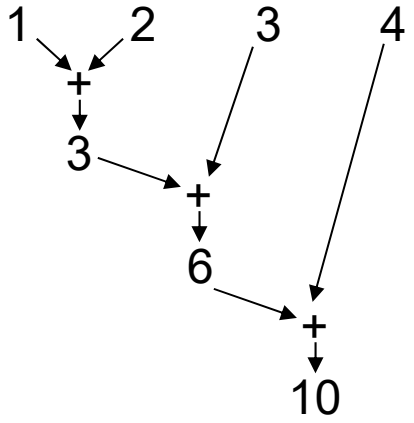
撰寫『程序式』程式的基本方法與要求

丁培毅

針對需要解決的問題設計範例

- 撰寫程序式程式需要教計算機一步一步處理輸入的資料，直到產生所要求的輸出資料為止
- 第一個要件是寫程式的人自己需要非常清楚解決問題的方法與步驟
- 因為人的記憶力與計算能力有限，所以可以針對比較小的問題設計範例，並且逐步得到解這個範例的完整運算過程
- 例如：
  - 級數 $\sum_{i=1}^{n} i$, n=4, 1+2+3+4=10



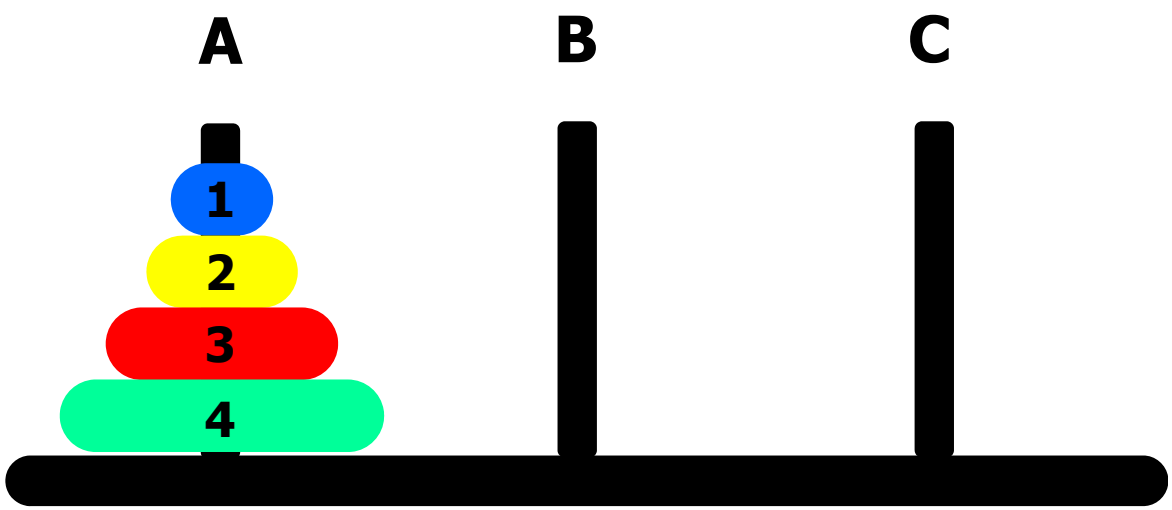
必須留意每一個步驟的切割與執行之先後順序

我們會逐步介紹每一個語法的要項與限制，例如加法一次只能將兩個數字加起來：如 1+2，這些都反應底層電腦的基本硬體功能，也影響到每一個步驟的切割與安排

設計範例 (cont'd)

- 選擇排序法　　未排序資料　18, 9, 13, -3, 10, 7, -1, 2
  由小到大排序完成之資料　-3, -1, 2, 7, 9, 10, 13, 18

18		-3	-3	-3	-3	-3	-3
9	9		-1	-1	-1	-1	-1
13	13	13		2	2	2	2
-3	18	18	18		7	7	7
10	10	10	10	10		9	9
7	7	7	7	18	18		10
-1	-1	9	9	9	10	18	
2	2	2	13	13	13	13	



