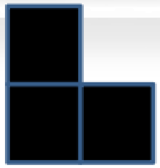
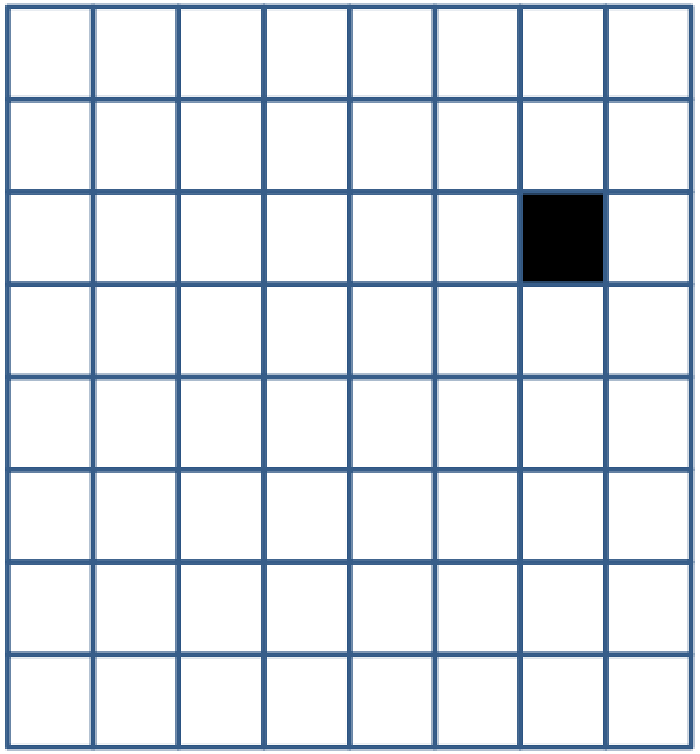
Algorithm Design Methods
Divide & Conquer

for Sprout 2014 by Yuan

Sprout

從一個例子開始…

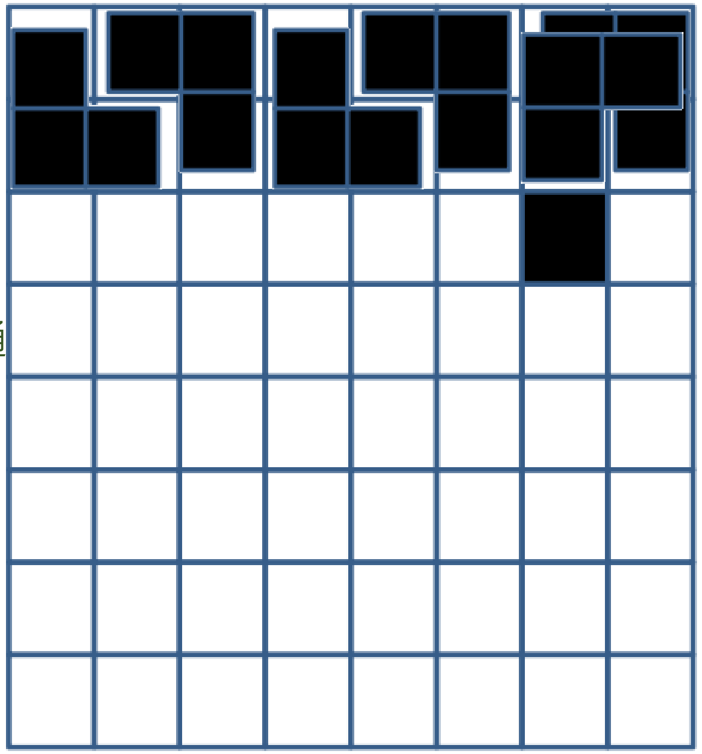
- 占三格的L形方塊 
- 是否可以不重疊的放入8x8，且缺了一格的棋盤中呢？



我們來試試看…
很「貪心」的盡量
把每個方格塞滿

糟糕了…好像行不通

怎麼辦？難道要回
去窮舉每一種可能
嗎?

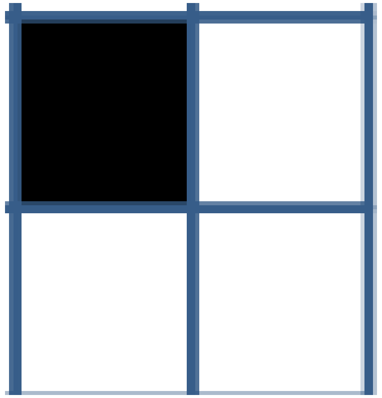


別緊張!!

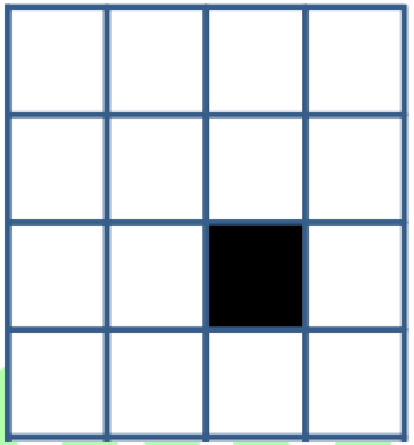
- 遇到看似不可解的問題怎麼辦？
- 大問題不會解，小問題總會解了吧？
- 不妨從較小規模的合理問題開始思考!

- 圖一是否可解呢？看起來易如反掌

- 放大一倍試試看！
- 圖二是否可解呢？看起來沒那麼難......
- 要怎麼從小問題的解法獲得一些線索呢?

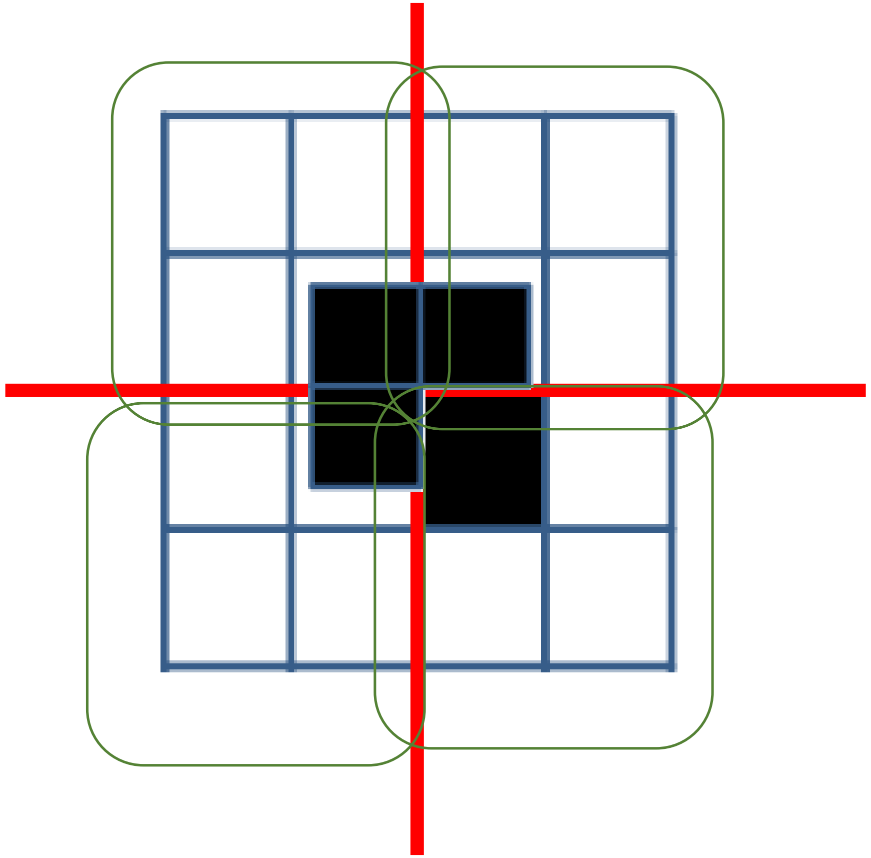


圖一



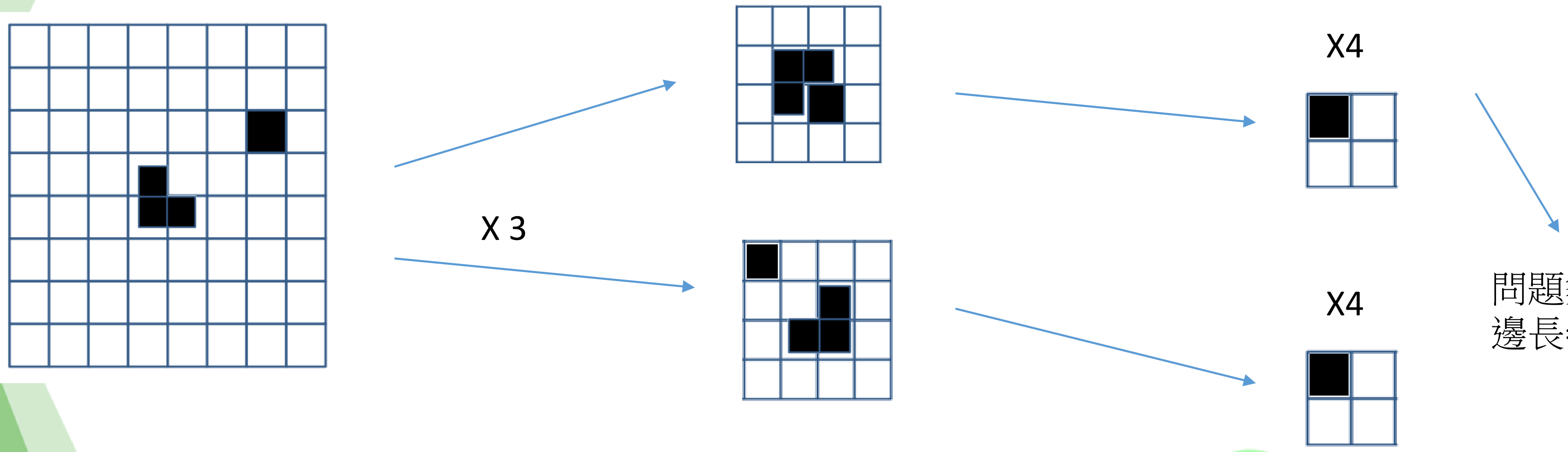
圖二

切割！



- 這樣一來，就很像解四次圖一的問題了！
- 可是有個小小的插曲：其中的三格並沒有缺格
- 我們正好可以放上一塊方塊完美的解決這個問題！
- 產生四個規模較小的子問題~

遞迴過程



問題夠小，我們會解了！
邊長=2為終止條件

什麼是「分治」？

- 切割問題，然後征服問題！
  - 把問題切割成相似的子問題
  - 利用相似的方法解決它

- 剛剛的例子，由「分治法」構造出一組正確的方案

- 遞迴是方便實做分治想法的好工具
  - 遞迴的本質是用堆疊保存每一層函數之區域變數的狀態

目慢，貪心不好嗎？

- 貪心很棒啊，總是拿目前最有利的解
- 剛剛的問題，不知道該怎麼貪心
- 也有些問題，太貪心，得不償失

- HW5, Problem 1
- 總是以面額較高之貨幣付款，能讓使用的貨幣數量最小化嗎？
- 如果硬幣的面額是 {1,2,4,8}, 想要湊出面額15, 怎麼做？
- 如果硬幣的面額是 {1,500,501}, 想要湊出面額1000, 怎麼做？

那，窮舉總行了吧

- 太過曠日費時，天荒地老
- 盲目的窮舉並沒有好好的利用問題的性質

- 剛剛的問題，如果對於每一個方格，窮舉四個方向
- 總複雜度需要 O(4^N) !!
- 其中，N是擺放的L型數量

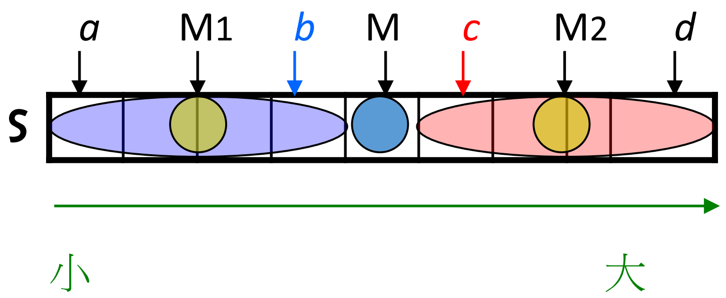
- 直接解決大問題很難…
- 要是問題滿足以下條件：
  - 規模小的時候，輕鬆簡單，易如反掌
  - 規模大的時候，既不能貪心，窮舉又不切實際……
    幸好，我們可以把大問題切割成小問題
  - 而且，小問題的答案有助於我們探尋大問題的答案！
- 就可以考慮使用分治的概念解題

分工合作的想法

- 把大問題變成一些相似的小問題
  - 該怎麼變，術語叫”切割問題”
  - 不一定會有多個子問題，可能一個就足夠
  - 有些不可能對答案造成影響的子問題，可以直接忽略
- 把小問題算出答案 (怎麼算的不重要，算得出來就好)
  - 就是”遞迴求解”囉
  - 只要問題的切割有遇到終止條件的一天，一定算得出來!
- 把小問題的答案變成大問題的答案
  - 術語叫”合併問題”
  - 善加利用我們已經得到的特性

