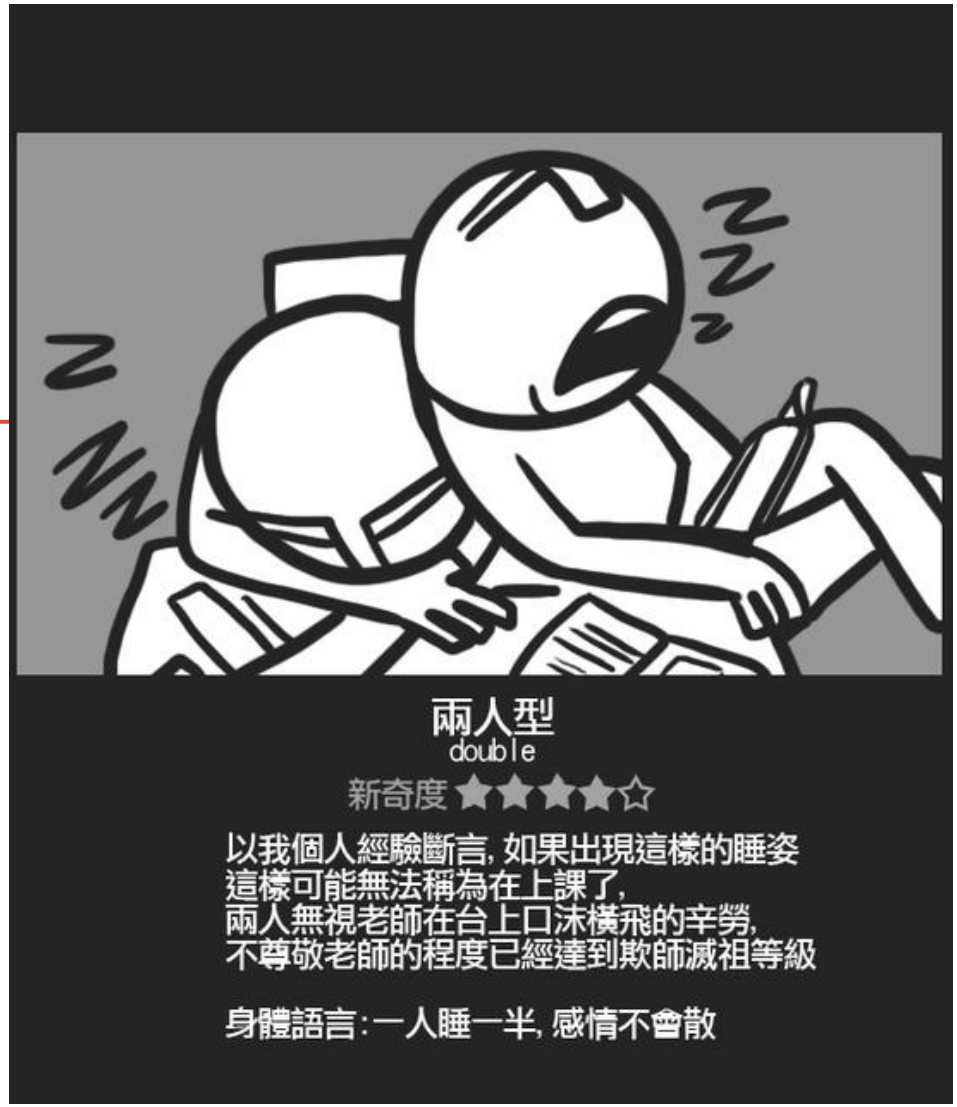
SORTING

Michael Tsai

2013/5/7



Sorting

- 定義:
- Input: $\langle a_1, a_2, \ldots, a_n \rangle$為n個數字的序列
- Output: $\langle a'_1, a'_2, \ldots, a'_n \rangle$為輸入之序列的重新排列，使得 $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

- 真實的情況中, a_i為一組(record)中的key (例如學號)
- record中除了key以外的資料稱為satellite data
- 如果satellite data很大， 我們通常在排序的時候只排列指到該record的pointer。

Sorting有什麼用?

- 例子一: 在一個list裡面找東西.

- 如果沒有sort過, 要怎麼找?
- 答: 只能苦工, 從頭找到尾 →$O(n)$

- 那如果sort過呢?
- 繼續苦工的話有幫助嗎?
- 有. 可以提早知道要找的數字不在裡面.
- 也可以binary search →$O(\log n)$

- 但是, sorting本身要花多少時間呢...

Sorting有什麼用?

- 例子二: 比對兩個list有沒有一樣 (列出所有不一樣的item). 兩個lists分別有n與m個items.

- 如果沒有sort要怎麼找呢?
- list 1的第1個, 比對list 2的1-m個
- list 1的第2個, 比對list 2的1-m個
- ...
- list 1的第n個, 比對list 2的1-m個
- 所以需要$O(nm)$

- 如果sort過呢?
- $O(n+m)$
- 不要忘了還有sorting的時間
- 所以sorting究竟要花多少時間呢?

Sorting的分類

- Internal Sort: 所有的資料全部可以一股腦地放在記憶體裡面
- External Sort: 資料太大了, 有些要放到別的地方 (硬碟, 記憶卡, 網路上的其他電腦上, 等等等)
- 我們只講internal sort的部分
- →現在電腦(及如手機, iPod, iPad等各種計算裝置)記憶體越來越大, 越來越便宜, 比較少有機會使用external sort.

