計算機概論四講

單維彰

AUGUST 25, 1994 (first draft)*
AUGUST 27, 1995 (revised)

前言

本文是根據我在中央大學數學系的計算機概論課程的部份授課內容寫成. 大約集結了四堂課的材料. 授課對象, 大部分是剛從高中畢業, 對電腦半知不解的數學系新生.

授這門課的時候, 我經常以交談的形式進行. 現在要書之成文, 才發現仍然很難寫得段落分明, 主題明確. 幾經掙扎之後, 決定仍按原來的漫談模式書寫. 我們從計算機程式語言的一個要素談起, 一路上涉獵些硬體梗概, 歷史故事和數學. 雖然本文的最終目的, 是突顯數學在計算上的重要性; 我同時還希望能夾帶傳播一些其他訊息. 諸如, 對硬體的認識, 可以增進我們運用計算機的效率; 對歷史的認識, 可以幫助我們理解電子計算機何以有今天的面貌.

我的學生們, 一邊學習概括的計算機理論, 一邊學習使用 C 語言. 所以這裡的舉例程式, 都是以 C 語言寫成. 有經驗的讀者一定覺得我的程式寫得有點笨. 但是我的目的在於闡述一些非常基礎的觀念, 而將程式技巧延到稍晚的課程中. 比如說, 我不用 `i++` 這個語法, 而直接用 `i=i+1`. 而且, 在這裡我也不便詳細解釋如何在各種計算機上執行這些範例程式. 我所使用的配備是一台 Sparc 10 工作站, 以及 gcc 編譯程式.

讀者會在文中發現“電子計算機”與“電腦”兩種名詞交互出現. 要特別聲明的是, 我並不認爲它倆是完全同義的名詞. 當意指機器的硬體本身時, 我頃向於說“電子計算機.” 當軟體和硬體有了美妙的結合, 以至於成了一個有點兒形而上的整體時, 我頃向於說“電腦.”

我假設讀者們瞭解數學歸納法和二進位的算術, 至少聽說過中央處理機 (CPU) 是作什麼的. 對於尚未具備此知識的讀者, 我仍希望您們對下面的某些段落產生興趣.

---

* 即將出現於數學傳播期刊.

第一講

- 程式

今天常見的計算機都跟隨 1940 年代 John von Neumann 所繪製的藍圖, 屬於序列執行類. 也就是說, 計算機的中央處理機一次只執行一個指令. 所以我們的程式語言, 也是大體上如此設計. 其實不須要特別理論說明, 直覺上大家都可以瞭解程式的寫作, 是一行一行地從上向下寫. 而基本上計算機也是一行一行地依序執行. 這種一次只作一件事的現象, 在普通的個人電腦上最爲明顯. 至於在多人或多工的作業系統上, 例如 UNIX, 電腦透過工時分享 (time sharing) 的技巧, 使得它看起來像是同時可以做很多事情.

雖然臺灣有幾台平行執行類的電子計算機, 但不是一般人有機會可以接觸到的. 雖說是平行, 其實每一個中央處理機仍是一次處理一個指令. 所謂平行, 指的是一個計算機裡面有許多個中央處理機, 而它們可以同時執行各自的指令.

但是, 如果計算機只能一根直腸子地從第一行程式執行到最後一行, 那麼它的功能就恐怕要歸零了. 這個基本的認識, 在 1840 年代已經由 Babbage 和 A. A. Byron 明確地寫了下來. 他們說:

> 自動計算機的眞正重要之處, 在於它可以重複執行一套給定的程序. 其重複次數可以在計算前確定, 也可以依計算結果而臨時決定. (附錄一)

"計算程式的核心技術在於重複," 我認爲這個說法並不誇張. 爲了要重複, 又爲了每次重複時某些參數的值可能要改變, 所以要有邏輯迴路. 每一種程式語言, 都必定會提供重複結構和邏輯迴路. 而一個程式的寫作, 基本上就是把數學的邏輯轉換爲符合一定規則的計算機指令. 所以, 數學與計算機除了有許多明確的理論關係, 更有一種形而上的關係, 值得我們注意. 那就是, 數學推理和程式寫作其實用到了人類的同一種思維能力.

我們都知道, 人類有幾種不同的思維能力; 有語言的, 肢體的, 邏輯的, 幾何的, 色彩的, 等等. 這些能力通常既不相生也不相剋, 但是其中的微妙關係是我還不能瞭解清楚的. 爲什麼我認爲數學推理和程式寫作用到同一種能力呢? 第一, 兩者都要看出問題的最癥結部份, 通常也就是最抽象的部份. 唯有掌握了這一部份, 數學定理才得以證明, 計算機程式才得已精確而且簡明. 第二, 兩者都須要絕對的邏輯正確. 這一點, 我想毋庸解釋. 第三, 兩者都須要 "逆向思考." 我要就這一點多說幾句. 各位多少都做過數學習題, 尤其是證明題. 正常情況下, 老師給我們的習題一定是利用已經學過的知識就能解決的. 可是, 即使老師把所有的等式, 不等式, 定義, 定理, 引理, 一股腦兒全列出來, 也可能有些學生不能解決眼前的習題. 要說這些學生不聰明, 這是誤解而且不公平的. 只能說這些學生可能稍欠逆向思考能力. 也許我們多少都有這種經驗, 當我們看到答案, 就捶胸頓足: 「明明我會的嘛, 只是當時沒想到.」 這裡的關鍵是, 當問題放在眼前, 所有的材料和工具也都在手邊, 您是否有能力選擇最適用的東西, 以正確而又最高效率的方式將它們結合在一起, 用以解決問題? 這就是我想要解釋的, 逆向思考的能力. 這是作證明題和寫電腦程式都須要的能力.

雖然我們的能力多半是與生俱來的, 但教育與練習總是有點兒作用. 我們從小接受數學的訓練, 其目的, 我想, 並不只是要我們能核對統一發票上的帳目, 而是要我們加強上述的幾種能力. 一旦這些能力得以發展, 倒未必要應用到數學問題上; 它將成爲一個人的內涵和本能, 隨時可能表現在日常生活中所面臨的, 種種無法預測的狀況上. 只要是從

事科學活動, 這些能力想必都是重要的; 但是我覺得, 數學與計算機理論有更清楚的類比關係.

- Babbage 和 Ada

Charles Babbage,* 英國人, 1791 年生, 享年八十. 他差不多是現代計算機歷史的開始人物, 但是帶一點點悲劇色彩. 在他以前, 有中國的算盤. 但是平心而論, 算盤實在不能稱作一個計算機; 它只是個計算輔助器. 因爲它不能自動執行任何程式, 更徨論執行不同的程式. 比 Babbage 稍早, 有 Pascal 的機械撥盤式計算器. 但嚴格來說那也不是我們今天所普遍認知的計算機. 因爲它不能因輸入不同的程式而執行不同的任務; 它只會作某些位數的正整數加減.

在 Babbage 的時代, 物理, 航海與天文等科學都有了相當的理論和經驗. 人類開始有更大的野心, 想要瞭解自然現象的細節, 更加掌握自然界的資源. 於是我們開始須要作較大量的計算. 科學家, 甚至哲學家, 開始想像, 如果有一個會自動計算的機器, 那該多好. 今天所謂的科學計算, 也在那時候開始了. 最戲劇性的例子, 該當是海王星的發現.

