Algorithm Design Methods
Enumeration

by Chin Huang Lin

Sprout

如何解題？

- 如果寫成一個程式……
  1. 觀察性質
  2. 分析結論
  3. 提出作法
  4. 評估現有作法，如果不夠好，goto 1
  5. 實作作法
  6. 如果實測不夠好，goto 1

- 有時會需要神來一筆，但大多數情況下會仰賴於既有的經驗
  - ex. 透過範圍大小設計針對範圍較小者下手的作法
  - ex. 如果有重複計算者，可以試圖避免
  - ex. 快速猜出一個複雜的演算法的複雜度
  - ex. 評估演算法實作上的困難度與複雜度常數

Sprout

枚舉—未知與已知之橋

- 知道的越多，能做的越多
- 如果只有有限個關鍵，那麼不妨一一枚舉！
- 優點：絕無遺漏、不求他人
- 缺點：錯殺一百、難敵無限
  - ex. 丟翻圖方程，為什麼不能夠用枚舉解決呢？

Sprout

牛刀小試

- 一個字串的一個連續區間構成的字串稱為這個字串的子字串，如 out、prou 都是 sprout 的子字串，而 spot 不是。
- 一個字串如果翻轉 (如 XDrz 翻轉後變為 zrDX) 後與原先的字串相同，我們稱這樣的字串為一個回文。
- 如果一個字串 S 的子字串 T 是一個回文，我們便說 T 是 S 的一個回文子字串。
- 現在給你一個字串 S，請問 S 內最長的回文字串有多長呢？

Sprout

天真的作法

- 我們可以枚舉該子串的開頭與結尾，並檢測枚舉的子串是否為回文
- 枚舉開頭與結尾複雜度是 $O(n^2)$、檢測回文的複雜度是 $O(n)$，總複雜度為 $O(n^3)$
- 好像有點糟糕，有沒有辦法做得更好呢？

Sprout

再觀察回文子串的性質……

- 以一個區間 $[l,r]$ 來表示一個子串
- 如果 $[l,r]$ 不是回文子串，那麼 $[l-1,r+1]$ 一定也不是回文子串
  → 還檢測 $[l-1,r+1]$ 實際上造成了浪費
- 如果 $[l,r]$ 是回文子串，$S[l-1]=S[r+1]$，那麼 $[l-1,r+1]$ 也是回文子串
  → 還重新檢測 $S[l..r]$ 實際上造成了浪費
- 發現浪費的情形，都有共同的字串中心！
- 枚舉回文的中心並且逐步向外拓展，中心的可能數不超過 $O(n)$ 個、對於每個中心拓展不會超過 $O(n)$ 次，總複雜度不超過 $O(n^2)$
- ex. S=QAQAO

Sprout

但，如果是這樣呢？

- 雄精英二號（簡稱雄二）在和他的妹妹（簡稱桐乃）都有養寵物，雄二養了一只手動壓縮高速水箭龜（簡稱龜王），桐乃養了一只 aaabaaajss（簡稱雕王）。
- 這一天，雄二和桐乃為了「誰的蛋比較硬」的事情在爭吵，彼此都覺得自己寵物的蛋比較堅固，於是他們想出了這樣的方法來檢測：首先他們找到一間非常高的大樓（你可以假設科技很進步，高樓高達 2147483647 層），接著試摔他們寵物的蛋。如果蛋在第 x 樓沒破，但在第 $x+1$ 樓恰好破了，我們就說這顆蛋的硬度是 x。測量完後，比較誰的蛋硬度比較大，誰就贏了。
- 問題是：假設不需要考慮蛋的數量的問題，有沒有辦法可以盡快地測量出蛋的硬度呢？

