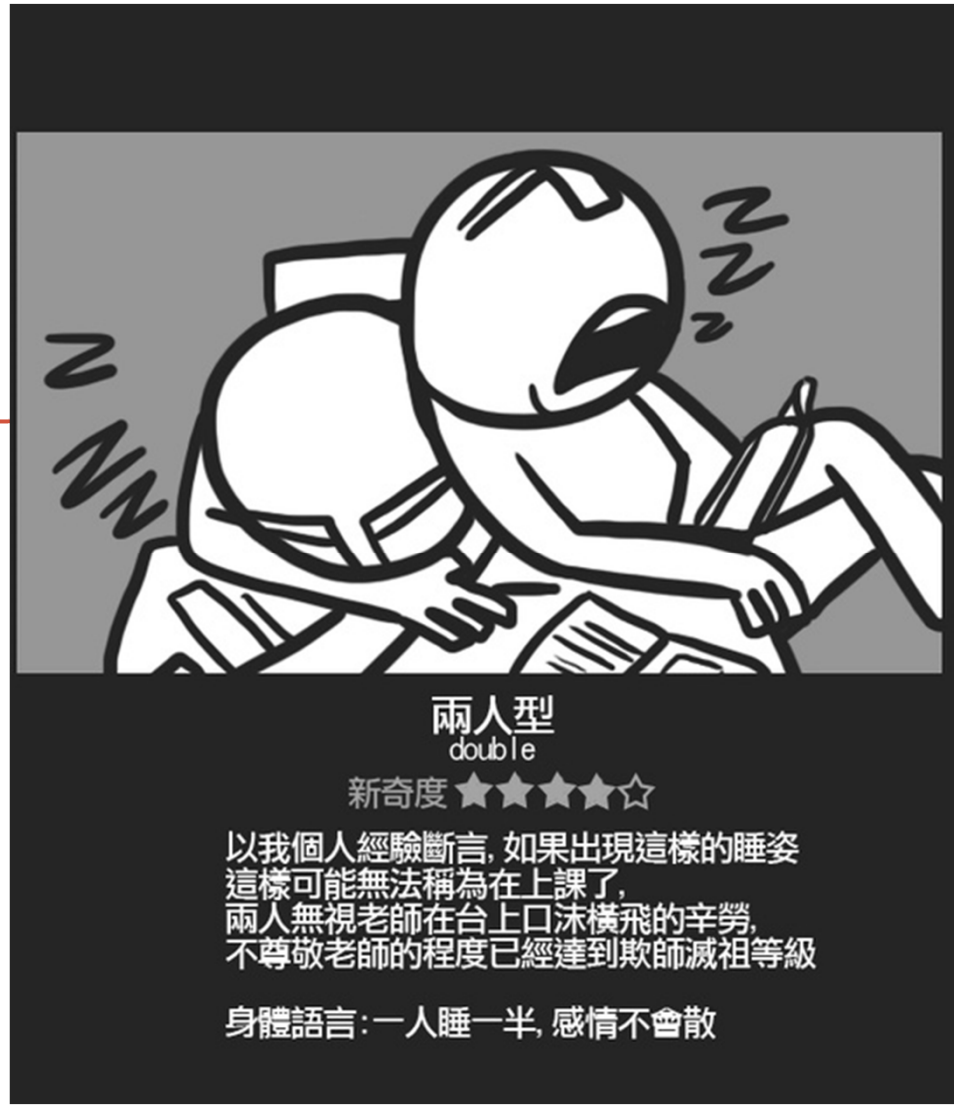
SORTING

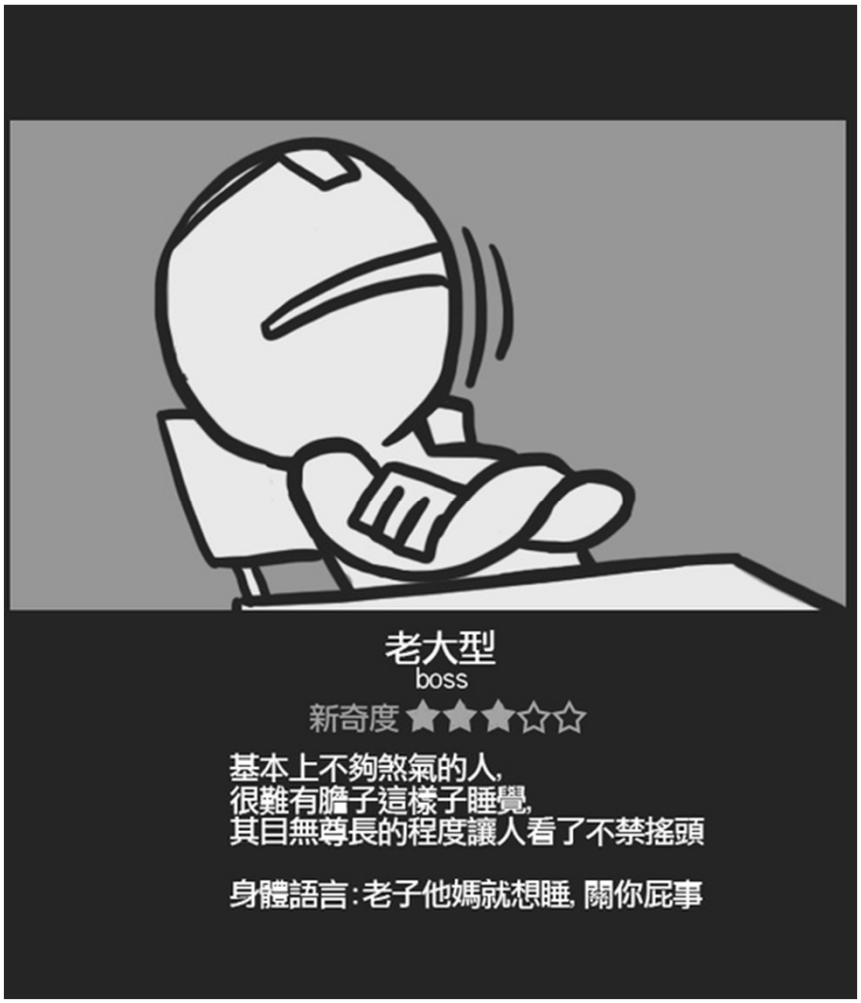
Michael Tsai

2010/12/10



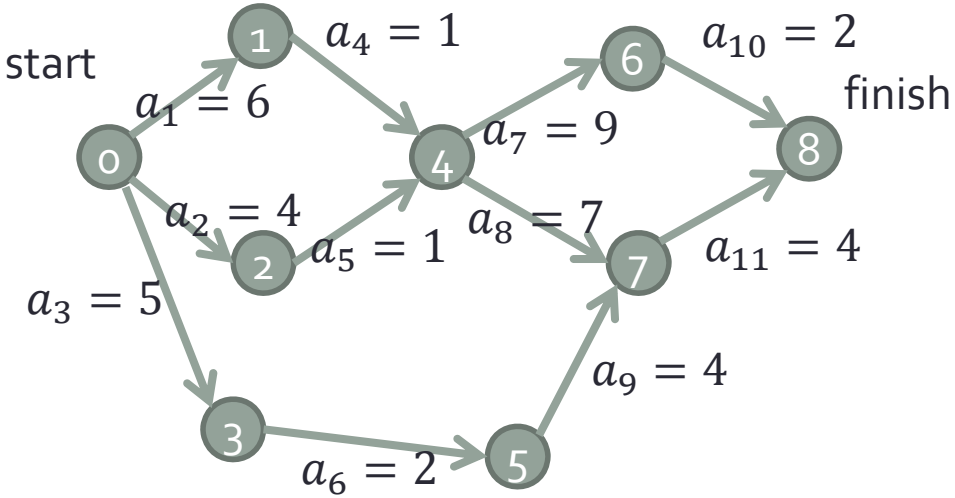
今日菜單

- 上次還沒講完的AOE網路
- Sorting
- 定義
- 最快可以sort多快?
- Insertion Sort
- Quick Sort
- Merge Sort
- Heap Sort



Activity-on-edge (AOE) Network

