本文圖片來自維基百科

程式人

用十分鐘

學會《資料結構、演算法和計算理論》

陳鍾誠

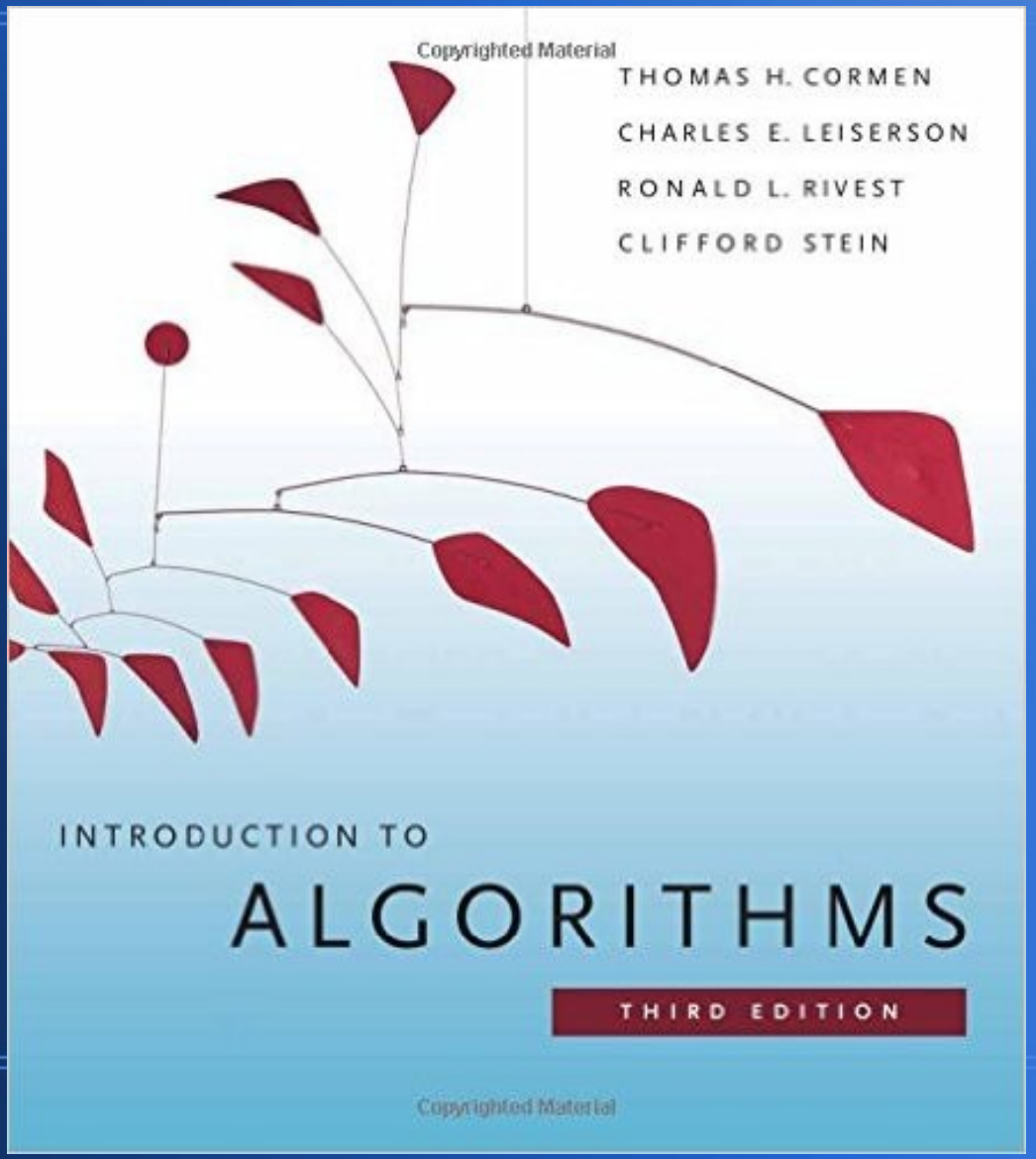
2016 年 1 月 11 日

二十五年前

- 我大學時讀的演算法課本是 300 頁

十五年前

- 我博士班時的演算法課本有一千多頁



但是

- 我比較喜歡大學時的版本
- 很討厭博士班的那個聖經版

問題是

- 現在很多大學也用那本聖經版

到底為甚麼

- 演算法的書要這麼厚呢？

而且老師們

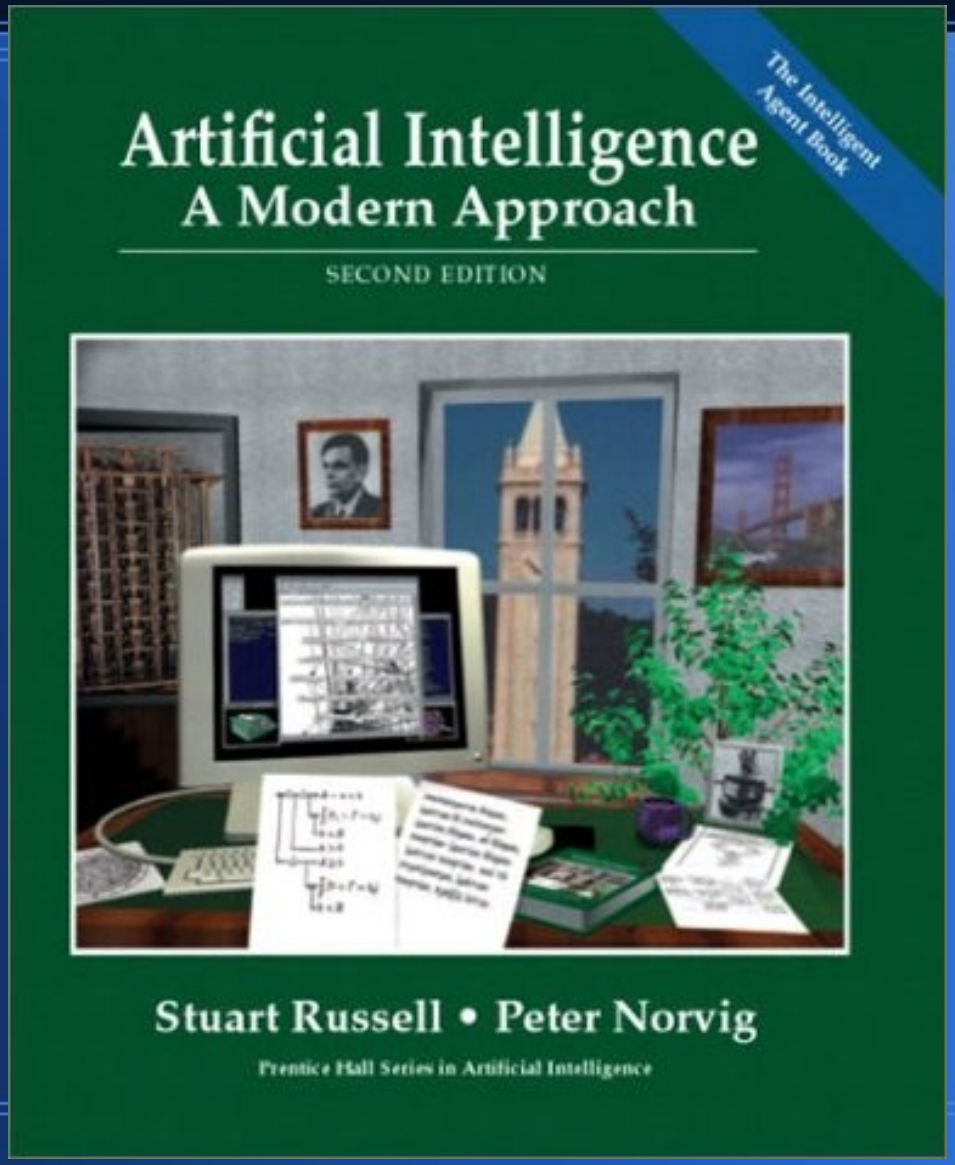
- 都爭相採用那種超厚的版本

更糟糕的是

- 我從來沒有看過任何一位老師可以把那本教完

就像我碩博士修了兩次人工智慧

- 用的都是這一本
- 同樣有一千多頁
- 而且讀起來絕對不會輕鬆愉快



但是每次老師

- 都只從第 1 章教到第 10 章

  然後學期就結束了！

- 問題是那本書有 26 章

  我讀了十年都還沒完全讀完！

我不禁想問

- 那剩下的呢？
- 既然教不完，難道就不能用本薄一點的嗎？
- 一定要用聖經版嗎？

除了這兩門課之外

- 還有
  - 微積分、線性代數、....
  - 電子學、數位邏輯、計算機結構....
- 等等族繁不及備載
- 課本都一本比一本厚

問題是學完之後

- 大部分的學生，都是處於完全失憶狀態…

舉例而言

- 數位邏輯學完之後
- 不會畫卡諾圖
- 畫不出七段顯示器的電路

其實、通常不用學那麼多

- 數位邏輯
  - 只要徹底學會七段顯示器電路
  - 就很夠用了！
  - 最多再加上正反器，就能學計算機結構了

同樣的、對於資料結構

- 你只要學會下列三種結構
  - 鏈結串列
  - 二元樹
  - 雜湊表
- 就差不多夠用了！

那麼為甚麼

- 我們要念那麼厚的教科書
- 讓自己十年都讀不完呢？

直到我開始教書之後

- 我終於想清楚一件事！

我們用那麼厚的教科書

- 只是因為
  - 那些作者寫了那麼厚的書

然後

- 老師們又採用了那本書

問題是

- 老師們為甚麼要採用那本書呢？

那是因為

- 那本書是聖經版！

那麼為何

- 那本書是聖經版呢？

那是因為

- 那本書大家都採用！
- 而且作者通常是該領域的優秀研究者！

那些作者

- 一輩子都在研究那個主題

更厲害的是

- 聖經版通常有兩個作者

這兩位作者

- 把畢生的功力，通通都寫進那本秘籍裏。

於是、你只要讀完一本

- 就相當於吸收了兩輩子的功力

問題是

- 你通常得花二十年才能讀完

無怪乎

- 我們的學生
- 讀完之後甚麼都忘了！