在 Herschel 以望遠鏡發現了天王星之後, 科學家發現其運行軌道與牛頓力學所預測的不合. 1845 年, 英國的 Adams 和法國的 Leverrier 分別以計算的結果, 斷言另一個行星的存在, 甚至預測了它的軌道. 一年後, 柏林天文臺按著計算的軌道而發現了一顆新的行星, 海王星. 這個計算問題牽涉到聯立微分方程式的數值解. Adams 發明的解法, 到今天仍出現在數值分析的教科書上.

Babbage 幾乎終身得到英國皇家學會的支持, 使他得以畢生盡力於自動計算機的設計與製造. 他的一生中, 至少企圖製造三種計算機. 因爲他的時代背景, 所想到的都是機械式的計算機. 但是他的確已經掌握了現代計算機的幾個重點; 尤其是: 可以由不同的程式而作不同的用途. 雖然 Babbage 是一位很優秀的科學家兼工程師, 可惜他的時代環境還不能配合他的發明. 這就好像一個人, 想要在橡膠, 蒸汽機都還未知之前, 就發明汽車.

事實上, Babbage 連一台機器都沒有眞的做完. 每當他快要完成一個設計的製造工作, 他就發現了這個設計的缺點和可改進之處, 進而改良甚至重塑他的想法, 然後開始下一個機器的設計. 後人從他的不斷反省更新的設計記錄中學習到寶貴的資料, 但是他自己從來沒有完成一個製造成品. 後來有人繼他的遺志而完成了他的計算機, 某些機器一直使用到本世紀初, 現在則存在博物館裡面.

年近半百的時候, Babbage 結識了當時的名才女, Augusta Ada Byron. 她是盛名詩人拜倫的女兒. 但是她在滿月的時候, 父母就離異了. 她一生中沒機會見到那位詩人父親. 後來根據她的遺囑, 她與父親並排葬在一起; 兩人都恰好各享年三十六歲. Ada 不但能詩善畫, 還是當時主要數學家 de Morgan 的學生. 她因某個機緣, 參觀了 Babbage 的工作室. 雖然零件散落滿地, 屋子中央座落著一台怪物般的半成品, 在聽到解釋之後, 她說她看到了一個 "偉大而美麗的發明."

Ada 參加了 Babbage 一部份的工作, 尤其是程式設計方面. 今天有些人認爲, 最早的計算機應該歸功給 Babbage, 而最早的程式設計應該歸於 Ada. 因而八十年代由美國國防部委託設計的一套程式語言, 就以 Ada 命名.

然而 Ada 卻把這種可以因軟體而改變硬體功能的設計理念, 歸功於法國人 Jacquard (

* Babbage 的發音與 cabbage 同韻.

附錄二). 1805 年, Jacquard 製造了可變程式型自動織布機. 它使用不同式樣的厚紙卡來控制紡錘和飛梭的交互作用, 始其編出不同花式的布料. 這個發明啓動了紡織工業的一次革命. 根據記錄, 到了 1812 年, 這種機器一共賣出一萬一千台.

第二講

- 疊代

計算機程式語言中, 最基本的重複結構就是疊代 (iteration). 我們以計算整數階乘爲例, 用它來解釋疊代的結構. 相信讀者們都知道階乘的定義.

如果不存在疊代的結構, 那麼可能我們必須要寫

```
4 * 3 * 2 * 1
```

來計算 4!, 或是寫

```
6 * 5 * 4 * 3 * 2
```

來計算 6!. 這樣寫出來的程式, 既冗長又沒有一般性.

所謂疊代, 就是重複執行一定的步驟, 只是每一次執行時所牽涉的參數可能不同. 而疊代的次數, 可以是在語法結構中就明確指定, 也可以是因每次疊代所產生的不同結果而決定. 前者通常是 for-loop, 後者通常是 while-loop.

例如我們現在要計算 6!. 我們先宣告兩個變數, 型態都是普通的整數.

```
int fac,  i;
```

用 C 語言的 for-loop 結構來計算 6! 是這樣的:

```
fac = 1;
for (i=1;  i≤6;  i=i+1) {
    fac = fac * i;
}
```

在這個程式片段的第一行, 我們先給 fac 初始值. 然後進入 for-loop. 先執行 i=1, 這一部份只作一遍. 接著開始疊代步驟, 先看看 i<=6 是否成立. 如果成立, 就執行後面一對大括號中的所有指令; 在此只有一句話, 就是 fac = fac * i. 大括號執行完了, 就作 i=i+1. 這就是一次疊代步驟的結束. 然後回到 i<=6 這一這個測驗, 開始下一個疊代步驟. 所以, 在每一次疊代步驟結束後, i 的值分別是

```
2  3  4  5  6  7
```

fac 的值分別是

```
1  2  6  24  120  720
```

剛接觸計算機程式的讀者, 看到 fac = fac*i 這樣的句子, 可能感到愕然. 在這裡, 計算機的處理方式如下: 中央處理機從記憶體中取得變數 fac 和 i 的值, 在處理機裡面作乘法計算, 然後把答案送回到記憶體, 存入 fac 所在的位置; 舊的 fac 的值就被蓋掉了.

用 C 語言的 while-loop 結構計算 6! 是這樣的:

```
fac = 1;
i = 6;
while (i > 0) {
    fac = fac * i;
    i = i−1;
```

```
}
```

在前兩行, 我們給初始值. 然後進入 while-loop. 疊代的步驟從檢查是否 $i > 0$ 開始. 如果成立, 就執行後面一對大括號中的所有指令; 在此有兩句話, 注意執行後 i 的值減少了. 大括號執行完了, 就是一次疊代步驟的結束. 然後回到 i>0 這個測驗, 開始下一個疊代步驟.

觀察上面的 for-loop 的例子, 很容易可以看出來它一定執行六遍. 但是在 while-loop 的例子, 這個事實並不非常清楚. 理論上, 任何 for-loop 都可以改寫成 while-loop, 反之亦然. 但實際上, 可能某一種結構會比另一種寫起來更自然. 可惜在我們這個簡單的例子上, 看不太出來.

- 資料的型態

前面的例子裡, 我們看到這句指令

```
int fac,  i;
```

這叫做宣告. 像 C, FORTRAN 這種高階程式語言, 要求寫作者在使用變數或子程式之前, 先行宣告. 以下, 我想解釋, 何以須要宣告變數, 對我們的影響又如何.

各位可能已經知道, 任何資料型態, 諸如整數, 小數, 文字, 都以電子數位訊號儲存於計算機中以備處理. 我們可以說, 今天所有電子計算機裡面的記憶單位, 都恰有兩個可能的狀態: 開或閉, 通或斷, 滿或空. 在數學模型上, 就是零或壹. 我們不管製造技術, 只看數學模型.

