無所不在的電腦科技
淺談計算機科學、演算法、及基因演算法

于天立／國立臺灣大學電機工程學系教授

隨著數位科技的發展，現代人的日常生活中總伴隨著各類電腦科技，諸如手機、數位相機、電腦、數位電視等等。然而大家對計算機科學，卻常常以為就只是在寫程式。其實程式設計其實僅僅是計算機科學中的一支，以下將由計算機科學的核心之一 — 演算法（Algorithm）開始談起，希望讓大家對計算機科學有進一步的認識。

演算法相關的研究最早起源於 1920 及 30 年代左右，這個詞看起來深奧，但其實說穿了就是「解決問題的方法及步驟」。例如下面是一個解決「想在家喝冰牛奶」問題的演算法（圖一）。

1. 看看冰箱有沒有牛奶，有的話跳到第 3 步。
2. 去最近的超市買冰牛奶，趁牛奶變熱前趕快回家。
3. 在家喝掉冰牛奶。

圖一　在家喝冰牛奶的演算法

這裡有個重點是演算法中每個步驟必須是有明確定義（well defined）而且可執行（executable）的。對人而言，這可能見人見智，但是以下這個「成為千萬富翁」的演算法可能多少執行起來有點困難（圖二）。

1. 弄到一千萬元。
2. 成為千萬富翁。

圖二　成為千萬富翁的演算法 (?)

臺灣大學電機工程學系

10617 台北市 大安區 羅斯福路四段 一號
Email: dept@cc.ee.ntu.edu.tw
http://www.ee.ntu.edu.tw/

因為如何「弄到一千萬元」可能對很多人來說並不是一個很清楚該怎麼做到的步驟。如此以上便不成為一個演算法。在計算機科學中，所謂「明確定義」及「可執行的」這部份，也花了前人不少心力。畢竟我們不希望指定是某一台實際的電腦可執行的，不然例如有些在蘋果電腦上可執行的，在微軟視窗系統下又不一定能跑。目前大家所廣為接受的是一種抽象的電腦概念，稱為 Turing Machine。目前你就先把它想像成一種可以模擬我們現在任何電腦的機器就行了 [1]。

我們先從簡單的問題開始。想像如我給你任意 n 個數字，如何找出最小的數字？有人可能覺得一眼就看出來了。當然啦，n 不大的時候是如此，但是如果 n 是一百萬呢？應該不難吧，如果想成有一大堆寫著數字的紙片，就先隨便拿一張起來，然後看下一張。如果下一張的數字比較小，就丟掉手上那一張改拿這張。不然的話，就留著手上的，丟掉剛看過的紙片。當然啦，請準備一個紙簍來裝被丟掉的紙片，不然混進還沒看過的就麻煩了。讓我們試著把它寫成演算法（圖三）

findMin ($a_1, a_2, \cdots, a_n$)
1. *result* ← a_1
2. *index* ← 2
3. *result* ← **min** (*result*, *aindex*)
4. *index* ← *index* + 1
5. **go to** step 3 **till** (*index* > *n*)
6. **return** *result*

圖三　找最小值的演算法

以上我們用了所謂 pseudo-code 的寫法，符號就不說明了。如果各位寫過一些程式的話，應該不難懂。

這個演算法需要執行多久呢？基本上用不同的程式語言，在不同的機器上執行，時間或多或少都有差異。但是有一點我們很肯定的是，執行時間基本上和 n 成正比。「基本上和 … 成正比」其實是個很重要的概念，這裡我不給正式的定義，但是為了方便起見，讓我用 $\Theta(n)$ 來表示這個概念。這個概念很重要是因為用計時的方式不容易正確評估一個演算法的好壞。在某一台電腦上跑個 5 分鐘的程式，拿到另一台電腦上可能只要 1 分鐘。為了讓演算法的效能評估獨立於實際的機器之外，我們常常使用 $\Theta(.)$ 做為演算法好壞的指標。