Sorting 相關名詞

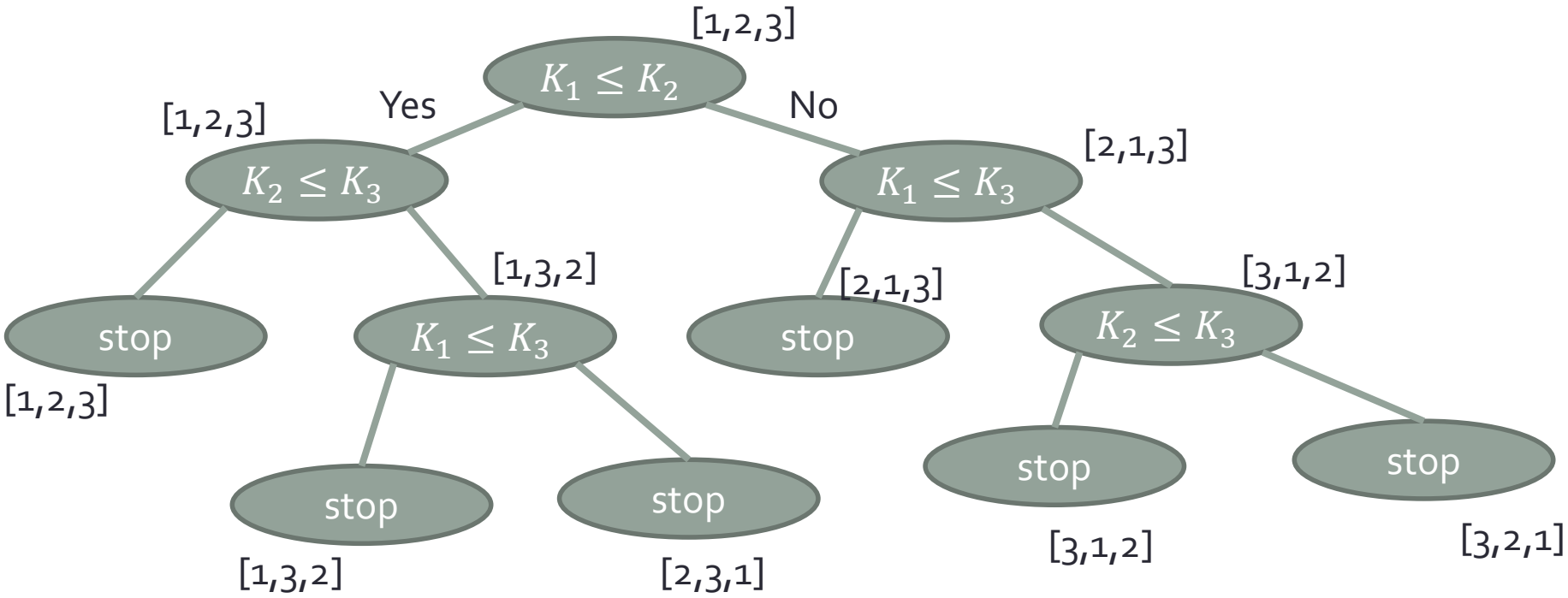
- Stability: 如果$a_i = a_j$ (一樣大的key)，那麼它們在sort前的順序和sort之後的順序是一樣的。

- In-place: 直接在原本儲存這些key的記憶體位置做sort。因此只需要額外O(1)的記憶體大小來幫助sorting。

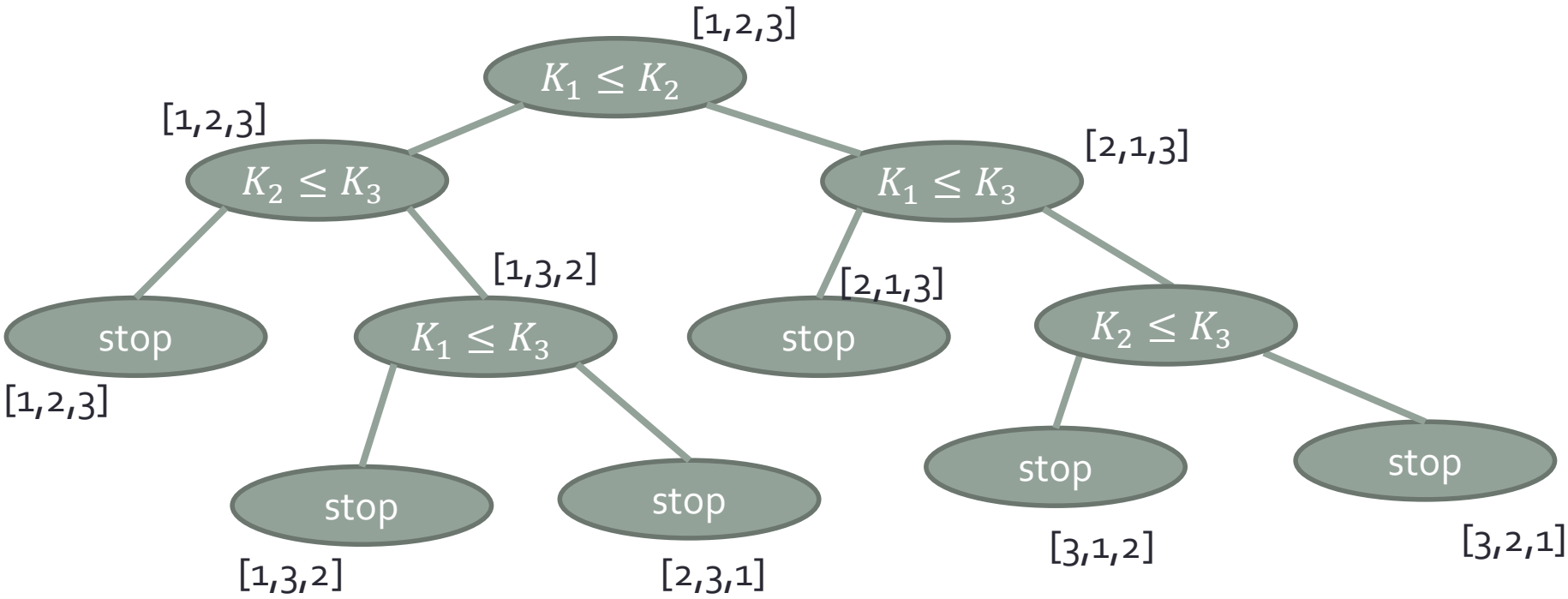
- Adaptability: 如果數列已經有部分排過序了，則sorting的time complexity可因此而降低。