當我們寫出一個整數, 我們用的是十進位制. 如果要把這個整數輸送給現代的電子計算機處理, 首先要把它改寫成二進位制. 假設讀者已經瞭解這個轉換過程. 例如,

$$5 \longrightarrow 101, \quad 6 \longrightarrow 110.$$

現在我們面臨三個技術性問題. 第一, 計算機的記憶體有限, 它不能處理任意長的整數. 所以我們要規定一個數或一筆資料的長度上限. 第二, 既然所有資料都已經轉成零或壹, 計算機不能分辨誰是誰. 設身處地的想一想. 假設您看到一串零零壹壹的記號, 您怎麼知道這是整數呢, 還是幾個英文字母的編碼, 甚至中央處理機的內部指令? 所以, 當一個程式拿到一筆存在計算機中的資料, 它必須要知道這筆資料的型態, 以便解讀. 第三, 怎麼記錄負整數?

現代常見的計算機程式語言, 例如 C, Pascal, FORTRAN, 都是將第一, 二項問題一併解決. 這些語言賦予每一筆資料一個固定的型態. 而當這個型態確定了之後, 它的資料長度與解讀方式也都一起確定了.

例如

```
int i=1;
unsigned int fac;
```

這兩句話告訴了計算機, 此程式將要用兩個變數. 一個叫做 i, 其型態是普通整數 (可能是負整數); 另一個叫做 fac, 其型態是正整數. 兩者的長度都一定是四個字元 (byte), 或是三十二個位元 (bit).

一個字元是由八個位元組成的. 雖然位元是計算機的最小儲存單位, 但是它本身的資料容量太小了: 只能儲存兩種狀態. 所以, 由於經驗的累積, 現在的電子計算機大多以字元

作爲最小的使用單位; 由於有八個位元, 它可以儲存二百五十六種狀態. 順便一提, 最早的電子計算機都以四十左右個位元作最小使用單位, 其原因是當時的應用以計算爲主, 每個字元就儲存一個數. 後來受到美國標準資訊交換碼 (ASCII) 的強烈影響, 定成以八個位元爲最小使用單位. 當時的設計者似乎根本沒想到電腦會成爲像今天一樣的世界性工具, 而記憶體會像今天一樣的便宜又快速. 所以當時的八位元設計, 到了今天變得不方便卻也很難改正. 不方便的原因是, 現在幾乎全世界的民族都參與了電子通訊網路, 爲了要使計算機處理幾乎全世界的文字, 八個位元的內部編碼設計顯然不敷使用, 十六位元看來比較合理.

在記憶體中, 每個字元有它自己的地址, 而中央處理機有足夠的線路, 可以根據地址直接接觸任何一個字元. 這種設計, 現在稱作隨機存取 (random accessible), 但是早期稱作平行存取. 1940 年代的第一代電子計算機不能隨意找到記憶體中的任一個地址, 它必須按照地址順序, 一個一個地讀過去.

int 與 unsigned int 的解讀方式當然不會相同. 當一個程式在計算機裡開始執行的時候, 作業系統就依照宣告的指令, 在記憶體中保留一串恰當長度的字元給這些變數, 並且記得兩件事: 一, 每個變數在記憶體中所存放的位置; 如果長度超過一個字元, 則記得其第一個字元所在的地址. 二, 每個變數的型態, 以備存取的時候知道該用多少字元, 該用什麼解讀方式.

正整數型態比較單純, 因爲它不考慮負號. 就用我們一般知道的方法, 在十進制與二進制之間互換. 例如 5 的表現方式就是

$$00000000000000000000000000000101$$

所以正整數的型態所能記錄的最小數就是 0, 最大的是

$$2^{32} - 1 = 4294967295$$

如果您不小心用這種型態來記憶一個太大的數, 也就是寫成二進位時超過三十二位的數, 就會發生溢位 (overflow) 的現象. 計算機可能會無視於溢出的位數, 因此而解讀出不正確的整數. 例如 2^{32} 寫成二進制應該是一個 1 後面跟三十二個 0. 但是 unsigned int 只記錄了右邊的三十二個位元, 所以這個數就變成了零.

但是也有套裝軟體或程式語言, 容許任意轉換資料的型態或長度. 例如 Maple. 這類軟體容許您計算任意長的整數, 例如 2^{300} 也不是問題. 問題是您的計算機也許沒有足夠多的記憶體, 讓它作這種揮霍的計算. 此處不多談.

- 零合記數法

至於記錄一個普通的整數就比較麻煩了. 因爲我們有負的整數. 最直覺的方法就是, 利用右邊的三十一個位元來記錄整數的絕對值, 用最左邊的一個位元來記錄正負號; 正號是 0, 負號是 1. 這個直覺的方法基本上是對得, 但是今天的計算機多半使用另一種方法. 我們稱它爲零合記數法, 讓我們來欣賞一下.

我們用上述直覺的方法來記錄正的整數. 所以這時候我們比 unsigned int 少了一個位元. 因此, 在正數這邊, 只能記錄介於 0 和 $2^{31} - 1$ 之間的整數. 至於負數, 要將我們的直覺改變一點: 首先, 若第一個位元是 1, 代表這是個負數. 但是怎樣知道這個負數的絕對值呢? 我們把它的每一個位元作零壹互換, 然後加一. 所以,

$$11111111111111111111111111111011$$

就代表了 -5.

當然, 我們知道

$$5 + (-5) = 0$$

在這種記數法之下, 這個基本的等式成立. 試作以下的二進位加法

$$\begin{array}{rl} & 00000000000000000000000000000101 \\ + & 11111111111111111111111111111011 \\ \hline & 00000000000000000000000000000000 \end{array}$$

本來最左邊應該還有一個進位的 1, 但是因爲溢位而被忽略了. 如果用我們上述的直覺方法來記錄負整數, 事情就沒這麼簡單了: +5 和 -5 的二進位相加結果不會是零.

讓我們再把一個小小的細節捕上. 想想該怎樣解讀第一位元是 1, 後面都是 0 的二進位數呢? 用直覺法, 這個數應是 -0. 用上述的零合法, 它的絕對值還是自己. 可是, 這樣一來就有兩個不同的編碼而代表同一個數, 這太浪費了. 這個數, 我們將它定作是 -2^{32}. 因此, 整數型態的資料, 可以記錄介於 -2^{32} 和 $2^{31} - 1$ 之間的 2^{32} 個整數. 我們稱這個區間叫 "可記錄區間."

在電子計算機裡面, 二進數加一或減一的過程都可以用很簡單的電路板來完成, 其基本動作就只是零和壹的比對和互換. 簡單地說, 加一的步驟就是從右向左, 逐一把壹換成零, 一直到第一個零被換成了壹爲止. 前面已經說明了, 以零合記數法, 如何從負數變號成正數. 反過來, 把一個正整數減一再逐一作零壹互換, 我們可以將一個正數變號成負數. 總之, 我們可以很快得將一個整數變號. 例如將 +5 變成 -5, 或反之.

由於零合的特性, 再加上變號容易, 所以在這種系統之下我們不須要整數減法電路. 當兩個同號的數相減的時候, 在電腦裡面其實可以將其中一個變號再相加. 零合法的另一個好處是, 很容易在電腦裡面察覺加法溢位.

如果兩個整數都在可記錄區間. 若它們異號, 則相加的結果必定仍在區間內, 所以不會溢位. 若它們同號, 就有可能發生加法溢位. 例如 $(-2^{32}) + (-1)$. 因爲同號的數相加之後必定還是同號; 又由於, 在此記數法之下, 發生溢位時一定會影響最高位的那個位元. 所以我們有一個簡單的方法檢查加法溢位: 只要是兩個同號整數相加之後變號了, 那就一定有溢位發生. 這時候電腦應該發出一個錯誤或警告的訊號.

