臺北市立第一女子高級中學

資訊學科能力競賽

選手訓練魔法書

Training Bible

Data Structure　Algorithm

& Problem Solving

Edit by Computer Science Team in TFG

July, 2011 5th Edition

要記得曾經 在 2011 年 和妳一起走過的

營隊指導老師

陳怡芬老師

編輯團隊

主編 陳怡芬老師
作者群 黃易學姐、南蓁學姐

學姐團隊

黃易、陳南蓁
吳雙、郭盈妤
薛祐婷、邱意晴
葉佩雯、陳琬嫣 、姜佳昀、廖乃萱

資訊學科諮詢團隊

陳怡芬老師 何雪溱老師
董致平老師 黃芳蘭老師

選手

班級	座號	姓名	班級	座號	姓名
一溫	09	林品君	二溫	14	陳姿穎
一溫	11	邱筱晴	二溫	01	王心瑜
一溫	25	楊尚蓉	二溫	05	林怡均
一良	02	王馨儀	二溫	12	陳怡靜
一良	09	林友嵐	二溫	11	陳　瑀
二公	19	張筱晴	二溫	04	房芷筠
			二良	06	吳虹熠

儲備選手

班級	座號	姓名	班級	座號	姓名
一平	15	洪士倢	一儉	02	王妤文

Content

第一階段 基礎概念篇 (on line)

單元	內容
Unit 1 - 資料表示法	二進位系統轉換、補數運算、浮點數表示、GrayCode、Haffman Code..
Unit 2 - 布林代數	基本定理定律與邏輯電路、布林代數化簡
Unit 3 - 資料通信與電腦網路	TCP/IP、IPv6、Domain Name
Unit 4 - 軟體系統	分時系統、即時系統、批次處理
Unit 5 - 數位影像簡介	影像解析度、色階與檔案大小
Unit 6 - 資料加密技術與電子交易安全	對稱式加密技術、非對稱式加密技術、DES、RSA… SSL, SET…數位簽章、數位信封
Unit 7 - 資料結構複習 (ppt)	Sort, Search, Tree, Graph, Shortest Path…

第二階段 程式實作篇 Program = Data Structure + Algorithm

Section I: 動態規畫法與實例解析

單元	內容
Unit 11- 動態規畫法	最短路徑問題 (Shortest Path) 、最長遞增子序列 (Longest Increasing Subsequence, LIS) 、最長共同子序列 (Longest Common Subsequence, LCS) 、背包問題 (Knapsack) 、零錢問題、最大連續和 (Maximum Consecutive Sum) 、最大子矩陣、最大矩形、矩陣相乘 (Matrix-Chain Multiplication) 、拿石頭、旅行推銷員問題 (Traveling Salesman Problem, TSP) 、爬樓梯問題、貼磁磚問題 (Tiling)......
Unit 12- 動態規畫法實例與解析	Problem 1 : 最大矩形 (Area) Problem 1-1 : Take the Land (ACM 10074) Problem 2 : 正直 DE (2007 NPSC 高中組決賽 b090: D. 正直 DE) Problem 2-1 : Matrix Chain Multiplication (ACM Q442) Problem 2-2 : Optimal Array Multiplication Sequence (ACM Q348) Problem 2-2-1 : Cutting Sticks (ACM Q10003) Problem 2-2-2 : Mixtures (SPOJ Problem Set 345. Mixtures) Problem 3 : Walking on the Safe Side (ACM Q825) Problem 3-1 : Expressions (ACM Q10157) Problem 3-1-1 : How Many Trees ? (ACM Q10303) Problem 3-1-2 : Count the Trees (ACM Q10007) Problem 3-1-3 : Safe Salutations (ACM 991) Problem 4 : Tiling (ACM Q10359) Problem 4-1 : Brick Wall Patterns (ACM 900 - Brick Wall Pattern) Problem 4-2 : Tri Tiling (ACM Q10918: Tri Tiling) Problem 4-3 : 鋪磁磚問題 (94 北市賽 -prob 3)

Section II: 演算法集錦

- Unit 21-排列組合
- Unit 22-產生可能的集合
- Unit 23-m 元素集合的 n 個元素子集
- Unit 24-數字拆解
- Unit 25-選擇、插入、氣泡排序
- Unit 26-Shell 排序法
- Unit 27-Heap 排序法
- Unit 28-快速排序法
- Unit 29-快速排序法(二)
- Unit 30-快速排序法(三)
- Unit 31-合併排序法
- Unit 32-循序搜尋法
- Unit 33-二分搜尋法
- Unit 34-費氏搜尋法
- Unit 35-老鼠走迷宮
- Unit 36-騎士走棋盤
- Unit 37-八個皇后
- Unit 38-字串核對
- Unit 39-背包問題