因為

- 他們看那本書的時間
- 不到二十天！

而且

- 其中的十五天
- 都處於恍神狀態！

所以、當他們學完資料結構

- 我問他們
  - 有沒有寫過《鏈結串列、二元樹和雜湊表》的程式呢？
  - 答案你們都知道了 …

我曾經考幾位學完《數位邏輯》的學生

- 請畫出七段顯示器最上面那根亮棒的《真值表、卡諾圖和電路》
- 結果、竟然幾乎沒有人會！

大部分的人反應是

- 那是啥？

然後是

- 我有學過嗎？

好了

- 我必須停止當一個酸民！

回到老師的角色

有教無類

- 因材施教！

讓我們來看看

- 資料結構　到底是甚麼碗糕？
- 演算法　我們到底應該會甚麼
- 計算理論　才算是夠用呢？

首先看看《資料結構》

- 顧名思義
  - 就是學習如何安排《程式》
    所需要用到《資料》
    的《結構》

而演算法

- 則是學習《演算的方法》
- 也就是《抽象的程式》之設計方法
- 探討的是《程式的方法論》

最後那個《計算理論》

- 則是《計算機的理論》
- 探討《電腦能力的極限》問題
- 包含《可不可解》，《要解多久》等問題。

《演算法》和《計算理論》

- 兩者的共同點就是都會探討《電腦要解多久》的問題
- 但計算理論還探討《可不可能解》的問題

而《電腦要解多久》的問題

- 有個正式的名詞
- 稱為演算法的複雜度
- 數學符號用 O() 表示
- 念為 Big O

話說股市有句名言

- 好的老師讓你上天堂
- 不好的老師讓你住套房

這句話對於《演算法》也適用

- 好的《資料結構》讓你上天堂
- 不好的《資料結構》讓你住牢房

程式要執行得快

- 資料結構一定要好
- 否則就會事倍而功半！

那麼、到底要怎麼安排資料的結構呢？

- 其實、這個問題並不太難。

在電腦中，通常只有兩種方法

- 一種是大區塊的結構
- 一種是小區塊的結構
- 大的整塊叫陣列
- 小的一塊一塊之間用連結連起來

對於陣列

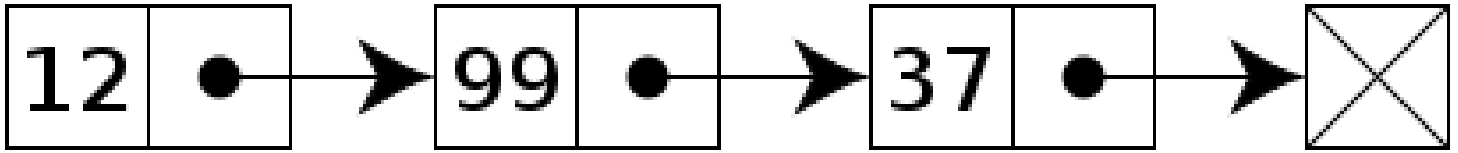
- 你可以用《排序＋二分搜尋》，來加快搜尋速度。
- 這就是資料結構第一部分的內容

對於小塊的結構

- 怎麼組織連結，那就是關鍵問題。
- 方法通常有三種
  - 一個連一個
  - 一個連兩個
  - 一個連多個

一個連一個的結構

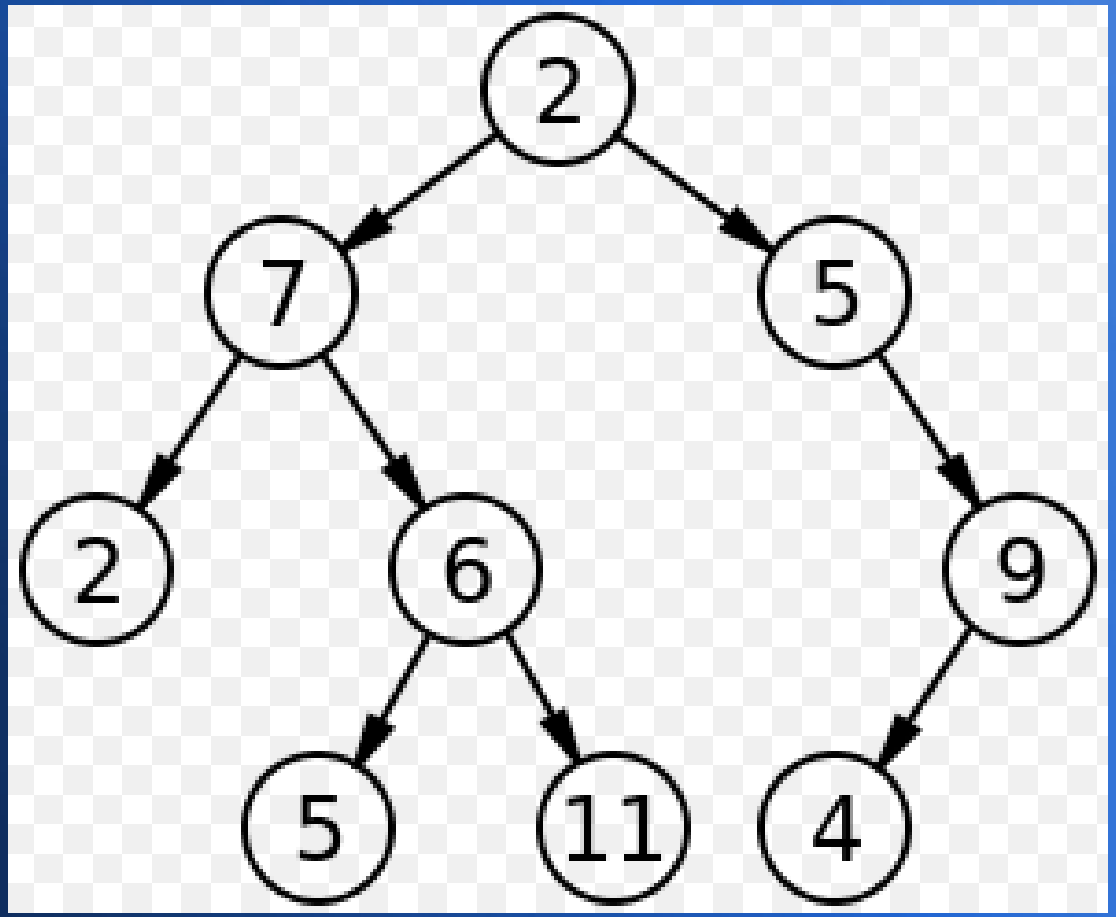
- 稱為《鏈結串列》



A singly linked list whose nodes contain two fields: an integer value and a link to the next node