到底我們可以sort多快?

- 假設我們使用"比較"+"交換"的方法.
- "比較": 看list裡面的兩個item誰大誰小
- "交換": 交換/移動兩個item在list裡面的位置
- 如何歸納出"最差的狀況要花多少時間sort?"



Decision tree for sorting



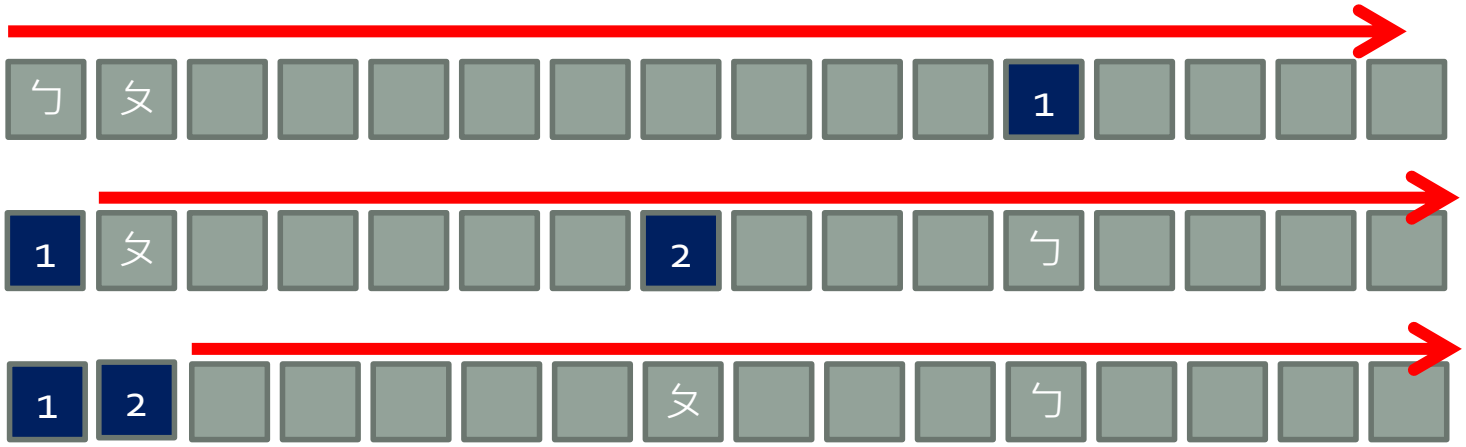
- 每個node代表一個comparison 及swap
- 到達leaf時表示sorting完畢
- 共有幾個leaf?
- 共有n個items的所有可能排列數目: $n!$

到底我們可以sort多快?

- 所以, worst case所需要花的時間, 為此binary tree的height.
- 如果decision tree height為h, 有l個leaves
- $l \geq n!$, 因為至少要有n!個leaves
- $l \leq 2^h$, 高度為h的binary tree (decision tree)最多有2^{h-1}個leaf
- $2^h \geq l \geq n!$
- $h \geq \log_2 n!$
- $n! = n(n-1)(n-2) \ldots 3 \cdot 2 \cdot 1 \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$
- $\log_2 n! \geq \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)$
- 結論: 任何以"比較"為基礎的sorting algorithm worst-case complexity為$\Omega(n \log n)$.

複習: Selection Sort

- 第一次選最小的, 移到最前面
- 第二次選第二小的,移到第二前面
- ....
- 直到剩一個(最大的), 會放在最後面



複習: Selection Sort

- 因為selection sort並不因為目前的狀況而改變演算法執行的步驟 (總是從頭看到尾)
- 所以best-case, worst-case, average-case都是$O(n^2)$
- (Not adaptive)

- In-place

Insertion Sort

- 方法: 每次把一個item加到已經排好的, 已經有i個item的list, 變成有i+1個item的排好的list

Insertion Sort

- 要花多少時間?
- 答: 最差的狀況, 每次都要插到最末端 (花目前sorted大小的時間)
- $\sum_1^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
- 平均complexity, 也是$O(n^2)$. (Why?)