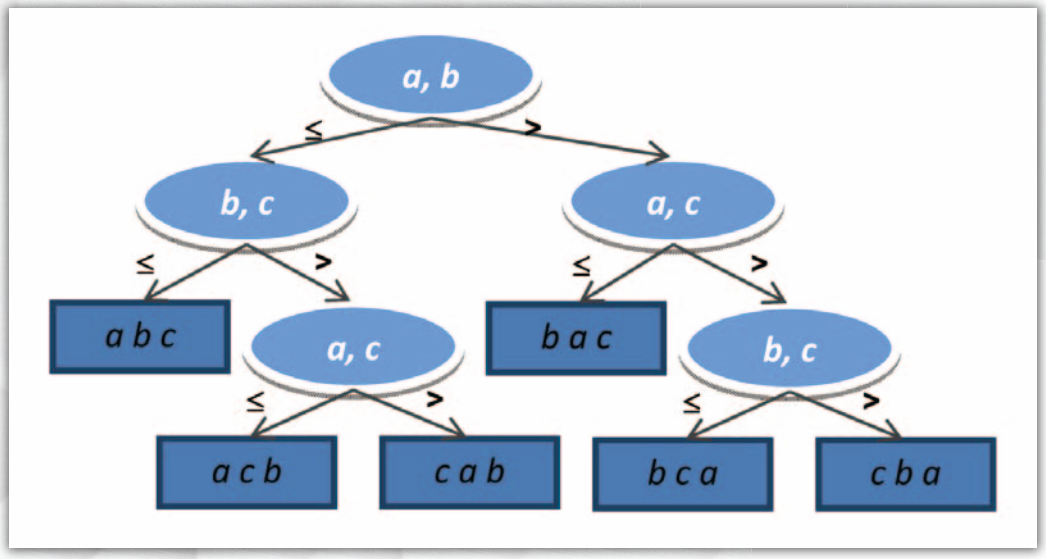
上述的演算法要花 $\Theta(n)$ 的時間，那可不可能更快呢？例如可不可能用 $\Theta(n^{0.5})$ 甚至 $\Theta(\log n)$ 的時間把最小值找出來呢？這裡我們想像一個搗蛋鬼，如果那 n 張紙片有任一個我們沒有看，他就可以偷偷的把其中的數字任意更換。想像如果有一張紙片我們沒看，然後我們的演算法說最小值是 –100。搗蛋鬼就把我們沒看的那張紙片數字換成 –200，然後攤開說：「哈哈，你錯了，最小值是 –200」。我們百口莫辯。因為我們既然沒有看過那張紙原來寫的是什麼，我們也無法確定這個 –200 是搗蛋鬼偷改的還是原本就是 –200。也就是說，為了確保找最小值的演算法正確，我們一定得看過每個數字，所以時間就最起碼正比於 n [2]。換句話說，不考慮常係數（constant factor）的話，我們的演算法是最佳的（asymptotically optimal）。

像這類問題，我們証明了下限（lower bound）是 $\Theta(n)$，而我們又有一個執行時間恰好是 $\Theta(n)$ 的演算法，我們稱之為 closed problem。意謂著日後的改進只能侷限在常係數的改進，例如只花原先一半的時間。當然在實用上，這樣的改進仍然是很重要的。但是至少大家不用再花腦筋去想有沒有 $\Theta(\log n)$ 的演算法了。寫到這裡，我要先恭喜各位，因為你已經初步了解了演算法的意義、如何設計演算法、如何表示演算法、如何分析演算法、以及如何証明演算法的最佳性。

接下來讓我們看看另一個在電腦常見的問題—排序問題：給任意 n 個數字，由小到大排列好。我們已經會了找最小值，所以排序其實很簡單，先把最小值拿出來，再對剩下的 n–1 個數字找最小值，依此類推。利用已有的程序的方法，相當於是把一個我們不會解的問題轉化成一個（或若干個）我們會解的問題。這種方式稱為 reduction，也是計算機科學中相當重要的一個概念。

我們來看看上述的演算法需要多少時間。第 1 次找最小值有 n 個數字，需要做 n–1 次的比較，第 2 次找最小值需要 n–2 次比較 … 第 n 次找最小值只剩 1 個數字，需要 0 次比較。所以總共我們比較了 $\frac{(n-1)(n-2)}{2}$ 次。用「基本上和 … 成正比」的概念，我們知道這個演算法執行時間是 $\Theta(n^2)$，現在你們了解這種「大而化之」函數的好處了吧？