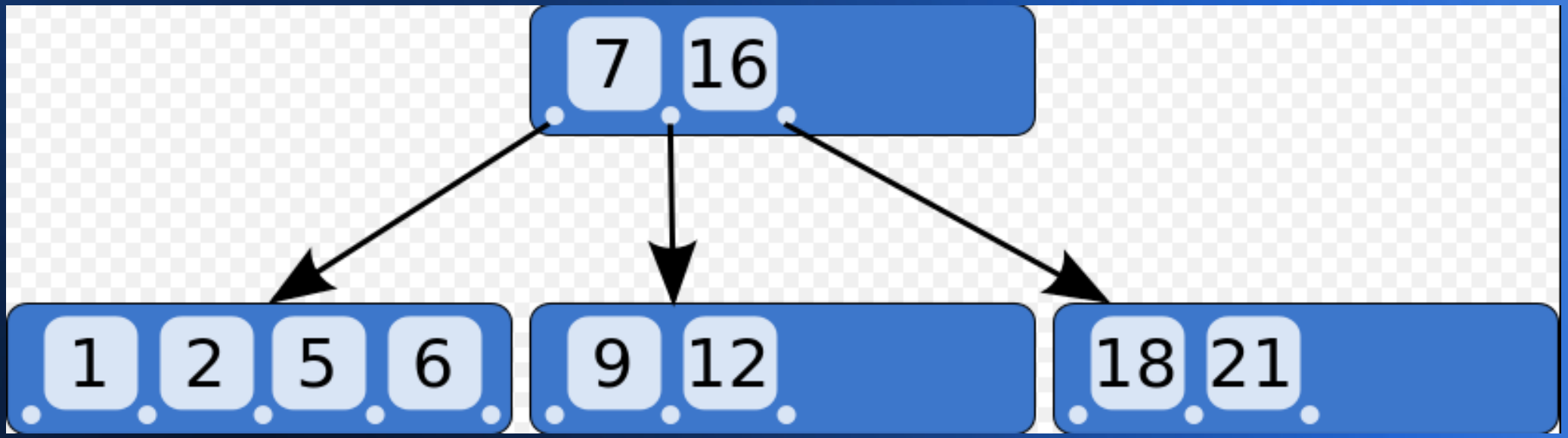
一個連兩個的結構

- 稱為《二元樹》



而一個連多個的結構稱為多元樹

- 像是 B-Tree 或 B^+ Tree 就是多元樹



這些結構

- 都必須要有相對應的
  - 新增、修改、刪除、查詢

演算法

通常

- 鏈結串列、二元樹
  - 會放在記憶體內
- 而 B-Tree 則是放在硬碟中
  - 是資料庫的主要結構

為何硬碟內要用 B-Tree 這種一對多的結構呢？

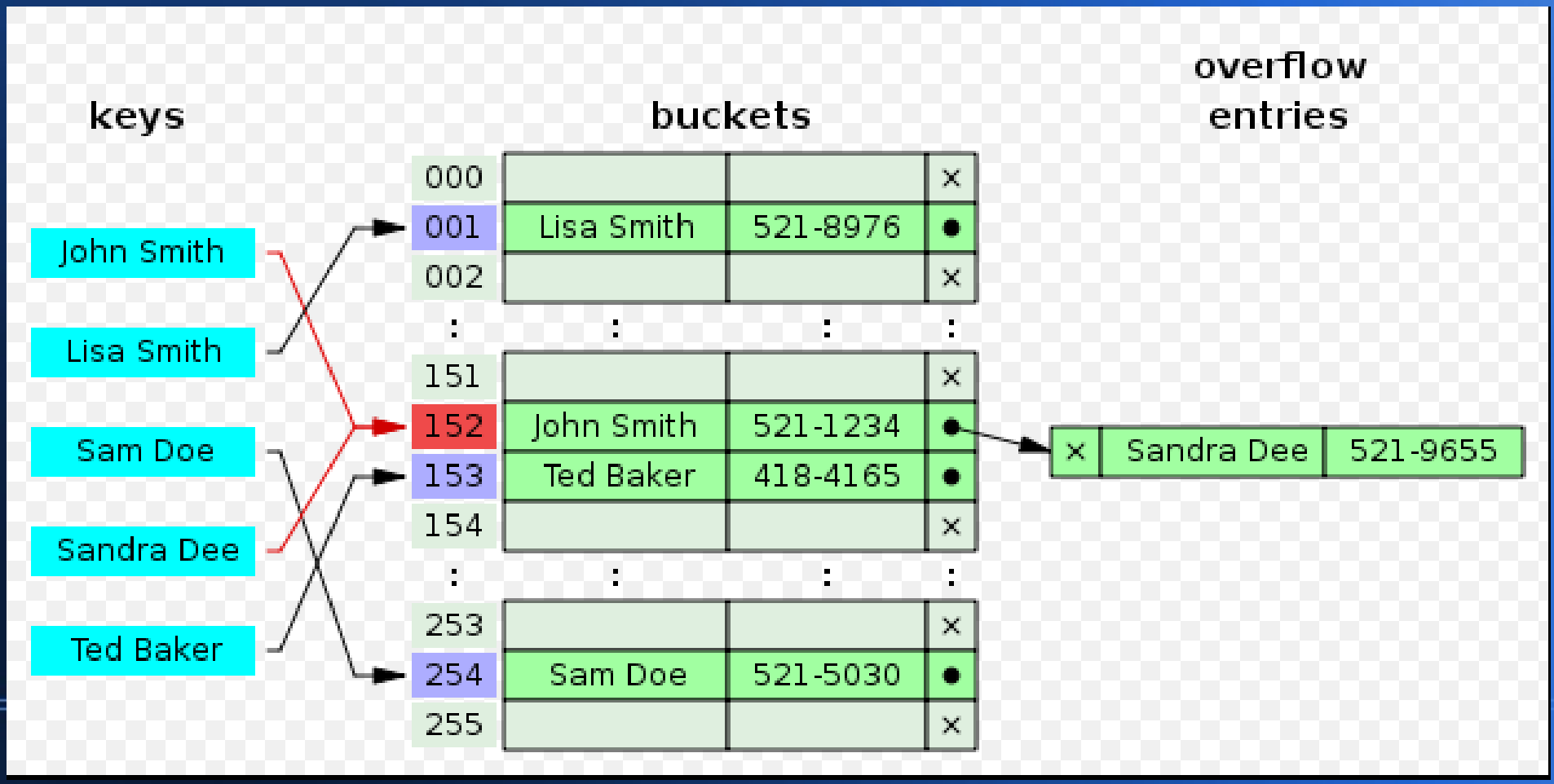
- 那是因為硬碟的旋轉速度是機械性的，相對電子速度很慢
- 於是每轉一次當然要多讀一些才划算
- 所以 B-Tree 通常是以一對百的結構就像《葉問》以一擋百那樣。