Sprout

一個明顯的性質

- 如果我們發現在第 x 樓還沒破，那麼我們可以確定在第 $y\ (y < x)$ 樓都不會破
- 如果我們發現在第 x 樓已破蛋，那麼我們可以確定在第 $y\ (y > x)$ 樓都會破蛋
- 方法 1：我們可以「跳著」測量！
  - 舉例來說，我們可以每兩層樓測一次；測到蛋破掉了，再往回測一次就可以知道答案了
  - 如果三層樓測一次呢？測到蛋破了，再往回測兩次
  - k 層樓測一次，在 $O(n/k)$ 回合內會蛋破，然後要往回測 $O(k)$ 次來確定答案
  - 當 $k = O(\sqrt{n})$ 時，複雜度為 $O(\sqrt{n})$
- 方法 2：利用詢問的結果，一刀兩斷
  - 一開始解答可能座落在 $[0, n]$ 區間內
  - 如果現在解答座落在 $[l, r]$ 區間內，我們測量第 $(l + r)/2$ 樓摔落的結果
  - 如果蛋破了，那麼解答座落在 $[l, (l + r)/2 - 1]$ 內
  - 否則座落在 $[(l + r)/2, r]$ 內
  - 每次範圍都縮小一半，複雜度不超過 $O(\log n)$！

Sprout

觸類旁通

- Zerojudge a439 田忌賽馬 (2011 NPSC 高中組決賽)
  - http://zerojudge.tw/ShowProblem?problemid=a439
- 當然也是田忌的策略！問題是，到底我方最快的是哪幾匹馬？
- 隨著時間不斷變動，最快的馬們也不斷變動；但如果知道哪一天要比賽，馬上就可以知道比賽結果—枚舉的動機
- 由於輸入皆非負，因此馬們只會不斷變強 (或者至少不會變弱)。也就是說，隨著天數增加，贏得場數具有單調性—二分法！
- 天數決定後，需要 $O(n\log n)$ 時間計算輸贏；天數枚舉次數不會超過 $O(\log C)$ 次，總複雜度為 $O(n\log n\log C)$。

Sprout

再下一城！

- 參加過解題比賽的人都知道，通過一道題目能獲得的快感度常常取決於解掉題目的時間。舉例而言，成為某題目的第一個通過者、解掉一道困難的題目或者是在最後一刻通過某題都會讓參賽者好興奮
- 然而，身為一個題目設計者，評估一道題目帶給參賽者的樂趣就是很嚴肅的一件事情了。根據統計數據，我們知道現在的 n 道題目中，通過第 i 道題目帶給參賽者的快感大約是一個只與比賽經過時間相關的函數
$$f_i(t) = ai(t - bi)^2 + ci$$
- 而一整套題的樂趣度則為當下最有樂趣的題目的樂趣度，即
$$S(t) = \max\{f_i(t) | 1 \leqq i \leqq n\}$$
- 整場比賽共有 300 分鐘，即 t 的值域為 $[0,300]$。為了可以最有效率地運用點心，你決定在整場比賽最令人低落（即整套題目樂趣度最低的時候）送上點心。問題來啦：到底應該在什麼時候送上點心才好呢？
- $n \leqq 10, 0 \leqq a_i, bi, ci \leqq 300$
- Problem Source: 2013 台清交程式設計競賽

Sprout

糟了，是無限！

- 解既然不一定是整數，乍看之下 x 是多少都可以，根本枚舉不完阿！
- 把這個問題假想成賽車比賽：很多選手在開賽車，我們想要知道的是在任意時間第一名是誰
- 如果有一天第一名更動了，那一定是有人超越了他！
- 重要的只有那些有「超車事件」發生的時間，以函數間的關係來看就是某兩個函數的交點
- 兩個不相等的二次函數最多只有兩個交點，所以總交點數只有 $O(n^2)$ 個
- 枚舉最小值發生的位置，總複雜度為 $O(n^3)$

Sprout

再回去看看……

- 二次函數都可以視為一個「斜率非嚴格遞增」的函數，這裡姑且稱這種函數為「U 型函數」
- 神秘的特性：U 型函數和 U 型函數取 max 後，還是 U 型函數！
- 也就是說，$S(x)$ 實際上也是一個 U 型函數
- 有沒有辦法在 U 型函數上二分呢……？

Sprout

野生的瓶頸

- 從 U 型函數中間切下去，要怎麼知道最小值會在哪一邊？
  - 其實可以，不過可能會需要微分的技巧
- 檢討一下失敗的原因……
  - 單調性分成兩段！
- 強化版本：三分法！