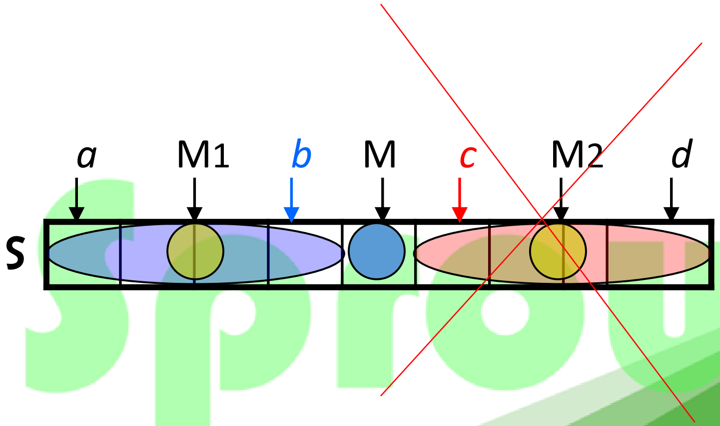
複習一下二分搜尋法

- 在S中搜尋某數



- 不可能對答案造成影響的子問題，可以直接忽略
- 於是拋棄不用求解的子問題
- 分割問題為兩個規模接近的子問題，子問題的規模是原問題的一半

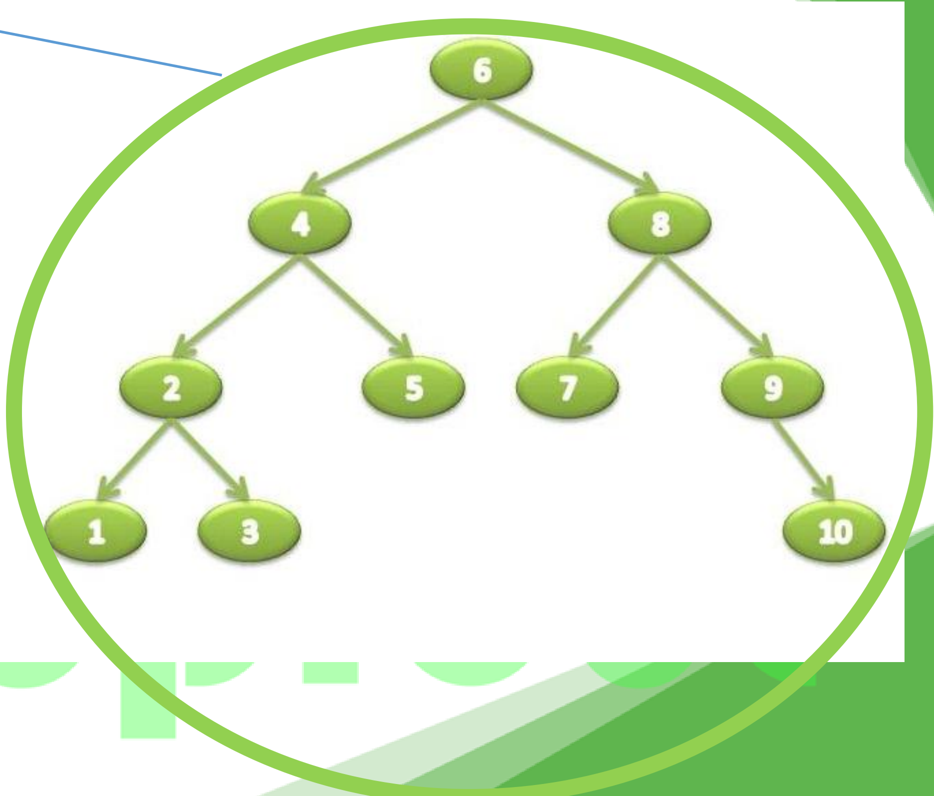
- 只有分，不太需要治
- 因為最後都只有一個可能的分支
- 子問題的解直接就會是答案



適用分治的時機

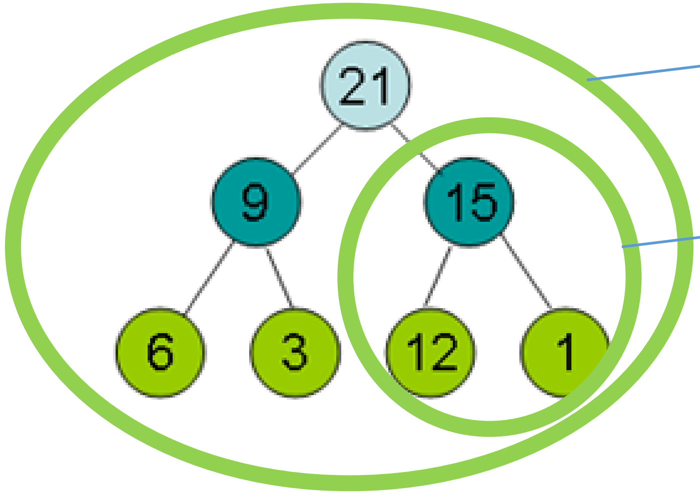
- 天生就適合分治的問題
  - 問題本身就長得很遞迴 (原問題可以切割成許多規模較小的問題)
  - 資料結構由遞迴定義而得
  - 用分治的想法可以讓程式很清楚！

我是二元搜尋樹!
我的子孫也是!

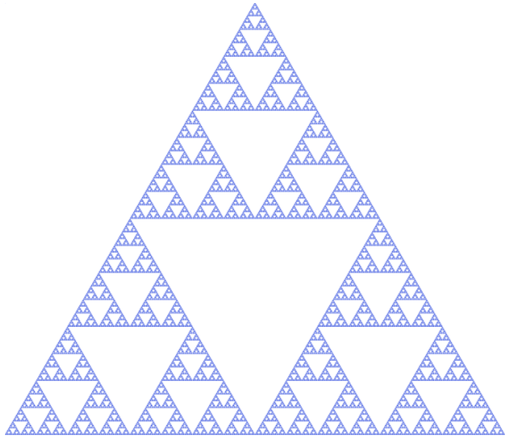
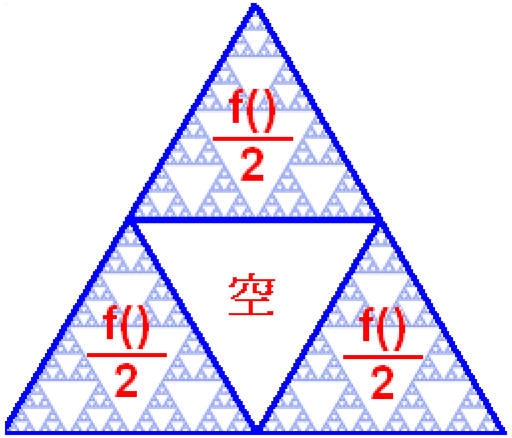


我是Heap!

我也是Heap



遞迴定義

f()=  = 

適用分治的時機

- 看起來貌似沒有分治結構的問題，如剛剛的L型方塊
- 但分治可以幫助我們求解

- 再舉一個例子
- 輸入**n**，請構造出一組**1~n**的排列，滿足任意選擇其中三個數，按照原本的順序排列，均不會形成等差數列。

- n=8
- 3 4 2 1 7 8 5 6

- 想要直接構造長度為n的解答，似乎很困難？

- 不妨考慮分治：如果我們在已有長度為n/2的解答X的情況下，是否有辦法構造出一組長度為n的解答X’呢？

- 既然解答X合法，表示從X中任意取三個數，必定不會形成等差數列

- 如果對於X中的每個元素加減一個數，是否還保持此性質？
- 如果對於X中的每個元素乘上一個數，是否還保持此性質？

- N=3 的解：3 1 2
- N=6 的解：要如何構造，使得1~6都會用到呢？
- 使用N=3 的解做一些變化，但避免變化前與變化後形成等差數列

- [3 1 2] 每個數值都加上3，得到 [6 4 5]
  - [3 1 2] 與 [6 4 5] 組合，這樣可以用到每個數
  - [3 1 2 6 4 5]
  - 似乎並未有效的防止等差數列產生

- 換個想法，利用乘法得到較大的數

於是乎

- [3 1 2] 每個數值都乘以2，得到 [6 2 4]
  - 剩下 1, 3, 5 不妨把[6 2 4]各減1，得到 [5 1 3] 這樣可以用到每個數
  - 該怎麼決定他們的位置呢？
  - 如果把奇數都擺在偶數的前面，無論是 [奇奇偶] 或 [奇偶偶]都不可能形成等差數列！當然，因為遞迴的性質，[奇奇奇]與[偶偶偶]也不可能

  我們成功找到一種由 N=3 的解 構造出 N=6 的解 的方法了！

- 有了這個思路後，剩下的想法就容易許多，不妨自己試試看，如果n是奇數的話，也可以這樣做嗎？

那那...

- 以上都是如果不把問題規模縮小，我們難以構造的問題
- 對於原本我們就會解答的問題
- 分治法有沒有任何幫助呢？

- 給定a,n,M (int範圍內)，請求出 (a^n)%M
- 還記得嗎~~ (a*b)%M = (a%M * b%M) % M
- 因此，提前取餘數並不會造成結果不同，
- 於是我們在此不用擔心溢位問題

- Ans = (a^n)%M
- 用迴圈求解
- for(ans=1,i=0;i<n;i++)
-     ans=ans*a%M;

(a^n)%M

- 時間複雜度為 O(n)!
- 可是n在int範圍內，這樣不夠快

- 其實，我們隱含著把求解「a^n」分割成求解「a^(n-1)」與「a^1」的兩個子問題，再利用相乘合併，求得「a^n」的答案
- 兩個子問題的規模懸殊太大，並不是個理想的分割方法

換個想法

- 在生活中，我們如何計算2^16呢?
- 2*2=4
- 4*4=16
- 16*16=256
- 256*256=65536

- 只要四次就足夠了
- 那2^17呢?
- 再多乘一次2就好!

- Divide (把問題分隔成相似的小問題)
  - a^n =
  - If n%2 == 0, a^n = [a^(n/2)]^2
  - If n%2 == 1, a^n = [a^((n-1)/2)]^2*a

  - a^(n/2)確實是「性質相近」且「規模較小」的子問題

- Recursive (求出小問題的解答)
  - 終止條件：If n = 1, 則 a 就是解答了!

- Conquer! (Merge 利用小問題的答案求解大問題)
- 設 b = a^(n/2), 則 If n%2 == 0, a^n = b*b, else a^n=b*b*a, 都是常數時間可以完成的工作！

分析一下時間

- 直觀的看來，每次可以約略把 n 的大小變為一半或更小
- 直到 n 變成 1 為止

- 從大問題到子問題的規模：n -> n/2 -> n/4 -> …… -> 1
- 總共有 (int)log_2(n)+1 層，每一層的合併時間為常數

- 因此，我們可以在 O(log n)時間內求出解答
- 盡量把問題分為兩個大小約略相等的子問題，讓子問題的最大規模盡量快速的變小，才能有效率的降低複雜度
- 思考：如何快速的計算 (a + a^2 + a^3 + … + a^n) 呢?

常見的排序問題

- 插入排序法
- 泡沫排序法
- 選擇排序法
- 時間複雜度 = ?

合併排序法 Merge Sort

- 現在想要排序一個陣列　長度為　n　的陣列

- 依樣畫葫蘆
- 分割：　把陣列分成左右長度差不多的兩部分
- 遞迴：　把左右兩部分都各自排序完成
- 合併：　把左右部分排序後的結果合併起來，成為大問題的答案

- 想一想：給定兩個已經排序過的陣列A,B，有沒有可以快速得到A與B所有元素一起排序過後的結果的方法呢？

由兩個分別排序過的陣列合併出整體的結果

假設兩部分都已排序完成, 該怎麼合併呢!?

- A = [1 5 6 8 9] B = [2 4 7 10]
  - C[1] = 目前 A 跟 B 當中最小的元素
  - 最小的在哪裡? **只可能在A的開頭或B的開頭**
  - C[1] = 1
- A = [~~1~~ 5 6 8 9] B = [2 4 7 10]
  - C[2] = 2
- A = [~~1~~ 5 6 8 9] B = [~~2~~ 4 7 10]
  - C[3] = 4
- A = [~~1~~ 5 6 8 9] B = [~~2~~ ~~4~~ 7 10] …….
  - C[4] = 5

繼續

- A = [~~1 5~~ 6 8 9] B = [~~2 4~~ 7 10] => C[5] = 6
- A = [~~1 5 6~~ 8 9] B = [~~2 4~~ 7 10] => C[6] = 7
- A = [~~1 5 6~~ 8 9] B = [~~2 4 7~~ 10] => C[7] = 8
- A = [~~1 5 6 8~~ 9] B = [~~2 4 7~~ 10] => C[8] = 9
- A = [~~1 5 6 8 9~~] B = [~~2 4 7~~ 10] => C[9] = 10
- A = [~~1 5 6 8 9~~] B = [~~2 4 7 10~~] => 完成!

- C = [1,2,4,5,6,7,8,9,10]

合併需要幾次運算

- 每次從A的開頭與B的開頭，挑選數值較小者
- 如果是空的就忽略他

- 決定C的每一項，　只需要一次比較 => O(1)

- 每次合併所需時間：該部分的數列長度
- A = [1 5 6 8 9]　B = [2 4 7 10]
- C = [1,2,4,5,6,7,8,9,10]
- 在這個例子中長度是 9

