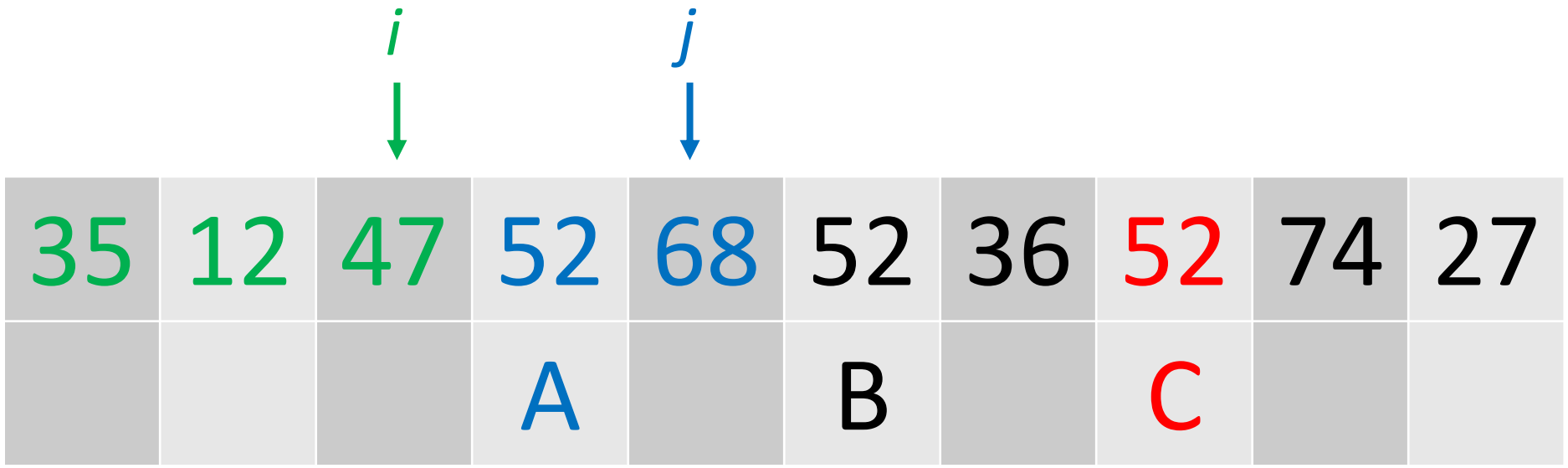
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



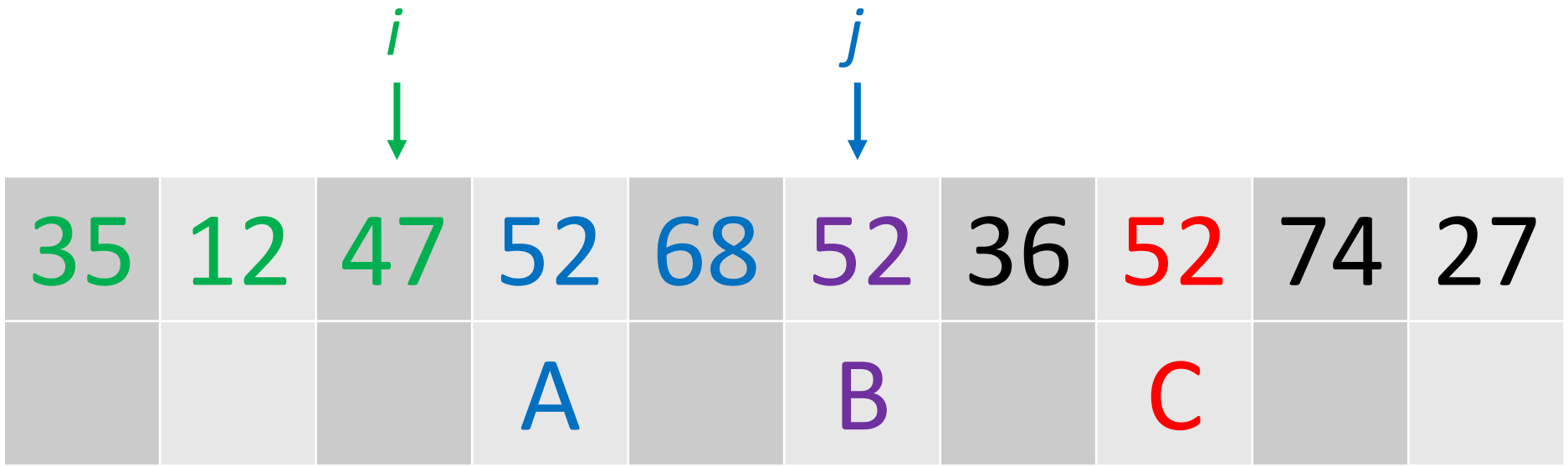
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



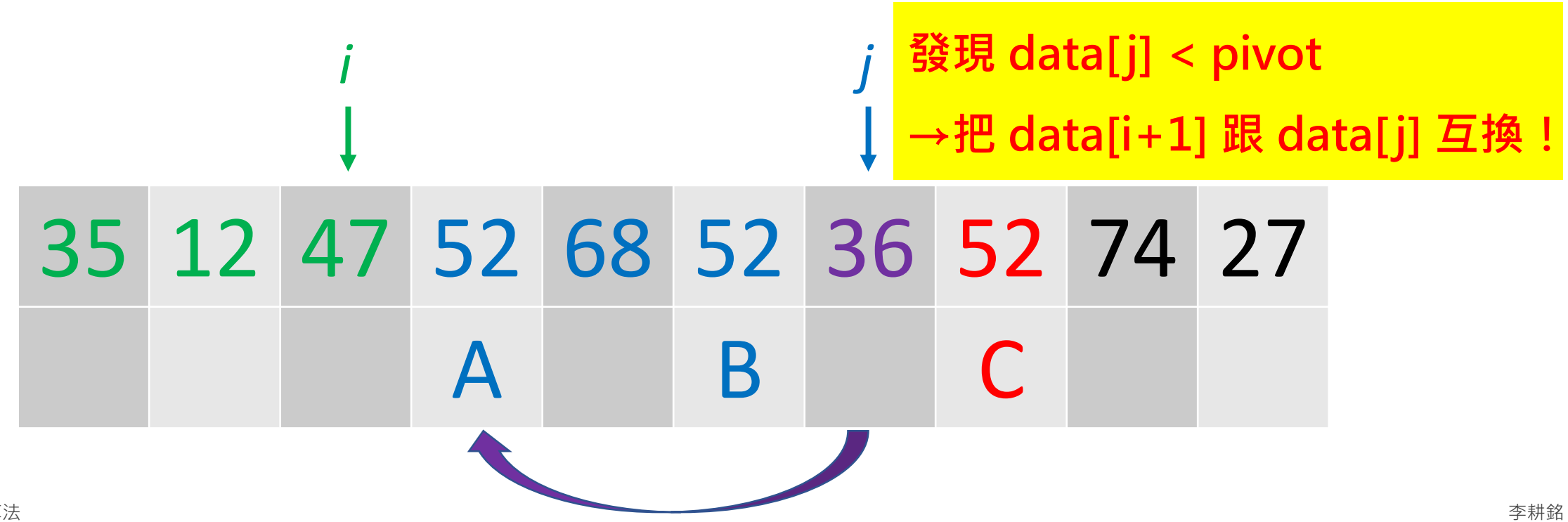
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