Sprout

三分怎麼判？

- 考慮三分後從左到右四個採樣點的關係……
- Case 1: $S(a) < S(b) < S(c) < S(d)$
  - 此時最小值一定不在最右邊
- Case 2: $S(a) > S(b) < S(c) < S(d)$
  - 此時最小值一定不在最右邊
- Case 3: $S(a) > S(b) > S(c) < S(d)$
  - 此時最小值一定不在最左邊
- Case 4: $S(a) > S(b) > S(c) > S(d)$
  - 此時最小值一定不在最左邊
- 每次都至少可以讓區間縮小 1/3！

Sprout

新的複雜度

- 假如我們做了 k 次縮小，區間大小變為原先的 $(\frac{2}{3})^k$
- 只要 k 足夠大，其實答案就夠精準了！
- 複雜度為 $O(nk) = O(n\log\frac{300}{10^5})$，在這題中比前面的作法還好
- 優勢：好寫、通用
- 劣勢：仰賴數字範圍

Sprout

for v.s. forall

- 到目前為止，我們所有的枚舉都能只用一兩個迴圈解決
- 如果是枚舉一個「狀況」呢？例如說，「對於所有可能的排列」、「對於所有可能的走法」、「對於所有可能的方案」……
- 通常會需要「樹狀圖」的概念！

Sprout

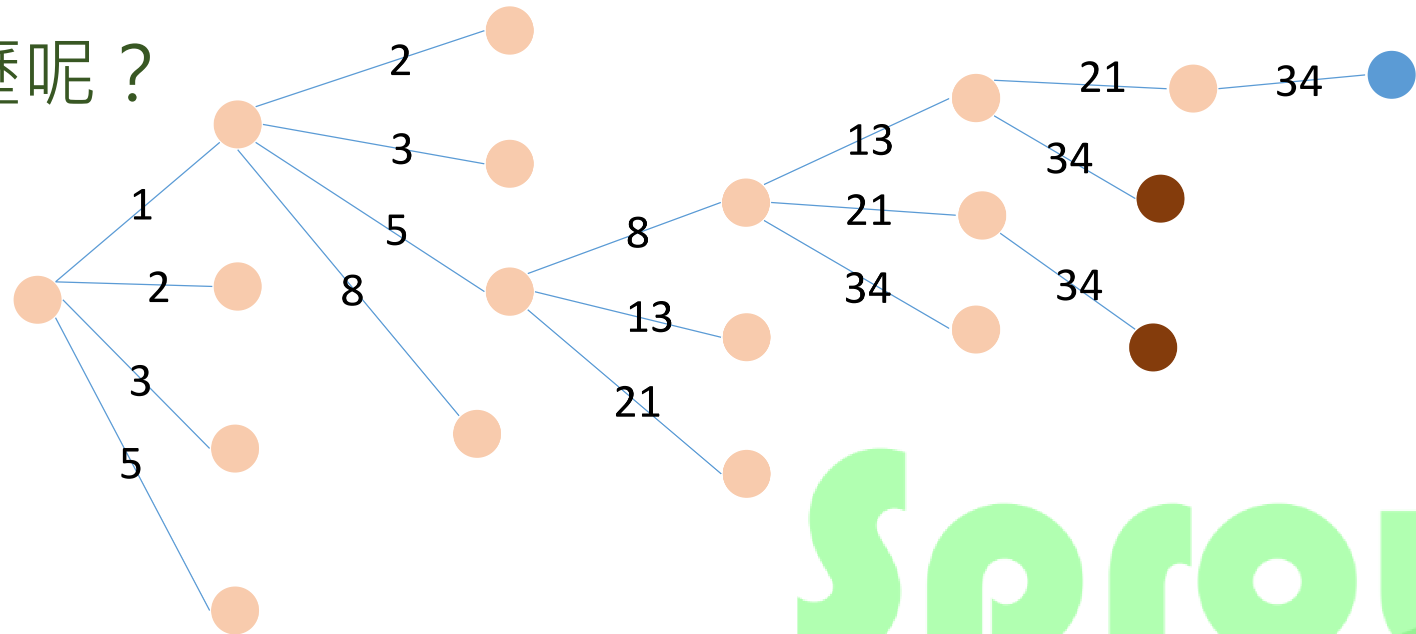
有夢最美

- Zerojudge c074. Lotto (UVa 441)
  - http://zerojudge.tw/ShowProblem?problemid=c074
- 列出組合（遞增排列），誰不會呢？說穿了，就是一棵排列樹