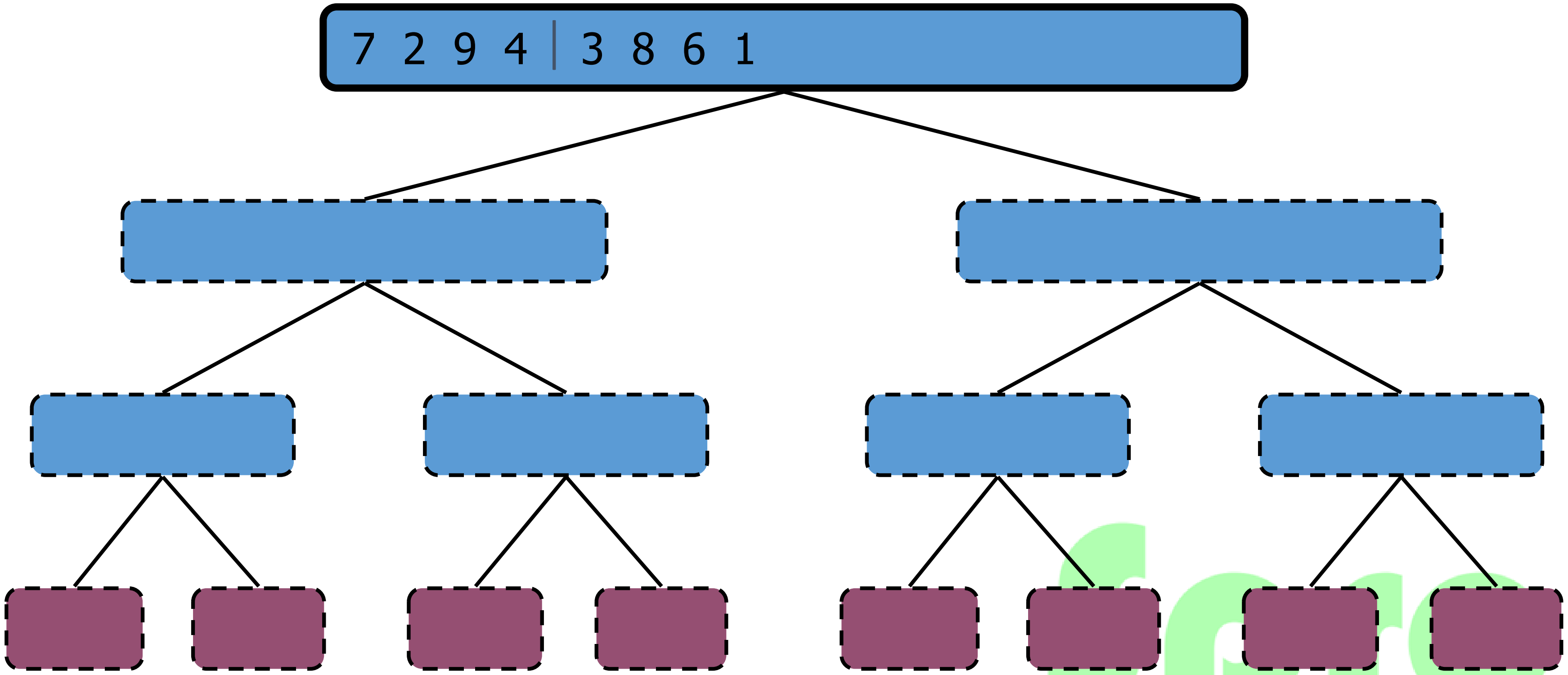
遞迴樹

- 雖然看不見，但實際上我們求解問題的過程可以畫成一個樹狀結構，我們稱之為「遞迴樹」，樹上的每個節點實際上代表每個子問題

- 不同於以往，貪心法是「直接走向最佳解所在的分支」，並得以證明(至少一組)最佳解在該分支中；

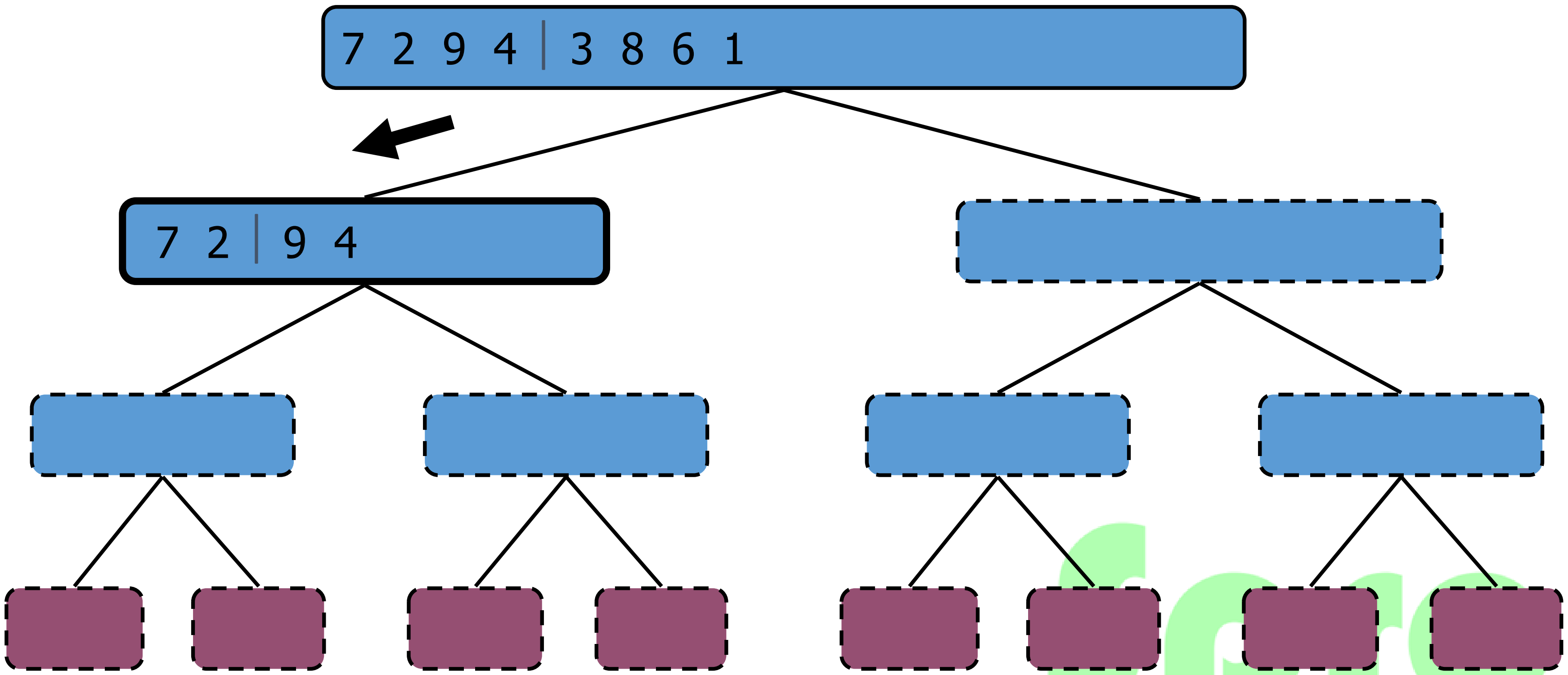
- 現在行不通了，必須「綜合考量可能分支所得到的解」，然後藉由這些小問題的解，找出原本大問題的解！