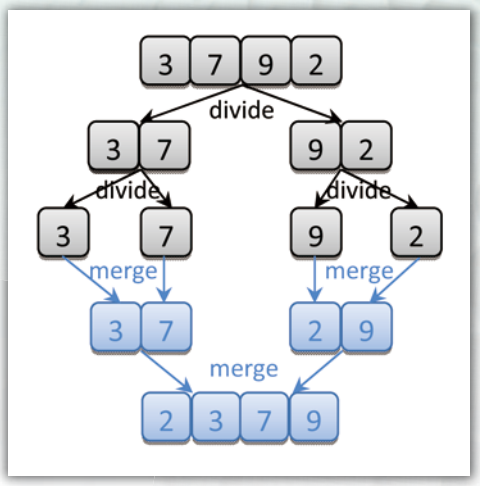
如果電腦的能力限制在數字只能兩兩比較大小（假設這種比較所花的時間是固定的），倒底排序需要多少時間呢？我們考慮 3 個數字 a, b, c。原本所有可能的排法就是 3! = 6。當我們比較 a 和 b 誰大誰小後，假設 $a > b$，則我們就知道 abc, acb, cab 這三個排法是不可能的（由小排到大）。如果我們把每次做的比較和最後的輸入列出來，就成為一個決策樹（圖四）。



圖四　對 3 個數字排序的決策樹

其中最糟的情況（需要最多次的比較），也就是決策樹中最長的路徑，也就相當於樹的高度（不算最後輸出）。當然如果我們做不同的比較，決策樹會長的不一樣。重點是我們知道有 6 種不同可能的輸出，每次比較時會有 2 種不同可能的結果，所以整個決策樹的高度至少是 $[\log_2 6] = 3$。各位同學可以試試看不同的排法，應該會發現最長路徑只可能更長，而不會更短。我們相信搗蛋鬼一定會給我們更糟的狀況（adversary argument），也就是說，要排任意 3 個數字，我們至少需要 3 次的比較。同理，推到 n 個數字，要排序 n 個數字，至少需要 $[\log_2 (n!)]$ 次比較。$n!$ 取 log 可能對同學有些困難。幸好數學家 Stirling 告訴我們 $n! \approx \sqrt{2\pi n}\left[\frac{n}{e}\right]^n$。有興趣的同學可以推導看看，你應該會得到 $[\log_2 (n!)] = \Theta(n \log n)$。

吁！終於証完了。咦？等等，我們上面給的排序演算法要花 $\Theta(n^2)$，難道還有改進空間嗎？接下來我要介給的方法，稱為 merge sort，是一種利用 divide-and-conquer 的解題法。Merge sort 整個重點是在合併 (merge) 上面。如果我們能很快的合併兩個排好序的數列，則針對 n 個數字，我們只要把它們切成兩個 $n/2$ 的數列，各自排序好，再合併就行了。怎麼「各自排序好」呢？沒錯，再切成兩個 $n/4$ 的數列 …。這樣一直切到整個數列只有 1 個數字，則不用做任何事情，它就是排好的了。整個 merge sort 的示意如圖五。



圖五　Merge sort 流程

那如何很快的合併兩個排好序的數列呢？天才數學家 von Neumann 在 1945 年給了如下的演算法 [3]。先比較兩個數列 $<a_i><b_i>$ 中最小的數 a_1, b_1，假設 a_1 比較小，則把 c_1 設為 a_1，然後比較 a_2 和 b_1，依此類推，最後 $<ci>$ 就會是合併後的有序數列（圖六）。合併兩個 k 個數字的數列最少需要 k 次比較，最多需要 $2k$-1 次。

這要花多少時間呢？這有一點點麻煩。如果我們假設排 n 個數字需要 $T(n)$ 的時間，則可以寫出下列算式：

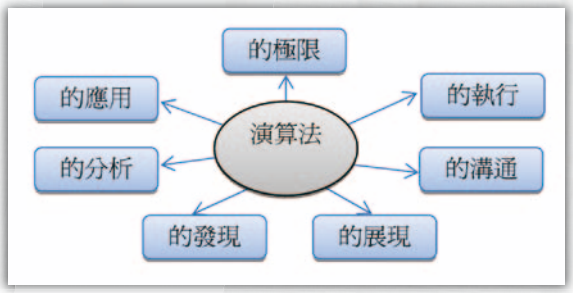
$$T(n) = 2T(n/2) + cn$$

其中 c 是某個接近 1 的常數，而且我們知道 $T(2) = 1$（兩個數字比一次就好了）。同學們可以試試看用 $n = 2^k$ 代進去試試。沒意外的話，應該會得到 $T(n) = \Theta(n\log n)$ 的結論。也就是說不考慮常係數的話，merge sort 已經是最佳的排序程式了。