在這一節裡, 我們看到一些非常簡單的數學觀念應用在計算機科學上. 同時, 我希望這些簡單的細節能夠減少電腦的神秘感.

第三講

- 階乘

現在我們瞭解了整數型態的記數法. 所以知道, 使用 C 或大部分其他的程式語言, 我們不能計算任意大的階乘. 其實, 別說任意大了, 稍微估計一下, 我們會發現, 根本就沒幾個階乘可以算. 上一節中, 我們看到, 整數型態能記錄的最大數是 $2^{31} - 1$. 因爲 $log\, 2$ 大約是 0.3, 這個最大數應有 $31\, log\, 2 + 1$ 大約是十個十進位數. 而 $6! = 720$ 有三位數, 此後每加一個階乘, 就至少多一位數. 所以最多只能算到 13!, 以後就溢位了. 實驗得到:

$$12! = 479001600, \quad 13! = 1932053504.$$

這 $13!$ 的計算值顯然是錯的. 因爲 $13! = 13 \cdot 12!$, 但是計算值甚至還不到 $10 \cdot 12!$.

像這種問題, 有沒有辦法解決呢? 簡單的說, 沒有. 像這一類的問題, 是數學與計算機的本質困難. 階乘的值, 本來就這麼大, 並不是可以換一個計算法就變小了. 在計算科學的領域裡, 還有許多這類本質性的困難, 並不是改變計算法就能解決的. 而計算工作者的要務, 是認清哪些爲計算性的困難 (而非數學性的困難), 並在演算法上尋求改善.

基本上, 想要計算更大的階乘, 必須訴諸硬體的擴大. 也就是說, 要使用更多的位元來儲存和計算整數. 最簡單的方法, 就是花錢買能作多重精度 (multi-precision) 的計算軟體, 例如 Maple. 它可以很快得算出來

$$\begin{aligned} 100! = 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000 \end{aligned}$$

甚至更大的階乘. 至於答案嘛, 信不信由你.

但是像 Maple 這樣的套裝軟體未必適合拿來作科學計算. 有時候我們還是必須或喜歡自己寫程式. 這時候, 比較簡單的方法就是用程式語言所提供的其他較多字元的整數型態. 第一個想到的, 該是正整數型態 (unsigned int), 因爲階乘都是正數. 但是這實在幫助不大, 因爲這種型態才多出一個字元, 在十進位才多一倍 (乘以 2). 實驗結果也顯示, 即使是用正整數型態, 我們仍然只能正確地算到 $12!$.

比較有實際效用的, 是使用長整數或雙精度浮點數型態. C 語言的長整數叫做 long. 但是未必每個 C 的編譯程式 (compiler) 都懂得這種型態. 有些較小型機器上的編譯程式, 會給一個訊息, 告訴您它不知道什麼是長整數; 有些會自動把它認作與普通整數一樣. 某些大型機器會使用六個字元, 也就是四十八個位元, 來記錄長整數. 其解讀方法還是和普通整數一樣, 只是使用更多的記憶體資源, 而使得長整數大約可以記錄十五位十進制的整數. 即使這樣, 也只能幫助我們多算幾個階乘.

可能只有少數的編譯程式懂得長整數型態, 但是幾乎所有的編譯程式都懂得雙精度浮點數. 利用這種型態, 可以得到比長整數計算還要好的效果, 只是在計算機的效率上稍差一點. 但是利用雙精度浮點數來計算大的整數, 步驟稍微複雜, 而且要更加小心. 我們留到下一節再談.

如果前兩種方法都不能用, 您可能就得自己寫乘法的程式了. 其實也不太難, 就是我們在小學裡所學的乘法. 在稍早幾年, 很多計算機概論或是類似的初級教科書, 都會介紹這種程式的寫法. 近來, 可能這一類的套裝軟體愈來愈多了, 就比較少人須要學習這類技術.

- 浮點數

前面我們介紹了整數在計算機中的記錄方法. 整數固然是最基本的數字型態, 但是在科學計算上, 其用途是很有限的. 您一定很難想像, 哪一個有關科學或物裡的計算問題, 可以從頭到尾都只用整數就算完了?

數學上, 除了整數之外, 我們至少還有有理數, 實數. 我們不能在此詳述這些數的意義. 讓我們總括地說, 計算機還須要記錄與處理實數. 但是很多實數, 在寫成十進位時, 有無窮多位小數部份. 就例如 $\frac{1}{3}$. 而計算機只有 (非常) 有限的資源, 它只能處理有限的資料.

這是我們在前面考慮過的. 因此, 當我們說計算機處理實數, 其實的意義是, 它處理實數的有限位數的近似值.

就像是整數的記錄和解讀一樣, 我們也必須定義一種資料型態, 並規定其位元長度與解讀方式, 用以記錄實數. 而且, 無可避免的, 我們只能記錄有限多個有限長的實數. 至於那些像三分之一這種 (在十進位制中) 無限長的實數, 就只好作一些逼近的工作, 通常是四捨五入 (round-off).

早期的計算機, 使用所謂的定點數 (fixed-point numbers) 來記錄小數. 也就是, 其資料型態已經確定了小數點之後有幾位數. 在金融界, 通常這種型態是夠用的. 因爲, 我們通常算錢, 只會準確到小數點下兩位; 而利率, 通常也只會準確到小數點下四位. 而且金融界的計算問題常常可以在事前估計最大可能用到的數. 但是這些現象, 並非普遍發生於一般的計算問題中. 如果固定了小數點的位置, 一則不能視情形需要而增加小數的位數; 二則, 如果小數點下的後幾位都是零, 這些零都不是有效數字, 卻仍要佔據記憶體, 使得整體的有效位數降低. 比較有效率的作法是, 規定一定位數的有效數字, 再用另一個參數來決定小數點所在的位置. 也就是, 小數點的位置是浮動的, 故名浮點數 (floating-point numbers).

計算機用二進位數記錄浮點數的作法, 就像我們使用十進位制的科學記數法. 例如, 與其寫六個零來表示一千七百萬 17000000, 我們寫:

$$.17 \times 10^8$$

在這裡, 10 叫做基數 (base number), .17 叫做底數 (mantissa), 8 叫做指數 (exponent). 我們要求

$$\frac{1}{\text{基數}} \leq \text{底數} < 1.$$

所以底數的第一位數必定不是零. 指數一定是個整數. 指數爲零, 則代表小數點在原定位置; 也就是, 底數的第一位之前. 正的指數 n, 表示把小數點向右移 n 位; 負的向左移.

一般的單精度 (single precision) 浮點數型態, 長度定爲三十二位元. 在 C 語言中, 稱作 `float`. 基數, 當然是 2. 保留一個位元, 用來表示正負號. 二十三個位元用來記錄底數, 注意, 因爲小數點下第一位必定是 1, 所以不必紀錄. 剩下的八個位元用來記錄指數. 由於指數是個整數, 可以用零合計數法, 但也有可能用了其他的計數法, 不再詳述了.