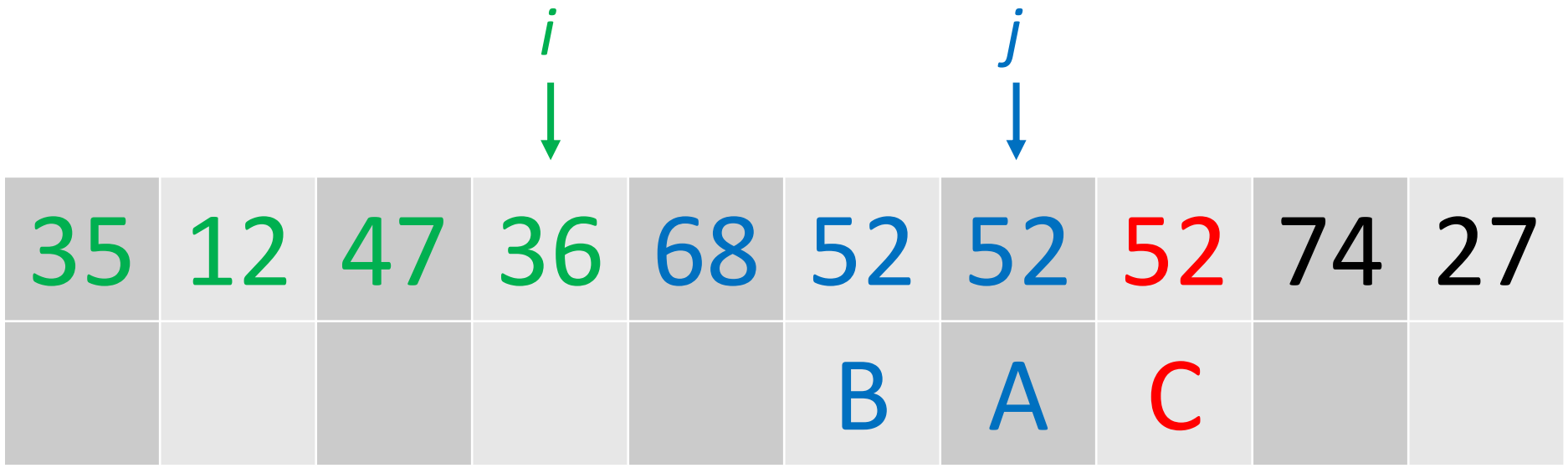
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



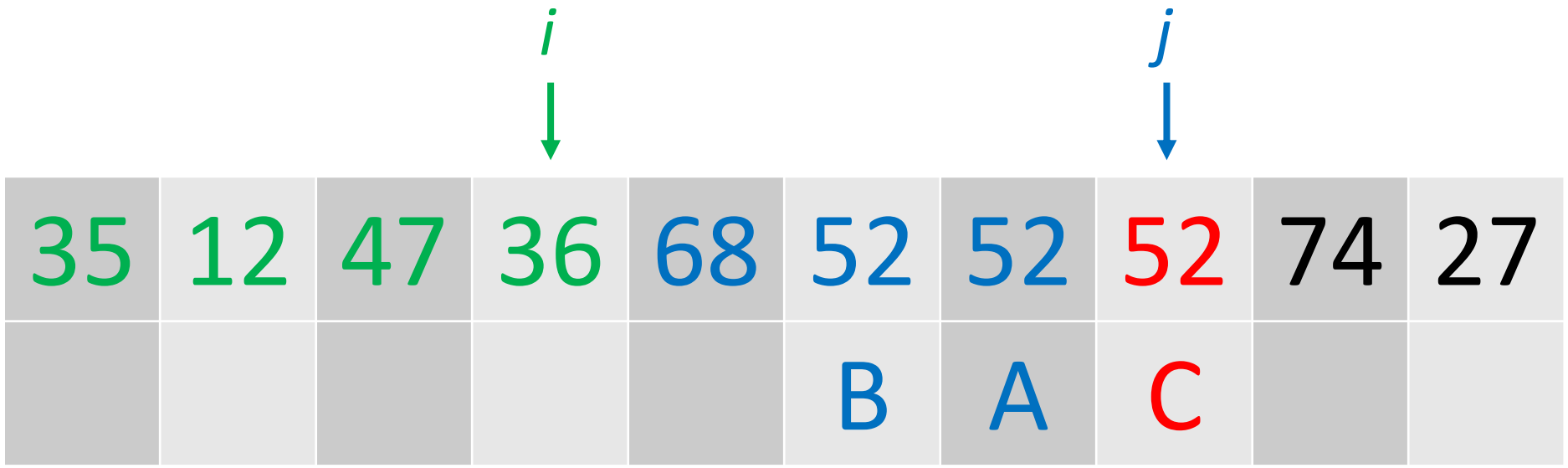
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