Sprout

希望相隨

- 排列樹上有許多節點，其中有些分支可以長到深達 6 層，就形成最後的一組解
- 我們可以按照字典序遍歷整棵樹，走到解節點時順便印出當前路徑即可
- 問題：怎麼遍歷呢？



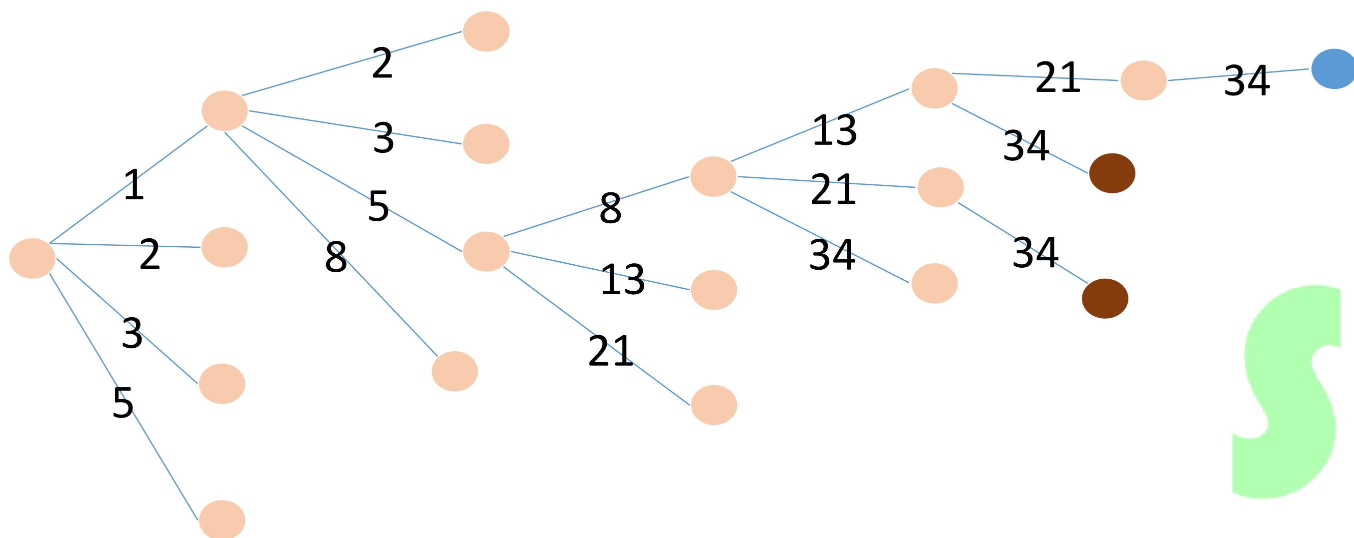
如果使用 BFS……？

- 過程中所有節點都必須記錄下來
- 總共有幾個節點呢？
  - 深度為 0 的 1 個
  - 深度為 1 的 k 個
  - 深度為 2 的 C_2^k 個
  - 深度為 3 的 C_3^k 個
  - ……
  - 總共有 $\sum_{i=0}^{6} C_i^k$ 個節點！
- 以本題而言，如果 $k = 12$，大約就有 2510 個節點
  - 其實還可以接受嘛
- 要注意的是，實作方法不同，使用的空間和效率也不同