假設指數是 0. 在這個指數之下, 最小的浮點數是二進制的 .1, 也就是 $\frac{1}{2}$. 最大的是

$$.11111111111111111111111$$

也就是

$$(\frac{1}{2}) + (\frac{1}{2})^2 + \cdots + (\frac{1}{2})^{24} = 1 - (\frac{1}{2})^{24}.$$

所以這些浮點數均勻分佈在

$$[\frac{1}{2}, 1 - (\frac{1}{2})^{24}]$$

這個區間裡面.

現在, 讓我們想像浮點數在實數線上面的分佈. 前面已經解釋, 當指數是 0 的時候, 一共有 2^{23} 個浮點數, 以 2^{-24} 爲間隔, 均勻分佈在

$$[\frac{1}{2}, 1)$$

這個區間裡面. 注意這個區間符號, 左邊是閉區間, 右邊是開區間. 當指數是 1 的時候, 相當於把所有指數是零的浮點數都乘上 2. 也就是, 仍有 2^{23} 個浮點數, 這時以 2^{-23} 爲間隔, 均勻分佈在

$$[1, 2)$$

這個區間裡面. 同理, 當指數是 -1 的時候, 相當於把所有指數是零的浮點數都乘上 $\frac{1}{2}$. 這 2^{23} 個浮點數, 就以 2^{-25} 爲間隔, 均勻分佈在

$$[\frac{1}{4}, \frac{1}{2})$$

這個區間裡面.

現在, 我們可以有一個比較整體的認識了. 浮點數並不是均勻分佈在實數線上, 也不是胡亂散落在實數線上. 它們是一段一段地均勻分佈在這些區間裡面:

$$[2^{k-1}, 2^k)$$

在每個這樣的區間裡面, 都均勻分佈了 2^{23} 個同指數的浮點數. 其中 k 代表指數; 由於它是以八個位元所記錄的整數, 所以

$$-128 \leq k \leq 127.$$

這是正的浮點數部份, 而負浮點數完全是正浮點數對零的鏡射. 注意, 以上描述的浮點數並不包括零. 零其實是個整數, 我們不去費心此中的細節.

如果有一個絕對值不大於最大浮點數的實數 x, 而我們要用浮點數來記錄它. 基本的機率概念告訴我們, x 恰好是一個浮點數的機率, 是零. 所以, 實際上的作法是, 選擇一個最接近的浮點數, 記作 $fl(x)$, 在電腦裡面記錄或代表 x. 於 x 與 $fl(x)$ 之間, 幾乎永遠有一個誤差. 在幾乎所有牽涉實數與浮點數的計算過程中, 都一定會發生這種誤差. 這種幾乎無法避免的誤差, 稱作 "機器誤差;" 其發生乃源於計算機的有限性, 而無關於計算的數學方法. 但是一個計算方法的設計, 必須要確定機器誤差不至於過分膨脹, 而搞壞了一整鍋粥. 這種計算方法的性質, 叫做穩定性 (stability); 它是計算研究者最重要的課題之一.

如果一個計算的數 (的絕對值) 超過最大可能的浮點數, 我們稱它爲溢位; 如果小於最小的浮點數而又不是零, 稱爲虧位 (underflow). 在計算的過程當中, 如果發生溢位或分母的虧位, 計算機會視情況用 NaN (Not a Number) 或 Inf (Infinity) 來記錄, 而讓計算繼續下去. 但某些虧位的情況, 會被當作是零. 有關這些情況的處理方式, 有一套 IEEE 的標準作爲準則. 這個準則源自多年的經驗累積, 所以通常是無害而且方便的. 現在大部分的計算機, 在浮點數計算方面, 都符合了 IEEE 的標準.

浮點數的計算, 比整數複雜得多. 前一代的中央處理機通常只能作整數計算, 而把浮點計算交由軟體或是一個輔助晶片來處理. 例如, 常見的 Intel 80386 就有一個浮點計算的

輔助晶片 80387. 現在由於超大型積體電路的製造技術一再進步, 浮點計算的電路也可以裝在中央處理機裡面了.

我們無法在此敘述浮點計算的細節, 但是我想在這一段裡解釋浮點加法的原則. 最重要的關鍵是, 在作加法之前, 要先統一指數, 再對齊小數點. 讓我們先用比較熟悉的十進位科學記數法來作例子. 試作以下的加法:

$$.678 \times 10^{-1} + .12345 \times 10^{3}$$

我們得先把其中一個數寫成不標準的科學記數法, 以求兩者有同樣的指數. 我們選擇把大的指數換成小的:

$$.12345 \times 10^{3} = 1234.5 \times 10^{-1}$$

現在這兩個數有相同的指數, 我們對齊底數的小數點, 作加法:

$$\begin{array}{rr} & .678 \times 10^{-1} \\ + & 1234.5 \quad \times 10^{-1} \\ \hline & 1235.178 \times 10^{-1} \end{array}$$

再改寫成標準的記數法 $.1235178 \times 10^{3}$.

注意, 在上面的例子裡, 兩個輸入數的底數各是三位及五位; 但是相加的結果有七位底數. 這種位數增加的情形當然也會在浮點數的計算中發生. 整個計算過程, 是在中央處理機裡面完成, 它會暫時允許較多的底數位元以求計算的準確. 當計算的結果, 從中央處理機轉移到記憶體中儲存的時候, 就必須要轉成某種資料型態了. 因此, 計算結果即使沒有溢位或虧位, 仍有可能因底數的位元太多而發生誤差. 例如, 前一段的例子中, 如果要求每一個數只能有五位底數, 而且處理超長的方式是四捨五入, 則計算結果就會被記成 $.12352 \times 10^{3}$. 造成 0.0022 的絕對誤差, 或是約 $.18 \times 10^{-4}$ 的相對誤差. 這一種誤差, 也歸類爲機器誤差.

基於這個認識, 我們就不難瞭解爲什麼在電腦中計算 $1 + x$, 即使 x 是一個正浮點數, 其結果也未必大於 1. 例如, 如果 x 和 y 都是單精度浮點數型態的變數; 也就是, 底數有二十四位. 若令

$$x = (\frac{1}{2})^{24}, \quad y = 1 + x,$$

則 y 得值仍然是 1. 原因是, 以二進位的記號寫出來, y 的計算結果是

$$\begin{array}{rr} & 1000000000000000000000000. \times 2^{-23} \\ + & .1 \times 2^{-23} \\ \hline & 1000000000000000000000000.1 \times 2^{-23} \end{array}$$

計算結果的底數有二十五位, 以浮點數的型態記錄到記憶體中 y 的位置時, 最後一位被捨去, 結果就成了 1. 但是, 如果

$$x = (\frac{1}{2})^{23},$$

則 $y = 1 + x$ 的結果就會大於 1.

所謂雙精度 (double-precision) 浮點數, 就是用兩倍的位元來記錄浮點數: 六十四位元. 其中五十二位元用來作底數. 在十進位下, 它可以記錄十六位有效數字. 單精度的浮點數則只能記錄七位. 如果一個整數的有效數字部份, 換成二進位之後, 在五十三位以內, 則可以把它記錄成雙精度的浮點數, 而不會造成誤差.