Section IV: 2010 阿南學姐集訓篇

- Unit 41-基礎字串函數 String (9 pages)
- Unit 42-函數與遞迴 Function & Recursive (6 pages)
- Unit 43-遞迴 Recursive (4 pages)
- Unit 44-排序 Sorting (2 pages)
- Unit 45-篩法 Sieve (1 pages)
- Unit 46-樹狀結構 Tree (3 pages)
- Unit 47-Advance C (7 pages)
- Unit 48-圖形結構 Graph + DFS (4 pages)
- Unit 49-動態演算法 Dynamic Programming (3 pages)
- Unit 50-DPbySuhorng (4 pages)
- Unit 51-貪婪演算法 Greedy Method (4 pages)

Group 學習進度

First Phase 筆試題

測驗編號	完成期限	完成日期	Note
Test-1			
Test-2			
Test-3			
Test-4			
Test-5			

Mid Phase 程式題

程式編號	完成期限	完成日期	Note
PROGRAM-1			
PROGRAM-2			
PROGRAM-3			
PROGRAM-4			
PROGRAM-5			
PROGRAM-6			
PROGRAM-7			
PROGRAM-8			
PROGRAM-9			
PROGRAM-10			

自由選題--ACM, NPSC, ...

完成題號	

100 學年度資訊學科能力競賽暑期選手訓練營

營隊時間：

- 第一階段 100 年 7 月 13-15 日(09:00~12:00)
- 第二階段 100 年 8 月 22-24 日(09:00~12:00; 13:30-16:30)

營隊地點：電腦教室 210
指導老師：陳怡芬

課程進度

第一階段

日期	活動名稱	主講者	活動內容	Note
7/13 (三)	Introduction to Computer Science	陳怡芬	布林代數簡介 網路概論簡介 試題解析	
7/14 (四)	Data Structure + Algorithm	陳怡芬	DS,Alg 概念講述 試題解析	
7/15 (五)	Dynamic Programming 初探	黃易	DP 概念說明 DP 程式題解析	

第二階段

日期	09:00-12:00	主講者	13:30-16:30	Note
08/22 (一)	Dynamic Programming Advanced	黃易	集訓模擬競賽(1)	
08/23 (二)	Graph-1	薛祐婷 葉佩雯	集訓模擬競賽(2)	
08/24 (三)	Graph-2	陳南蓁	集訓模擬競賽(3)	

Unit 1：資料表示法

¤ 數字系統

- 十進位(Decimal) 0,1,2,3,4,5,6,7,8,9
- 二進位(Binary) 0,1
- 八進位(Octal) 0,1,2,3,4,5,6,7
- 十六進位(Hexadecimal) 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

數制間的轉換:

整數部分: 用短除法連除 2(或 8 或 16)，由下而上寫出餘數

小數部分: 用乘法乘 2(或 8 或 16)，直到小數為 0。由上而下寫出乘積的整數。

範例 1: 將十進位數字 0 到 16 分別以十進位（Decimal），二進位（Binary），八進位（Octal），及十六進位（Hexadecimal）表示出來。

解答：

1. 十進位逢 10 即進位，而二進位逢 2 即進，八進位逢 8 即進位，十六進位逢 16 即進位。
2. 十六進位，由於基底爲 16，故其數元分別 0,1,2,......,14,15，但 10 到 15 由於有兩個數元，故借來 A,B,C,D,E,F 等數元以分別表示 10 到 15。
3. 若是二進位，則其一個數元（digit）被稱爲二進數元或位元（binary digit, or bit）。

十進制（Decimal）	二進制（Binary）	八進制（Octal）	十六進制（Hexadecimal）
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

範例 2：試將下列數字表示成十進位數字：

① $(903.87)_{10}$
② $(10101.01)_2$
③ $(754.34)_8$
④ $(ABC.5)_{16}$

解答：

① $(903.87)_{10} = 9\times 10^2 + 0\times 10^1 + 3\times 10^0 + 8\times 10^{-1} + 7\times 10^{-2}$
$= 900 + 3 + 0.8 + 0.07$
$= 903.87$

② $(10101.01)_2 = 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 + 0\times 2^{-1} + 1\times 2^{-2}$
$= 16 + 4 + 1 + 0.25$
$= 21.25$