- 河內塔　　程式輸出

1, A -> B	1, C -> A	1, B -> C	1, A -> B
2, A -> C	2, C -> B	2, B -> A	2, A -> C
1, B -> C	1, A -> B	1, C -> A	1, B -> C
3, A -> B	4, A -> C	3, B -> C	

Get Your Foot In the Door

- 每一個問題都有一定的複雜度，要能夠直接看到完整沒有缺點的答案並不容易，大部分能夠快速解決問題的人都有很好的抽象化能力、問題轉換能力、問題切割能力、或是問題簡化能力，轉換成一個看起來可以掌握的問題，看起來和過去處理過的問題有一些相似性，於是就可以把過去的經驗投射進來，找到一個適合的解法，後續再逐步修改、擴充功能，以便處理原本複雜的問題。
- 在這個解決問題的過程中，很重要的是要有一個有系統、有信心的方式描述問題、勾勒子問題的解法、逐步靠近答案，逐步釐清問題，最後可以完整解決問題

- 接下去談的就是拿到一個程式問題，在設計範例問題、教會自己用規律化且符合計算機運作模型的方法來解決範例問題之後，基本的構思步驟，以及跨出第一步的方法

運算思維: 簡化問題/切割問題

- 很多時候你覺得不知道怎麼下手開始寫程式解決問題，有可能是標準太高了，太理想化了，有點兒偷懶、累積了很多小問題沒有確定的答案，卻想要一次把 錯誤1＋不確定2＋…+錯誤n 通通合起來，還保有期待答案正確的想法，聽起來就不太容易吧!
  - 在寫程式控制電腦的這個層次上，電腦本身是沒有太大智慧的，你的任何一點小錯誤，它沒有辦法幫你更正，它沒有辦法猜測你想要做什麼，需要你自己改成正確的，它才能用它的速度和沒有誤差的記憶來幫你處理複雜以及大量的工作
- 重點在於: 你**不需要一次做出具備完整功能的程式**，你需要練習
  - 簡化問題 – 忽略多變的細節/只看重要部份，太多細節常常讓人覺得複雜無法掌握，有時細節是可以慢慢調整的，有了主要的程式架構以後，會覺得問題已經解決了一大部分，構思細部的方法也會更精確
  - 切割問題 – 有些子問題之間沒有太大關聯性，可以分別解決的

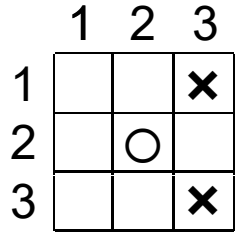
不論是上面哪一種方式，你的目標是處理比較單純的問題，在介紹各種語法時，常常也會給你一些**最基本的範例**，這些範例其實是絕對有用的，它只是還沒有轉換、沒有進化過而已，沒辦法直接拿來解決複雜的問題，但是單純的問題就綽綽有餘了，比對一下範例的結果和簡化過後的問題所要求的結果，常常就可以看出關聯性了；真的還沒有就繼續簡化/切割

以程式實作解決範例問題的演算法

- 調整已知的範例程式：課堂上或是課本上解釋各種語法時，都會以範例程式來說明，如果範例程式輸出和簡化問題要求的輸出相似、或有某種關聯性，就有機會修改範例程式來解決問題，程式修改不多時，比較不會出問題，程式的輸出可以想像和預測，目標是直接解決簡化過的問題，前提是你需要完全瞭解看過的基本範例，別輕視它們
- 設計概念性的步驟：因為你一開始有設計範例問題，也針對這些範例問題思考過規律化、步驟式的解決方法，每一個步驟會有一些產物，通常是一些資料，這些中間步驟的結果會慢慢地接近最終的要求
- 設計變數來存放資料：包括輸入的資料、輸出的資料、中間步驟的暫時性資料、在程式中都需要運用不同型態與結構的變數來存放
- 將概念性的步驟配合上面設計的變數轉換為程式敘述，一開始時程式不需要解決全部的問題，只需要完成各個階段部份的資料轉換即可
- 正確性確認：各個部份的結果可以和前面手動解決範例問題的中間步驟答案比對來確認其正確性 (雖然寫程式的時候都好期待一次就看到最終正確的結果，但這不常發生，事先規劃好逐步驗證中間步驟的結果，常常讓你比較快完成整個程式的功能，這可以說是欲速則不達，但更可以看到你務實的經驗與態度)