講了兩個例子，都有已知最佳的演算法。那有沒有所謂的 open problems 呢？當然有，而且問題的描述本身並不一定很複雜。例如著名的 3SUM 問題：給任意 n 個數字，請驗証是否其中有某 3 個數字加起來是 0? 當然很簡單的我們可以用暴力演算法對所有 C_3^n 的可能都檢查看看，這樣基本上要花 $\Theta(n^3)$ 的時間。能不能更快呢？其實這個問題有很簡單的 $\Theta(n^2)$ 的演算法。這邊給一點提示。首先對這 n 個數字排序，我們已經知道這要花 $\Theta(n\log n)$ 的時間。然後我們想檢查是否存在 $a + b + c = 0$，就等於針對每一個 c，檢查在一個排好序的數列是否有 $a + b = -c$。換句話說，如果我們能在 $\Theta(n)$ 的時間內，檢查在一個排好序的數列是否有某兩個數字相加是 0，則 3SUM 就可以在 $\Theta(n^2)$ 的時間內解完（因為對每個 c 都要花 $\Theta(n)$）。剩下的就留給大家去思考。3SUM 最大問題在於我們知道 $\Theta(n^2)$ 可以解開，但是最快能多快目前沒有人知道。之前有學者証明過在比 Turing machine 弱的一種計算模型上，$\Theta(n^2)$ 已經是最快的了。2008 也有學者給出了比 $\Theta(n^2)$ 還快的演算法，但是我們目前尚不知道此問題在 Turing machine 上倒底極限在哪。

還有像是著名的整數分解問題：給一個 n 位數（很大）的正整數，對它做質因數分解。這個問題目前我們不知道有沒有存在多項式時間（polynomial time）的演算法。所謂多項式時間，指的是 $\Theta(n^c)$，其中 c 是某個常數，可以大到像是一百萬，一兆都好。只可惜目前已知最快的演算法仍然比這個慢，而且也沒有人能証明是否存在或是不存在多項式時間的演算法。這不是件小事。目前許多密碼機制是採用所謂 RSA。而如果有很快的方法可以做到大數質因數分解，則我們可以很快的破解 RSA，以致於現在使用 RSA 的銀行交易密碼機制都必須重新設計。

圍繞著演算法，計算機科學發展出了各個不同領域（圖六）。在本文中，我們已經簡單的聊了演算法的發現、分析、應用、及極限。演算法的執行，主要和計算機的架構有關，其中的技術包含了如何把高階語言（如 C/C++）轉成計算機能執行的代碼，以及如何利用快取，多核心等技術，使的演算法執行的更快，當然也包括了作業系統（如 Windows7、Android、Linux）的研究。有些大量運算的演算法是設計成可在多台電腦上執行的。這時演算法之間的溝通就變的很重要。相關的研究領域包含了叢集電腦、分散式運算、以及近年來很熱門的雲端技術。演算法之間一旦需要溝通，又會涉及資訊安全的議題。畢竟我們總不希望存在雲端的個人資料隨意的被第三者讀取。至於演算法的展現，主要的研究則是在所謂程式語言的部份。像各位可能學過的 C/C++、Basic、Java 等都是一種程式語言。程式語言的發展通常有各種不同的目標。例如 Java 的目標之一在於跨平台，正因這個特性，Java applet 在網頁設計上變得非常普及，各位手機上小遊戲大部份也都是由 Java 寫成的。另外則有一些語言，其設計的目的在於幫助減少設計師的錯誤。當然計算機科學中還有許許多多的應用，例如數位相機的人臉辨識功能，及電腦棋是屬於人工智慧範疇，生物晶片技術中運用的資料探戡，… 實在是列舉不完。其中我想對基因演算法（genetic algorithms）稍加著墨。