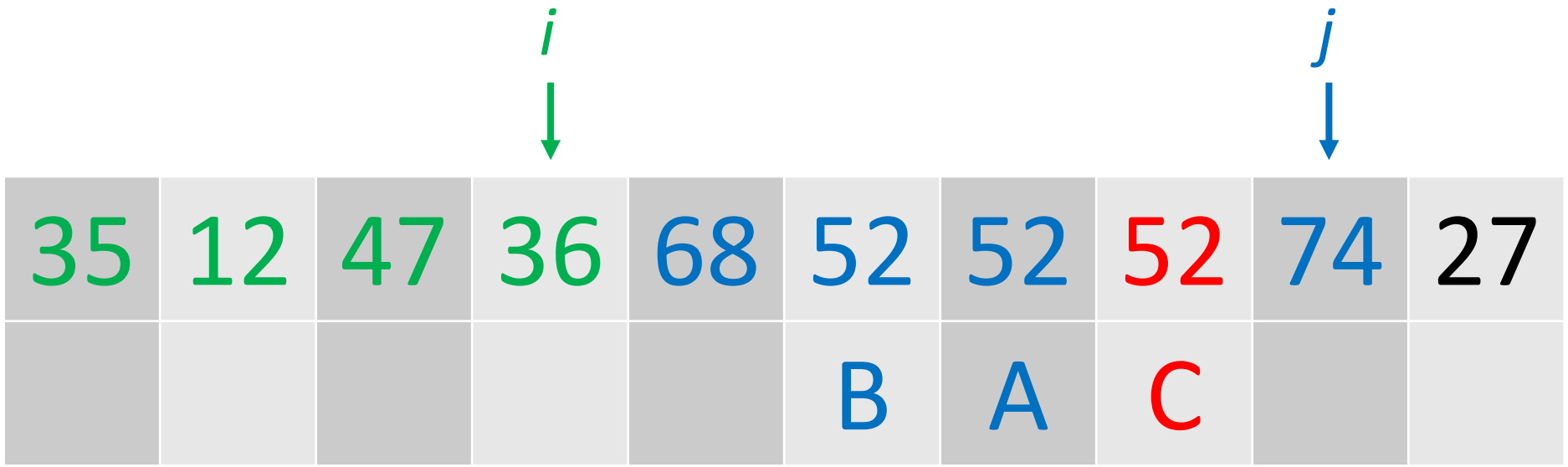
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



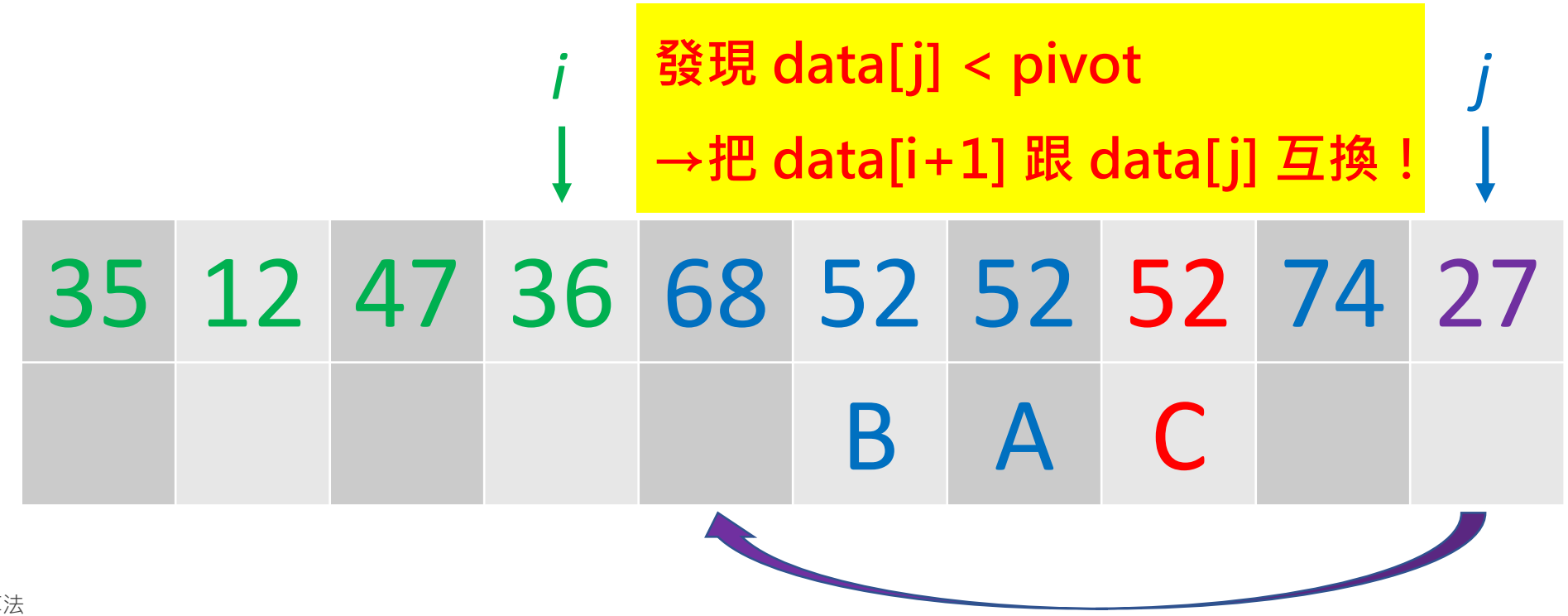
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