Merge sort ~

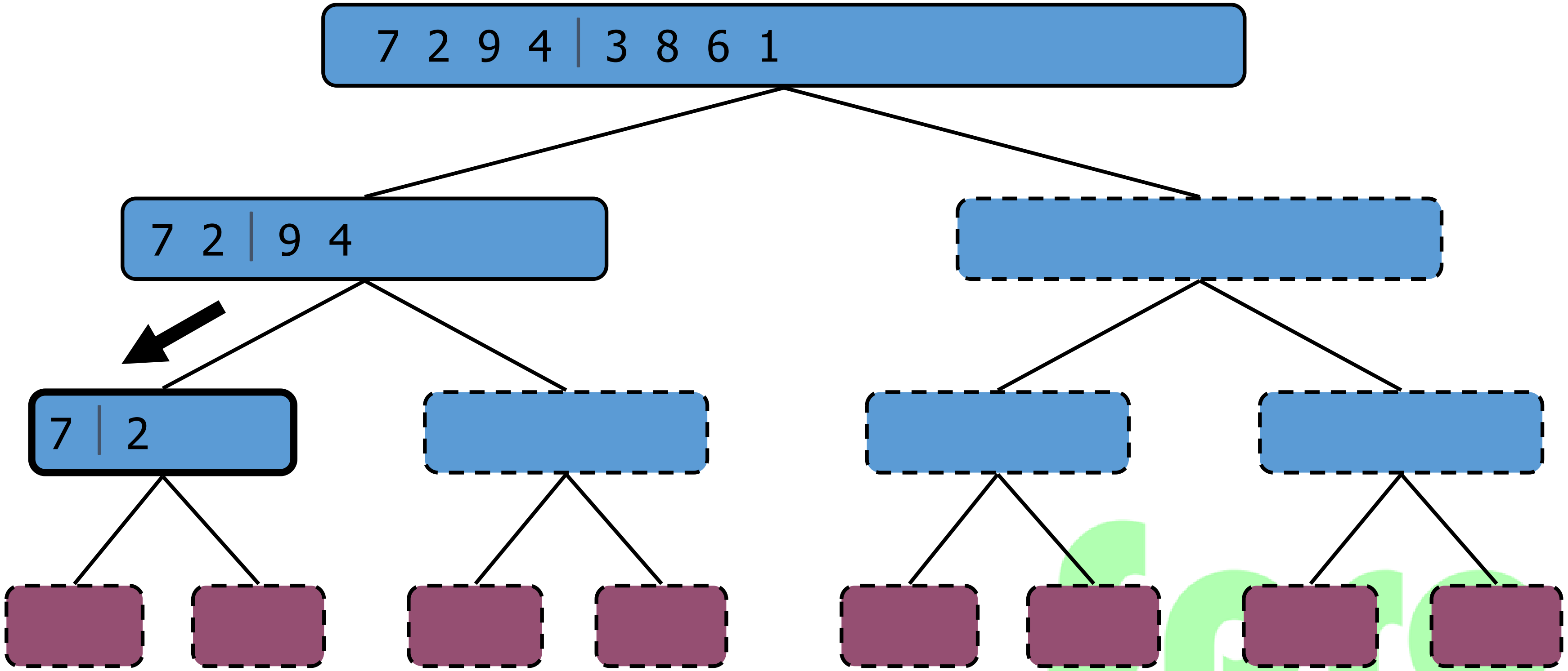


Merge sort ~

- 求左半邊

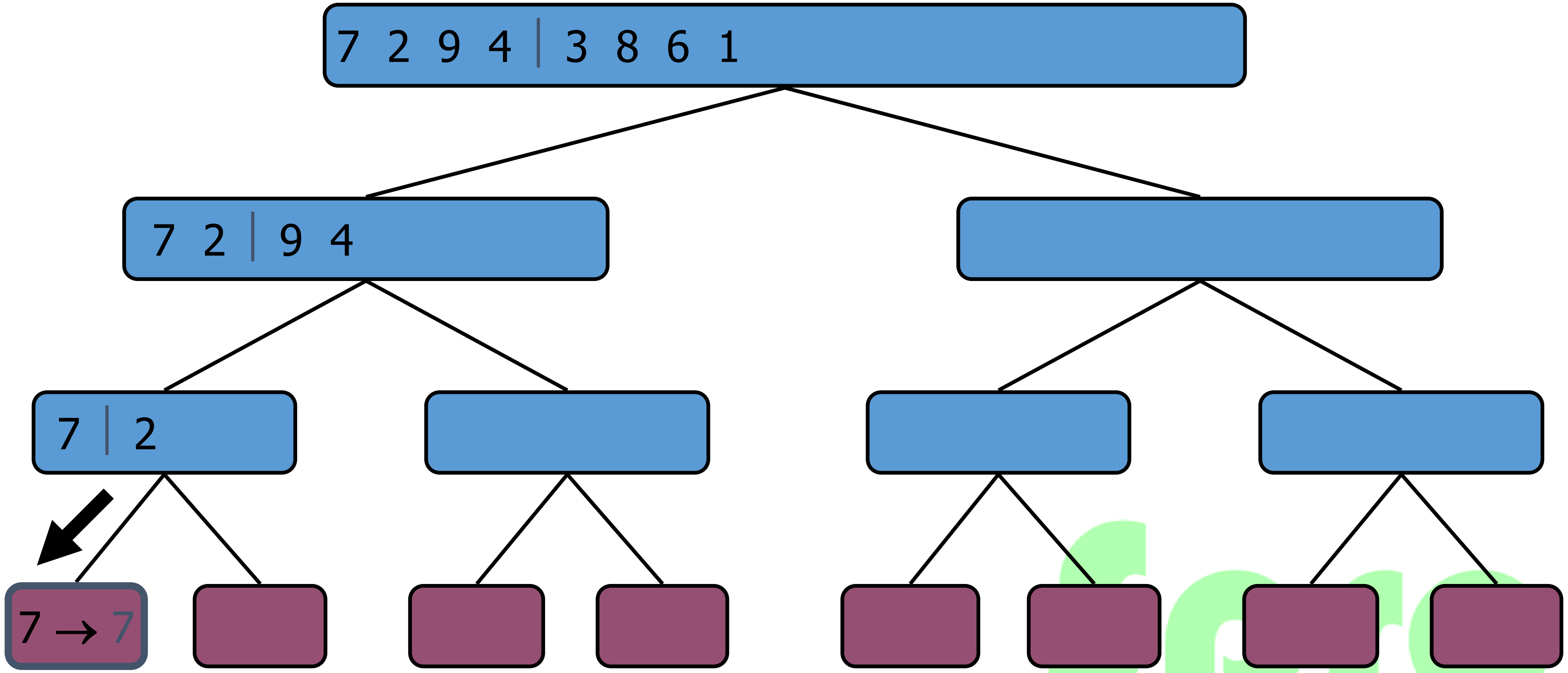


- 左半邊的左半邊

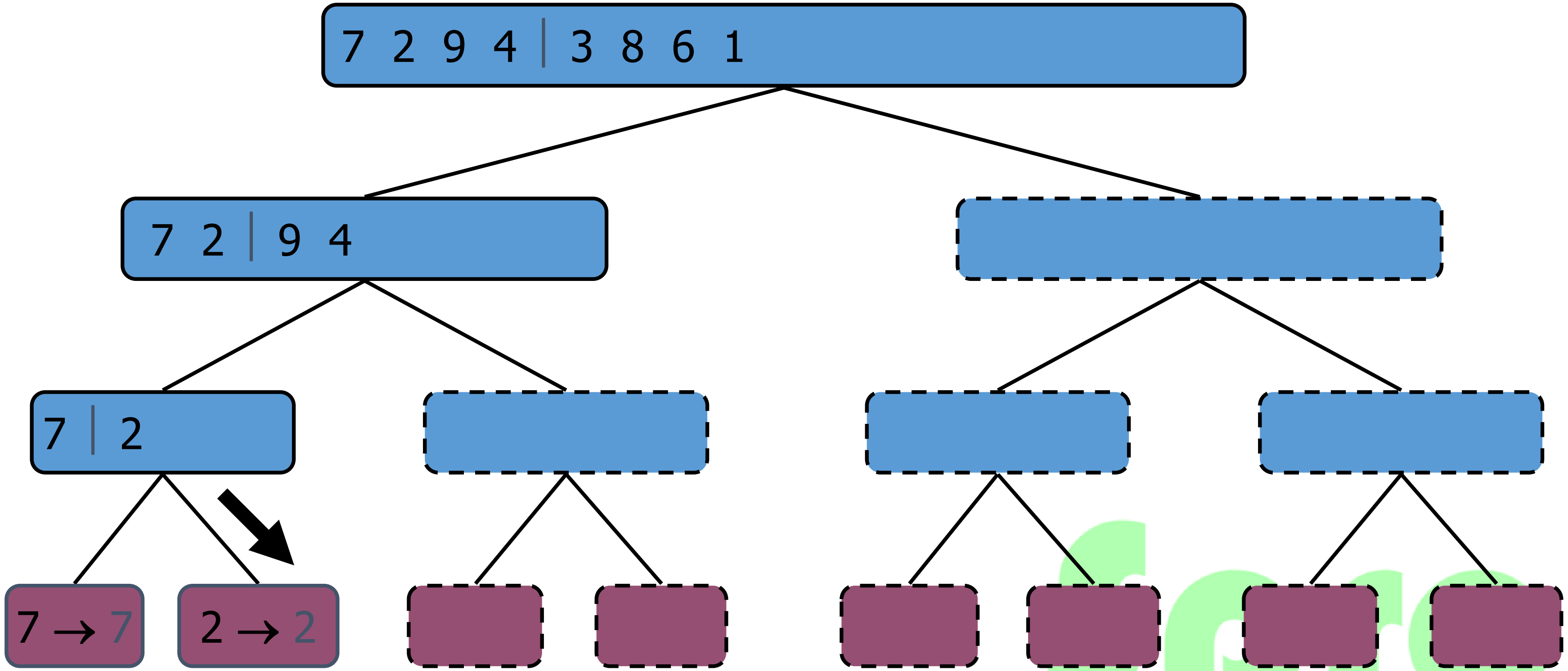


Merge sort ~

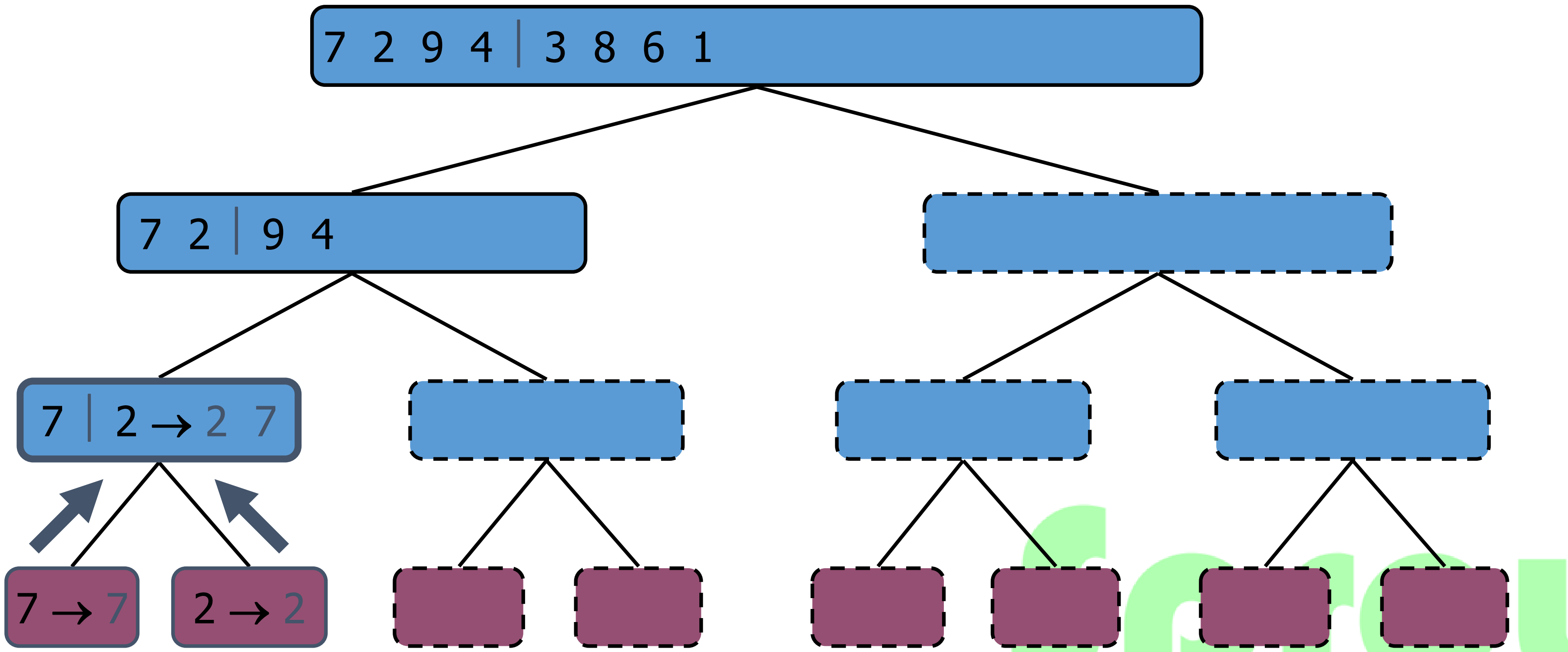
- 到"終止問題"囉，剩下一個！



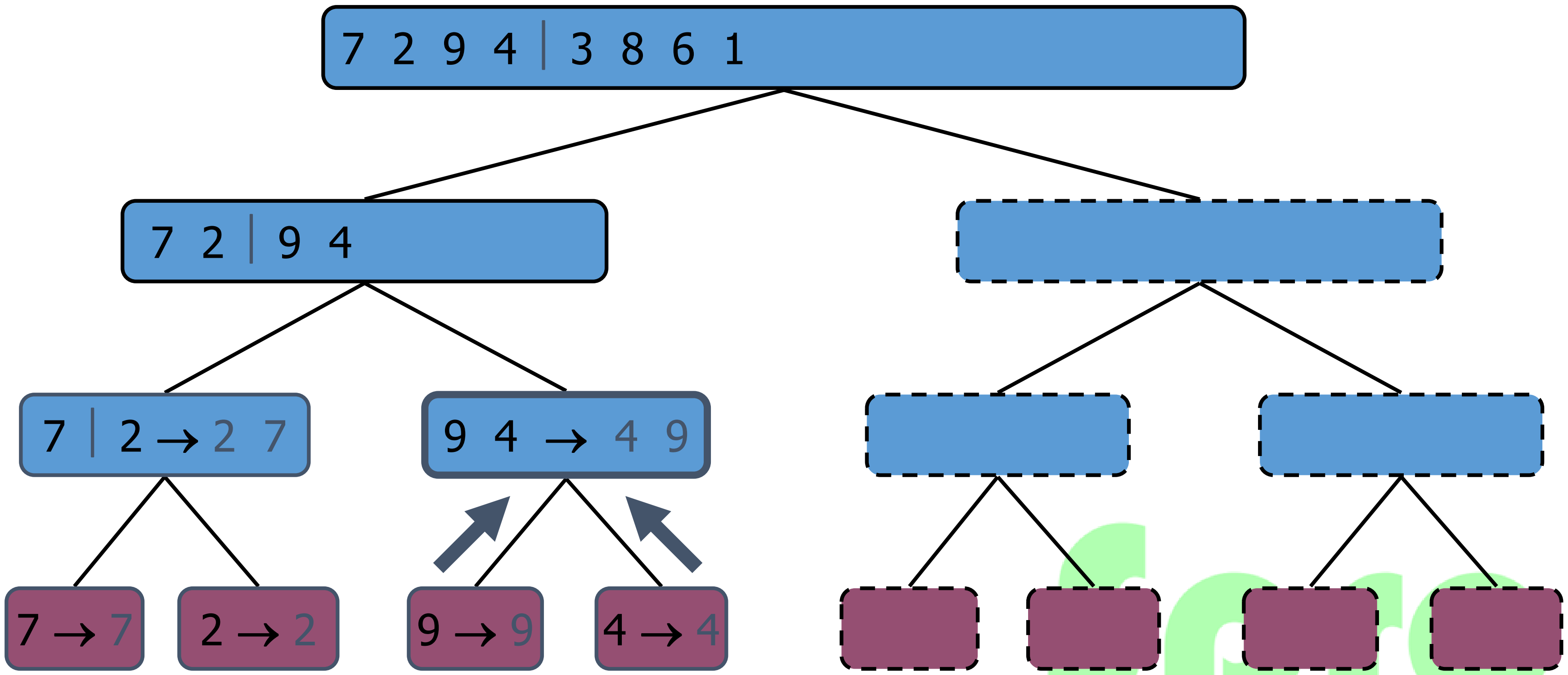
- 繼續做



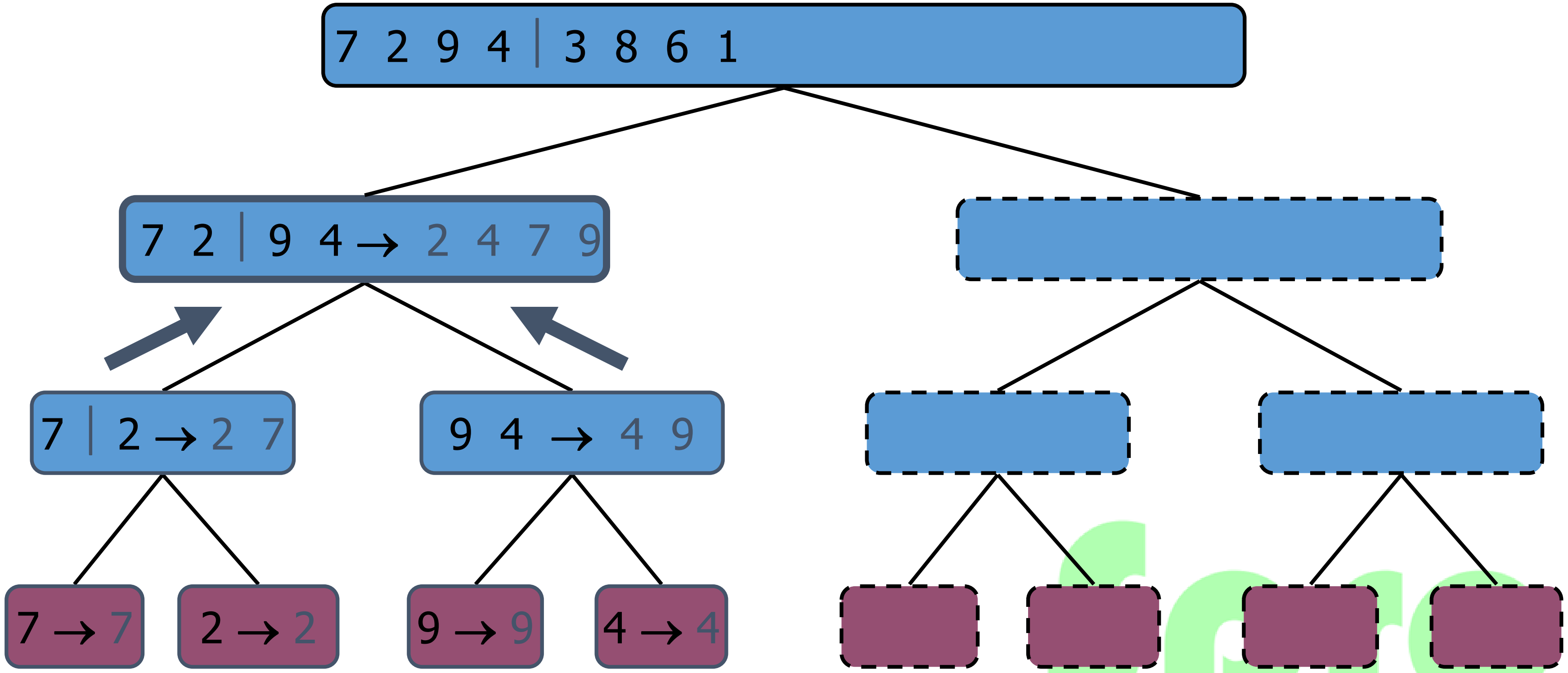
- 合併!



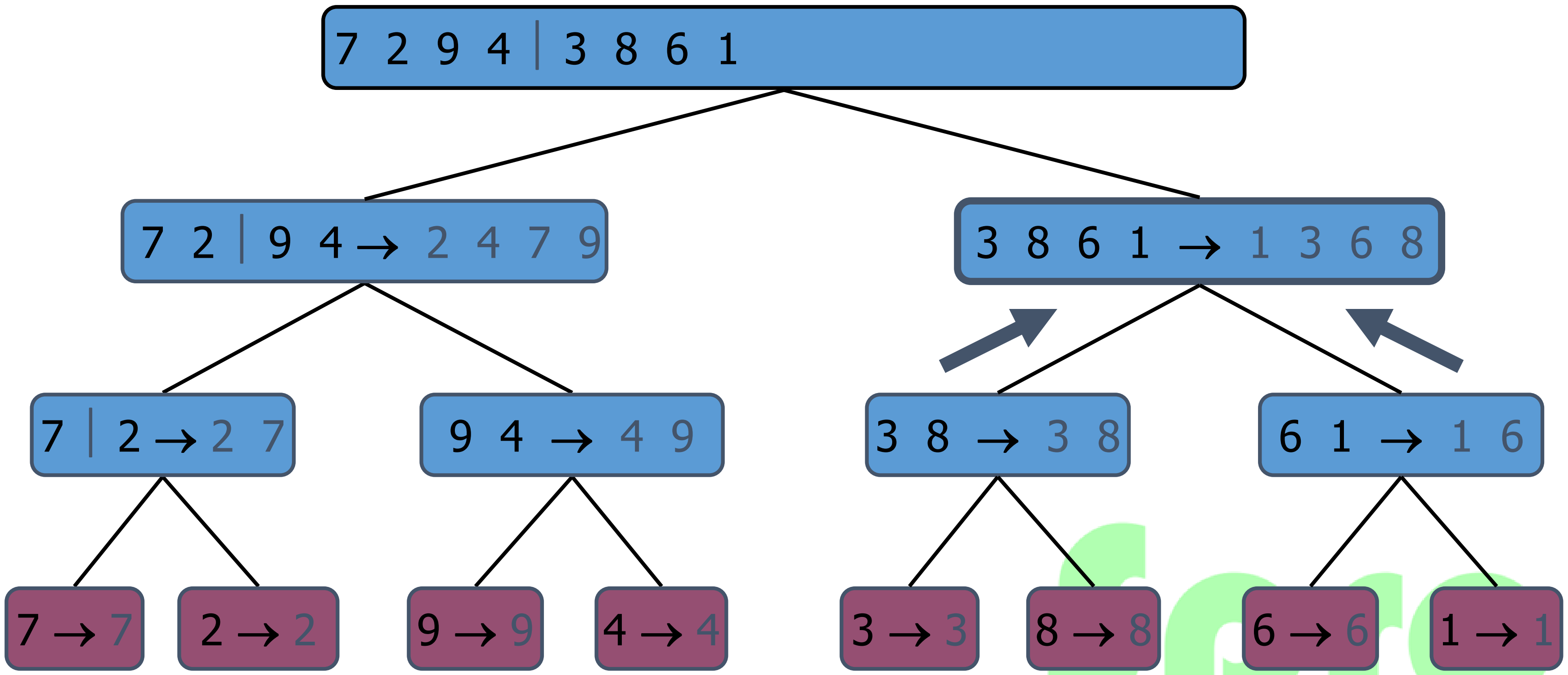
- 合併!