由此可見, 運用雙精度浮點數, 其實可以比整數型態處理更多的整數. 但是以浮點數記錄並計算整數時, 要特別小心機器誤差. 因爲計算機不會通知使用者, 什麼時候發生了這種誤差. 基本守則是, (十進位) 有效數字在十六位以內, 才是安全的. 例如, 以雙精度浮點數計算得

$$21! = 51090942171709440000$$

這是對的. 其實, 雖然 22! 有十八位的有效數字, 我們發現它以雙精度浮點數的計算結果仍是對的. 這是因爲我們還不清楚某些浮點計算法的細節. 但是 23! 的計算值就錯了.

第四講

- 子程式

疊代的結構, 只能在一個程式裡重複使用. 其實很多計算或資料處理的程序, 都會在不同的問題中重複出現. 如果我們每次都把它們重寫一遍, 並且重新檢查一遍它們的正確性, 那就太沒效率了. 子程式 (subprogram) 就是用來處理這種, 在不同程式中重複使用的情形. 原則上, 任何一個可以獨立出來的計算或資料處理的程序, 都應該寫成子程式, 經過仔細檢查之後保存起來, 以備日後重複使用. 呼叫 (call) 這個子程式出來替它工作的程式, 叫作控制程式 (control program). 子程式與控制程式之間, 只是相對的關係; 一個子程式如果再呼叫了另一個子程式, 前者就成了後者的控制程式. 很明顯的, 一個程式可以呼叫很多子程式, 但是它只有至多一個控制程式.

例如我們造一個子程式, 用來計算階乘.

```
int factorial(int n) {
    int fac,  i;
    fac = 1;
    for (i=1;  i≤n;  i=i+1) {
    fac = fac * i;
    }
    return fac;
}
```

這個 C 程式片段的意思是, 定義一個子程式, 它的名字是 factorial; 它須要一個輸入參數, 型態爲整數而在此稱之爲 n; 它執行結束後, 將交回一筆資料, 其型態是整數. 隨後的一對大括號中, 就是這個子程式的內容. 注意, 由控制程式交付的參數 n 不必在子程式中宣告. 在最後一行的 return 指令, 表示要把 fac 這個變數的值送回給控制程式.

提醒讀者, 此處的 C 程式均依 ANSI C 的標準寫成. 舊型的 C 格式現在習稱爲 K&R C. 如果您的編譯程式會抱怨以上的子程式, 它也許是 K&R C 的編譯程式, 您可能該換掉它了. 暫時您可以將第一行改寫成

```
factorial(n) {
```

而躲避一些麻煩.

那麼, 在任一個程式裡面, 可以寫

```
tmp = factorial(6);
```

或者

```
tmp = factorial(k) * factorial(k−1);
```

其中 k 是一個已經有給定值的整數變數. 諸如此類. 總之, 這個子程式可以在任何其他程式中被使用. 比如說, 計算泰勒展開式的數值, 或是計算組合數

$$\binom{n}{k} = \frac{\texttt{factorial(n)}}{\texttt{factorial(k) * factorial(n-k)}}$$

一個子程式可以再呼叫其他子程式; 相對地, 自己就成爲一個控制程式. 例如下面這個子程式就是計算組合數的, 它須要兩個輸入參數:

```
int choose(int n,  int k) {
    int tmp;
    tmp = factorial(n) / (factorial(k) * factorial(n−k));
    return tmp;
}
```

在這裡我要聲明, 我們必須忽略兩個細節. 第一, 我們暫時不考慮控制程式呼叫子程式的操作手續. 這將牽涉到計算機的作業系統. 總之, 通常有三種方法, 可以使控制程式知道去哪裡找子程式. 這些方法隨著不同的作業系統而有些許不同.

第二, 我們不詳談程式的強韌性 (robustness). 當我們寫作一個程式, 首先它當然得是正確的. 但是在實際情況下, 常常光是正確的還不夠. 它還要經得起"折磨;" 也就是, 要有強韌性. 比如說, 上面 factorial 的例子, 如果使用者不愼輸入了一個負數, 那該如何? 您的處理方式, 就不單是數學的考慮了. 您應該選擇一個正確而且最方便的處理方式. 這種例外情形的處理, 也許超出了科學而須要許多經驗法則. 因爲它牽涉廣泛, 必須考慮周全. 例如, 如果使用 choose 時, 輸入的 k 比 n 大, 就會輸入一個負數給 factorial. 一般說來, factorial 不知道如何處理負的輸入值. 於是, factorial 的強韌性就間接地影響了 choose 的功能.

- 控制流程

想像一個計算機的工作模型: 基本上, 它的工作指令記錄在一條帶子上 (管它是紙帶還是磁帶), 帶子劃分成一格一格的, 每一格寫著一個指令. 我們稱它作"控制帶." 另外想像一條"資料帶," 相當於電腦的記憶體. 爲了方便, 您可以想像, 控制帶是水平放著, 第一格在最左邊. 通常電腦依照控制帶的順序, 執行完一格的指令, 就移到右邊一格去執行, 依此類推. 必要時, 電腦會依控制帶的指示到資料帶去存取資料. 當邏輯迴路發生的時候, 電腦可能從控制帶的一格跳到另一格, 而不依慣例走到右邊的格子. 當疊代發生的時候, 電腦可能在某一串格子當中反覆執行.

以上所述的帶子觀念, 起源自英國的數學家 Alan Turing. 他生於我們的民國元年, 享年四十二歲. 在二十四歲的時候, 他發表了一篇論文, 基本上在討論可計算和不可計算的數. 從這裡, 引發他一連串在計算理論上的貢獻. 他的機器模型, 現在稱作是圖靈機 (Turing machine); 而他的理論, 現在發展成一門學科, 稱作自動機理論 (automata). 在

1950 年前後, 他在英國的 Manchester 實際參與了電子計算機的設計與製造. 這台機器的設計, 是針對邏輯問題, 而不是計算問題. 也許這是歷史上的一個重要腳步: 人們開始意識到, 計算機不只是能作計算, 它還可以拿來處理一些比較須要智慧的事情.

上述的流程控制帶和指令格, 在某些課本裡面有嚴格的規定; 而且在更抽象的結構之中, 我們不須要另一條資料帶. 但是在這裡, 我只想借用這個模型, 不想嚴肅地符合那些遊戲規則. 現在, 讓我們把這個帶狀模型推展到二維空間.

當一個指令呼叫子程式的時候, 想像在這個格子上方, 堆置 (stack) 另一條帶子. 說到堆置, 它不同於排隊 (queue). 前者是譬如置薪, 後來居上, 先進去的後出來; 後者像是 (正常的) 排隊買票, 先進去的先出來. 子程式的控制帶, 被堆置在控制程式之上, 也是從左向右, 一一執行格子裡的指令. 爲了方便起見, 我們假設子程式有它自己的資料帶. 當子程式啓動的時候, 它從控制程式抄過來某些參數, 放在自己的資料帶上. 當子程式的控制帶跑完了, 就把它拿掉, 回到控制程式的格子繼續向右邊的格子進行. 依此類推, 如果子程式又呼叫了另一個子程式, 那就堆置了兩條控制帶在上方. 如此這般, 可以一直往上面堆置帶子. 在下方的是它上方的控制程式, 而上方是下方的子程式. 電腦的工作流程, 總是要把最上面的帶子最先結束, 拿掉, 降到下一層, 如此一層層地把控制權降到最下面的主程式.