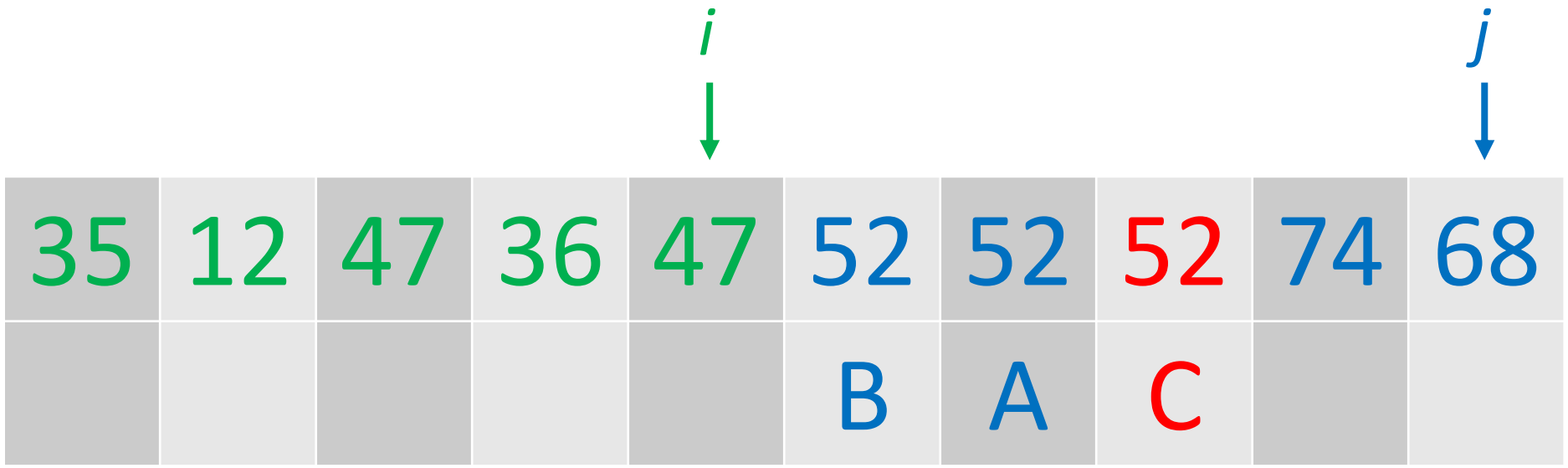
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



Quick Sort

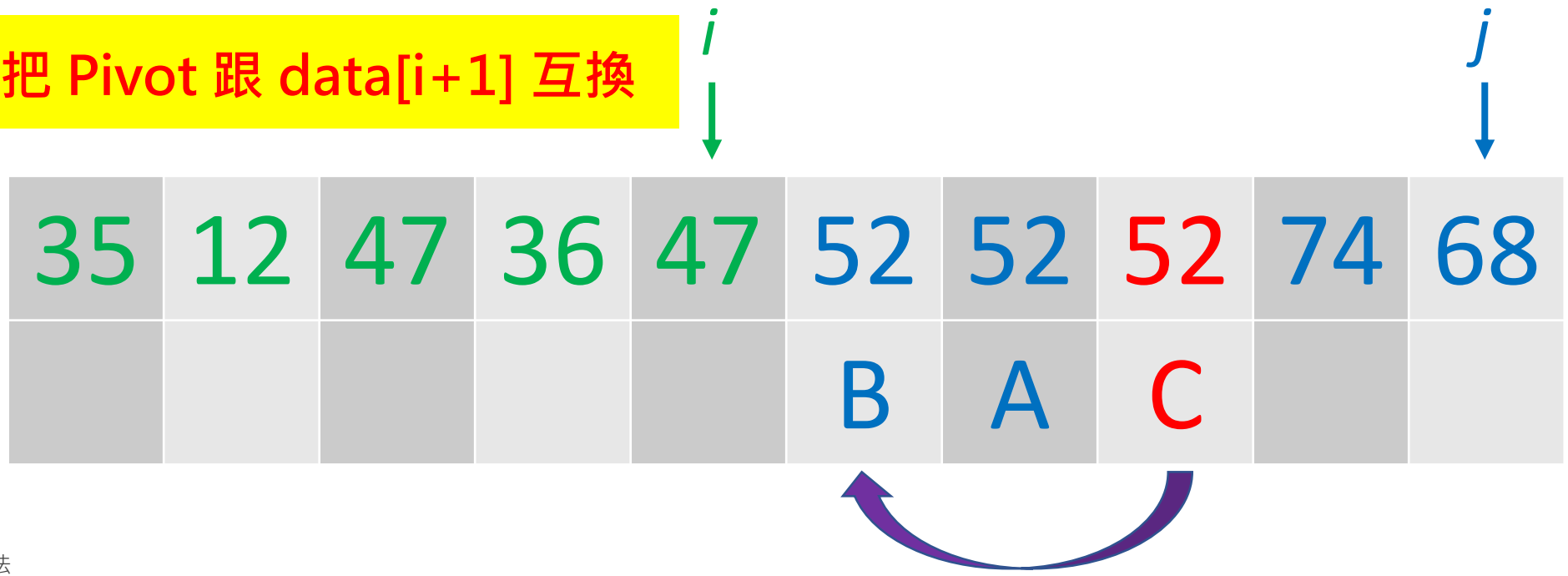
- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



Quick Sort

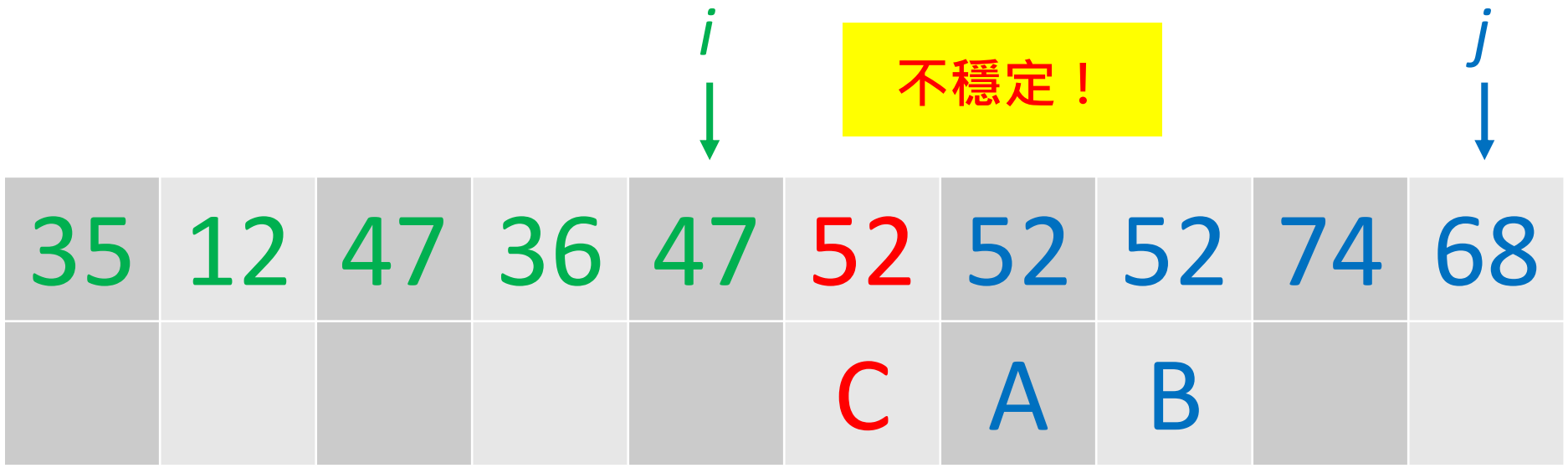
- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點