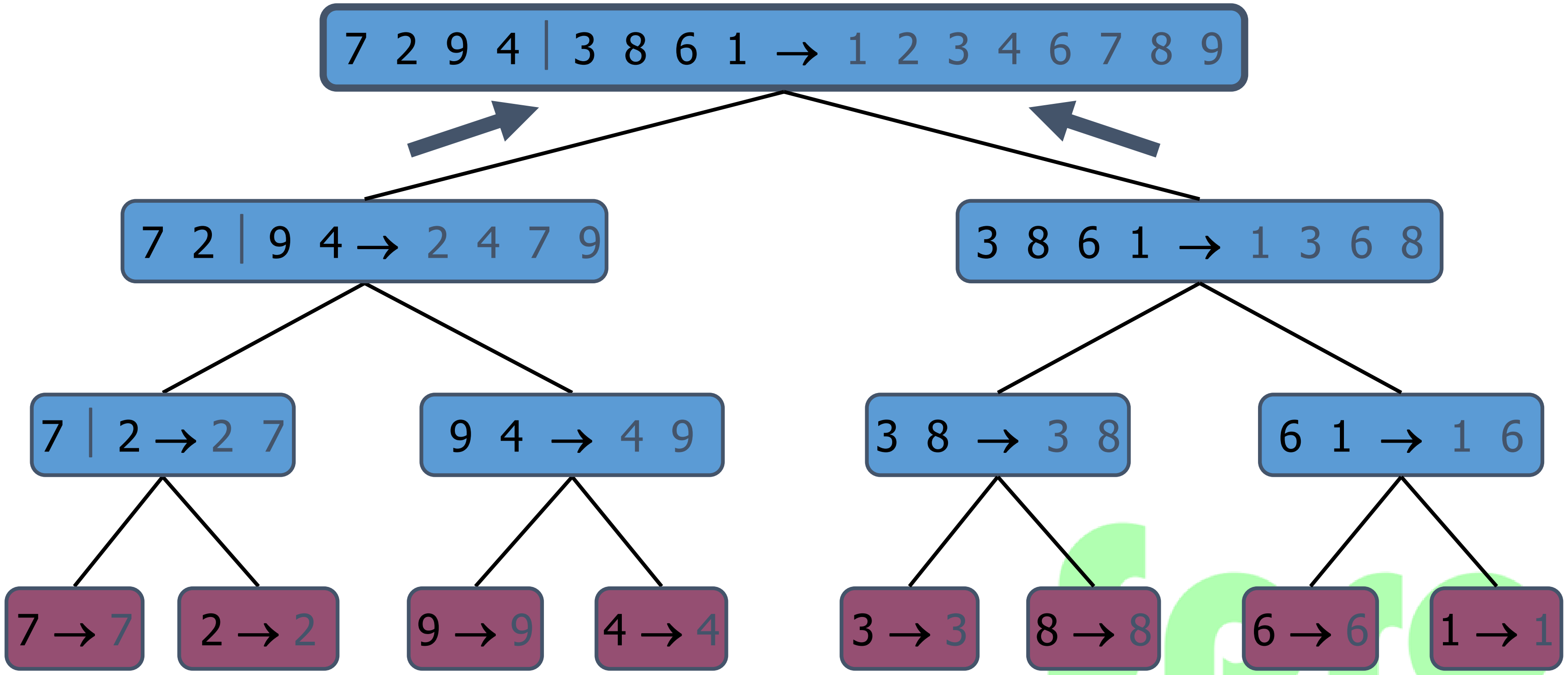
- 繼續合併



- 右邊也做一做

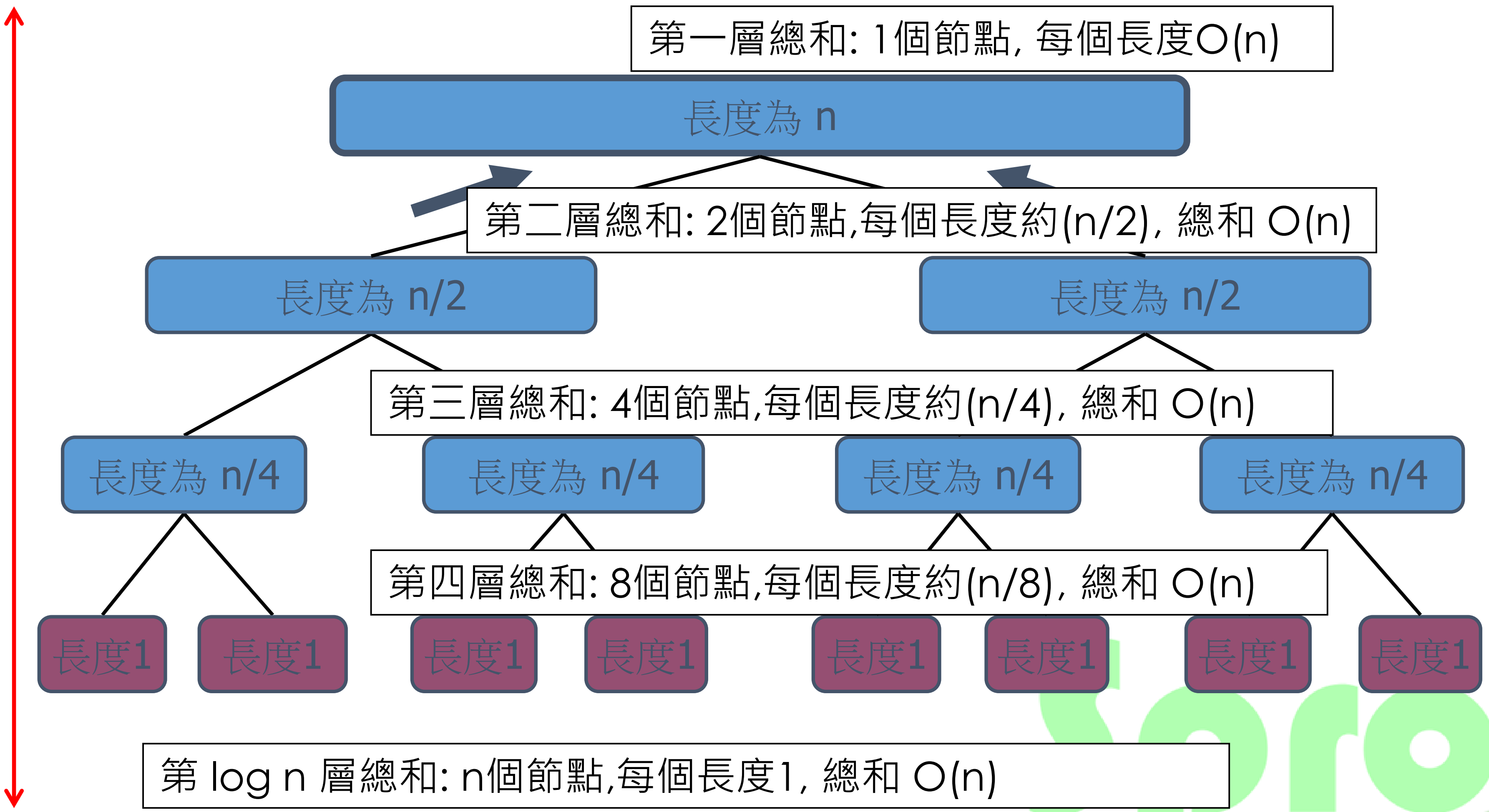

7 2 9 4 | 3 8 6 1
7 2 | 9 4 → 2 4 7 9
3 8 6 1 → 1 3 6 8
7 | 2 → 2 7
9 4 → 4 9
3 8 → 3 8
6 1 → 1 6
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1

- 完成囉！



7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9
7 2 | 9 4 → 2 4 7 9
3 8 6 1 → 1 3 6 8
7 | 2 → 2 7
9 4 → 4 9
3 8 → 3 8
6 1 → 1 6
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1

分析一下執行時間


第一層總和: 1個節點, 每個長度O(n)
長度為 n
第二層總和: 2個節點,每個長度約(n/2), 總和 O(n)
長度為 n/2
長度為 n/2
第三層總和: 4個節點,每個長度約(n/4), 總和 O(n)
長度為 n/4
長度為 n/4
長度為 n/4
長度為 n/4
第四層總和: 8個節點,每個長度約(n/8), 總和 O(n)
長度1
長度1
長度1
長度1
長度1
長度1
長度1
長度1
第 log n 層總和: n個節點,每個長度1, 總和 O(n)

每一層的時間都是 O(n) ！

- 終止條件為n=1，總共有幾層呢？
- 顯然由一些簡單的數學，會發現共有 log N 層!
- 總時間 O(n log n) !!
- 空間呢? 我們只要利用原本的陣列操作就可以囉!
- 需要一個輔助陣列暫存合併後的結果
- 不可以覆蓋到原本的兩個要合併的陣列

Merge-sort function

- N、A[0~(N-1)]
- mergesort(0, N):   ← 其中包含Left, 不包含Right
- Example: mergesort(3, 7) => 排序 A[3], A[4], A[5], A[6]

```
mergesort(Left, Right):
 if(Left+1==Right) return;  ←  長度為1, 終止條件
 int Mid = (Left + Right) / 2;  ←  找出中間的位置
 mergesort(Left, Mid); ← 遞迴求解,  排序好 A[Left], …, A[Mid-1]
 mergesort(Mid, Right); ←遞迴求解,  排序好 A[Mid], …, A[Right-1]
 int L=Left, R=Mid, K=Left;
 while(L<Mid || R<Right) ← 開始合併 左半邊開頭 = L, 右半邊 = R
    if( L<Mid && (R>=Right || A[L]<=A[R]) )
                // 若左半邊還有東西 , 且 (1)右半邊空了 (2)左半邊的開頭較小
                B[K++] = A[L++]; ← 放進左半邊的開頭, 而開頭位置L加1
         else
                B[K++] = A[R++]; ← 放進右半邊的開頭, 而開頭位置R加1
 for(L=Left; L<Right; L++) ← 丟回去原本的陣列
        A[L]=B[L];
```

另外的一種思路

- 快速排序法
- 對問題先好好的分割，讓合併時不那麼麻煩！
- 分割：
  - 從陣列中選一個值 x
  - 亂排一下，把陣列分成三個區域 順序保持 L; M; R
    - L: <x的元素 (順序不重要)
    - M: =x的元素
    - R: >x的元素 (順序不重要)
- 遞迴：用快速排序法分別把 L 跟 R 排好
- 合併：把左右各自的結果合併起來
  - 怎麼合併? **不用合併**,遞迴完成自然而然整個都是遞增的

快速排序法~

- 選擇 x 值 = 6，讓數列滿足 小於6 等於6 大於6
- 在此為了講解方便，三個部份我們依照原本的順序排列
- 這步有很多種不同的實作方式
- (L) 7 2 9 4 3 7 6 1 (R)
- 反覆進行以下操作：直到 L>=R 的位置為止
  - 從L往右找第一個>=6的數 p
  - 從R往左找第一個<=6的數 q
  - 交換 p,q ，L往右移一格，R往左移一格
- 7 (L) 2 9 4 3 7 6 (R) 1
- 1 2 9 (L) 4 3 7 (R) 6 7
- 1 2 6 4 3 7 9 7
- 在此為了講解方便,我們假設結果為 *[2 4 3 1]* 6 *[7 9 7]*