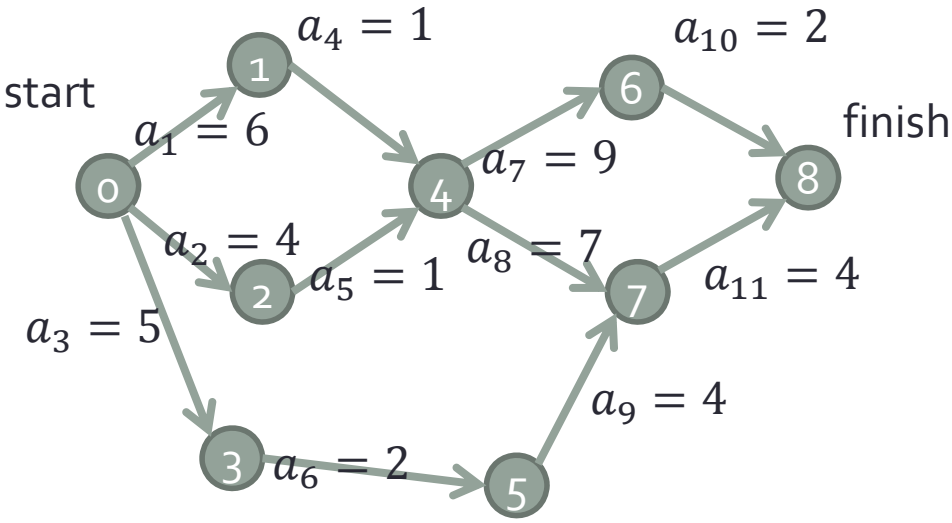
- 不是Ages of Empires (爛梗)
- "Activities"發生在edge上
- edge cost=所需要完成activity的時間
- 例如: vertex 4可以解釋為完成a_4及a_5後的時間



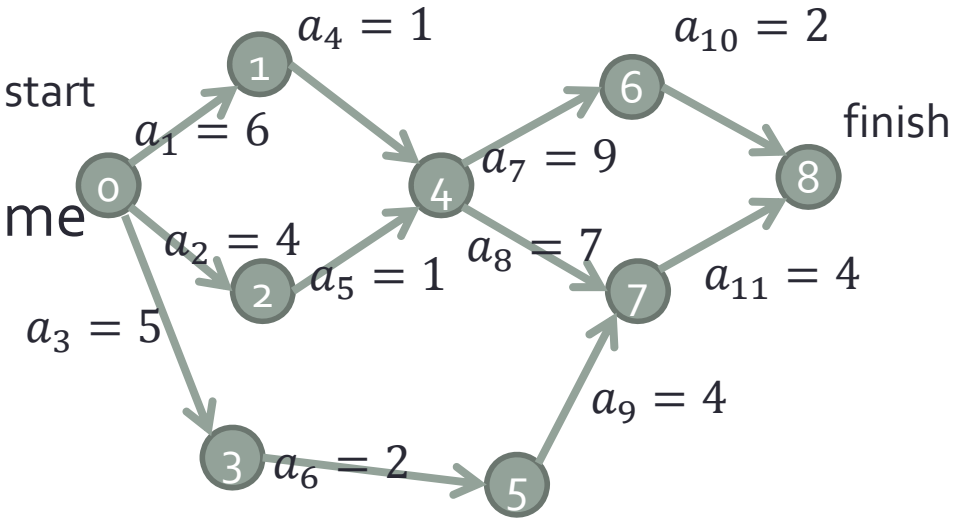
可以思考的一些問題

- vertex 8=完成所有工作. 完成的時間為?
- 完成的時間為從start →finish的**最長**path.
- 又稱為critical path
- 想想看為什麼?