最後記得把 Pivot 跟 data[i+1] 互換



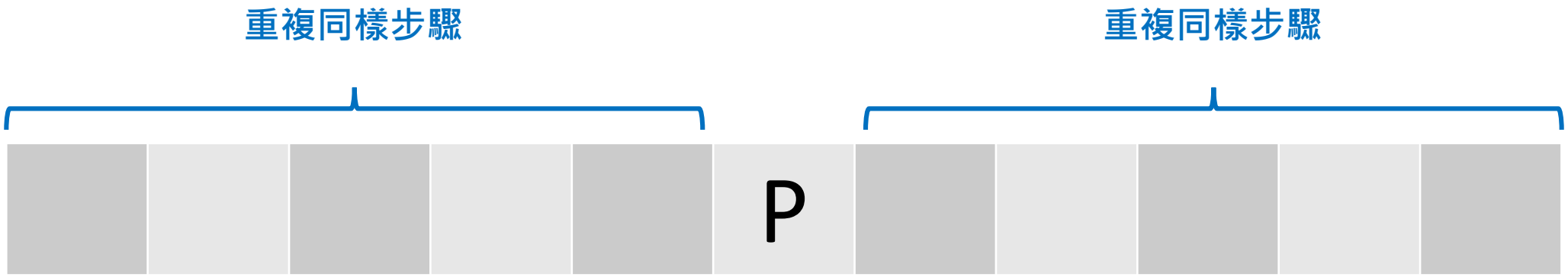
Quick Sort

- 如何實作 Partition？
  - 把資料分成：比 Pivot 小跟比 Pivot 大兩組
  - 宣告索引值變數指向兩組資料的交界點



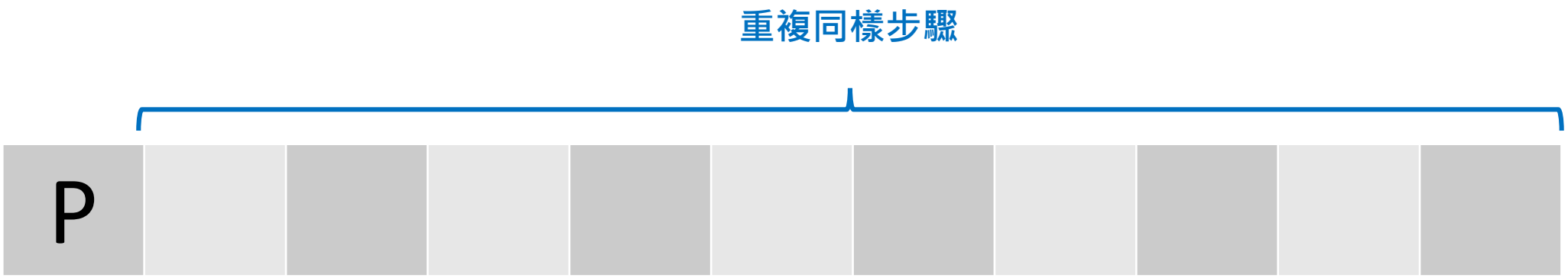
Quick Sort

- Best case：
  - 選到的 Pivot 剛好把資料切割成左右各半
  - Partiton 的複雜度：O(n)
  - $T(n) = 2T\left(\frac{n}{2}\right) + n$
  - $T(n) \in O(nlog_2 n)$



Quick Sort

- Worst case：
  - 選到的 Pivot 剛好是最大/最小
  - Partiton 的複雜度：O(n)
  - $T(n) = T(n-1) + n$
  - $T(n) \in O(n^2)$



Quick Sort

- 不適用於已經排序好的狀態
- 或是修改 Pivot 的取法跟遞迴方式
  1. 隨機取
  2. 取陣列中第一、中間、最後三個值的中位數
  3. 長度太短就停止遞迴，改用 insertion sort

	Complexity
Best	$O(nlog_2n)$
Average	$O(nlog_2n)$
Worst	$O(n^2)$
Memory	$O(n)$
Memory (Auxiliary)	Depends (Recursion)
Stable	Depends