7 2 9 4 3 7 6 1 → *2 4 3 1* 6 *7 9 7*

快速排序法~

- 開始排序，左右部分已經分好囉

7 2 9 4 3 7 <u>6</u> 1→ *2 4 3 1* <u>6</u> *7 9 7*

快速排序法~

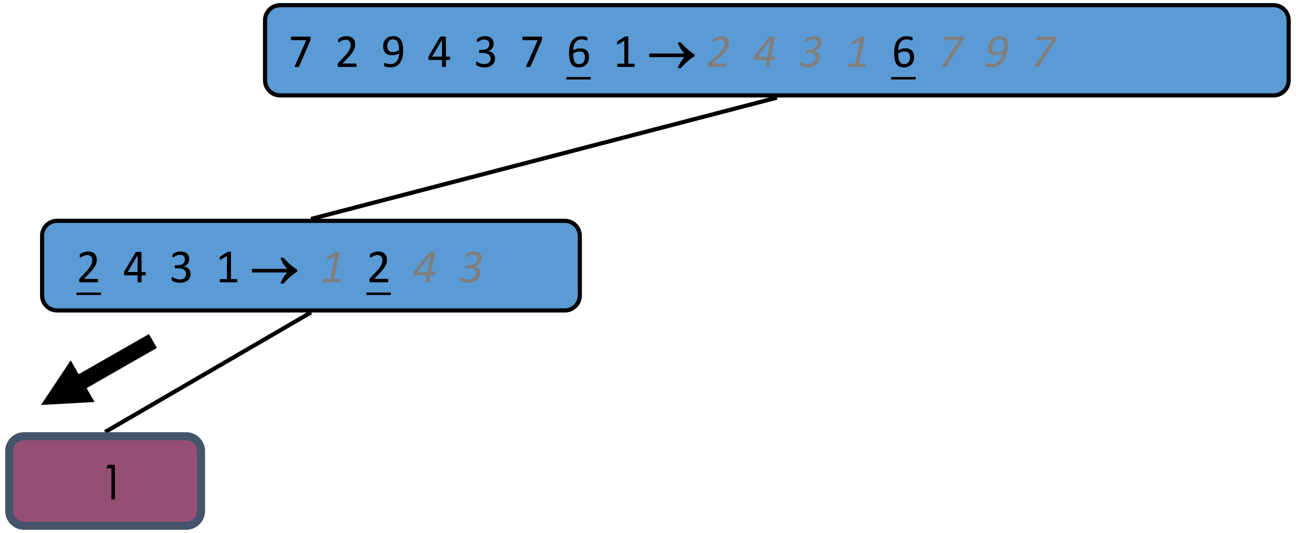
- 遞迴排好左半邊

7 2 9 4 3 7 <u>6</u> 1 → *2 4 3 1* <u>6</u> *7 9 7*

<u>2</u> 4 3 1 → *1* <u>2</u> *4 3*

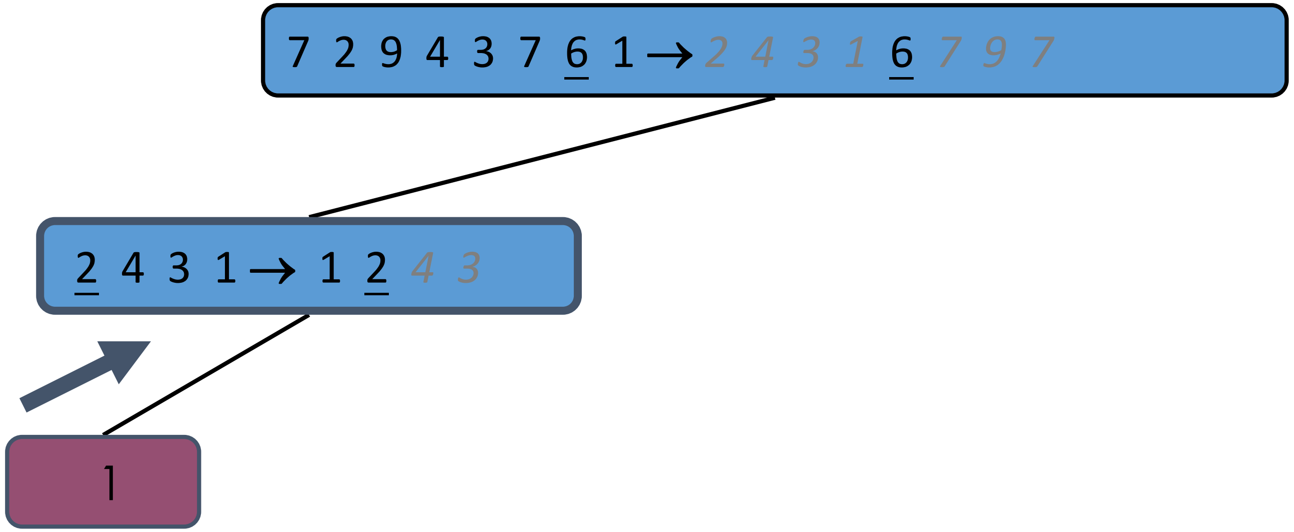
快速排序法~

- 遇到終止條件



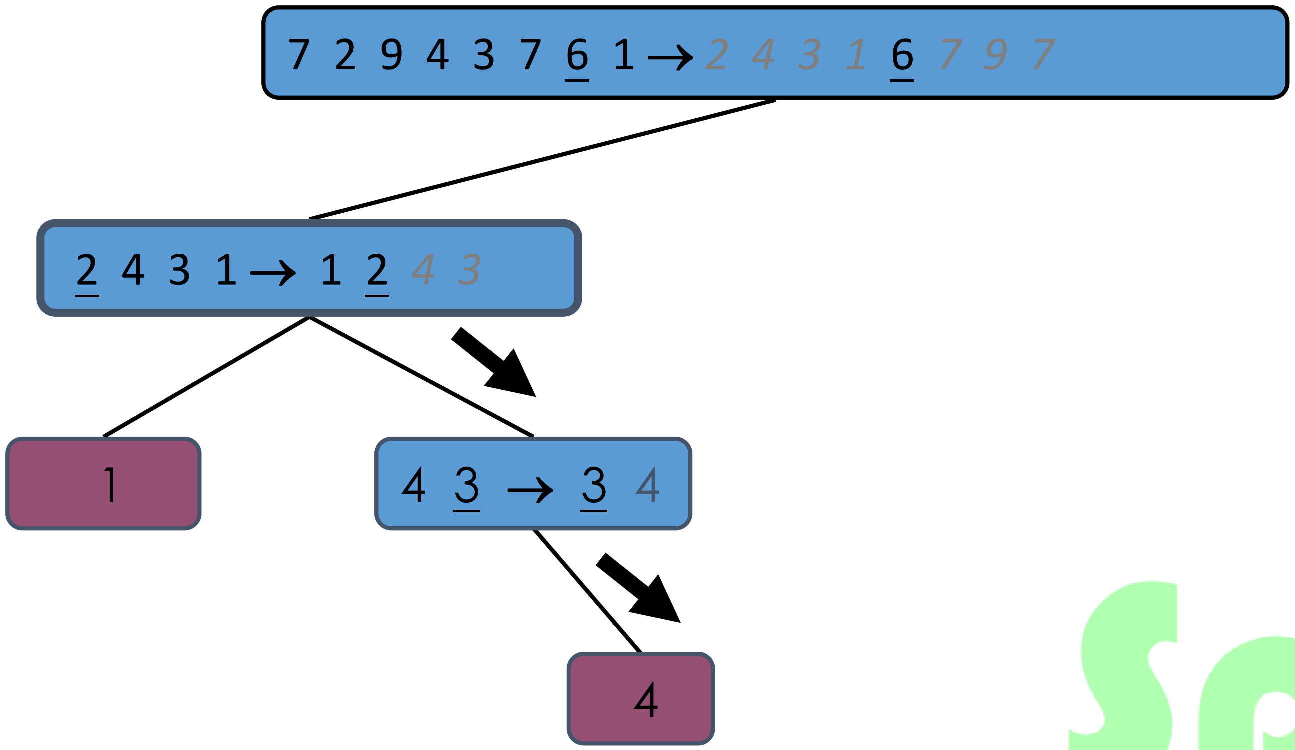
快速排序法~

- 1 的順序確定了



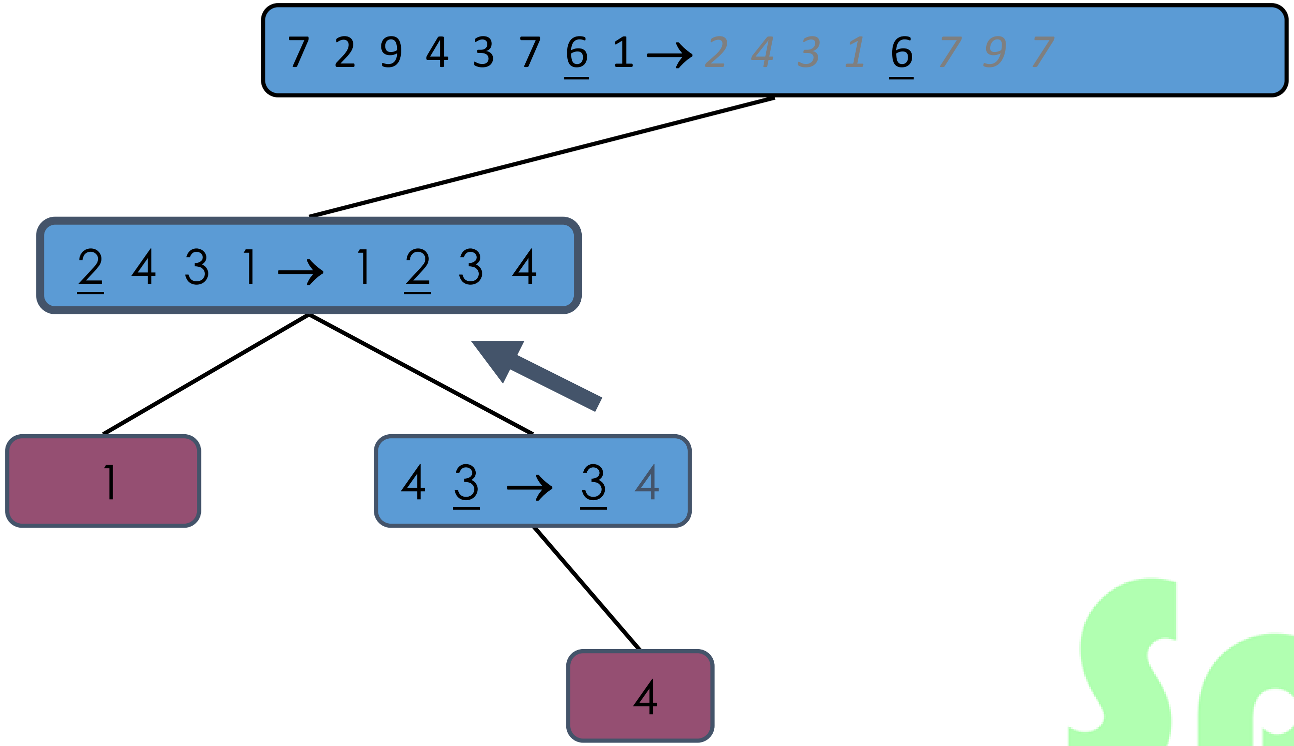
快速排序法~

- 排右半邊



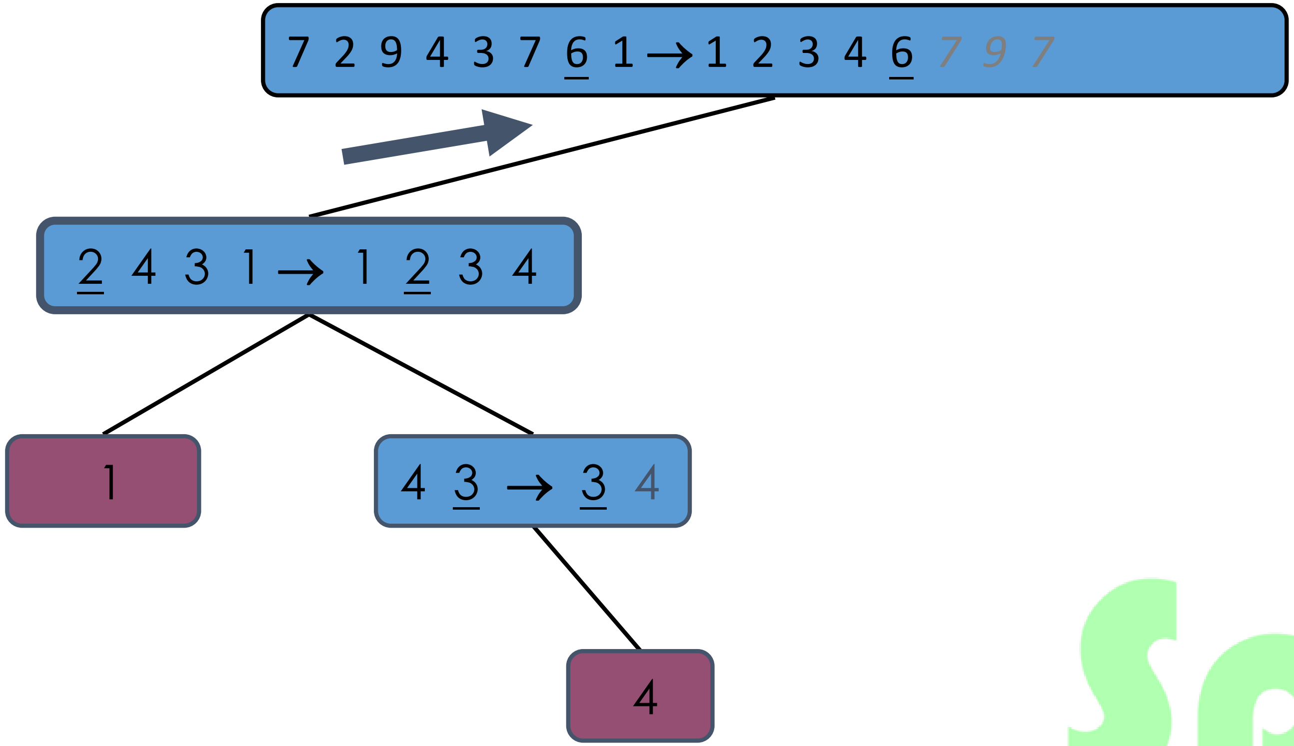
快速排序法~

- 3 與 4 的順序確定了， 由於原數列滿足 小 等 大
- 所以兩個部分各自排好後，整體就滿足從小到大的關係



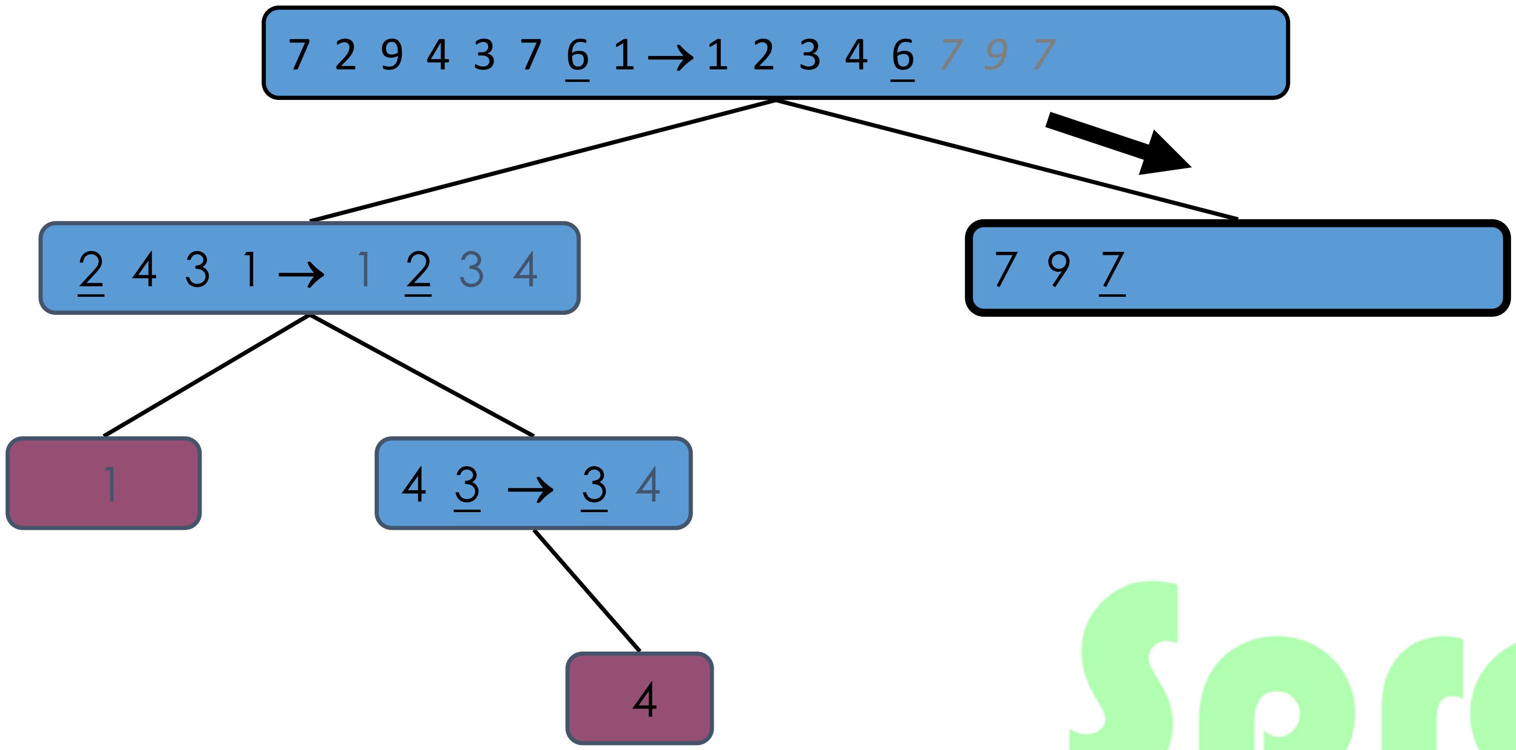
快速排序法~

- 左半邊都排序完成了



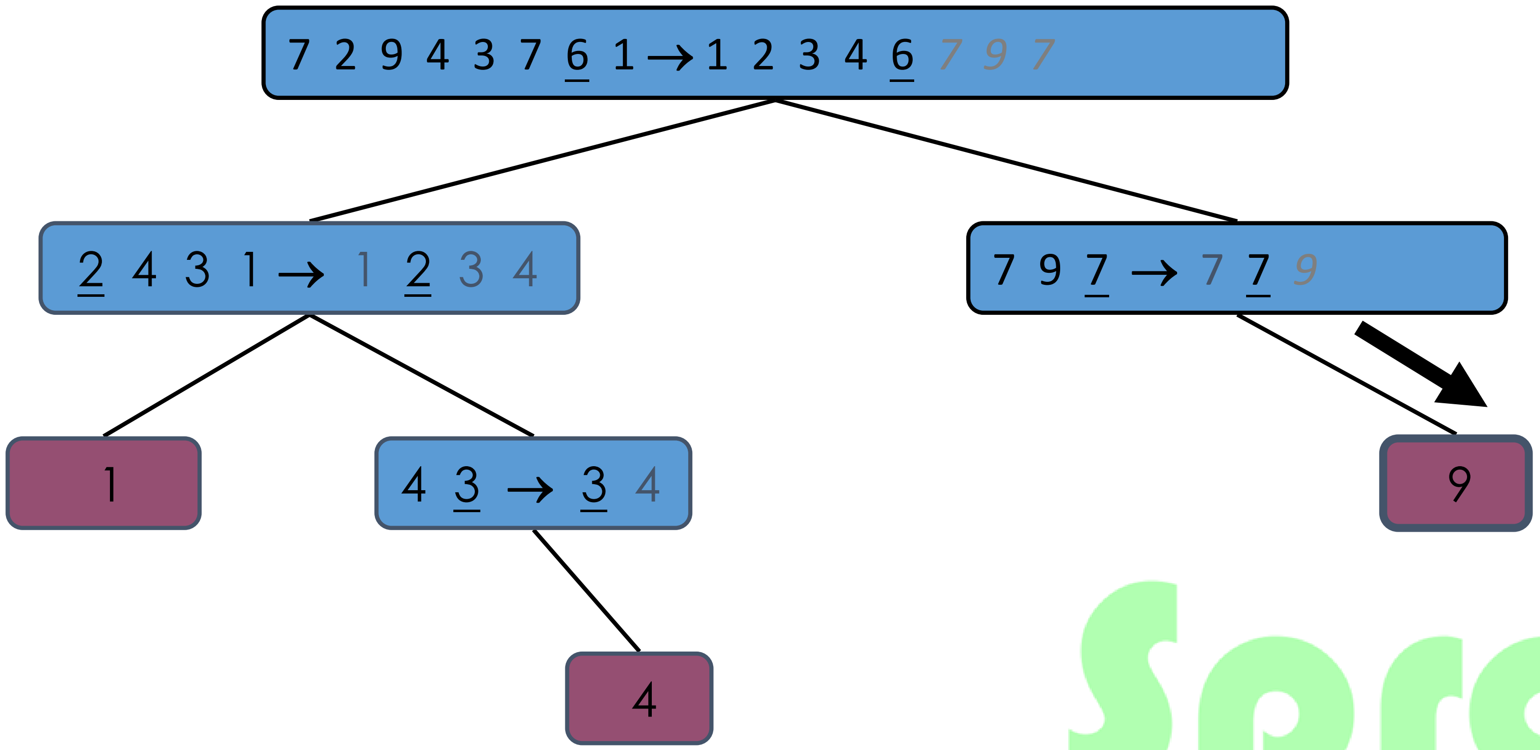
快速排序法~