- earliest time(edge e): 為該activity最早可以開始的時間
- latest time(edge e):
  不延誤最後完成時間的前提下
  activity可以開始的最晚時間
- earliest time(e) == latest time(e)
- then e為critical activity
- (完全不能延誤)



AOE Network

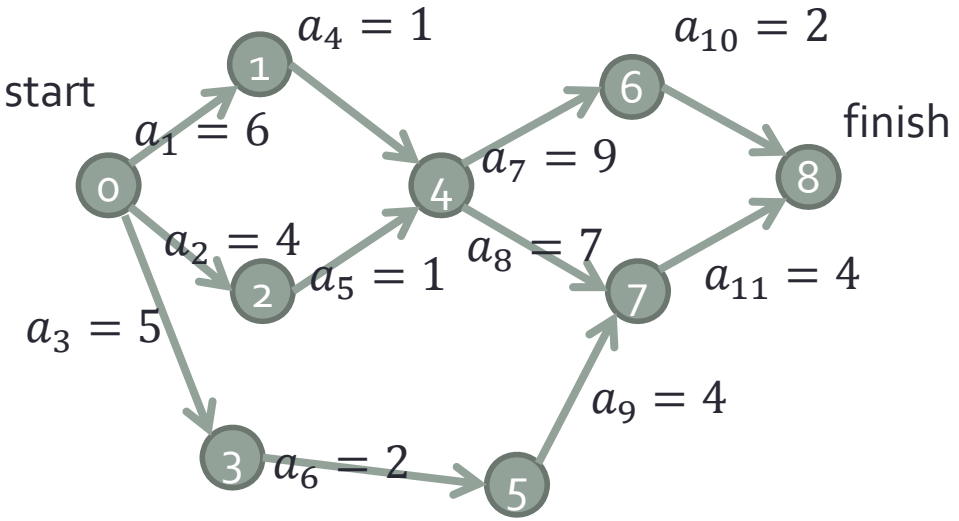


ee[i]=i-th event's earliest event time

a_i =<k,l> then
e(i)=ee[k]

ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	stack
Initial	0	0	0	0	0	0	0	0	0	[0]
0	0	6	4	5	0	0	0	0	0	[3 2 1]
3	0	6	4	5	0	7	0	0	0	[5 2 1]
5	0	6	4	5	0	7	0	11	0	[2 1]
2	0	6	4	5	5	7	0	11	0	[1]
1	0	6	4	5	7	7	0	11	0	[4]

AOE Network

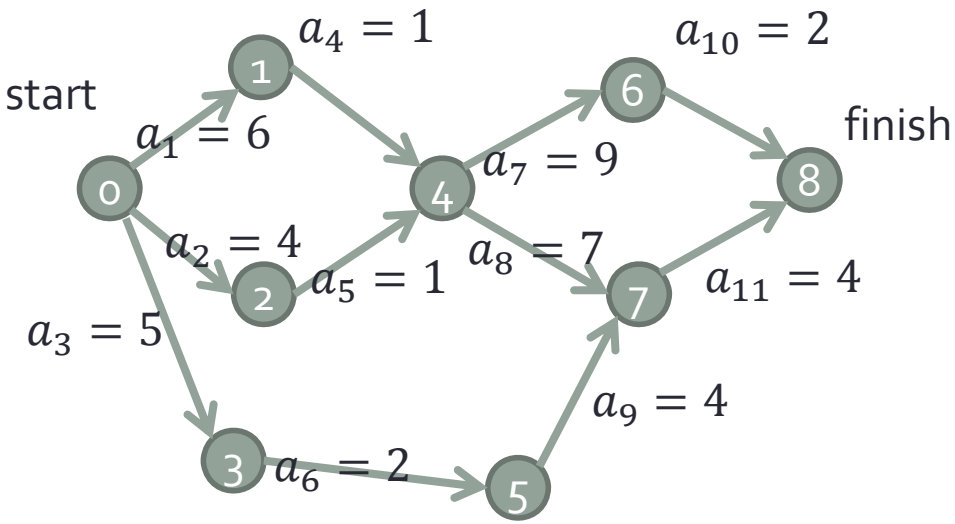


ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	stack
1	0	6	4	5	7	7	0	11	0	[4]
4	0	6	4	5	7	7	16	14	0	[7 6]
7	0	6	4	5	7	7	16	14	18	[6]
6	0	6	4	5	7	7	16	14	18	[8]

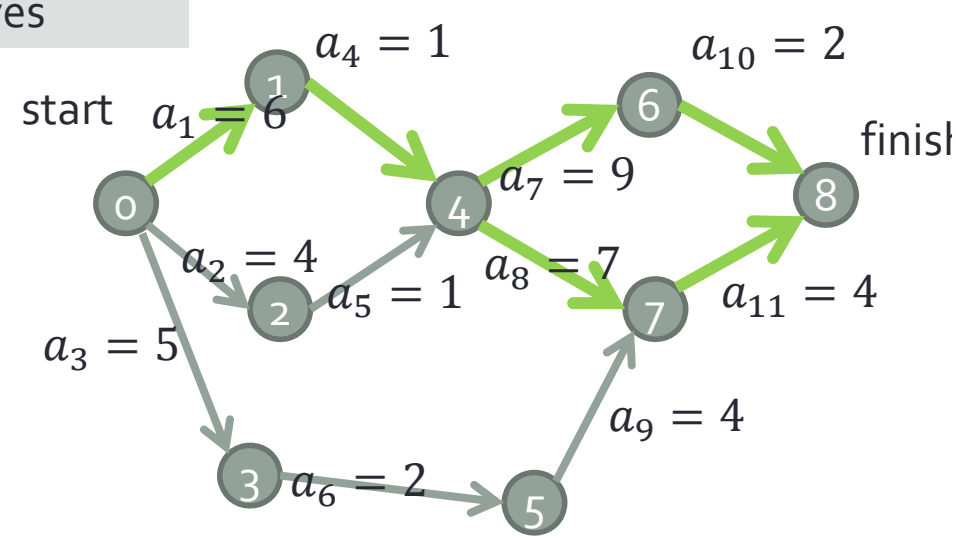
接下來怎麼算e(i)?