轉換資料的表示形式

- 設計程式解決問題的時候，常常需要思考相同資料的**不同表示方法**，不同的表示方法常常對應到**不同的解決方法**，效率和複雜度都不同
  - 例如在篩選質數的時候，常常需要很快判斷那些比較小的質數，程式裡可以把一個一個找到的質數用整數的格式直接記錄下來，但是如果這樣子做，當拿到一個數字需要判斷是不是質數時就要去比對搜尋、速度較慢，如果是用一個很大的陣列，把範圍中每一個數字是不是質數直接標示起來的話，就可以直接檢查是否為質數 2,3,5,7,…. 0,0,1,1,0,1,0,1,…
  - 在右圖的井字遊戲裡，可以用三組資料 (1,1,3), (1,3,3), (0,2,2) 來記錄盤面上的三個符號，也可以用 (0, 0, 1, 0, -1, 0, 0, 0, 1) 9 個數字來記錄，也可以壓縮在一起用9個位元來記錄，記錄方法會使得判斷輸贏的程式不同，也會使得記錄狀態由電腦進行最佳策略搜尋的資料量有所不同，如果是西洋棋，象棋，圍棋的差異就更大了



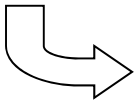
  - 一個撲克牌遊戲中玩家手上 13 張牌，各有不同花色不同點數，如果只是 13 筆 (花色, 點數) 的資料，要讓程式判斷裡面什麼牌型會比較麻煩，應該可以考慮記錄不管花色的情況下各個點數有幾張，或是不管點數的情況下各個花色有幾張

善用函式抽象化處理的細節

- 程序化的程式設計需要
  - 運用函式抽象化處理的細節，使得程式的運作簡潔，看起來清晰、不複雜、可以清楚掌握步驟

低階的程序

```
int i, j,k, m=5, n=6, o=6, p=7, a[5][6], b[6][7], c[5][7];
for (i=0; i<m; i++)
    for (j=0; j<p; j++)
        for (k=c[i][j]=0; k<n; k++)
            c[i][j] += a[i][k] * b[k][j];
```



高階的程序

```
matrixMultiply(a, m, n, b, o, p, c, m, p);
```

  - 運用函式隔離變數和不相關的處理程序，減低程式的複雜度，修改及維護程式時的風險可以降低，減少發生錯誤的機會，降低除錯的困難度
  - 函式最基本的功能當然可以使得同一段程式重複運用在程式裡許多地方，避免重複的程式造成維護的困難
  - 其實有太多的好處講不完，為什麼好多學長/學姊寫出來的程式就是長長一串不分函式的程式?? 覺得需要的時候再用就好了，平常懶得練習

程式設計課程的要求

1. 熟悉 C 的各種語法，瞭解各種語法的細部動作與運用目的，記住基本運用範例與運用情境
2. 分析問題，簡化/切割問題，找出規律化的方式解決問題，比對基本應用範例，產生 C 程式實作上面的解法
3. 遇見程式表現不如預期時，分析哪一個關鍵點程式開始表現與預期不同，分析並且設計輸入資料來顯示出程式的錯誤，運用原始碼偵錯程式以及格式化輸出來找出程式的錯誤，解釋錯誤的原因並且修正程式
4. 程式的易讀性、模組性、和正確性一樣重要

不是讓你練習

1. 修改或是組合 Google 到的程式碼繳交作業
2. 翻譯某一段寫好的演算法，管它在作什麼、在計算什麼，反正照著翻譯，讓它能夠編譯能夠執行就有正確結果
3. 在討論區裡求救，讓高手幫你把不太對的程式改對，讓高手給你一段看不懂 (好像很高深) 但是可以運作的程式