當然

- 有大塊的陣列結構
  和小塊的連結結構
  當然就會有《混合結構》

雜湊表就是一種混合大塊和小塊的結構

- 用陣列當大容器，每一格當中都可以放一個鏈結串列



至於每個東西要放哪一格

- 就由雜湊函數來決定！

```
hash = hashfunc(key)
index = hash % array_size
```

- 只要大家盡量分開，不要湊再一起，那每個鏈結串列就都很短，找起來就會很快

學會上述幾種結構

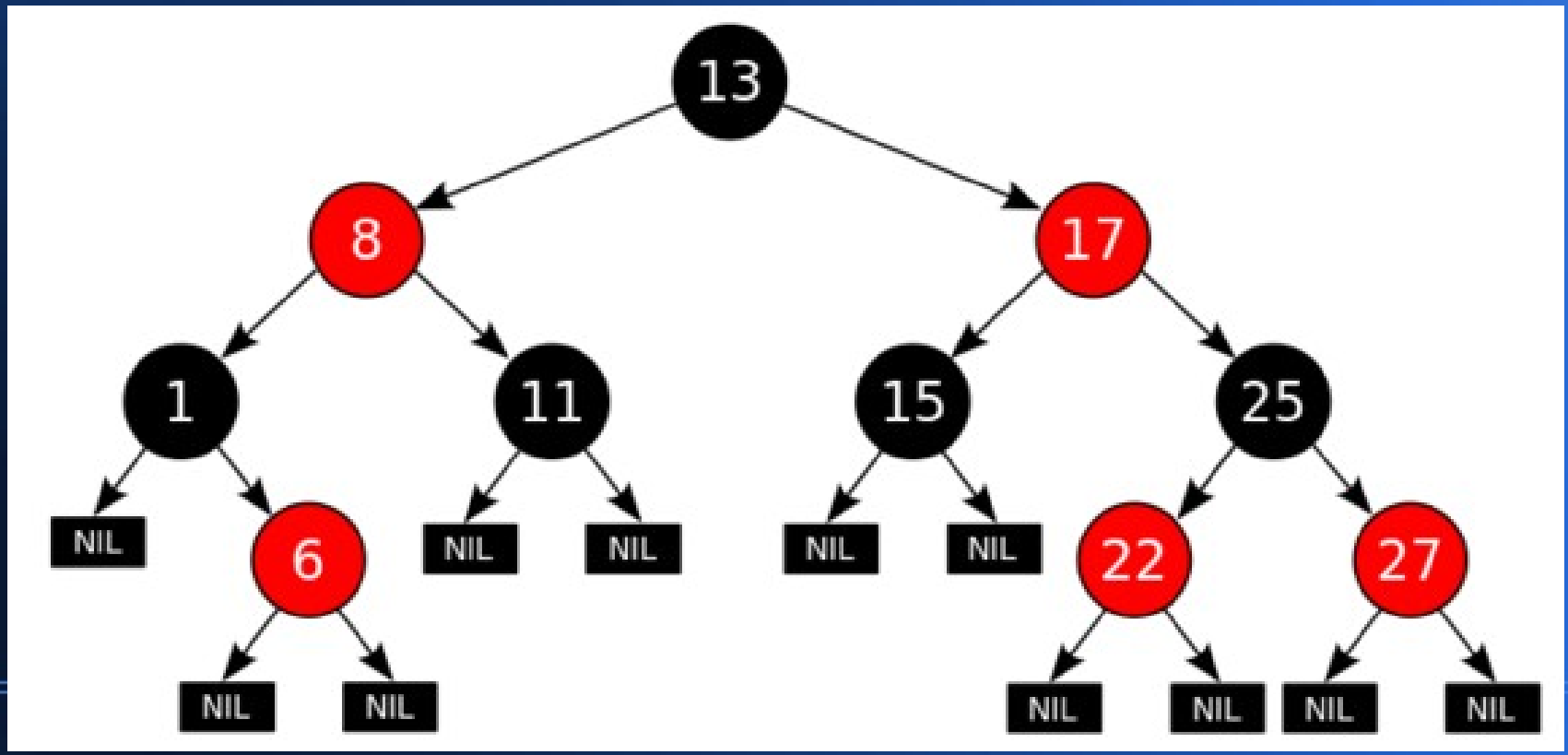
- 你一定要自己寫一遍
- 否則不要說你學會了！

而其他的資料結構

- 通常是特殊用途
- 或者是對上述幾種結構的改良。

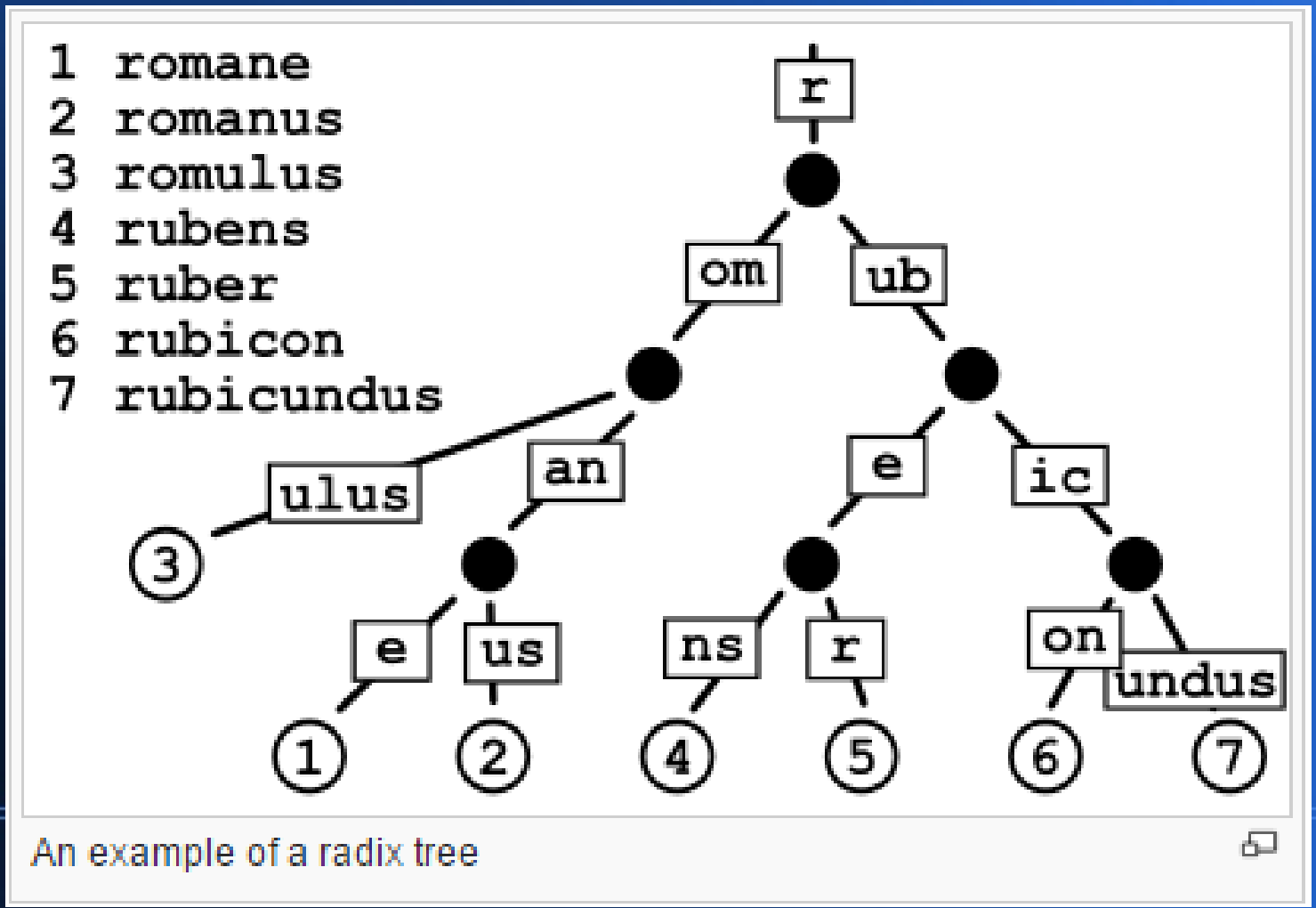
像是紅黑樹

- 就是一種自動平衡的二元樹，讓樹的層次盡量減少，不會太過傾斜導致搜尋很慢而已！



而 radix tree 則通常用在字串上

- 對字串結構的搜尋與統計特別有用



An example of a radix tree

這樣

- 資料結構就講完了！

接著讓我們來看看《演算法》

如前所述

- 演算法就是《抽象的程式》

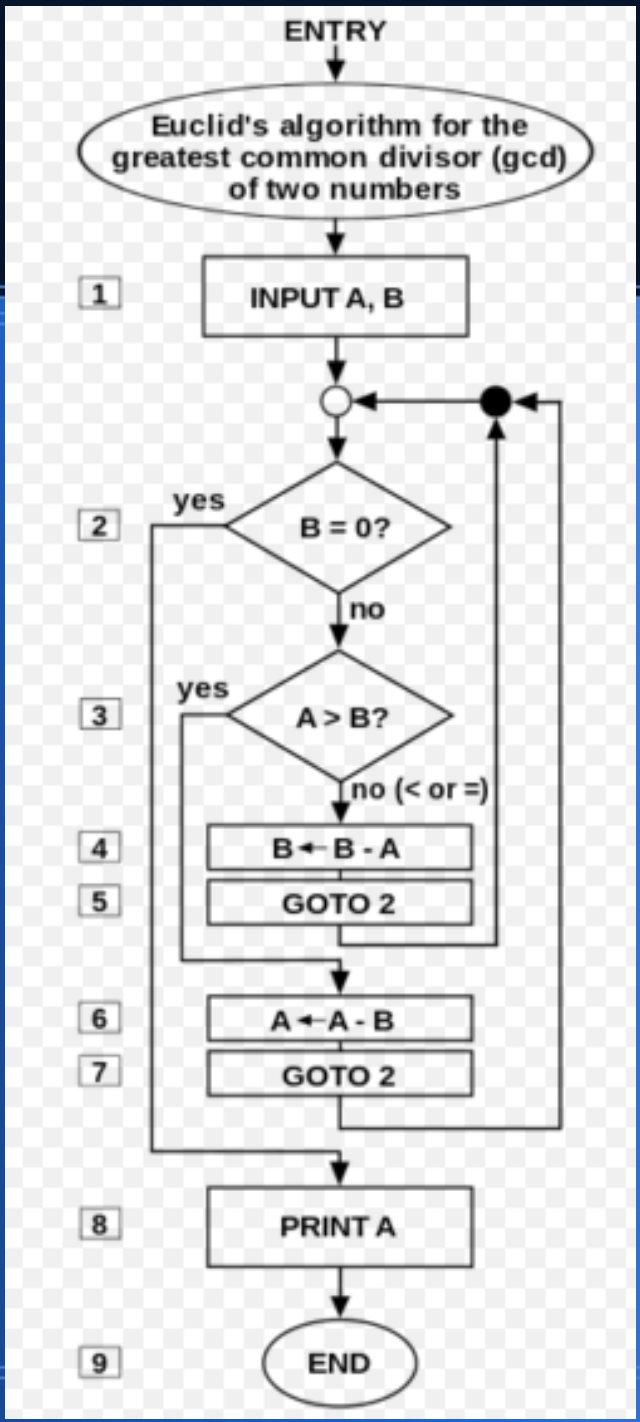
像是這樣

```
Discrete Space Hill Climbing Algorithm
   currentNode = startNode;
   loop do
      L = NEIGHBORS(currentNode);
      nextEval = -INF;
      nextNode = NULL;
      for all x in L
         if (EVAL(x) > nextEval)
               nextNode = x;
               nextEval = EVAL(x);
      if nextEval <= EVAL(currentNode)
         //Return current node since no better neighbors exist
         return currentNode;
      currentNode = nextNode;
```