AOE Network

- $le[j] = \min_i\{le[i] - cost(< j, i >\}$
- 用reverse topological order來依序算出
- 請同學來算算看☺



Activity	e	l	l-e (slack)	l-e=o
a_1	0	0	0	Yes
a_2	0	2	2	
a_3	0	3	3	
a_4	6	6	0	Yes
a_5	4	6	2	
a_6	5	8	3	
a_7	7	7	0	Yes
a_8	7	7	0	Yes
a_9	7	10	3	
a_{10}	16	16	0	Yes
a_{11}	14	14	0	yes



Sorting

- 定義:
- 有$R_1, R_2, R_3, \ldots, R_n$等n個record
- 每個record有個key.
- $K_1, K_2, K_3, \ldots, K_n$

- 我們會定義對key們定義>, =, <
- 這些operations是transitive, 就是說 a>b, b>c的話, a>c
- 要找到一個$R_1, R_2, R_3, \ldots, R_n$的排列$\sigma_S$使得
- 1. $K_{\sigma_S(i)} \leq K_{\sigma_S(i+1)}, 1 \leq i \leq n-1$
- 2. if i<j and $K_i == K_j$,then R_i precedes R_j
- 如果同時兩個條件成立則此sorting方法為 **stable**.

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$O(nm)$

Sorting有什麼用?

- 如果sort過呢?

- 可以利用類似之前講過polynomial加法的方法.
- (請同學回憶並解釋?)
- 可參考課本program 7.3 (p.337)

- 則需要花多少時間來比對?
- $O(n+m)$
- 不要忘了還有sorting的時間
- 所以sorting究竟要花多少時間呢?

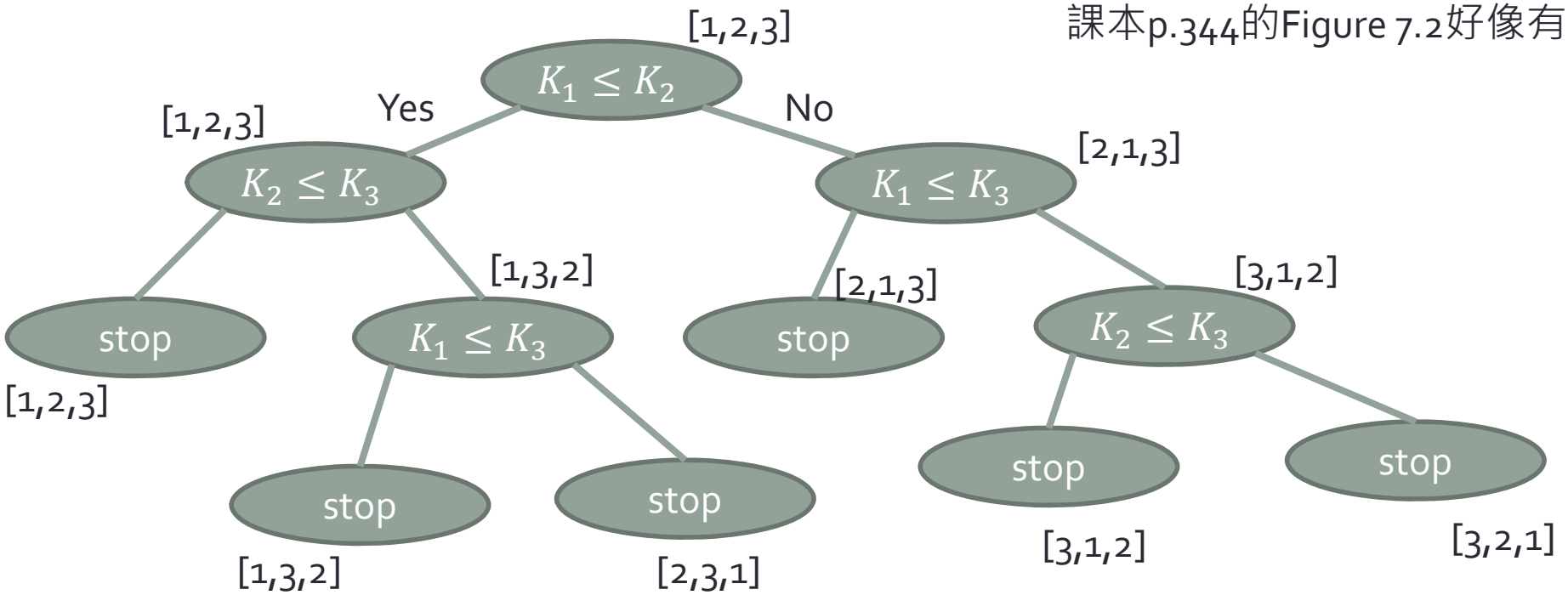
Sorting的分類

- 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.
- 有興趣的同學可以看一看課本7.10 (p.376)

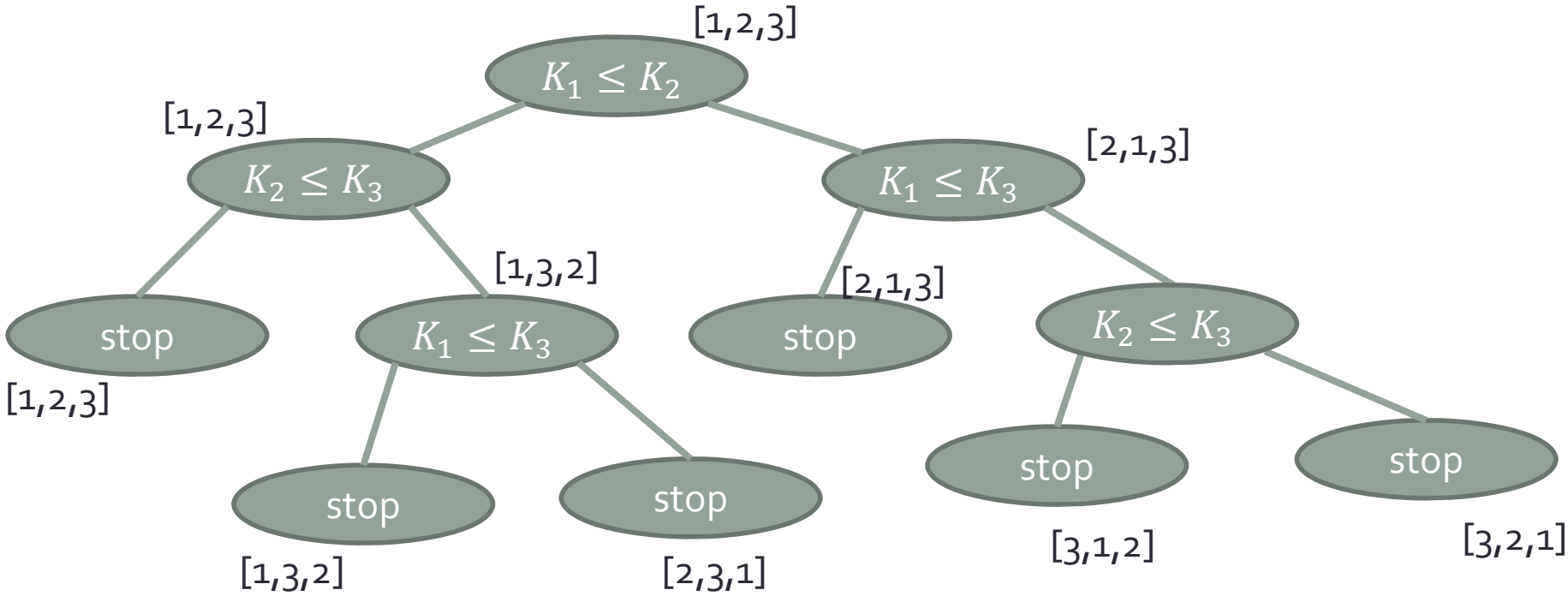
到底我們可以sort多快?