- 變形(蟲):
- 1. 在找正確的應該插入的地方的時候, 使用binary search. (但是移動還是$O(n)$)
- 2. 使用linked list來表示 整個list的item, 則插入時不需要移動, 為$O(1)$. (但是尋找插入的地方的時候還是需要$O(n)$)

Insertion Sort的好性質

- 簡單 (time complexity中的constant小)
  - 當需要sort的東西很少的時候, 常常被拿來使用
- Stable
- In-place
- Adaptive: 當有部分排好的時候會比較快
  - 舉例: ⟨1,2,5,3,4⟩中, 只有<5,3>, <5,4>兩組數字反了(inversion)
  - 使用insertion sort的時間為O(n+d), d為inversion數目 (上例中為2)
  - (根據以上) Best case: O(n) (沒有inversion, 已經排好了)
- Online:
  不需要一開始就知道所有需要被排序的數字, 可以一面收入一面排序

Merge Sort

- 使用Divide-and-Conquer的策略
- Divide-and-Conquer:
  - Divide: 把大問題切成小問題
  - Conquer: 解決小問題
  - Combine: 把小問題的解答合起來變成大問題的解答

- Merge sort:
  - Divide: 把n個數字切成兩個數列(各n/2個數字)
  - Conquer: 將兩個子數列各自排序(事實上是用recursive call交給下面處理)
  - Combine: 將return回來的兩個各自排好的子數列合併起來變成大的數列

Merge Sort

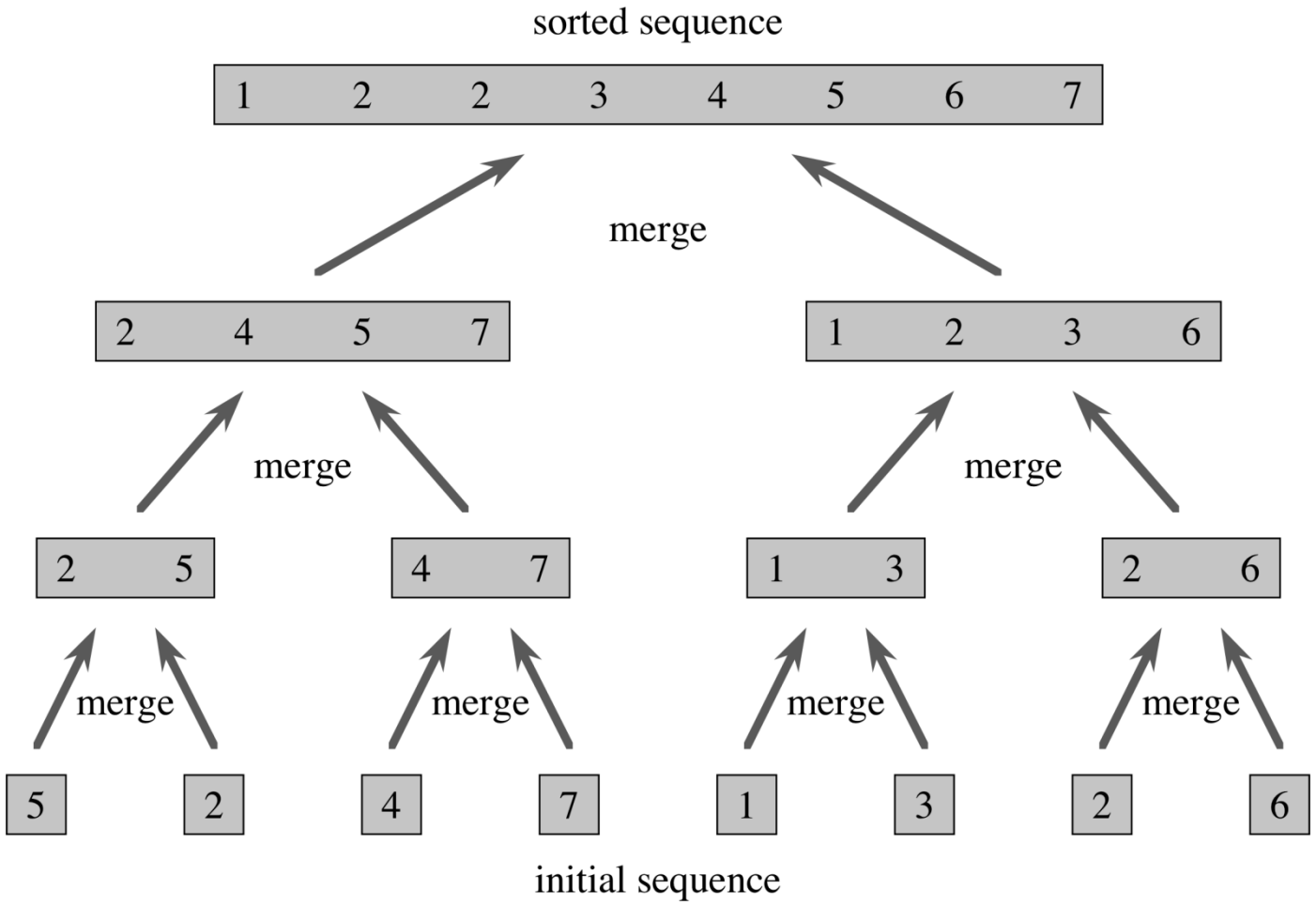
```
void Mergesort(int A[], int temp, int left, int right) {
        int mid;
        if (right > left) {
                mid=(right+left)/2;
                Mergesort(A,temp,left,mid);
                Mergesort(A,temp,mid+1,right);
                Merge(A,temp,left,mid+1,right);
        }
}
```