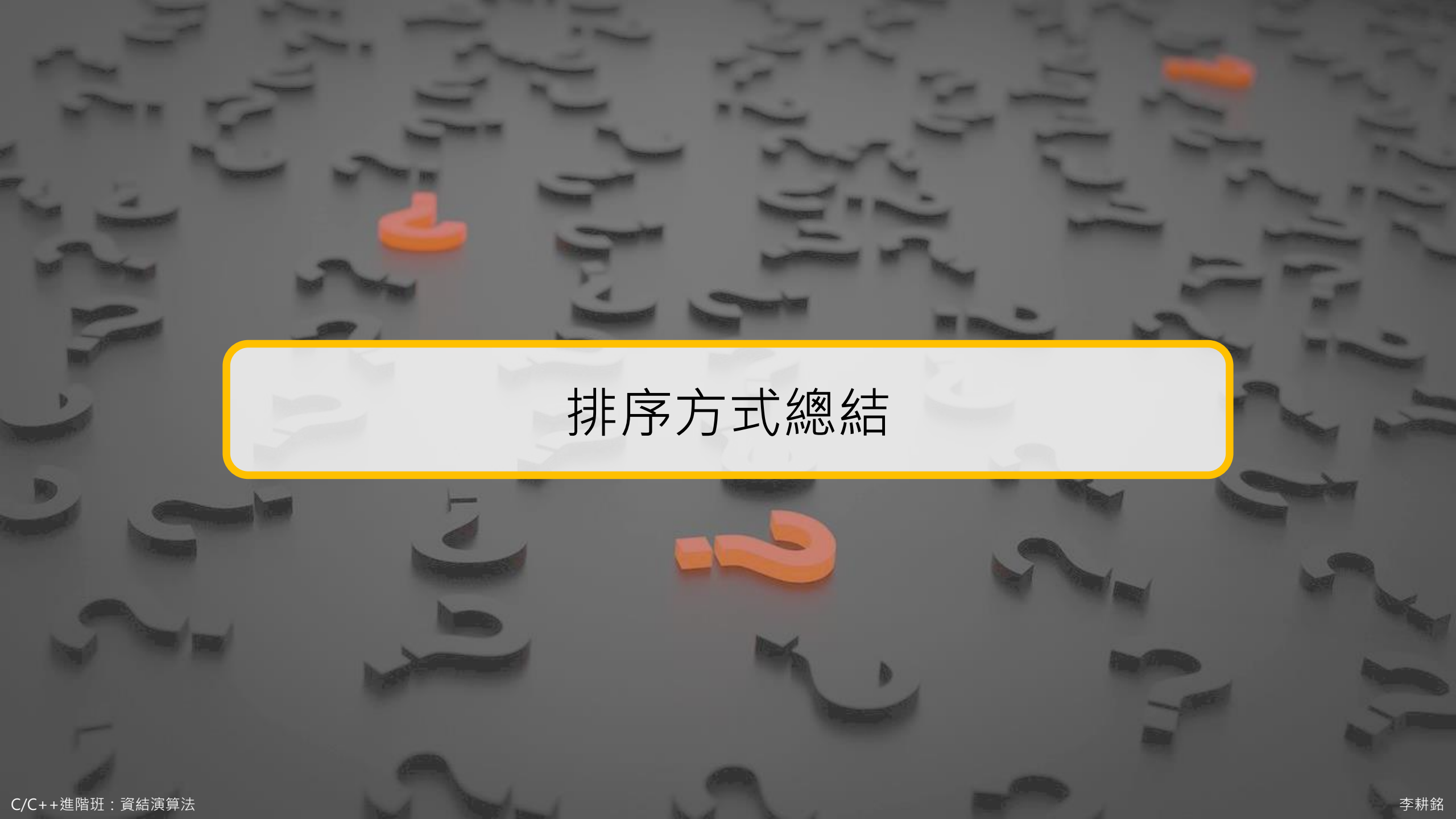
Radix Sort

依序比較每一位數(可轉成二進位)

Radix Sort

- 每個位數逐一比較
  - d：最大位數
  - k：幾進位、一個位數有多少可能

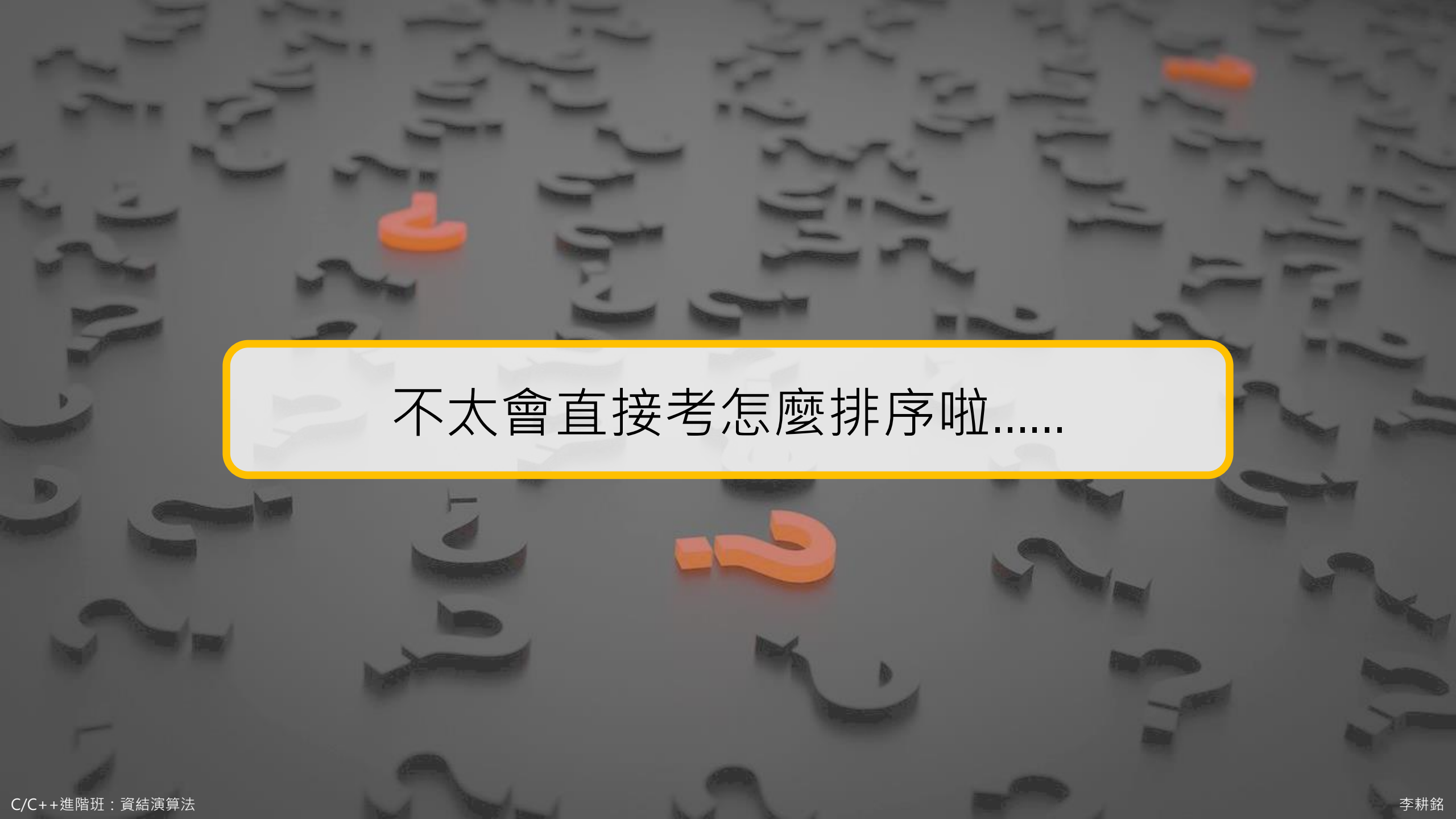
	Complexity
Best	$O(d(n+k))$
Average	$O(d(n+k))$
Worst	$O(d(n+k))$
Memory	$O(n)$
Memory (Auxiliary)	$O(d+n)$
Stable	True



排序方式總結

排序方式總結

Sort	Best Case	Average Case	Worst Case
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Shell	Depends	$O(n^{1.25})$	$O(n^{1.5})$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble	$O(n)$~$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
Heap	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
Merge	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$


不太會直接考怎麼排序啦.....