- 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?"

課本p.344的Figure 7.2好像有錯



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: $n!$

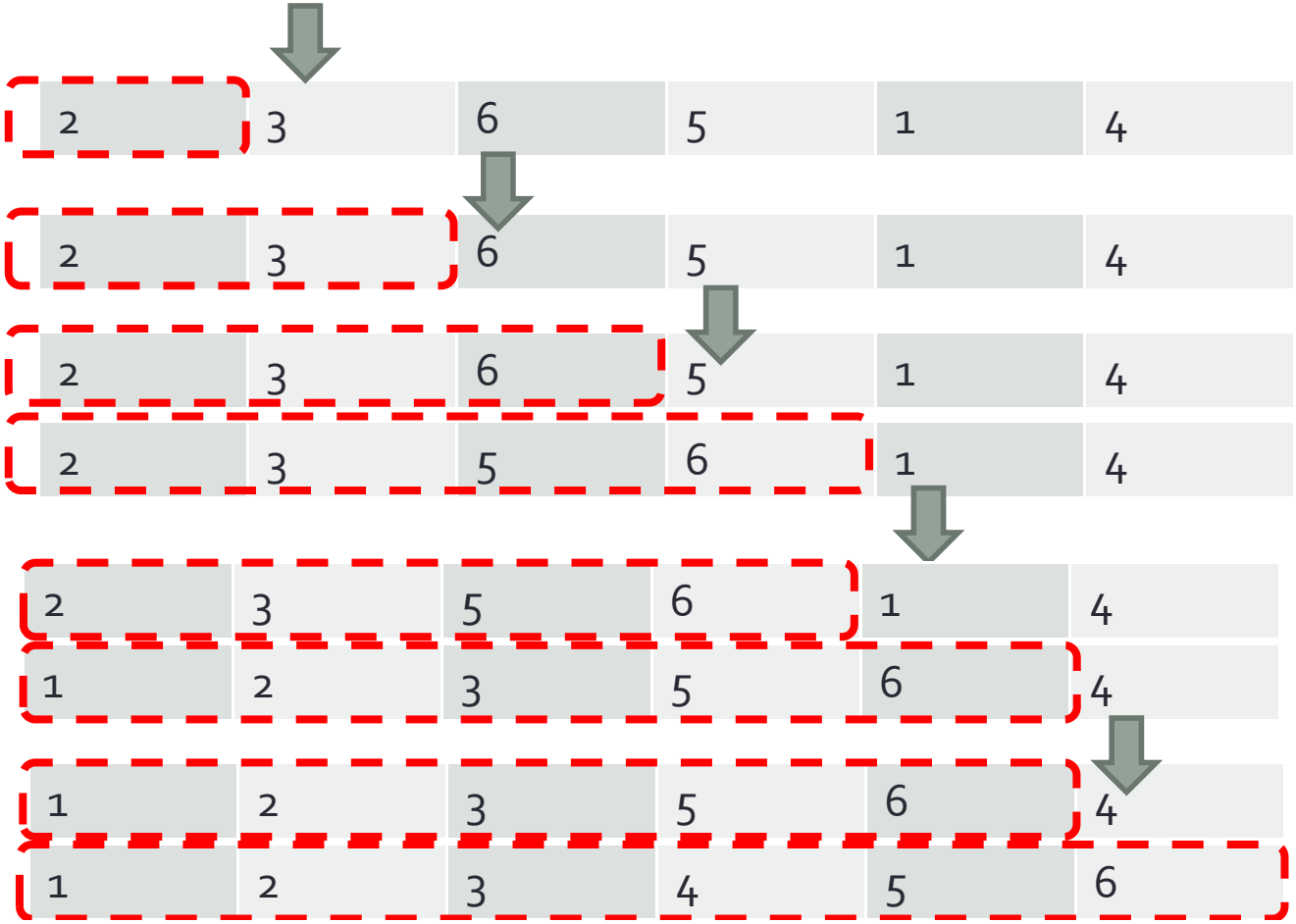
到底我們可以sort多快?

- 所以, worst case所需要花的時間, 為此binary tree的height.
- Binary tree的性質: 如果height為k, 則最多有2^{k-1}個leaves
- 所以有$n!$個leaves的話, 至少height為$\log_2 n! + 1$

- $n! = n(n-1)(n-2)\ldots 3 \cdot 2 \cdot 1 \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$
- $\log_2 n! \geq \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\log_2 \frac{n}{2} = \Omega(n \log n)$
- 結論: 任何以"比較"為基礎的sorting algorithm worst-case complexity為$\Omega(n \log n)$.
- <動腦時間> 如何證明average complexity也是$\Omega(n \log n)$?
- 答:算算看leaf到root的distance的平均值.