- 遞迴

重複的形式有兩種, 一是疊代, 二是遞迴 (recursion). 疊代的結構比較簡單, 前面已經介紹過. 遞迴則依賴於編譯程式, 視其能否容許一個子程式呼叫它自己.

在思考模型上, 疊代的控制過程, 仍然是一維空間; 而遞迴, 因爲其結構是自我呼叫的子程式, 可以想像成二維空間. 以下我們仍用階乘作爲例子來表現遞迴的結構.

數學上, 階乘可以由遞迴公式定義:

$$n! = n \cdot (n-1)! \qquad n\text{是個正整數}$$

所以, 如果我們知道什麼是 $(n-1)!$, 那就知道了 $n!$. 依此遞減, 我們須要一個初始值. 通常, 定義 $0! = 1$. 比如說, 什麼是 $4!$? 根據遞迴定義, $4! = 4 \cdot 3!$. 我們還不能執行這個乘法, 因爲 $3!$ 還未知; 再用一次遞迴定義, 得到 $4! = 4 \cdot (3 \cdot 2!)$. 我們仍然得不到答案, 因爲 $2!$ 還未知; 再遞迴一次, 得到.

$$4! = 4 \cdot (3 \cdot (2 \cdot 1!))$$

再來一次,

$$4! = 4 \cdot (3 \cdot (2 \cdot (1 \cdot 0!)))$$

現在我們可以開始作乘法了; 因爲 $0! = 1$ 是知道的. 我們從最內層的括號開始作乘積, 一一解開括號. 這種作法, 就叫做遞迴.

在計算機上應用遞迴公式, 就要先定義一個子程式; 例如 `Factorial(n)`. 當 $n > 0$ 的情形, 就要算 $n \cdot$`Factorial`$(n-1)$, 當 $n = 0$ 就給定 `Factorial(0) = 1`. 以下是一個完整的子程式:

```
int Factorial (int n) {
    int fac;
    if (n == 0) {
```

```
        fac = 1;
        }
        else {
        fac = n * Factorial(n-1);
        }
        return fac;
    }
```

在這裡我們第一次使用了 if-else 的邏輯結構. 注意 == 這個符號. 這是一個邏輯運算, 意思是左邊是否等於右邊. 寫 C 語言常犯的錯誤之一, 就是把 == 誤寫成 =. n==0 的意思是, 檢查 n 這個變數的值, 看看是否等於 0; 而 n=0 的意思, 是把 n 這個變數的值定成是 0.

如果有一個程式, 呼叫 Factorial(4). 那麼想像在這個程式的控制帶上面, 會加上一條 Facorial 的控制帶, 它的輸入值是 4. 但是它又必須呼叫 Facotrial(3), 以至在它上面再加一條 Factorial 的控制帶, 這一回輸入值是 3. 依此類推, 往上一共堆置五條帶子. 最上面一層的輸入值是 0, 所以它可以不必呼叫其他子程式而輸出答案, 1, 然後結束, 控制帶被拿掉. 它輸出的數字, 就掉到下一層 (由於 return 指令), 於是這一層的工作可以繼續, 輸出 2 然後結束. 這樣一層一層從上到下輸出結果的步驟, 就像在數學上, 一層一層地從內向外解開括號一樣.

利用遞迴作計算, 是一個優雅的技術. 使用遞迴寫出來的程式, 通常比用疊代結構的更爲簡潔. 但是電腦可能要多花一點力氣, 來處理比較複雜的控制流程. 也許因爲這個理由, 並不是所有的程式語言都容許遞迴的結構.

- 組合數

前面寫的 choose 子程式, 根據組合數的定義, 並藉用 factorial 來作計算. 但是我們也看到了, 階乘數通常太大了, 以至於很難在計算機中運算. 比如說, 如果用這個原始的 choose 子程式, 我們甚至不能算 $\binom{13}{1}$. 因爲, 如果只使用普通整數型態, 我們不能算 13!. 可是 $\binom{13}{1}$ 是一個很小的數: 13. 所以, 像這種問題, 就不是本質性的了; 這是一個可以藉更優良的計算法而解決的問題.

我們可以證明, 給定任一個正整數 n, 和一個整數 $0 \leq k \leq n$, 組合數 $\binom{n}{k}$ 一定是一個整數. 換句話說, 以下這個分數

$$\frac{n(n-1)\cdots(k+1)}{1\cdot 2\cdot \ldots \cdot (n-k)}$$

一定整除. (附錄三)

基於這個事實, 我們可以改寫 choose 子程式, 使它不依賴於階乘, 而且不容易溢位. 單純的想法是, 把分子和分母的整數先乘出來, 再相除. 這樣作雖然比直接用階乘稍好, 但是不夠好. 我們應該把這個分數, 保持得愈小愈好; 不要造出不必要的大分子或大分母. 想法是這樣的:

$$\frac{n}{1}\cdot\frac{n-1}{2}\cdot\frac{n-2}{3}\cdot\cdots\cdot\frac{k+1}{n-k}$$

一定是個整數. 以下就是我們的新的 choose 子程式.

```
    int choose(int n,  int k) {
```

```
    int i,  j,  tmp;
    tmp = 1;
    j = n;
    for (i=1;  i≤n−k;  i=i+1) {
    tmp = (tmp ∗ j) / i;
    j = j−1;
    }
    return tmp;
}
```

注意, 要先乘分子, 再除分母.

如果用舊的 choose 子程式, 最大的可計算組合數是 $\binom{12}{6}$. 換成新的作法, 我們可以算到 $\binom{28}{14}$. 如果換成雙精度浮點數計算, 舊的 choose 子程式能算到

$$\binom{40}{20} = 137846528820$$

把這個新的 choose 子程式改成雙精度浮點數的計算, 最大的可以算到

$$\binom{54}{27} = 1946939425648112.$$

我們仍不考慮強韌性, 但是我想指出兩個效率的問題. 第一, 在這個 for-loop 裡面, i<=n-k 要檢查 n-k 遍. 所以 n-k 這個減法就要算 n-k 遍. 雖然是個小事, 但值得改善. 我們可以多宣告一個整數, 例如

```
int limit;
limit = n−k;
```

然後在 for-loop 裡面說 i <= limit. 今天大部分的編譯程式都有選擇性的優化 (optimization) 功能; 它們會看出這種浪費計算時間的小瑕疵, 而自動予以修改.

第二, 在數學課上, 我們一定都學過這個等式:

$$\binom{n}{k} = \binom{n}{n-k}$$

在原來的程式裡, $\binom{n}{1}$ 的計算大約須要 $2n$ 次乘除法. 若是應用這個數學性質, 使得電腦知道去算 $\binom{n}{n-1}$, 那就只須要兩次乘除法. 當然, 要加上這個性質的考慮, 必須要加上一段邏輯迴路, 使得整個子程式多了一些附額 (overhead). 比如說, 加上這一段:

```
if (k < n/2)
    limit = k;
else
    limit = n−k;
```

但是一般經驗認爲, 這種附額是值得的.