Divide

Conquer

Combine

Merge Sort: Example



How to combine? (Karumanchi p251)

原本的位置

暫時的儲存

- 要怎麼把兩個已經排好的list merge成一個?
- 所花時間: $O(n+m)$, n和m為兩個要合併的list長度
- Merge完需要再從暫時的儲存把資料搬回原本的input array

Merge sort

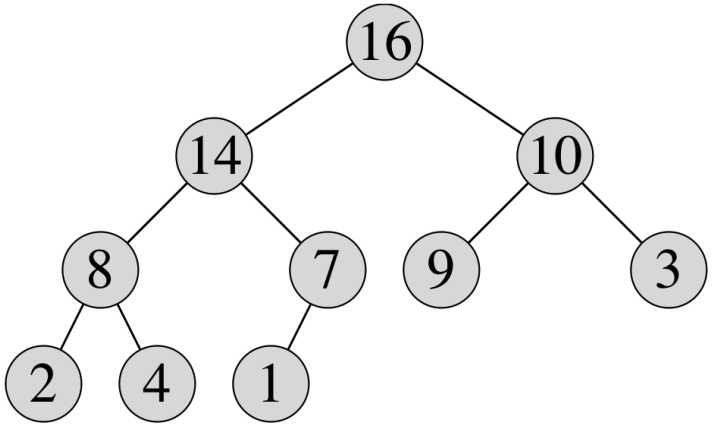
- 每個"pass", 正好每個要排序的數字都process了一次. 所以為$O(n)$
- 總共需要多少"pass"呢?
- 每次一個子序列的長度會變兩倍, 最後變成一個大序列有n個數
- 所以需要$\lceil \log_2 n \rceil$passes.
- 總共所需時間為$O(n \log_2 n) = O(n \log n)$
- Worst-case, best-case, average-case時間都一樣: $O(n \log n)$

- 需要額外的空間來sorting: 用來儲存merge好的結果
- 額外空間: O(n)

複習：Heapsort

- 如何利用heap排序?

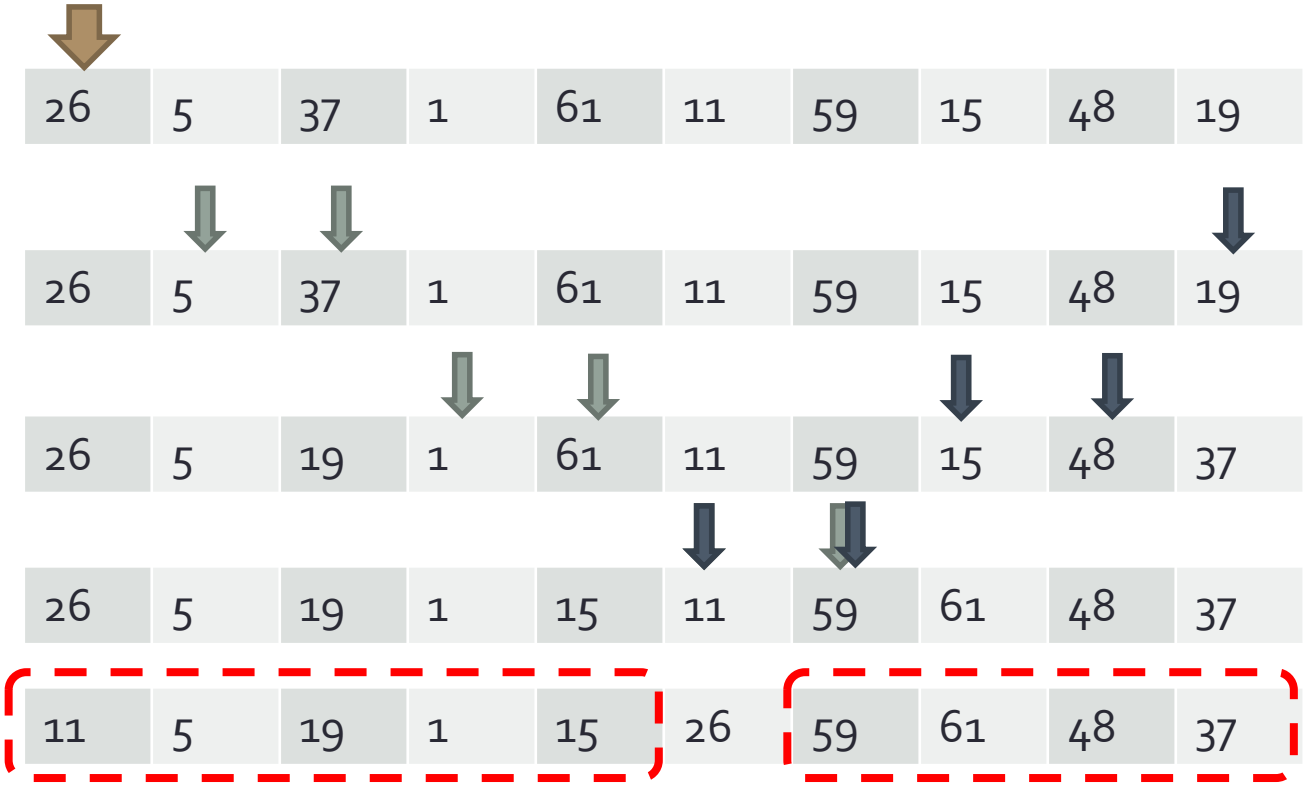
1. 先用heapify方法把整個array變成heap. $O(n)$
2. 每次從heap拿出一個最大的, (和尾巴交換), 然後把原本尾巴的element依序往下檢查/交換直到符合heap性質.
3. 重複步驟2一直到heap變成空的. $O(n \log n)$