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



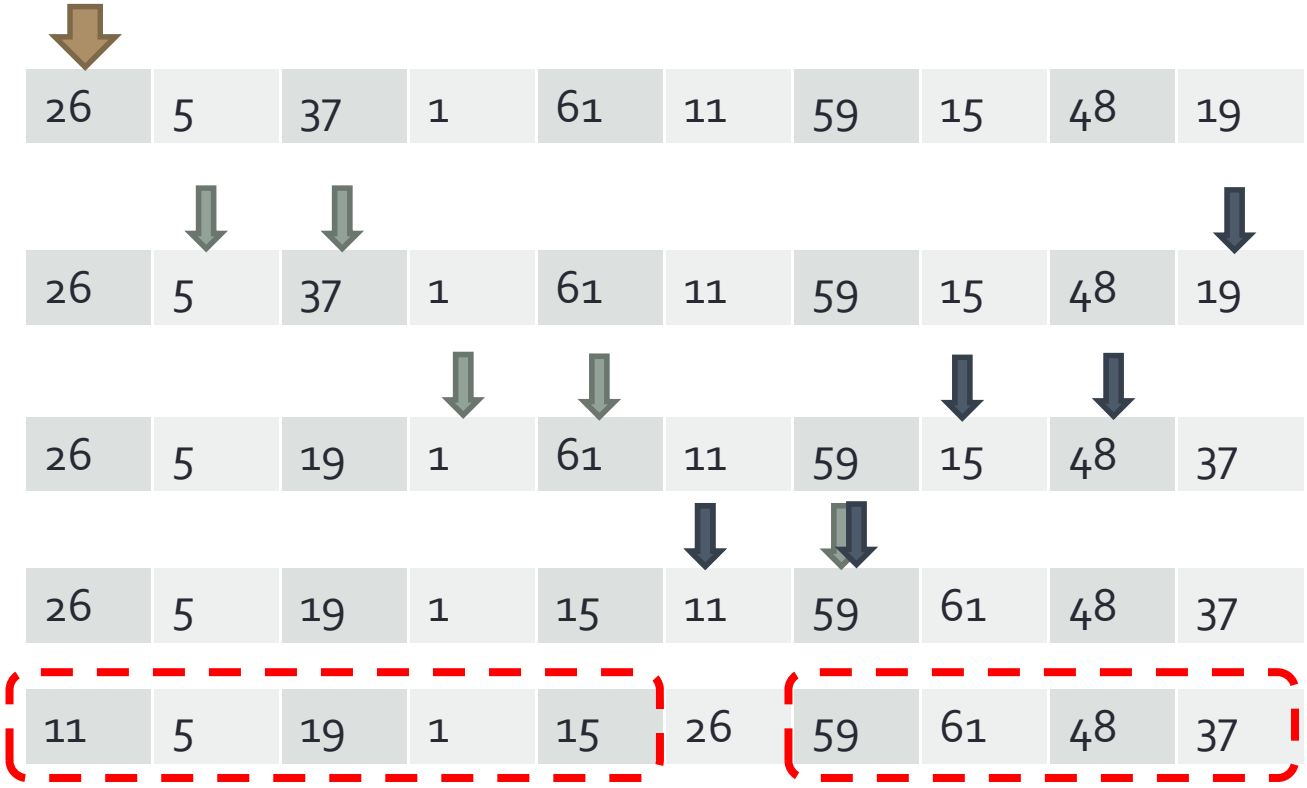
Insertion Sort

- 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)
- $\sum_2^n i = \frac{n(1+n)}{2} - 1 = O(n^2)$
- 平均complexity, 也是$O(n^2)$.

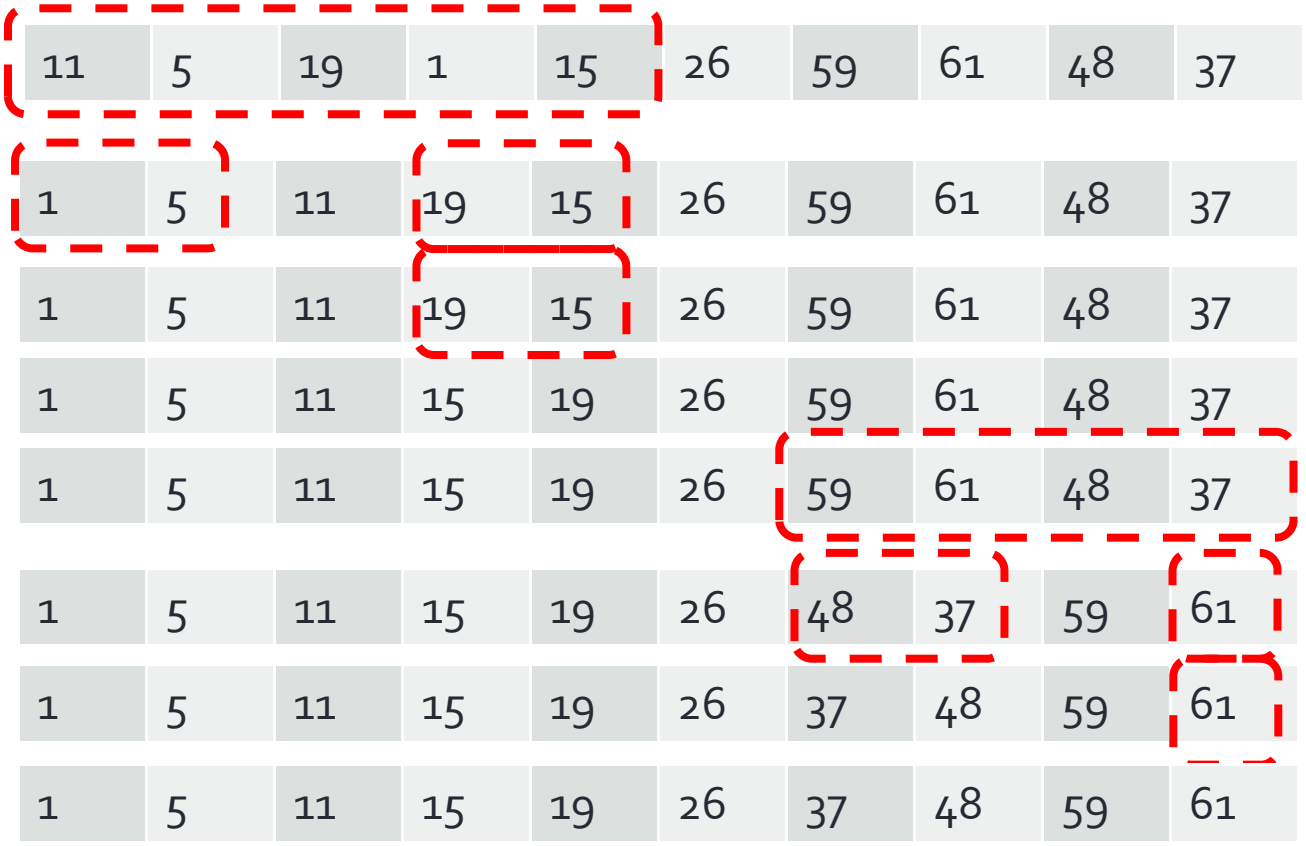
- 變形(蟲):
- 1. 在找正確的應該插入的地方的時候, 使用binary search. (但是移動還是$O(n)$)
- 2. 使用linked list來表示 整個list的item, 則插入時不需要移動, 為$O(1)$. (但是尋找插入的地方的時候還是需要$O(n)$)