C++ STL 中的排序

Algorithm(演算法)

排序 (sort)

sort(起點,終點,函式指標)

- 起點、終點都是 iterator
- 回傳 void
- 函式可以為空，預設由小到大
  1. greater：由大到小
  2. less：由小到大 (預設)
- 對 container 裡的資料做排序

```
template<class T1>
void sort(T1 first, T1 last);
```

```
template<class T1, class T2>
void sort(T1 first, T1 last, T2 pointer);
```

Algorithm(演算法)

排序 (sort)

sort(起點,終點,函式指標)

- STL 用的排序法是內觀排序 (introsort)
  1. 原則上是 Quick sort
  2. 監控遞迴的深度，太深改採 Heap sort
  3. 長度下降到一定程度，採用 Insertion sort
  4. 故為混合排序法 (hybrid sorting algorithm)

```
template<class T1>
void sort(T1 first, T1 last);
```

```
template<class T1, class T2>
void sort(T1 first, T1 last, T2 pointer);
```

Algorithm(演算法)

```
vector<int> data = {8,-5,-1,4,-3,6,2,-2,3,4};
```

```
sort(data.begin(), data.end());
for(vector<int>::iterator iter=data.begin();iter!=data.end();iter++)
    cout << *(iter) << " ";
```

-5, -3, -2, -1, 2, 3, 4, 4, 6, 8

```
sort(data.begin(), data.end(), greater<int>());
for(vector<int>::iterator iter=data.begin();iter!=data.end();iter++)
    cout << *(iter) << " ";
```

8, 6, 4, 4, 3, 2, -1, -2, -3, -5

```
sort(data.begin(), data.end(), less<int>());
for(vector<int>::iterator iter=data.begin();iter!=data.end();iter++)
    cout << *(iter) << " ";
```

-5, -3, -2, -1, 2, 3, 4, 4, 6, 8

函式指標

函式會載入記憶體中，有記憶體位置

有記憶體位置就可以透過指標操作！

回傳資料型態 (*指標名稱)(引數型態);

```
int sum(int a,int b) {
    return a+b;
}

int (*fp)(int,int) = sum;
```

函式指標

回傳資料型態 (*指標名稱)(引數型態);

要有上面紅色的小括號！

回傳資料型態 *指標名稱(引數型態);

Q：沒有的話會變甚麼？

函式指標

呼叫函式指標指到的函式時

可以不使用間接取值運算子 *

```
int sum(int a,int b) {
    return a+b;
}

int (*fp)(int,int) = sum;
```

```
(*fp)(20,10);
fp(20,10);
```

函式指標

可用 typedef 重新命名函式指標

typedef 回傳資料型態 (*指標名稱)(引數型態);

重新命名後 指標名稱 就是資料型態

```
typedef int (*fp)(int,int);
int sum(int a,int b) {
    return a+b;
}

void test(int a,int b,fp func){
    cout << func(a,b);
}
```

函式指標

C++ 11後可用 decltype 取得函式的資料型態

decltype(函式名稱) 引數名稱

```
int sum(int a,int b) {
      return a+b;
 }

 void test(int a,int b,decltype(sum) func){
      cout << func(a,b);
 }
```

函式指標

C++ 11 後可用 using

using 指標名稱 = 回傳資料型態 (*)(引數型態);

重新命名後 指標名稱 就是資料型態

```
using fp = int (*)(int,int);
int sum(int a,int b) {
    return a+b;
}

void test(int a,int b,fp func){
    cout << func(a,b);
}
```

函式指標

當然也可以用函式參考

回傳資料型態 (&參考名稱)(引數型態) = 被參考函式名稱;

```
int sum(int a,int b) {
    return a+b;
}

int (&fp)(int,int) = sum;
```

實戰練習

Practice 3

Mission

Try LeetCode #912. Sort an Array

Given an array of integers nums, sort the array in ascending order.

- Example 1:
  - Input: nums = [5,2,3,1]
  - Output: [1,2,3,5]
- Example 2:
  - Input: nums = [5,1,1,2,0,0]
  - Output: [0,0,1,1,2,5]

Ref： https://leetcode.com/problems/sort-an-array/

Practice 4

Mission

Try LeetCode #274. H-Index

Given an array of integers citations where citations[i] is the number of citations a researcher received for their ith paper, return compute the researcher's h-index.

According to the definition of h-index on Wikipedia: A scientist has an index h if h of their n papers have at least h citations each, and the other n − h papers have no more than h citations each.

If there are several possible values for h, the maximum one is taken as the h-index.

Ref：https://leetcode.com/problems/h-index/

Practice 4

6	5	3	1	0

index = 0, 有 0+1 篇文章的引用數 ≥ 6

6	5	3	1	0

index = 1, 有 1+1 篇文章的引用數 ≥ 5

6	5	3	1	0

index = 2, 有 2+1 篇文章的引用數 ≥ 3

6	5	3	1	0

index = 3, 有 3+1 篇文章的引用數 ≥ 1
結束，回傳 index = 3

Practice 5

APCS: 物品堆疊

某個自動化系統中有一個存取物品的子系統，該系統是將 N 個物品堆在一個垂直的貨架上，每個物品各佔一層。系統運作的方式如下：每次只會取用一個物品，取用時必須先將在其上方的物品貨架升高，取用後必須將該物品放回，然後將剛才升起的貨架降回原始位置，之後才會進行下一個物品的取用。

每一次升高某些物品所需要消耗的能量是以這些物品的總重來計算，在此我們忽略貨架的重量以及其他可能的消耗。現在有 N 個物品，第 i 個物品的重量是 w(i)而需要取用的次數為 f(i)，我們需要決定如何擺放這些物品的順序來讓消耗的能量越小越好。舉例來說，有兩個物品 w(1)=1、w(2)=2、f(1)=3、f(2)=4，也就是說物品 1 的重量是 1 需取用 3 次，物品 2 的重量是 2 需取用 4 次。