爬山演算法

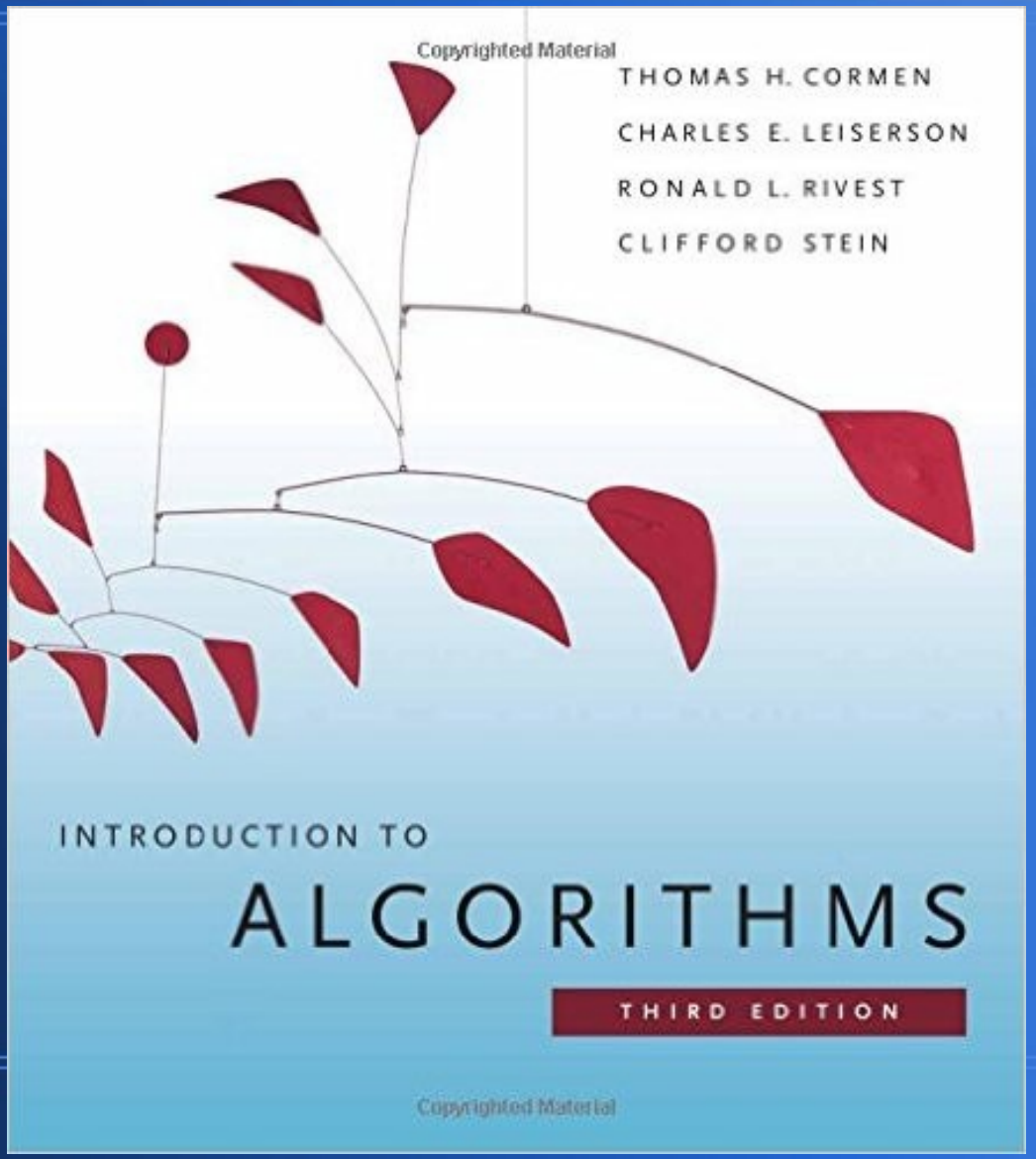
很久很久以前

- 教程式的老師們
  會教你畫流程圖
  那其實也算是
  一種圖示演算法



演算法就像程式語言

- 多得讓你學不完
- 想學完的話可以試著念右邊這本書！
- 不過我實在念不下去



但是如果因為我沒念完上面那本

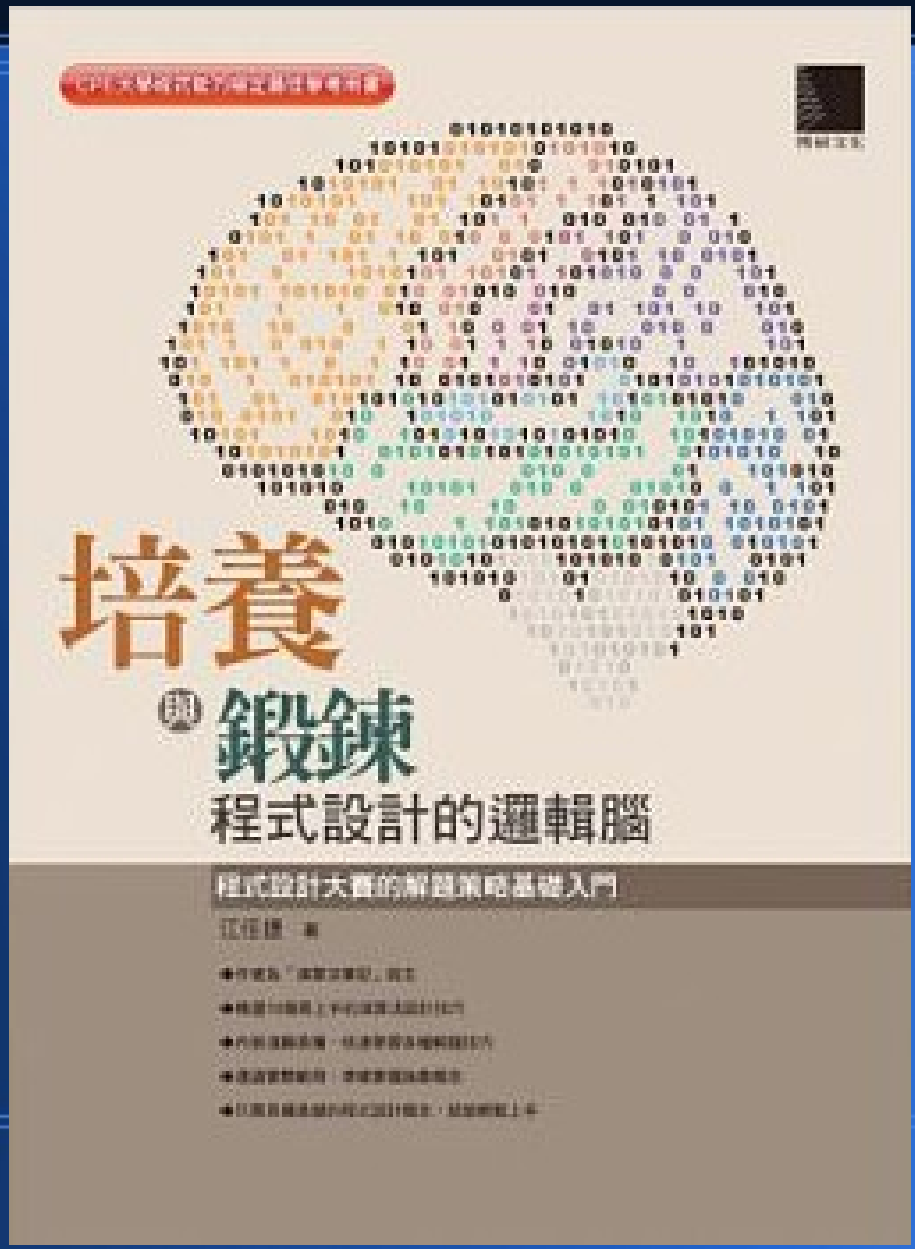
- 你就說我演算法很爛！
- 那我是不能服氣的。

我比較喜歡小本的

- 像這本一兩百頁的就很夠用了
- 重點是你要去寫
- 而不光是看完！

該書為《演算法筆記網站》的作者所寫的：

http://www.csie.ntnu.edu.tw/~u91029/index.html



而且我喜歡按方法分類的

- 像上面那本小的
- 再補上一些方法
- （像傅立葉轉換）
- 就差不多夠了！

Chapter 1 遞增法 Incremental Method
Chapter 2 記憶法 Memoization
Chapter 3 枚舉法 Enumerative Method
Chapter 4 遞推法 Iterative Method
Chapter 5 遞歸法 Recursive Method
Chapter 6 分治法 Divide and Conquer
Chapter 7 動態規劃 Dynamic Programming
Chapter 8 貪心法 Greedy Method
Chapter 9 縮放法 Scaling Method
Chapter 10 套用模型 Modeling

讓我們看看上述方法

- 方法一：遞增法(Incremental)就一個一個來囉！
  - 例如從 1 加到 n ，就寫個迴圈慢慢加囉！