Sprout

實作比一比

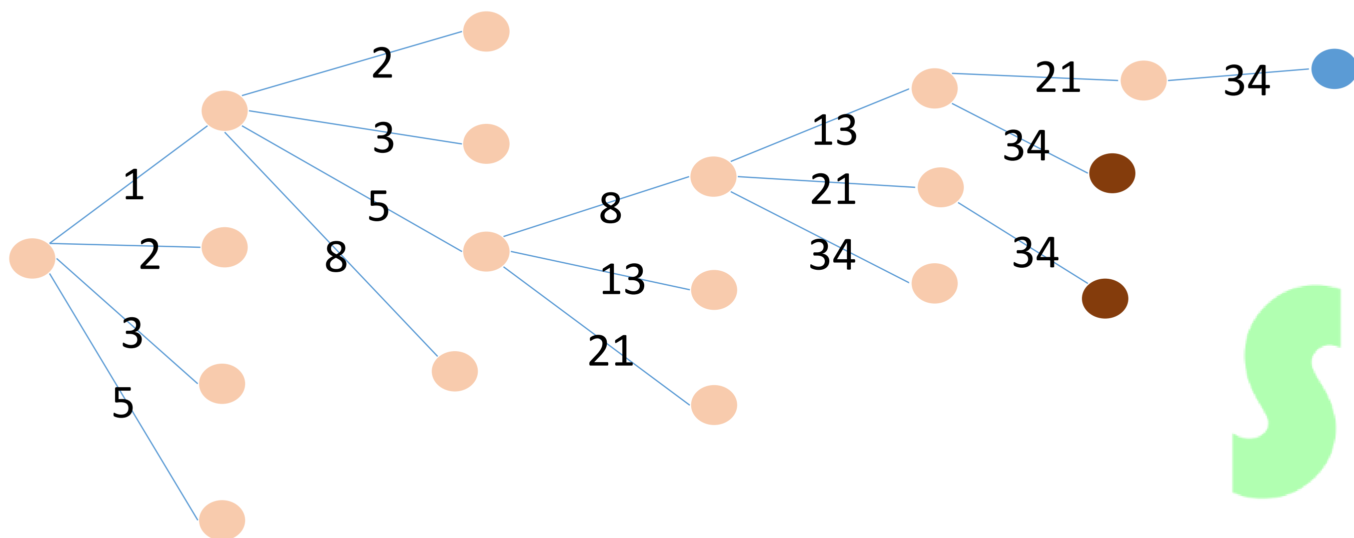
- 方法 1. 在每個節點都詳細記錄對應到的路徑
  - 舉例來說，開一個 `struct Node{int path[6],depth;};`
  - 每次從 queue 裡面拿出一個元素，找出所有比該條路徑上最末端號碼大的幸運號碼，然後依序加入末端形成新節點，並推入 queue 中
  - 過程中順便維護節點深度 (depth)，如果發現深度為 6 則輸出路徑
  - 推入新節點和每個點所需空間都為 O(路徑長)



Sprout

實作比一比

- 方法 2. 在每個節點記錄該節點的父親以及路徑上的最末元素
  - 在輸出時直接順著父親指針走回去，就可以輸出解了
  - 推入新節點和每個點所需空間都為 $O(1)$



Sprout

如果使用 DFS……？

- 直接開一個 stack 記錄當前節點的路徑即可！

```
int stack[MAXLV];
void dfs(int depth,int last){
    if( depth == MAXLV ){
        output(stack[0..MAXLV-1]);
    }
    else {
        for(int i=last+1;i<k;++i){
            stack[depth] = lucky[i];
            dfs(depth+1,i);
        }
    }
}
```

Sprout

BFS v.s. DFS

- 在本題中，使用 DFS 似乎在全方面都更有優勢

項目	BFS	DFS
時間複雜度	$O(\sum_{i=0}^{M} C_i^k + MC_M^k)$	$O(\sum_{i=0}^{M} C_i^k + MC_M^k)$
空間複雜度	$O(\sum_{i=0}^{M} C_i^k)$	$O(M)$
實作複雜度	高	低
額外優勢	無	M 較大時仍適用

Sprout

如果是這題呢？

- UVa 571. Jugs
  - http://luckycat.kshs.kh.edu.tw/homework/q571.htm
- 這題是存在漂亮的數學解的，但如果想不到的話就只好先枚舉所有可能的方式了……
- 我們發現用一個二元組就可以描述一個「狀態」：$\{A\text{內水量}, B\text{內水量}\}$
- 狀態 1 如果可以透過一個步驟變成狀態 2，那可以視為狀態 1 對應到的節點有一條邊連到狀態 2
- 所有的狀態之間形成一張「看不見的圖」！