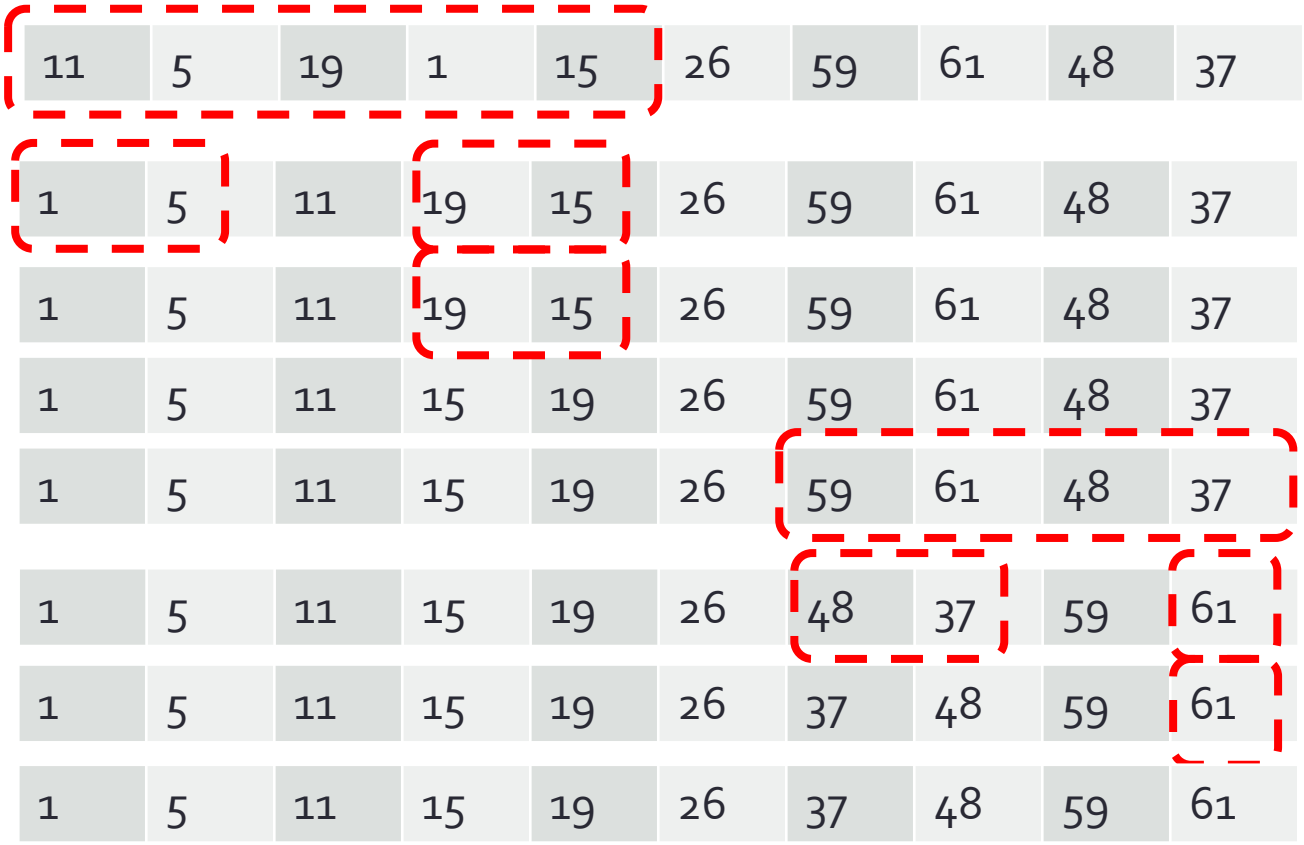
Total: $O(n \log n)$

Quick Sort

- 方法: 每次找出一個pivot(支點), 所有它左邊都比它小(但是沒有sort好), 所有它右邊都比它大, 然後再call自己去把pivot左邊與pivot右邊排好.



Quick Sort


11 5 19 1 15 26 59 61 48 37
1 5 11 19 15 26 59 61 48 37
1 5 11 19 15 26 59 61 48 37
1 5 11 15 19 26 59 61 48 37
1 5 11 15 19 26 59 61 48 37
1 5 11 15 19 26 48 37 59 61
1 5 11 15 19 26 37 48 59 61
1 5 11 15 19 26 37 48 59 61

Quick Sort: Worst & Best case

- 但是Worst case時所需時間還是$O(n^2)$
- 想想看, 什麼時候會變成這樣呢?
- 答: 每次pivot都是最大的. (or 每次都是最小的) →已經排好了!
- 此時居然執行時間為 $O(n^2)$