- 方法二：記憶法(Memoization)
  - 記住後，要用的時候再查。
  - 像是排序搜尋雜湊表等結構
  - 都是為了記住後再查出來而已。

方法三：枚舉法
Enumeration

- 就一個一個列出來檢查囉！
- 像是解《拼圖遊戲》或《八皇后問題》，就可以用一個系統性的方法列出解答後進行檢查，看是否符合，符合就輸出答案。

方法四：遞迴法 Recursive

- 就是讓函數自己呼叫自己的方法。
- 遞迴法也常與《方法五》的《分治法》一起用，將大問題化為小問題解決後，再組合成大問題的方法。
- 像是《合併排序，快速排序》等都是《遞迴分治法》的範例。

方法六：動態規劃法
Dynamic Programming

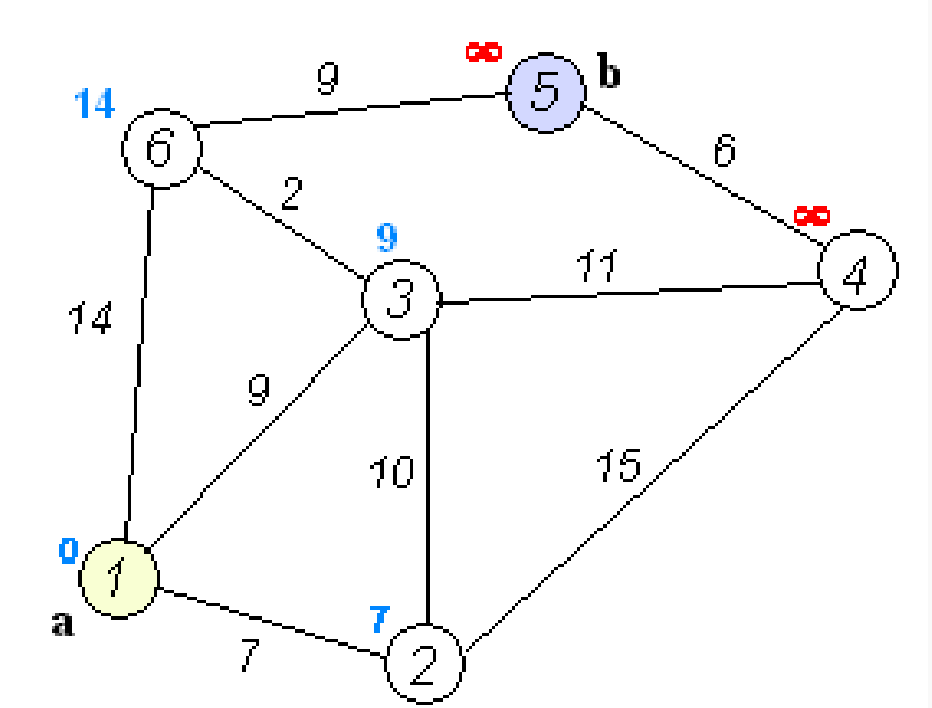
- 有時《分而治之》會重複計算很多次，這時若用個表格從下而上系統性的建構，就可避免重複計算，讓速度大大提升。

像是最短路徑問題

- 可用 Dijkstra 算法解決

  這就是動態規劃法的範例
- 另外像是 DNA 序列比對中的最小編輯距離問題，也可採用動態規劃方法解決
- 甚至早期的手寫辨識系統也通常是採用最小距離的動態規劃法所設計的。

Dijkstra's algorithm



Dijkstra's algorithm to find the shortest path between *a* and *b*. It picks the unvisited vertex with the lowest-distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\|E\| + \|V\| \log \|V\|)$