未來的課程銜接

- 在這學期裡如果你按部就班地學到前面所談到的基本方法，接下來的幾個學期裡，有適當地課程會引導你加強各個部份的知識與技能：

① 二上的資料結構會介紹過去常常運用在程式裡表示資料的方法，但是課程裡不會再告訴你怎樣怎樣分割問題、簡化問題、設計範例、產生程式的主要架構、怎樣測試程式，過去很多學長會赫然發現程式的要求常常就只是輸入和輸出了，也許還指定了程式裡資料的表達方法，但是卻不再引導你做出程式，如果你不在現在這個課程裡熟悉整個過程，建立自己規劃整個過程的能力，你下學期慘了

② 二下的演算法會介紹過去四十年裡在程式裡常常見到的問題解決方法，需要運用什麼樣的資料結構，會分析那樣子的解決方法的效率，有一些課程/課本也許會說明抽象化原始問題轉換為演算法的設計過程，但是大部分可能都希望你藉由程式設計課程得到的基本方法，配合對於程式系統單一步驟能力的了解，自己去思考為什麼會得到這樣的設計，當然大部分課程也不再要求你用某一種語言把演

算法做出來了，這並不是說演算法不用寫成程式，而是因為程式設計的課已經教會你製作程式的完整程序了，演算法只是強調過去一些比較具有代表性的程序與作法而已，基本上所有程序式的語言都有共通的語法架構，每一種語言的每一個語法都可以互相對應，過去也有很多學長，因為大一上的課程裡沒有弄清楚需要學會的東西，到了演算法課程的時後，竟然不曉得上課講到的東西是自己應該可以花半小時把程式寫出來的，常常到了大二下的時候告訴我：整個暑假加上大二上學期好像已經八九個月沒有碰過程式了，差不多都忘了，整個傻眼

③ 二上還有一個組合語言的課程，基本上這是低階的程式，直接用 CPU 提供的指令來撰寫程式，比較不會用來解決電腦應用中大的問題，最主要是希望你藉由這個課程清楚了解底層機器的支援到底有些什麼

④ 大三會運用 C++ (或是 Java) 介紹物件導向程式設計，和這學期一樣都是用同一個 C/C++ 編譯器，但是模型化的方法不一樣，寫出來的程式大大不同，如果你在這學期裡沒有學好基本的方法，

一定 GG，在那個課程裡有太多其它的東西要說明，有些根本和程序式的程式設計在理念上是衝突的，需要你針對不同的應用去評量使用的時機，所以課程裡不會再重複前面三個課程裡所交代的，還會一直用到這三個課程的內容來思考如何支援大型軟體的設計

⑤ 三上有 **Java 的程式設計**，有 Android 環境的程式設計，有 iOS/OS-X 環境，有 Linux 系統程式設計，有 Win32 環境的程式設計，都是整合運用前面的四個課程，這些和人機介面相關的課程都是物件導向的方式為主體的，如果你在大二下已經放棄，那麼應該不需要想說能不能應付或是能不能接受了，這時候可能要強迫自己接受把資訊系當成數學系唸的概念了

⑥ 三下有 **編譯器設計**，告訴你該怎樣處理用語法規範的語言，怎樣檢查它的語法的正確性，怎樣翻譯一個高階語言的程式成為 CPU 可以執行的低階指令，有人說這個課程可以把 3 個學分當成 9 個學分來唸，課程裡面當然談的每一樣事情都是程式，是寫程式處理程式…

⑦ 從大二下開始，系上很多課程是關於自然界模型和法則的課程 (電腦視覺，電腦繪圖，訊號與系統，影像處理，圖論，人工智慧，機器學習，圖形辨識，密碼學，資訊安全，物聯網，生物資訊)，終究電腦自己沒辦法學習知識沒有辦法自己整理應用知識，你需要學好了以後幫它分析問題的解法，才能寫出操控它的程式，這時候環境和工具都齊全了，你可以全力了解這些進階的模型，這是由其它行業轉進資訊業界的人最欠缺的東西，你在這個階段因為可以清楚掌握電腦如何實作這些模型，可以很快地累積你在以後業界工作的本錢，知識、經驗、和腦力是關鍵，不是只要爆肝寫程式或是偵錯程式就能夠愉快生存的