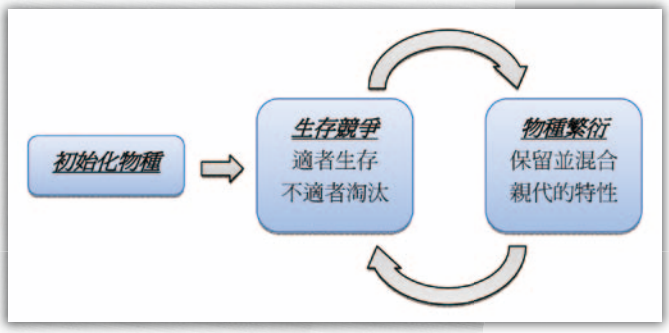
圖六　以演算法為中心的計算機科學，圖翻譯自 Computer Science: An Overview by Brookshear, PEARSON 一書

各位有寫過程式經驗的同學應該都有種感覺，電腦會非常忠實的執行我們給它的指令，所以大部份的時候程式執行的結果或多或少我們是知道的。傳統的人工智慧其實可以視為是把程式設計師的智慧程式化之後的結果。例如想到寫個下五子棋的程式，可能就會想到用一些規則：如果能連成 5 個就先連，不然的話，對方有連成 4 個的要先擋等等。這說穿了是設計師的智慧。當然這些規則可以是由電腦透過大師們的棋譜去統計、歸納出來的，但統計歸納的方式，以及大師們的棋譜畢竟不是電腦「原生」出來的。

基因演算法跟這些有點不同。它是利用 Darwin 的演化論，試著把環境設定好後，試著讓程式自行演化出來。基因演算法要解的問題我們通稱為黑盒子最佳化（圖七）。想像 f 是未知的，我們能做的事是丟一個 x 進去，看看 $f(x)$ 是多少。要解的問題則是找一個 x，使得 $f(x)$ 最大。舉例來說，x 可以是固定材料下的橋樑結構，$f(x)$ 則是此結構的載重量，則目標就是在固定材料下，尋找一橋樑結構，使其載重量最大。日本新幹線車頭就是利用基因演算法設計的，其目標在尋找一車頭結構，使其風阻最小。當然啦，我們對其截面積必須要有限制，不然演化的結果會變成一根針 …（汗）。

圖七　黑盒子最佳化及日本新幹線車頭

所謂環境的設定包含哪些呢？以五子棋為例，首先設定的物種就是「會下五子棋的程式」。要做到這點並不難，只要有一個「評估函式」即可。評估函式的輸入是盤面狀況，輸出則是分數。電腦只要把還可以下的地方都下下看，把產生的盤面狀況交給評估函式，然後選分數最高的那一步來下就好了。可以看的出來評估函式直接決定了程式的棋力。一開始沒有任何資訊下，評估函式可以直接由亂數產生。最開始我們就利用亂數產生一堆會亂下棋的程式。環境設定中還包含了「適者生存」，也就是讓程式捉對廝殺，勝者生存，敗者淘汰。可以想見的是，一開始大家都是在亂下，但是總有程式「不小心」勝過對手，而這些似乎下的比較好的程式就有比較高的機率生存下來。然後重頭戲就是讓這些生存下來的程式自行繁衍出下一代。繁衍的方式有非常多種，不過重點是在於某個程度保留而且混合父母的特性。基因演算法的流程可參考圖八。



圖八　基因演算法流程

不可諱言的，環境的設定或多或少也利用了設計者的智慧，但是演化的部份大部份是電腦原生的。而且寫這種程式好玩的地方在於設計者也不太能掌控程式的行為。就上面的例子而言，當然我們知道產生的程式都能下五子棋。可是它們活三或衝四會不會擋，就看它們演化的結果了。單就學術面而言，我們也很想知道這樣的演化會一直進步嗎？如果會的話，進步的上限又在哪裡？這些目前都是 open problems。處在數位電腦科技時代，我們期待有更多生力軍加入，不論是理論或是應用，能一起開發計算機科學所帶來的便利。

註腳

[1] Alan Turing 被譽為人工智慧之父（也有人稱他為計算機科學之父）。現在計算機科學最高榮譽 Turing award 即為紀念他所設立。他在 1936 年提出 Turing machine，大約比最早的電腦還早了十多年。後來由 Church 推動（Church-Turing thesis），Turing machine 成為計算機理論科學中最重要的抽象計算模型。

[2] 此類論証稱為 adversary argument，是一種重要的論証法。

[3] von Neumann 被譽為電腦之父，現在的電腦用的即是由他提出的 von Neumann 架構，他在電腦科學、經濟、量子力學、數學領域都有重大貢獻。