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

先大概算一下

- $T(n) \leq \mathrm{cn} + 2\mathrm{T}\left(\frac{\mathrm{n}}{2}\right) \leq \mathrm{cn} + 2\left(\frac{\mathrm{cn}}{2} + 2\mathrm{T}\left(\frac{\mathrm{n}}{4}\right)\right)$
- $\leq cn\log_2 n + nT(1) = O(n\log n)$

- 大約是$O(n\log n)$

- 接下來要稍微嚴謹一點證明:
- $T_{avg}(n) \leq kn\log_e n\,, for\ n \geq 2$

有點長的證明

- 用歸納法
- 首先:
- $T_{avg}(n) \leq cn + \frac{1}{\mathrm{n}}\sum_{j=1}^{n}(T_{avg}(j-1) + T_{avg}(n-j))$
- $= cn + \frac{2}{n}\sum_{j=0}^{n-1} T_{avg}(j)$
- 假設$T_{avg}(0) \leq b$ and $T_{avg}(1) \leq b$.
- n=2時, $T_{avg}(2) \leq 2c + 2b \leq kn\log_e 2$
- 假設$1 \leq n < m$時, $T_{avg}(n) \leq kn\log_e n$

有點長的證明

- $T_{avg}(m) \leq \mathrm{cm} + \frac{4\mathrm{b}}{\mathrm{m}} + \frac{2}{\mathrm{m}} \sum_{2}^{m-1} T_{avg}(j)$
- $\leq \mathrm{cm} + \frac{4\mathrm{b}}{\mathrm{m}} + \frac{2\mathrm{k}}{\mathrm{m}} \sum_{2}^{m-1} j \log_e j$
- $\leq \mathrm{cm} + \frac{4\mathrm{b}}{\mathrm{m}} + \frac{2\mathrm{k}}{\mathrm{m}} \int_{2}^{m} x \log_e x \, dx$
- $= cm + \frac{4b}{m} + \frac{2k}{m} \left[\frac{m^2 \log_e m}{2} - \frac{m^2}{4} \right]$
- $= cm + \frac{4b}{m} + km \log_e m - \frac{km}{2}$
- $\leq km \log_e m$

Quick Sort的其他性質

- 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)

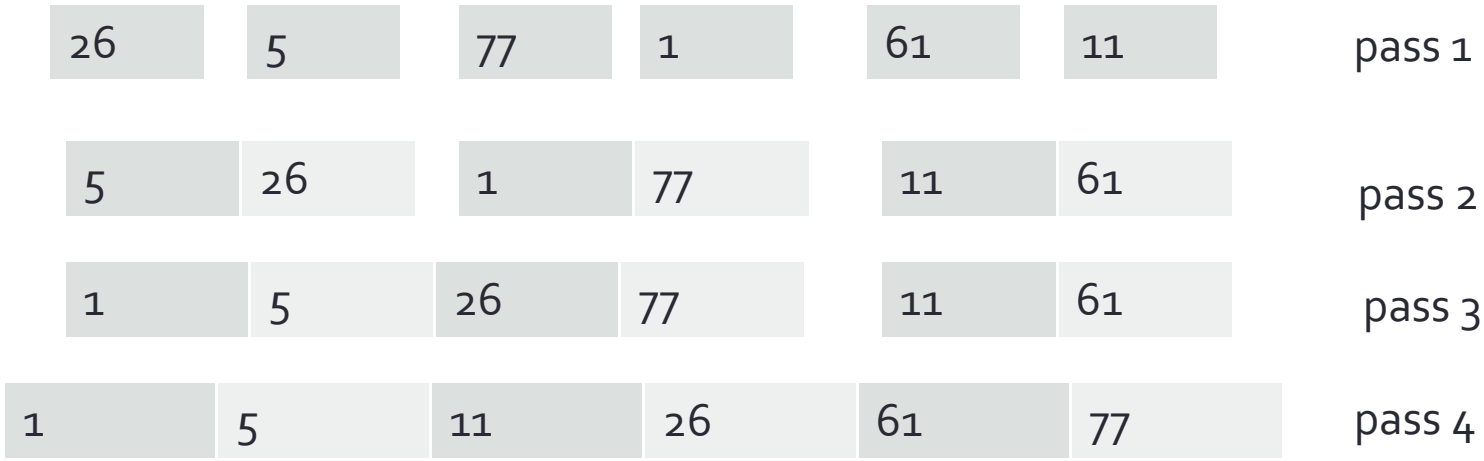
- 所需空間:
- 需要stack (處理recursive call用)
- stack需要多大呢?
- Worst case: 需要$O(n)$

Merge Sort

- 先看看: 要怎麼把兩個已經排好的list merge成一個?
- 請同學講講看☺ 也類似於之前polynomial加法
- 所花時間: $O(n+m)$

- Merge sort版本一
- 方法: 每個item一開始當作一個有一個item已經排好的list
- 然後每次把兩個merge成一個
- 一直到全部變成一個list為止

Merge sort


26 5 77 1 61 11
pass 1
5 26 1 77 11 61
pass 2
1 5 26 77 11 61
pass 3
1 5 11 26 61 77
pass 4