- 開始做右半邊



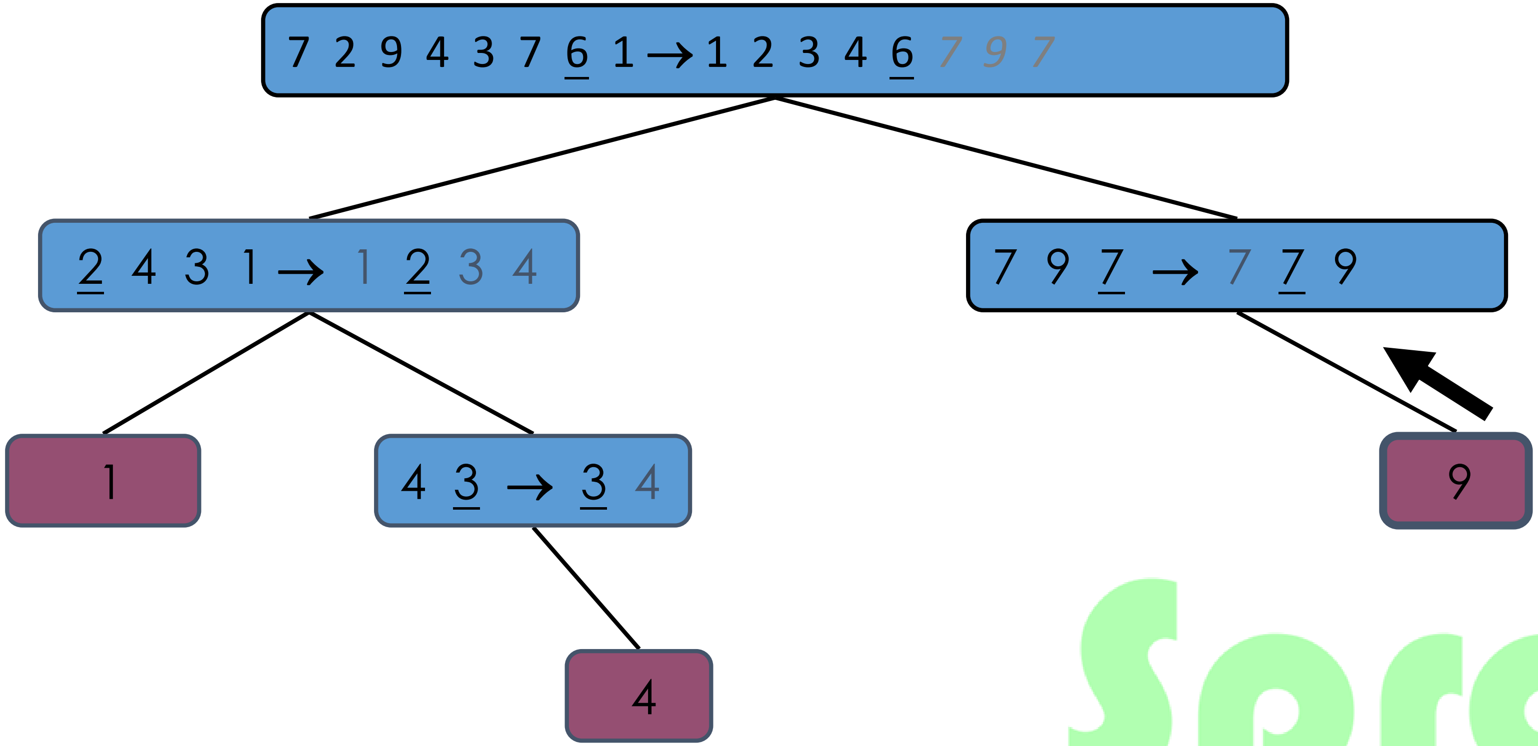
快速排序法~

- 小於7的部分是空的，不需要遞迴求解



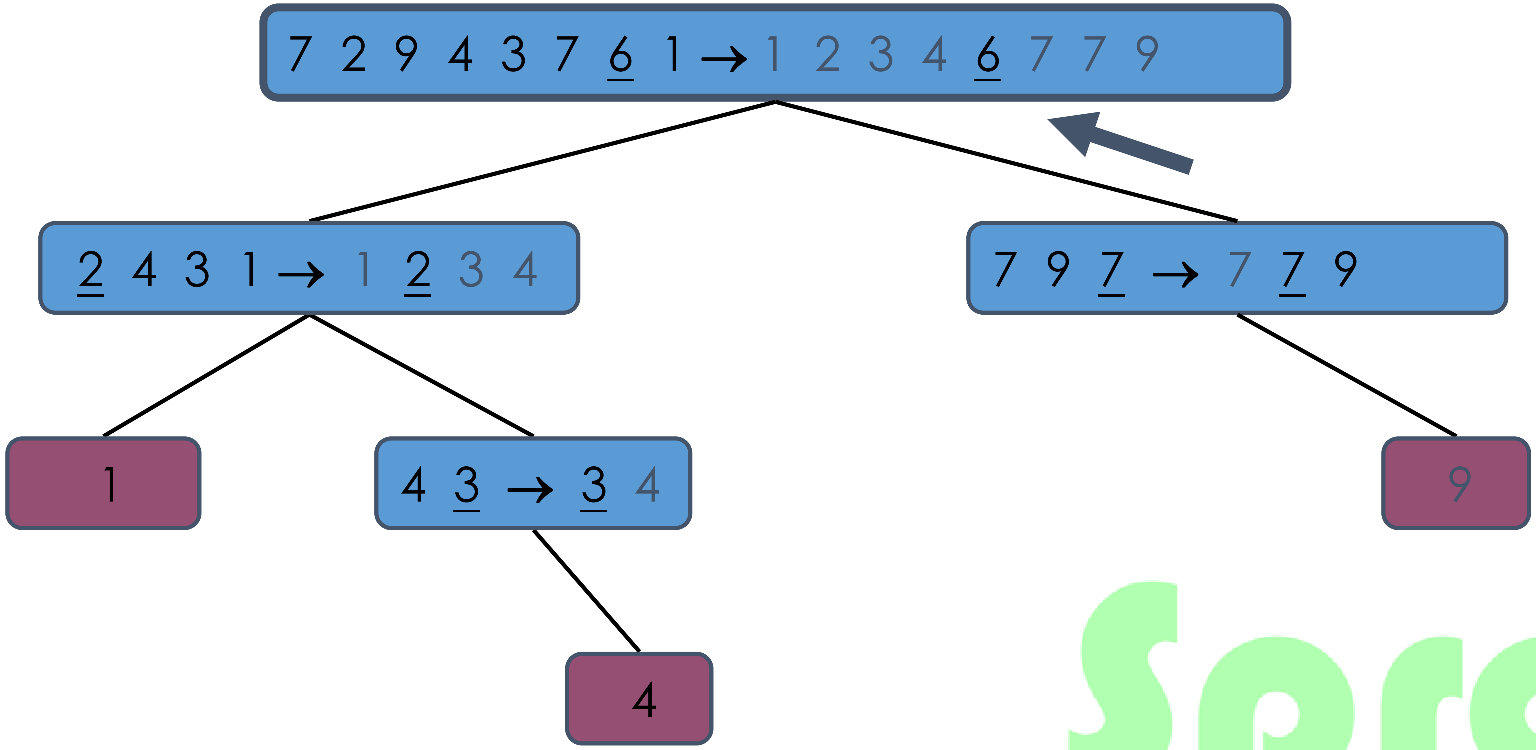
快速排序法~

- 更新回去（雖然只有一個元素）



快速排序法~

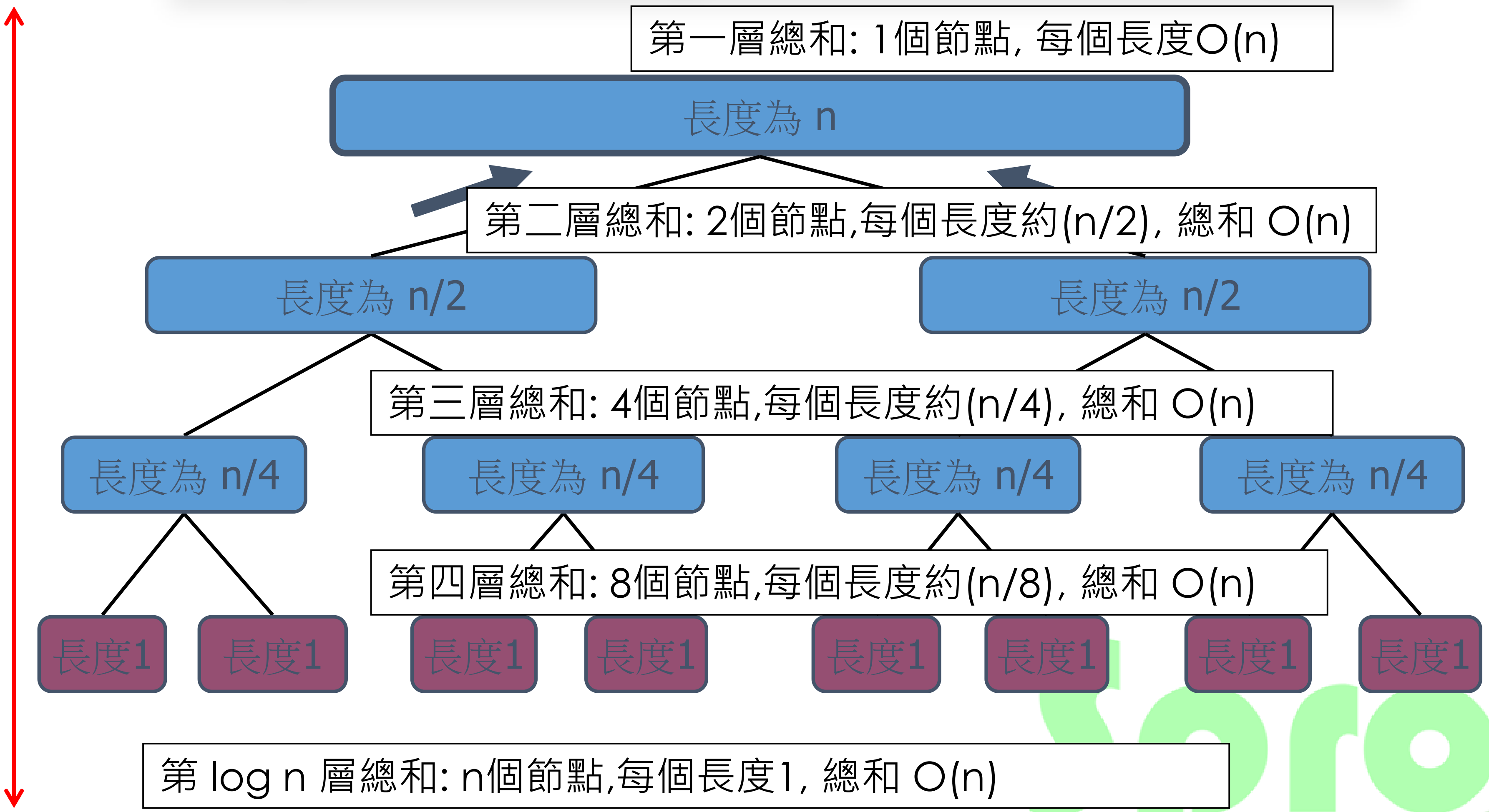
- 兩個部分都做好了，大功告成 @_@



分析時間

- 每次需要比較一個元素與x的大小關係，決定他的位置在左邊還是右邊
- 所需時間與該部分元素數量呈線性關係
- 每一層總和都接近 O(n)等級

好的情況
(長相跟Merge sort一模一樣)



分析時間

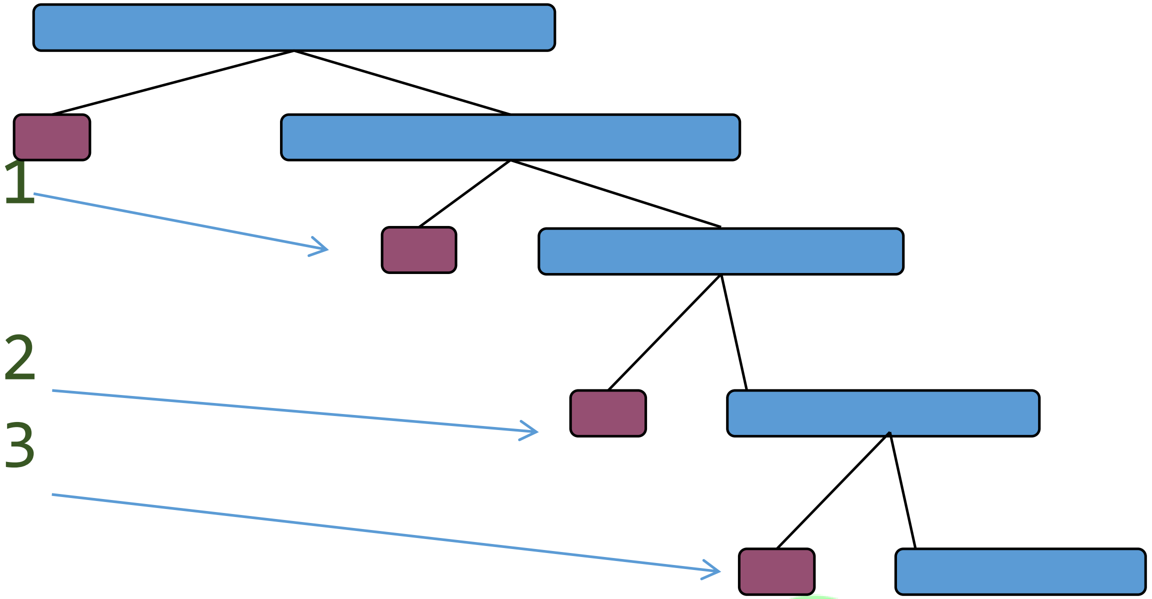
- 在一般的情形下 (x介於中間，使得L與R的長度差不多)也只會有log n層， 於是在這種情形下的時間複雜度與Merge sort相等
- O(n) * O(log n) = O(n log n)

- 還記得演算法的執行時間中，除了時間複雜度(執行時間與問題規模的相對增長速率)以外，執行時間的常數也會稍微影響執行速度

  快速排序法在好的情況略比Merge Sort快

糟糕的情況

- 長度總和n
- 長度總和n-1
- 長度總和n-2
- 長度總和n-3
- ....
- 長度總和1
- 1 + 2 + 3 + … + n = n(n+1)/2 = O(n^2) !!