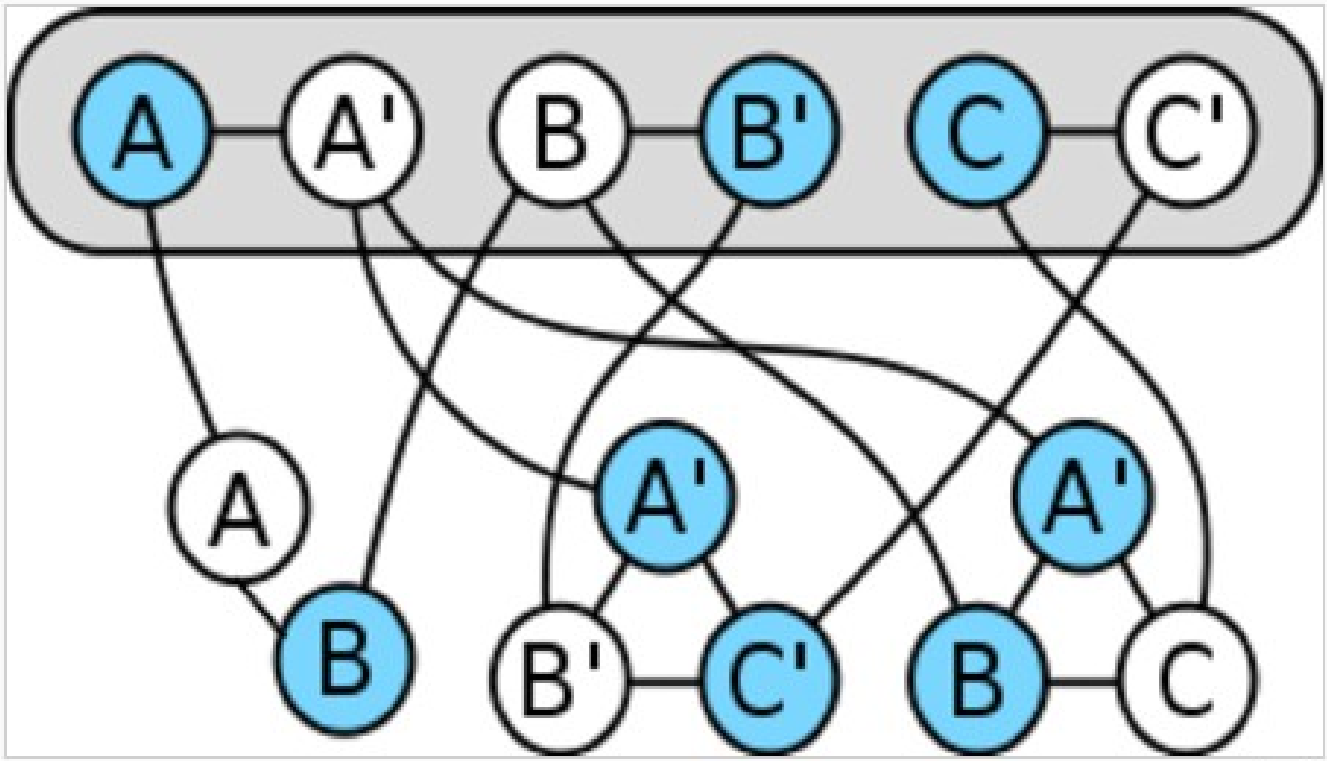
方法七：貪心法
Greedy Algorithm

- 就每次都找《賺最多》的那個加進來
- 像是要設計找零錢程式的話，就每次都找比差額小但最接近的錢，不斷重複就行了。
- 範例：1000-873 = 127 = 100+2*10+7*1

方法八：規約法
Reduction

- 將 A 問題轉換成 B 問題的特例，然後套用 B 的解法解決。

例如：邏輯滿足問題 SAT 竟然可以被轉成圖論的頂點涵蓋問題 VCP ，於是只要找到解 VCP 的演算法就可以用來解 SAT



Example of a reduction from the boolean satisfiability problem $(A \vee B) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (\neg A \vee B \vee C)$ to a vertex cover problem. The blue vertices form a minimum vertex cover, and the blue vertices in the gray oval correspond to a satisfying truth assignment for the original formula.

然後、就有一位偉人出現了

- 那就是

  Stephen A. Cook

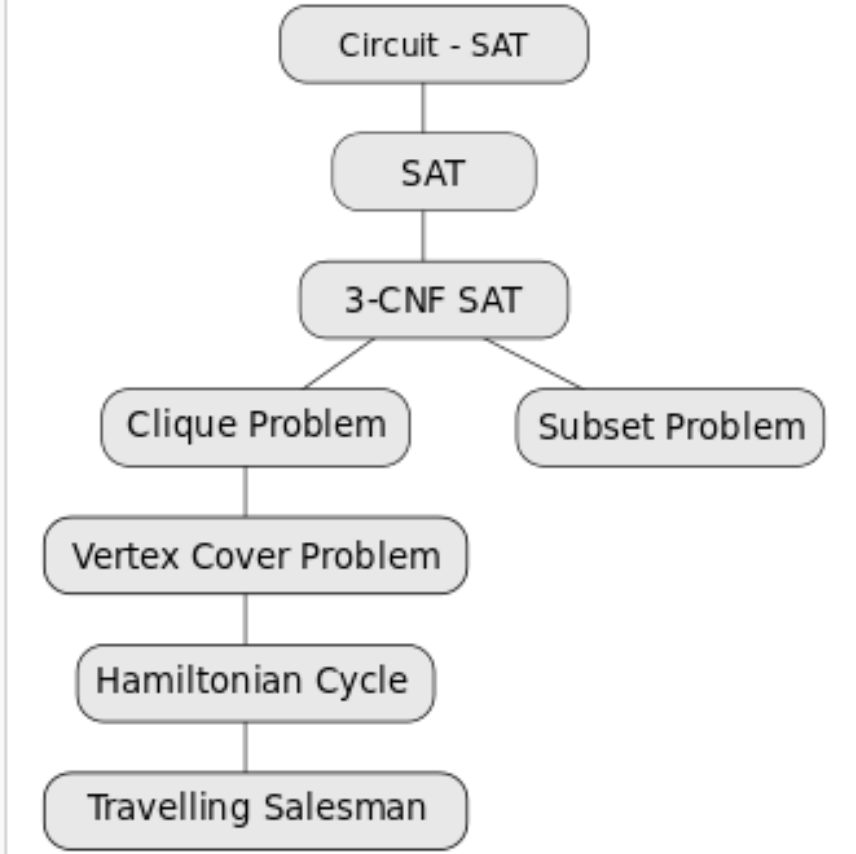
- 計算理論中

  NP-Complete

  問題的創造者

NP-Complete

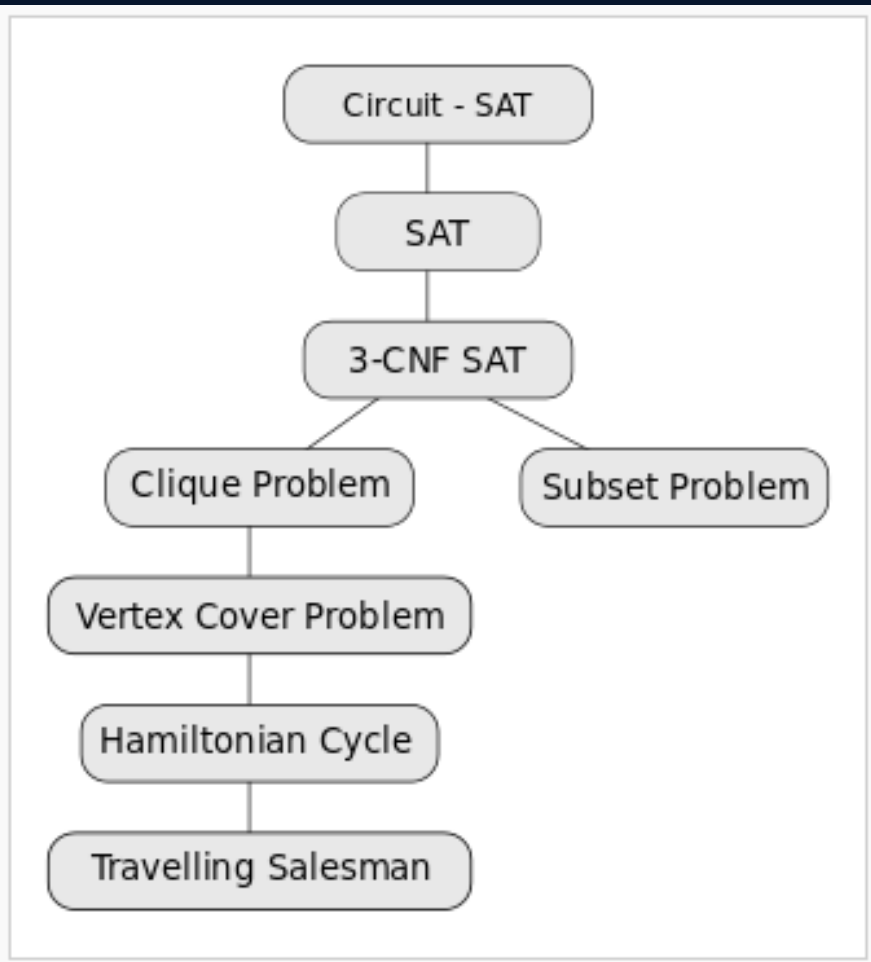
Stephen A. Cook 在 1971 年發表了 -《The Complexity of Theorem Proving Procedures》這篇論文，提出了 NP-Complete 這一大類問題，只要能在多項式時間內解掉其中一個，就可以解掉全部。