Merge sort

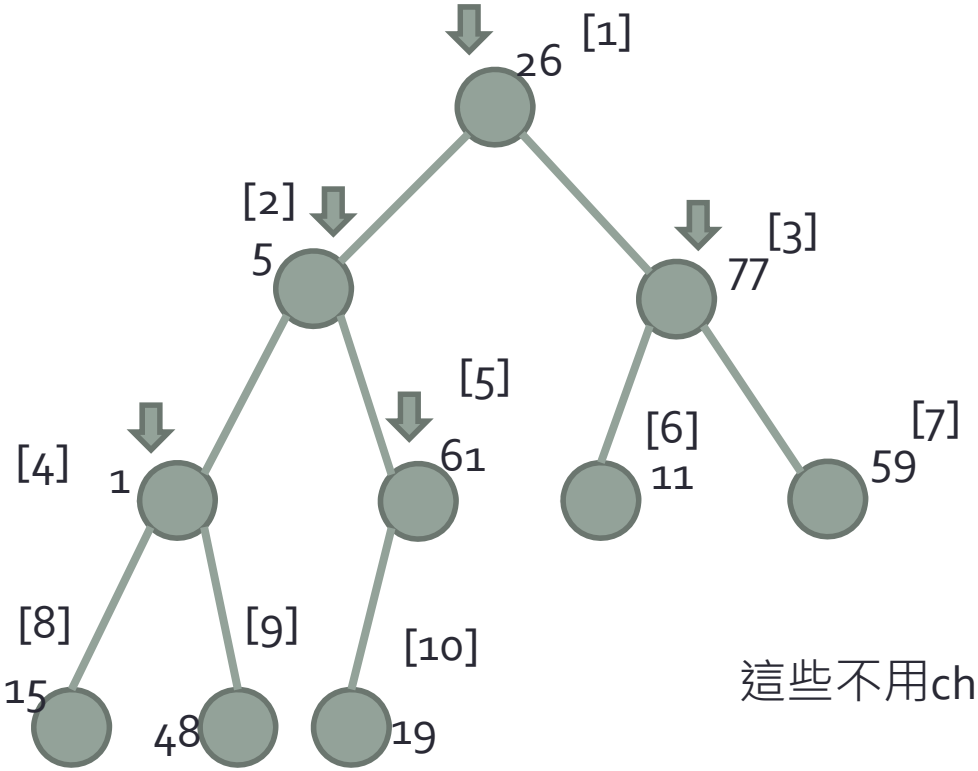
- 每個pass, 正好每個item都process了一次(worst case). 所以為 $O(n)$
- 總共需要多少pass呢?
- 每次數量減少一半
- 所以需要$\lceil \log_2 n \rceil$passes.
- 總共所需時間為$O(n \log_2 n)$

- 需要額外的空間來sorting
- 使用兩組跟n一樣大的空間, 交互存入merge過後的list
- 也可以用recursive的方法寫. 請參見program 7.10 (p. 350)

Heap Sort

- 方法: 用minimum heap 或 maximum heap來sort list. 使用所有item(的key)來建立heap. 之後每次從heap取出一個item, 放入sorted list. 則拿完以後就sorting完畢.

- 1. 稍微改變一下: 是否可以做到不用額外空間來建heap呢?
- 可以. array of items想是binary tree.



這些不用check, 因為沒有children

從n/2之前開始往回check,
只要檢查該node為root的binary tree是否為heap即可.

接下來只要每次把root跟最後一個node交換
重新整理維持heap即可.
當拿完以後就是完成sorting時.
(過程可參考課本figure 7.8 p 355)

Heap sort

- 兩個步驟:
- 1. 一開始建heap
- 2. 每次拿一個item出來

- 1. 開始建heap的時候
- 花的時間為
- $\sum_{1\le i\le k} 2^{i-1}(k-1) = \sum_{1\le i\le k} 2^{k-i-1}\, i \le n \sum_{1\le i\le k-1} \frac{i}{2^i}$
- $< 2n = O(n)$

第k層到leaf的distance

第k層有幾個node

- 2. 每次拿一個item出來, 花$\lceil \log_2(\mathrm{n}+1) \rceil = O(\log n)$
- 總共n-1次, 共$O(n \log n)$
- 只需要少數額外空間.

比較四大金剛

	Worst	Average
Insertion sort	n^2	n^2
Heap sort	$n \log n$	$n \log n$
Merge sort	$n \log n$	$n \log n$
Quick sort	n^2	$n \log n$

- Insertion sort: n小的時候非常快速. (因為常數c小)
- Quick sort: average performance最好
- 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.

Sorting on several keys

- 假設K有很多個sub-key
- $K = (K_1, K_2, \ldots, K_r)$

Most significant key　　Least significant key

- 則$K_x \leq K_y$ iff
- $K_{x,i} = K_{y,i}, 1 \leq i \leq j \; and \; K_{x,j+1} < K_{y,j+1}$ for some $j < r$, or
- $K_{x,i} = K_{y,i}, 1 \leq i \leq r$

- 則我們可以有以下的sorting 方法.
- 先依照most significant key sort, 然後依序往least significant key sort過去: Most Significant Digit first (MSD) sorting
- 先依照least significant key sort, 然後依序往most significant key sort過去: Least Significant Digit first (LSD) sorting

Sorting on several keys

- 哪一種比較好?

- 如果使用stable sorting algorithm, 則LSD sort比較單純
- sort成"一桶一桶"以後不需要分開sort
- 用黑板舉例吧~~ (投影片做到發瘋了)

Radix Sort

- 方法: 每次用某種方法把items分為r堆. 然後再concatenate起來.

- 怎麼分為r堆?
- 舉例: 每次使用某一位數的數字分堆.
- 第一次依照個位數分為十堆, 然後再放在一起.
- 第二次依照十位數分為十堆, 然後再放在一起.
- ...
- 直到完成(共d個pass).
- 最後放在一起的items即為sort好的list.
- $O(d(n+r))$
- 用黑板舉例. ☺

重要事項提醒

- ptt2開板完畢 (HsinMu) 歡迎大家問問題
- 作業五周日會上線, 期限兩周
- 作業四這周日deadline
- 下週五停課一次, 於下下周二晚上6-9pm補課