今天的電子計算機都有驚人的執行速度, 以至於學生們可能忽略了這些看似微不足道的效率問題. 但是, 所謂聚沙成塔, 在一個數學或統計問題裡面, 可能要重複某個計算步驟數百萬, 千萬次. 這時候, 一個關鍵性子程式的一點點效率上的不同, 可能就是整個程式的優劣所在. 所以, 我認爲, 優良的數學基礎, 適當的 (不必太多) 撰寫程式經驗, 再加上一點點追求完美的執著, 是造成一個優秀程式設計師的要素.

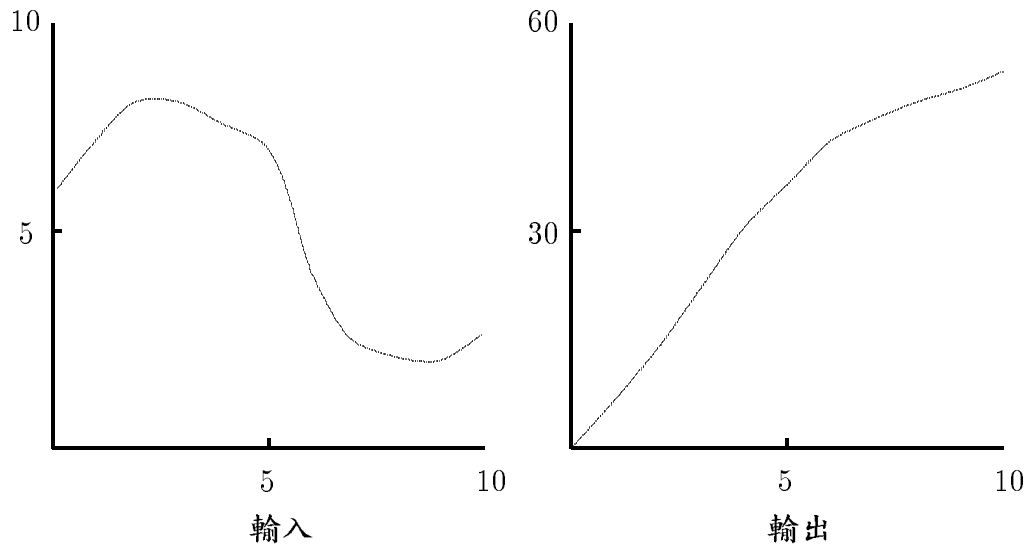
- 類比計算機

其實計算機未必一定是數位 (digital) 型的, 也有類比 (analog) 型的. 所謂類比型, 基本上就是能作連續形態的輸出. 比如, 類似心電圖在一個網格紙上畫出連續的曲線, 再從網格上讀出數據. 這種計算機從古早就用來作積分.

最早有記錄的類比積分器出現於 1814 年. 1855 年, 寫下電磁波方程的物理學大師麥斯威爾 (J. C. Maxwell) 因工作需要也發明了一個同類的計算器. 這個年代, 也就是 Babbage 致力於發明計算機的時代. Maxwell 當然也像當時的許多科學家一樣, 開始感到自動計算機的必要性. 他曾經說:

> 當一個人做著計算機的工作, 他的心靈很難會感到滿足; 而且必定不會發揮出他的所有能力. (附錄四)

假設在 x 軸的 $[0,1]$ 區間之內, 有一個正值的函數 $y = f(x) > 0$. 想像這個類比積分器有兩支筆桿. 輸入者握著一支筆桿描繪函數 $f(x)$ 的圖形. 透過兩片互相垂直摩擦轉動的圓盤, 牽動另外一支筆桿在另一張紙上畫出一條正值且漸增的連續曲線. 設定輸出的曲線也是在 $[0,1]$ 區間上. 則在輸出的這張紙上, 對應任一點 $x = c$ 的輸出值就是從 0 到 c 之間, $f(x)$ 與 x 軸之間所圍成的面積.



在 1880 左右, 英國的另一位物理學家 Lord Kelvin 應用這種類比計算機來分析和預測海潮的漲落. 一直到電子計算機即將問世的 1930 年代, 在美國麻省理工學院和其他地方, 仍有許多人盡力研究如何改良類比型的計算機. 例如 MIT 就製造了一個可以解聯立微分方程式的類比機 (基本上還是作積分的步驟).

理論上, 類比型計算機因爲有連續的輸出, 所以其精確度可以是無限大. 以上所說的有限長位元所造成的機器誤差將不再值得擔心. 實際上, 一個人要從畫著連續曲線的紙張上讀取數據的時候, 仍難免發生誤差. 而且這個讀取的步驟並不容易自動化. 再者, 由於類比型計算機的內部設計無可避免地要使用互相摩擦的轉盤, 使得這種計算機的機械複雜度有相當的上限, 否則過大的摩擦阻力將妨礙整個機器的運作. 所以, 這種計算機很快地就讓位給數位型的電子計算機了.

但是類比積分器並沒有眞的死去. 雖然它不至於活在我們每個人的心中, 它卻活在每一家的電表裡面.

所謂一度電, 就是持續一小時消耗一千佤的電力. 想像 x 軸代表以小時爲單位的時間, y 軸代表以千佤爲單位的耗電量. 則一個用户的用電量可以畫成一個正值的分片連續函數. 那麼這個函數和 x 軸所圍成的面積, 就是用電的度數. 每一個電表裡面, 都有一個類比積分器, 不停地以積分計算您用電的度數.

附錄

一 (The real importance of an automatic computer) lies in the possibility of using a given sequence of instructions repeatedly, the number of times being either preassigned or dependent upon the results of the computation. — Charles Babbage and Augusta Ada Byron

二 We may say most aptly that the Analytical Engine (指 Babbage 最後的機器) weave algebraicl patterns just as the Jacquard-loom weaves flowers and leaves. Here, it seems to us, resides much more of originality than the Difference Engine (指 Babbage 稍早的機器) can be fairly entitled to claim. — Augusta Ada Byron

三 我想, 讀者們都知道楊輝三角, 西方人稱作 Pascal triangle. 它的前幾列是

$$\begin{array}{ccccccccccc} & & & & & 1 & & & & & \\ & & & & 1 & & 1 & & & & \\ & & & 1 & & 2 & & 1 & & & \\ & & 1 & & 3 & & 3 & & 1 & & \\ & 1 & & 4 & & 6 & & 4 & & 1 & \end{array}$$

最上面的位置, 其實也就是 $\binom{0}{0}$; 第二列從左到右依序是 $\binom{1}{0}$ 和 $\binom{1}{1}$; 第三列則是 $\binom{2}{0}$, $\binom{2}{1}$ 和 $\binom{2}{2}$. 依此類推. 根據這個觀察, 可以猜到:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad 1 \le k \le n-1.$$

而且

$$\binom{n}{0} = \binom{n}{n} = 1.$$

從

$$\binom{0}{0} = 1$$

開始, 利用數學歸納法, 可以證明, 對任何正整數 n, 上述的猜想都成立. 利用這個等式, 再配合數學歸納法, 就可以證明任何組合數都是正整數.

四 The human mind is seldom satisfied, and is certainly never exercising its highest functions, when it is doing the work of a calculating machine. — James Clerk Maxwell