APCS: 物品堆疊

我們有兩個可能的擺放順序(由上而下)：

1. (1,2)，也就是物品 1 放在上方，2 在下方。那麼，取用 1 的時候不需要能量，而每次取用 2 的能量消耗是 w(1)=1，因為 2 需取用 f(2)=4 次，所以消耗能量數為 w(1)*f(2)=4。

2. (2,1)，也就是物品 2 放在 1 的上方。那麼，取用 2 的時候不需要能量，而每次取用1的能量消耗是w(2)=2，因為1需取用f(1)=3次，所以消耗能量數=w(2)*f(1)=6。

在所有可能的兩種擺放順序中，最少的能量是 4，所以答案是 4。再舉一例，若有三物品而 w(1)=3、w(2)=4、w(3)=5、f(1)=1、f(2)=2、f(3)=3。假設由上而下以(3,2,1)的順序，此時能量計算方式如下：取用物品 3 不需要能量，取用物品 2 消耗 w(3)*f(2)=10，取用物品 1 消耗(w(3)+w(2))*f(1)=9，總計能量為 19。如果以(1,2,3)的順序，則消耗能量為 3*2+(3+4)*3=27。事實上，我們一共有 3! =6 種可能的擺放順序，其中順序(3,2,1)可以得到最小消耗能量 19。

APCS: 物品堆疊

- 輸入格式：輸入的第一行是物品件數 N，第二行有 N 個正整數，依序是各物品的重量 w(1)、w(2)、...、w(N)，重量皆不超過 1000 且以一個空白間隔。第三行有 N 個正整數，依序是各物品的取用次數 f(1)、f(2)、...、f(N)，次數皆為 1000 以內的正整數，以一個空白間隔。
- 輸出格式：輸出最小能量消耗值，以換行結尾。所求答案不會超過 63 個位元所能表示的正整數。

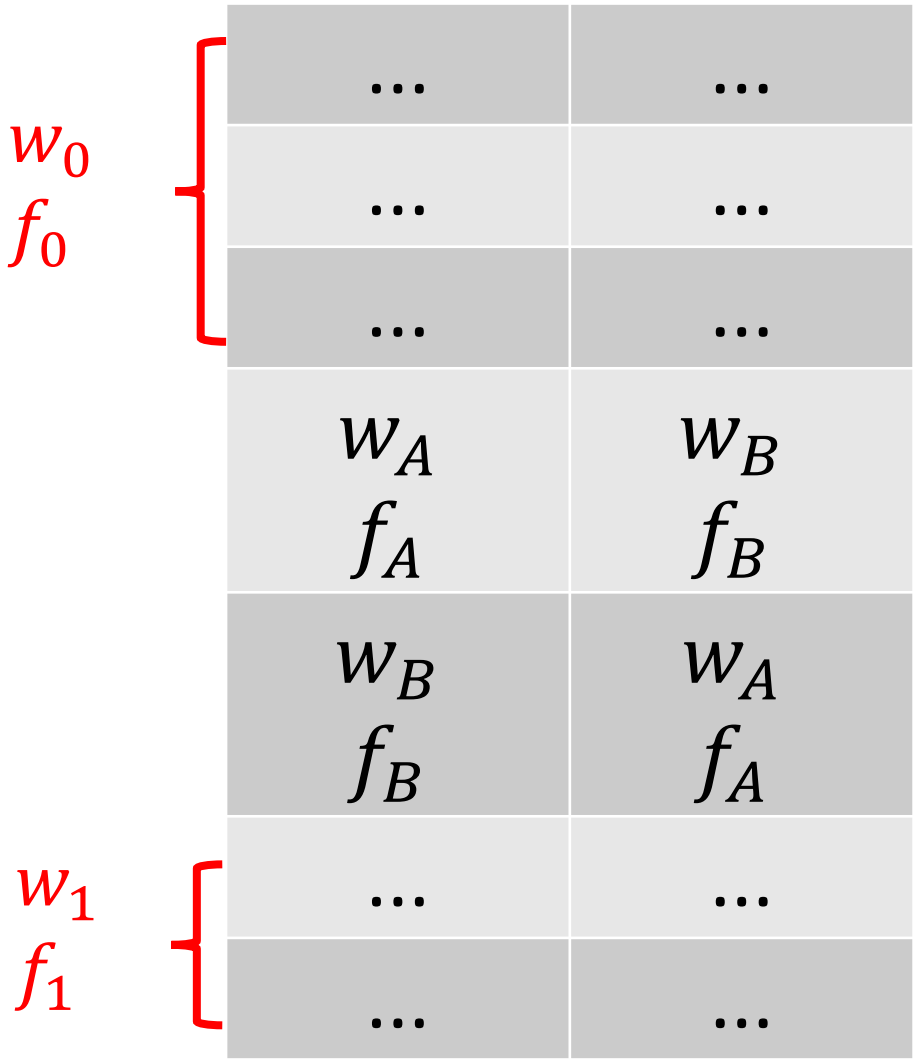
➢ 範例：輸入

```
2
20 10
1 1
```

➢ 範例：正確輸出

```
10
```

APCS: 物品堆疊



Given 2 object A and B :
W_A *is the weight of* A
f_A *is the frequency of* A
W_B *is the weight of* B
f_B *is the frequency of* B

Condition 1: *B is under A*

$$Total\ cost(1) = f_1 \times (w_0 + w_A + W_B) + f_B \times (w_0 + w_A) + f_A \times (w_0)$$

Condition2: *A is under B*

$$Total\ cost(2) = f_1 \times (w_0 + w_A + W_B) + f_A \times (w_0 + w_B) + f_B \times (w_0)$$

$$Total\ cost(1) - Total\ cost(2) = f_B w_A - f_A w_B$$

if $f_B w_A > f_A w_B$:
put A under B

if $f_B w_A < f_A w_B$:
put B under A

Conclustion :
Sort the objects by $f_B w_A$

Take Home Message

- 請簡述以下排序演算法的原理
  1. 插入式排序法 (insertion sort)
  2. 謝爾排序法 (shell sort)
  3. 選擇排序法 (selection sort)
  4. 泡沫排序法 (bubble sort)
  5. 合併排序法 (merge sort)
  6. 堆積排序法 (heap sort)
  7. 快速排序法 (quick sort)
  8. 基數排序法 (radix sort)
  9. 桶排序法 (bucket sort)