Some NP-complete problems, indicating the reductions typically used to prove their NP-completeness

您可以看到圖中位於最上面的問題

- 正是 CircuitSAT 問題
- 這是前面說的那個邏輯滿足問題 SAT 的一個變形。



Some NP-complete problems, indicating the reductions typically used to prove their NP-completeness

Cook 巧妙的把《非決定圖靈機》的執行過程轉換成 SAT 問題

- 這讓所有只要能在《多項式時間》被《非決定圖靈機》執行完畢的問題，通通都可以變成 SAT 滿足問題。
- 於是只要 SAT 能在《多項式時間》內被解決，那所有的 NP-Complete 問題就都能在《多項式時間》內被解決。

更詳細的內容請參考

- 維基百科
  - NP-Complete
  - Cook–Levin theorem
  - Boolean satisfiability problem

不幸的是

- 從 1971 年到現在
- 沒有發明出任何演算法可以在多項式時間內解決任何一個 NP-Complete 問題。
- 所以這是個 Open Problem，只要你能解決其中一個，應該就可以得到圖靈獎。

這意味著

- 有很多看來不見得很困難的問題，實際上對電腦是很難解的。
- 目前至少必須要花上 $O(2^n)$ 的時間才能解決。
- 但這是很難接受的，要是 $n>1000$ 你可能會一直算到人類滅絕都還算不出來。

更糟的是

- 有些問題你永遠解決不了
- 不管算多久都一樣。
- 像是《停止問題》就是一個被《圖靈》證明無法解決的問題。

在說明《停止問題》之前

- 請讓我們先看幾個簡單一點的問題，這些問題稱為《悖論》。

問題一：羅素理髮師悖論

- 在一個小世界裡，有一位理髮師，他宣稱
  - 『要為世界上所有不自己理頭髮的人理髮，但是不為任何一個自己理頭髮的人理髮』。
- 請問他做得到嗎？

請您仔細想一分鐘

想到了嗎？

- 他一定做不到！
- 為什麼呢？

問題在他自己

- 假如他為自己理頭髮，那麼他就《為一個自己理頭髮的人理髮》，違反了宣示。

- 假如他不為自己理頭髮，那麼他就《沒有為『所有』不自己理頭髮的人理髮》，又違反了誓言。

所以

- 理髮師肯定無法完成宣誓的任務。

同樣的，這種悖論可以用來論證

- 上帝是不是萬能（無所不能）的這件事情！

假如、上帝是萬能的

- 那請問，他是否能創造出一個難題，困難到讓他自己解不出來呢？

弔詭的是

- 假如上帝能，那他就不是萬能的
- 假如上帝不能，那他也不是萬能的。

所以結論

- 即使是上帝，也不是萬能（無所不能）的！

而圖靈的停止問題

- 也是如法炮製

停止問題的定義如下

- 請問您是否有辦法寫一個程式，判斷另一個程式會不會停
  如果會停就輸出 1，不會停就輸出 0。

- 換言之，就是要寫出下列函數：

  isHalt(code, data) = 1　假如 code(data) 會停就輸出 1
  = 0　假如 code(data) 不停就輸出 0

問題是、假如 isHalt() 函數真的存在且永遠做正確判斷

- 那麼我們就可以寫出下列這個函數：

```
function U(code) // 故意用來為難 isHalt(code, data) 的函數。
  if (isHalt(code, code)==1) // 如果 isHalt(U, U)=1，代表判斷會停
    loop forever //   那 U 就進入無窮迴圈不停了，所以 isHalt(U,U) 判斷錯誤了。
  else           // 如果 isHalt(U, U)=0，代表判斷不停
    halt         //   那 U 就立刻停止，所以 isHalt(U,U) 又判斷錯誤了。
end
```

- 意思是，如果你說不停，我就停，如果你說停，我就不停。

這樣的話

- isHalt() 就一定會做出錯誤的判斷，所以這個問題根本就不可能被電腦百分百正確的解掉。

停止問題

- 是《圖靈》在 1936 年提出並證明不可解的
- 但事實上數學家《哥德爾》在 1931 年就證明了一個很類似的定律，稱為《哥德爾不完備定理》。

如果用程式人的想法解讀《哥德爾不完備定理》

- 那麼可以改寫如下：
- 哥德爾不完備定理：
  - 不存在一個程式，可以正確判斷一個「包含算術的一階邏輯字串」是否為定理。

證明方法和停止問題很像

- 都是用反證法，先假設這樣一個程式存在，那麼可以寫成下列的 Proveable 算法

```
function Proveable(str)
  if (str is a theorem)
    return 1;
  else
    return 0;
end
```

首先

- 讓我們用 T 代表

$$\neg\exists_s \neg\text{Provable}(s) \;\&\; \neg\text{Provable}(\neg s)$$

- 這個邏輯式的字串

然後分別討論「真假」這兩個情況

註：$T = \exists_s$ -Provable(s) & -Provable(-s)

- 如果 isTheorem(T) 為真
  - 那麼代表存在無法證明的定理
  - 也就是 Provable() 沒辦法證明所有的定理
- 如果 isTheorem(T) 為假
  - 那麼代表 -T 應該為真，但這條路將會產生矛盾

對於 -T 為真的情況

註：$T = \exists_s$ -Provable(s) & -Provable(-s)

```
function Proveable(-T)
  if (-T is a theorem) // 2.1 這代表 -T 是個定理，也就是 Provable() 可以正確證明所有定理
    return 1;   // 但這樣的話，就違反了上述「 2. 如果 isTheorem(T) 為假」的條件了。
  else          // 2.2 否則代表 -T 不是個定理，也就是存在 (∃) 某些定理 s 是無法證明的。
    return 0;   // 但這樣的話，又違反上述「 2. 如果 isTheorem(T) 為假」的條件了。
end
```

就這樣，三條路都被封死了，根據矛盾證法，這代表
Proveable 算法可以正確判斷一個「包含算術的一階邏輯字串」
是否為定理的假定是錯誤的。

換句話說，我們不可能寫出一個能完全正確判斷字串是否為定理的程式

於是、我們講完《哥德爾不完備定理》了

- 加上前面的《停止問題》與 NP-Complete 理論，這差不多就是《計算理論》這門課的主要內容了

如果您讀到現在都可以理解我們所說的內容

- 而且沒有花超過十分鐘的時間。
- 那麼我們就成功的完成在十分鐘內講完《資料結構、演算法和計算理論》三門課的這個不可能的任務了。

當然、關於詳細的細節

- 還是得花很多時間去實作，去思考，去體會才能真正學會的。
- 但我們希望這份投影片能讓你先有個基礎，才不會在面對課本時毫無頭緒

其實

- 這些課程，也可以不用講得那麼深，那麼難，或者一定要學那麼久的，不是嗎？

再會了

- 希望您喜歡我們的《十分鐘》系列課程！