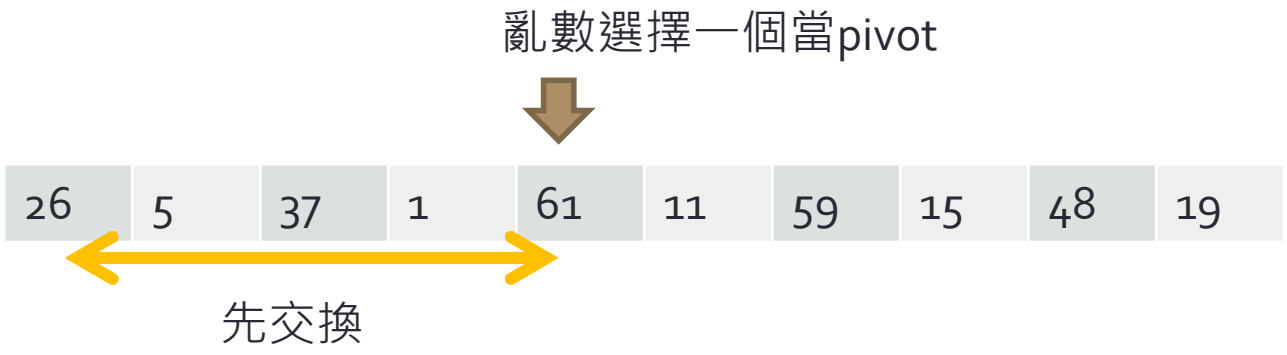
- Best case?
- 每次pivot正好都可以把數列很平均地分成兩半
- 因此T(n)=2T(n/2)+$O(n)$
- T(n)=$O(n \log n)$

Randomized Quick Sort

- 避免常常碰到worst case (已經排好的狀況)

- 選擇pivot的時候, 不要固定選最左邊的
- 亂數選擇需要分類的範圍中其中一個作為pivot

- 這樣減少碰到worst的機率 (但碰到worst case還是$O(n^2)$)