Sprout

看不見的圖

- 有些書中把這樣的圖稱為「隱式圖」
- 問題的起點是 $\{0,0\}$，終點是 $\{N,x\}$ 或 $\{x,N\}$，其中 x 可以是任意值
- 如果把二元組看成二維平面上的點，起點只有一個、終點有很多個，走到哪一個都可以—有沒有感覺很像*喵喵抓老鼠*？！
- Yes, we can use BFS！
- 使用 DFS 當然也可以，兩者的優劣就跟喵喵抓老鼠時分析的相同，不同的是本題中不要求最少步驟解，BFS 的優勢少了一個

Sprout

回過頭看一下 Lotto……

- 其實 Lotto 某種程度上也可以看成一張隱式圖 (只是保證會是一棵樹)，節點之間以某種「規則」形成邊，然後有些節點是解節點
- 方才的兩個例子中，總節點數都很少，我們可以利用 Flood Fill 把全部的節點都走訪；如果總節點數多得不像話呢？
- 我們只能夠在龐大的隱式圖中想辦法「找到」解，卻很難把每個節點都走過，此時我們常把解法稱為「暴力搜索」(brute-force search algorithm)
- 當我們要暴搜時，事情就變得不一樣了！

Sprout

暴搜下的 BFS,DFS

- 記錄節點狀態變得很困難
  - 舉例而言，如果圖很大，前一題就不能夠直接開陣列記錄每個節點距離原點多遠了—甚至連記錄是否拜訪過都很困難
- 拜訪全數節點接近不可能
  - 必須仰賴許多的加速技巧來排除根本不可能的分支 (這步驟又稱為「剪枝 (pruning)」)
  - 盡早拜訪到解節點非常重要
- BFS 的空間消耗變得非常致命
  - 要有 10 億次的運算很簡單，但要有 10 億單位的空間很困難
  - 狀態空間很大的題目中幾乎都使用 DFS，除非有特殊要求，如要求與原點最近的解

□ 有些書本或者網站會把此時的 DFS 算法稱為「回溯法 (backtracking)」

Sprout

來練習一下吧！

- 在數獨遊戲中，遊戲給你一個 9 x 9 的方陣，其中還分成 9 個 3 x 3 的子方陣，如圖：
- 數獨的遊戲規則是這樣的，最後完成數獨的時候，每個格子都必須填上 1~9 中一個數字，並且在每一行、每一列、每一個子方陣中，都不能有重複的數字。
- 現在給你一個已經填好一些數字的數獨，寫個程式完成它吧！

♪ 寫好之後拿去~~騙騙同學~~和同學分享，效果還不錯唷~

.	2	7	3	8	.	.	1	.
.	1	.	.	.	6	7	3	5
.	.	.	.	.	.	.	2	9
3	.	5	6	9	2	.	8	.
.	.	.	.	.	.	.	.	.
.	6	.	1	7	4	5	.	3
6	4	.	.	.	.	.	.	.
9	5	1	8	.	.	.	7	.
.	8	.	.	6	5	3	4	.

想一想

- 在 Lotto 中，如果每次不是固定輸出 $M = 6$ 個數字，而是可以自由決定 M，如 $M = k$ 等，那麼 BFS 和 DFS 的空間差距會變得多大？試著以複雜度表示出來，代入幾個數值看看兩者的差異。
- 在數獨問題中，如果完全不做任何加速，並且真的遍歷完所有節點（例如無解狀態），複雜度是多少呢？這結果告訴我們，如果存在暴力搜索以外的解法，我們通常會選擇哪種解法？
- 假如我們用了一個暴力搜索的作法，共搜索了 k 層，那麼通常大部分的時間會花在哪一層的搜索？為什麼？
- 假設你現在有一個「估價函數」，在搜索到某一個節點時，只要呼叫一次估價函數，它就可以回傳這個節點是否有希望拓展出一組可行解，從而提供剪枝參考，但是呼叫一次成本為 $O(n^2)$。這種情況下，要選擇使用該函數（但是讓整體複雜度乘上 $O(n^2)$）好，還是選擇不使用（損失一些剪枝機會）好呢？

Sprout