所以……

- 總共可能高達n層!!!
- 最差情況: O(n) * O(n) = O(n^2) 遠大於 O(n log n)
- 幸運的是，這並不常發生

- 邪惡的出題者表示：依據莫非定律，你覺得不會發生的情況有時特別容易發生OAOAOAO

比較一下兩種排序的方法

- 合併排序:
- 隨意分割問題(左右切一半), 然後好好的合併
- 因為分割的兩個問題大小接近, 所以遞迴的深度不會超過 O(log n), 總時間 O(n log n).

- 快速排序: (聽起來很迅速)
- 好好的分割問題(排列好大小關係), 然後自然就合併了
- 因為分割的兩個問題大小可能差異很大, 所以遞迴的深度高達O(n), 總時間最差 O(n^2), 平均來說 O(n log n).
- 我們可以隨機的選取中心點, 這樣期望複雜度為O(n log n)

經典問題 – 逆序數對數量計算

- A = [2, 4, 1, 3]
- 對於一個陣列 A 中，任取兩個元素，順序不變，如果前者大於後者，則我們稱為”逆序數對”
- (2,4)
- (2,1) => 逆序數對
- (2,3)
- (4,1) => 逆序數對
- (4,3) => 逆序數對
- (1,3)

計算逆序數對的數量

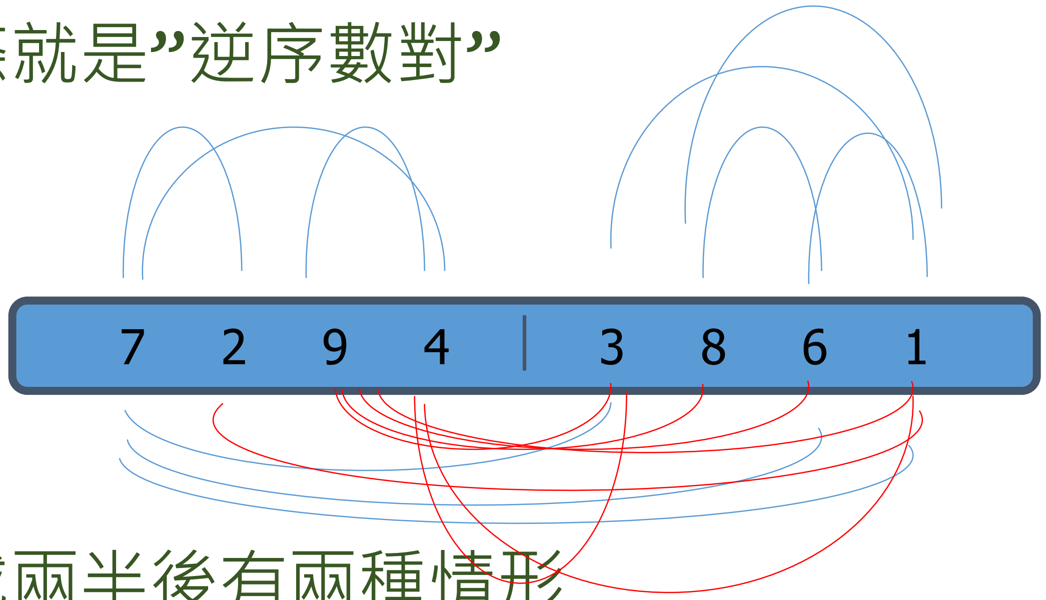
- 既然不知道答案，就每一種組合都試試看！
- 還記得傳說中的枚舉法
- 枚舉所有數對…檢查一下大小就好！

- 有 n(n-1)/2 = O(n^2)種！
- 成本太高囉

- 仔細想想，我們並不是真的需要找出所有逆序數對

分治！

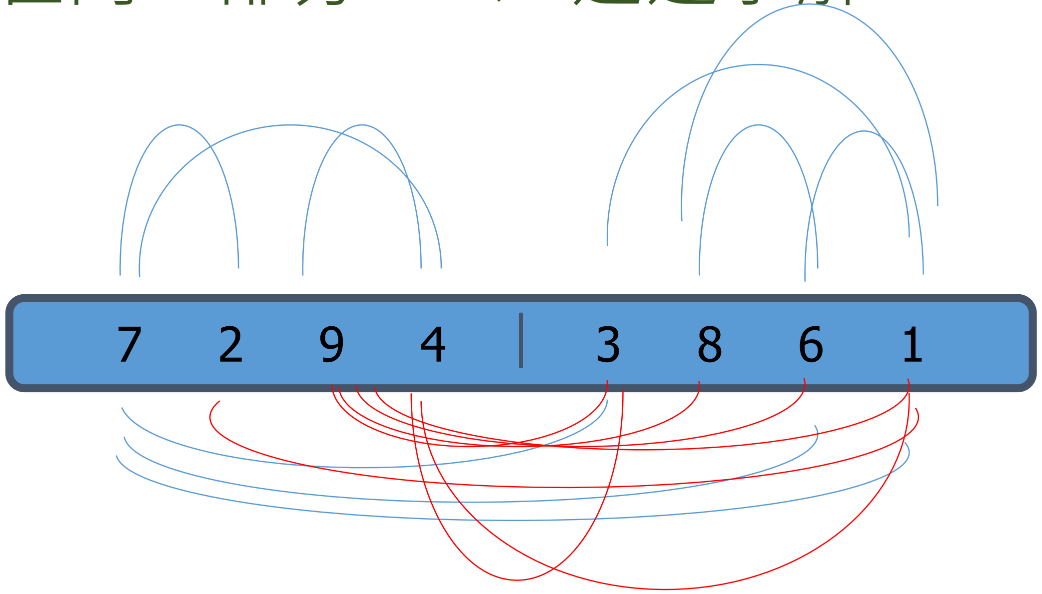
- 我們試著把陣列分成兩半
- 線條就是”逆序數對”



- 分成兩半後有兩種情形
- 1. 在同一部分（上面的線條）
- 2. 在不同部分（下面的線條），橫跨兩邊

分割問題

- 1. 在同一部分 => 遞迴求解



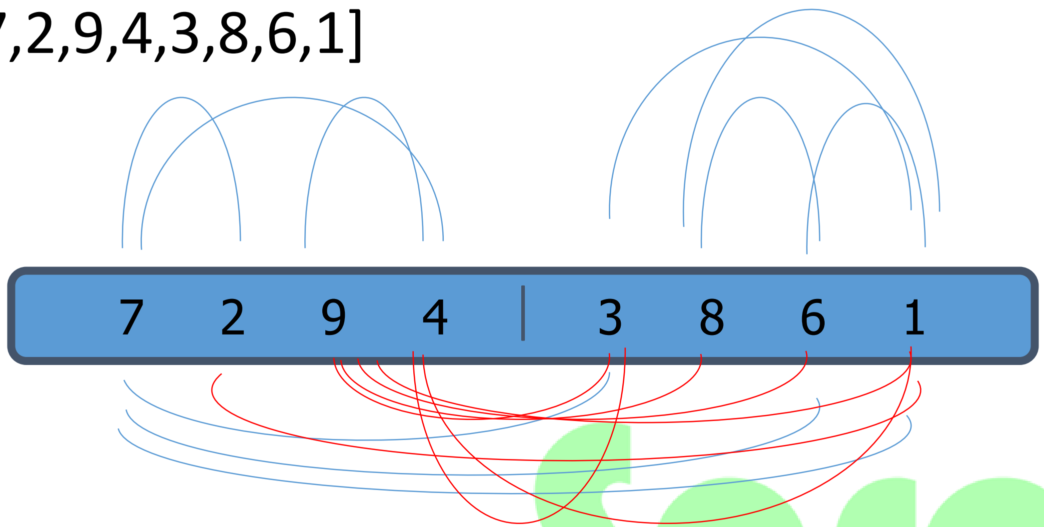
- 2. 在不同部分 => 合併的時候處理

遞迴形式

- 因為分割問題的方法與 Merge Sort 非常類似，因此就沿用一樣的定義吧!

寫成函數

- **N、A[0~(N-1)]**
- int counting(Left, Right): ← 其中包含Left, 不含Right
- 回傳逆序數對的數量
- N=8, A[0~(N-1)] = [7,2,9,4,3,8,6,1]
- counting(0,4) = 3
- counting(5,8) = 4
- Mid=(Left,Right)/2



- counting(Left,Right)=
  - counting(Left, Mid) (都在左邊)+counting(Mid, Right) (都在右邊)
  - 橫跨左右部分的逆序數對 (下面的線條)

合併問題

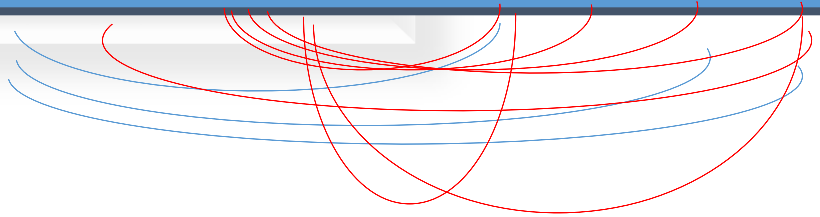
- counting(Left,Right)=
  - counting(Left, Mid) (都在左邊)+counting(Mid, Right) (都在右邊)
  - 橫跨左右部分的逆序數對 (下面的線條)
- 橫跨左右部分的逆序數對： 怎麼求呢?

- 枚舉左邊的每一項,右邊的每一項
- 在第一層就 O(n/2) * O(n/2) = O(n^2) !

- 辛苦的分治白忙一場， 問題的合併變成瓶頸
- 有沒有更好的合併方法呢

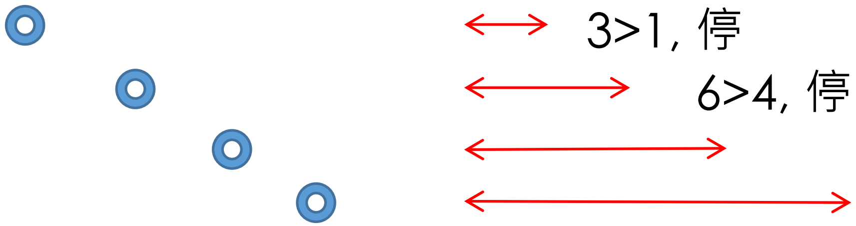
觀察性質

7	2	9	4	3	8	6	1



- 觀察1：因為這些數對橫跨兩邊，因此：
- 即使將左右兩部分排序，這部分的數量依然相同！

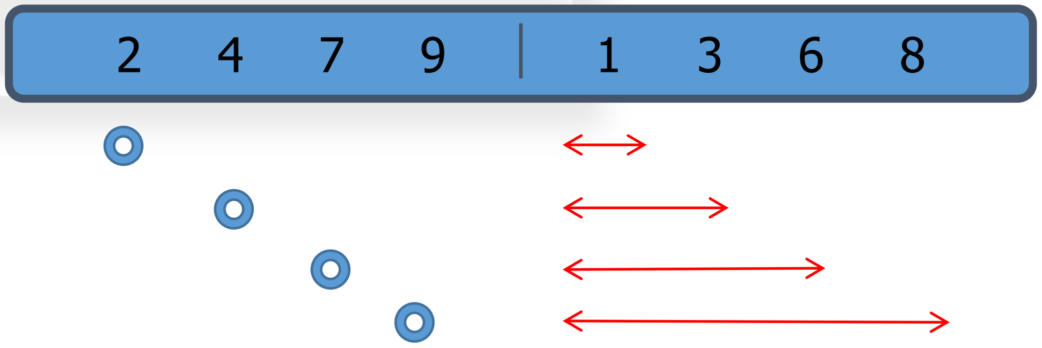
2	4	7	9	1	3	6	8



1對: (2,1)
2對: (4,1)/(4,3)
3對
4對
共有10對!

- 觀察2：經過排序以後瓊舉左邊的每一項，與它有關的逆序數對是原本右邊滿足的部分，加上往右找比它小的那些數值！

時間呢



- 回傳：左半部分與右半部分(排序前)的逆序數對數量：3+4
- 加上上面所有箭頭的長度和：10 = 3+4+10 = 17
- 每enumerate左邊的一項x，就只要檢查上一次enumerate的部分箭頭終點的下一項y，如果y<x，那箭頭就可以延伸!
- 設該層共有n個元素，左右各n/2個
- 左邊枚舉n/2次
- 箭頭也至多只會延伸n/2次
- O(n) !!
- 與合併排序法的時間分析一模一樣

寫寫程式吧

- 以合併排序法為基礎
- 在合併前額外加上計算“橫跨左右部分的逆序數對數量”
- 回傳三個部份的總和，就是在處理範圍中的逆序數對數量
- 順便也幫忙排序完成了！

- 在此問題中，子問題的答案並不只有「該區間內逆序數對的數量」，也隱含了「該區間內排序過的結果」

counting function

- N、A[0~(N-1)]
- int counting(Left, Right): ← 其中包含Left，不含Right
- 回傳逆序數對的數量

```
int counting (Left, Right):
 if(Left+1==Right) return 0;  ←  長度為1，終止條件
 int Mid = (Left + Right) / 2;  ←  找出中間的位置
 int Cnt = counting(Left, Mid); ← 遞迴求解，順便排序A[Left]~A[Mid-1]
 Cnt += counting(Mid, Right); ←遞迴求解,順便排序A[Mid]~A[Right-1]
 int L, R=Mid; ← R: 箭頭的起點永遠在中間
 for(L=Left; L<Mid; L++){ ← 枚舉左邊的每一項
    while(R<Right && A[R]<A[L]) ← 如果箭頭可以延伸的話
          R++; ← 就延伸吧!
    Cnt += R-Mid; ← 加上箭頭的長度
 }
 //Merge sort: 合併左右兩個部分
 return Cnt;
```