Average running time

- 越能選到pivot可以平均的分配成兩個subset越好

- 以下我們將說明為什麼average running time會接近best-case
- 假設很糟的一個狀況: 每次都分成1:9

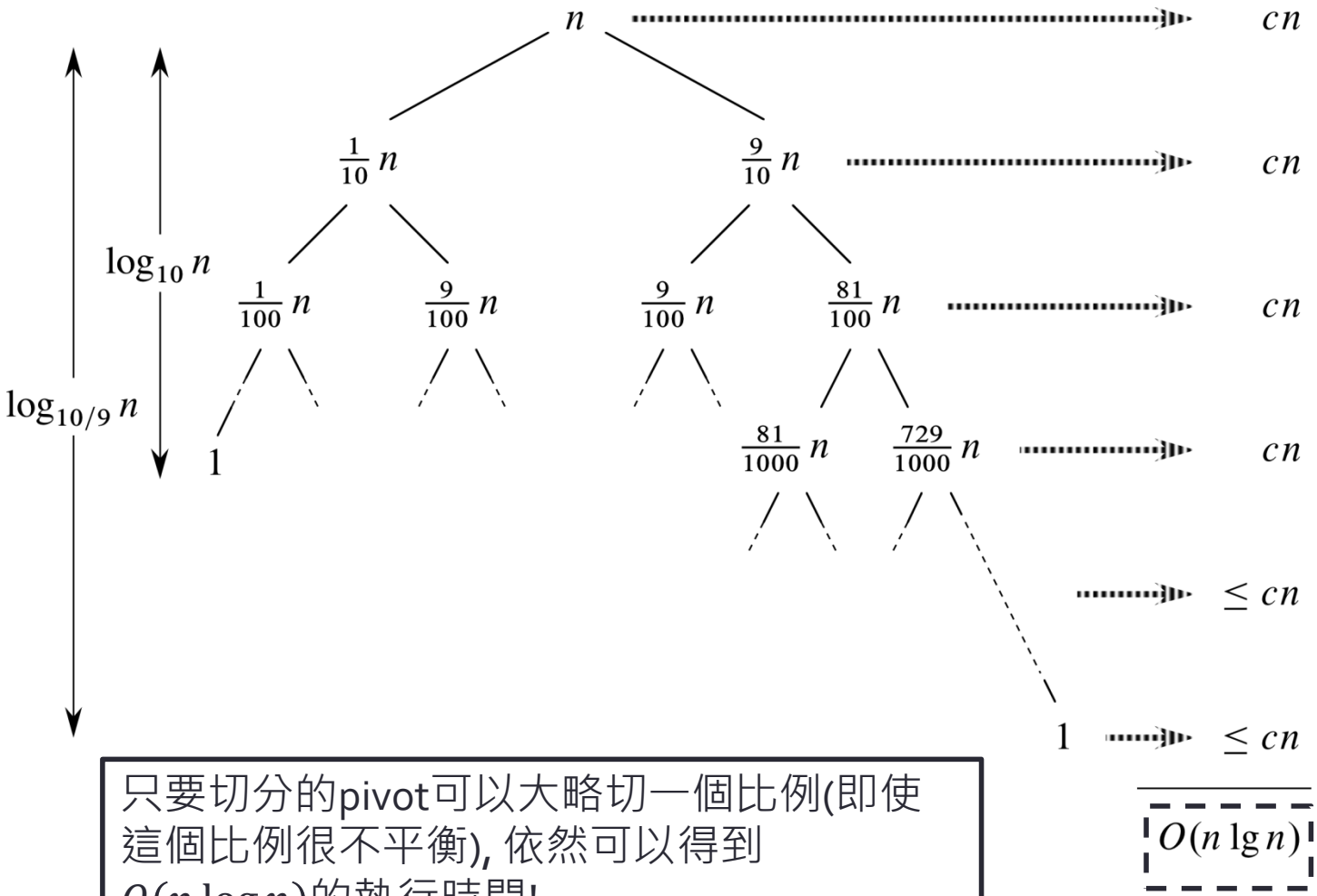
分成9/10的那一份所需花的時間　　分成比pivot大或小所花的時間

- $T(n) = T(9n/10) + T(n/10) + cn$

分成1/10的那一份所需花的時間

- $= \left(T\left(\frac{81n}{100}\right) + T\left(\frac{9n}{100}\right) + \frac{9cn}{10}\right) + \left(T\left(\frac{9n}{100}\right) + T\left(\frac{1n}{100}\right) + \frac{cn}{10}\right) + cn$
- $= \cdots$

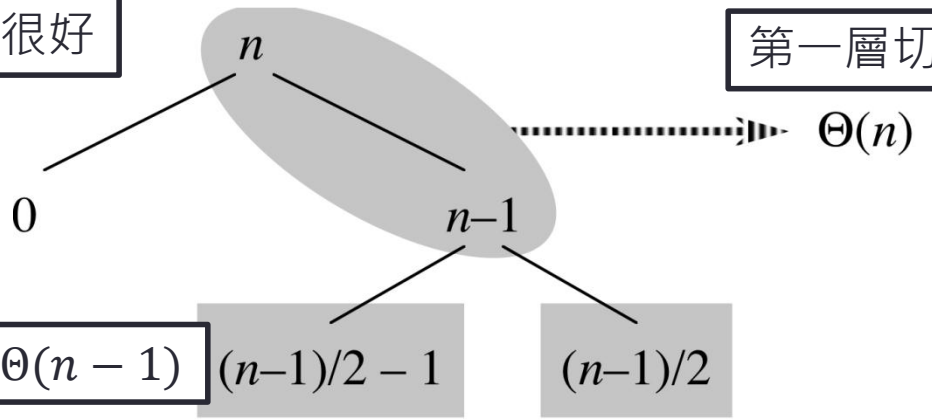
Average running time



只要切分的pivot可以大略切一個比例(即使這個比例很不平衡), 依然可以得到 $O(n \log n)$的執行時間!

Average running time

第一層分得很差
但是下一層分得很好



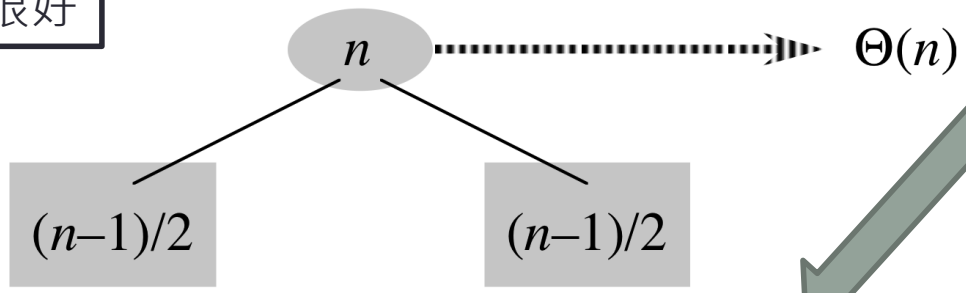
第一層切分花了$\Theta(n)$

$\Theta(n)$

第二層切分花了$\Theta(n-1)$

$\Theta(n)+\Theta(n-1)=\Theta(n)$

第一層就分得很好



$\Theta(n)$

第一層切分花了$\Theta(n)$

居然一樣!
(只是分得比較差的constant比較大而已)
分得比較好的層會"吸收"分得比較差的層造成的額外時間

比較四大金剛

	Worst	Average	Additional Space?
Insertion sort	$O(n^2)$	$O(n^2)$	O(1)
Heap sort	$O(n \log n)$	$O(n \log n)$	O(1)
Merge sort	$O(n \log n)$	$O(n \log n)$	O(n)
Quick sort	$O(n^2)$	$O(n \log n)$	O(1)

- Insertion sort: n小的時候非常快速. (因為常數c小)
- Quick sort: average performance最好 (constant也小)
- Merge sort: worst-case performance 最好
- Heap sort: worst-case performance不錯, 且不用花多的空間
- 可以combine insertion sort和其他sort
- 怎麼combine?
- 答: 看n的大小決定要用哪一種sorting algorithm.

Today's Reading Assignments

- Cormen p146-150 (preface of section II):
  - 了解排序演算法的背景
- Cormen 7.2-7.3
- Karumanchi 10.5-7, 10.8-10.11

- **Errata of Karumanchi Ch10**:
- P.245: Some sorting algorithms are "in place" and they need O(1) ~~or O(log n)~~ memory to create auxiliary locations for sorting the data temporarily.
- P.248: Performance of Selection Sort:
  - Best case complexity should be $O(n^2)$, not $O(n)$
- P.249: Performance of Insertion Sort:
  - Best case complexity should be $O(n)$, not $O(n^2)$
  - Worst case space complexity should be $O(n)$ total, not $O(n^2)$ total.