③ $(754.34)_8 = 7\times 8^2 + 5\times 8^1 + 4\times 8^0 + 3\times 8^{-1} + 4\times 8^{-2}$
$= 448 + 40 + 4 + 0.375 + 0.0625$
$= 492.4375$

④ $(ABC.5)_{16} = 10\times 16^2 + 11\times 16^1 + 12\times 16^0 + 5\times 16^{-1}$
$= 2560 + 176 + 12 + 0.3125$
$= 2748.3125$

範例 3： 試將 $(18.75)_{10}$ 轉成二進位數字，或以二進位來表示之。

解答：

$(18.75)_{10} = (18)_{10} + (0.75)_{10}$

我們首先將整數部分與小數部分分開，並分別找出它們的二進位表示法，再合併之。

（一）整數部分：用除 2 的方式	（二）小數部分：用乘 2 的方式
2 | 18 --- 0 (18÷2 的餘數) 1 | 9 --- 1 (9÷2 的餘數) 2 | 4 --- 0 (4÷2 的餘數) 2 | 2 --- 0 (2÷2 的餘數) 2 | 1 --- 1 ← MSD 0 所以 $(18)_{10} = (10010)_2$	0.75 × 2 小數點後第一位 1.50 0.50 × 2 小數點後第二位 1.00 當小數部分=0時則停止。 所以 $(0.75)_{10} = (0.11)_2$

將整數及小數合併，$(18.75)_{10} = (10010.11)_2$

讀者可自行練習將 $(13.3)_{10}$ 以二進位來表示。下面範例說明如何將十進位數字轉成八進位數字。

範例 5：試將 $(43969.75)_{10}$ 以十六進位表示之。

解答：

$(43969.75)_{10} = (43969)_{10} + (0.75)_{10}$

（一）整數部分：用除 16 的方式	（二）小數部分：用乘 16 的方式
16 | 43969 --- 1 (43969÷16 的餘數) 16 | 2748 --- 12 (2748÷16 的餘數) 16 | 171 --- 11 (2171÷16 的餘數) 16 | 10 ---10 ← MSD 0 故 $(43969)_{10} = (ABC1)_{16}$	0.75 × 16 小數點後第一位 12.000 ↑因=0，故停止 $(0.75)_{10} = (0.C)_{16}$

將整數與小數合併 $(43969.75)_{10} = (ABC1.C)_{16}$

讀者可自行練習將 $(1000.625)_{10}$ 轉換成十六進位。

範例 6：① 將二進位數字 $(10101.0101)_2$ 以八進位表示之。
② 將八進位數字 $(123.45)_8$ 轉成二進位數字。

解答：

① 將二進位轉成八進位，其做法是以小數點爲分界，分別往左右每三個位元爲一組，若不夠三位元則補 0，之後每一組的三個位元轉成八進位的數元。

```
 ←──────          ──────→
 0 1 0 1 0 1  .  0 1 0 1 0 0
 |___| |___|     |___| |___|
⇨  2     5    .    2     4
```

故 $(10101.0101)_2 = (25.24)_8$

② 將八進位轉成二進位，其做法是將每個八位數元以三位元的二進位數元表示之即可。

$(123.45)_8 = (\underset{1}{\underline{001}}\ \underset{2}{\underline{010}}\ \underset{3}{\underline{011}} . \underset{4}{\underline{100}}\ \underset{5}{\underline{101}})_2$

$= (1010011.100101)_2$

範例 7：① 試將 $(11101.0101)_2$ 以十六進位表示之。 ② 試將 $(2A5.A8)_{16}$ 轉成二進位數字。

解答：

① 與由二進位轉成八進位的方法類似，不過二進位轉十六進位，要四個位元組成一組，不足四個則補 0。

$$\underline{0001}\ \underline{1101}\ .\ \underline{0101}$$
$$1 \quad D \quad 5$$

故 $(11101.0101)_2 = (1D.5)_{16}$

② 將十六進位的每個位元轉成以四位元的二進位即可。

$$(2A5.A8)_{16} = (\underline{0010}\ \underline{1010}\ \underline{0101}.\underline{1010}\ \underline{1000})_2$$
$$2 \quad A \quad 5 \quad A \quad 8$$
$$= (1010100101.10101)_2$$

最左邊與最右邊的 "0" 皆可略去。

範例 8：① 試將 $(365.5)_8$ 轉成十六進位數字。 ② 試將 $(293.AC)_{16}$ 轉成八進位數字。

解答:

① 將八進位數字先轉成二進位數字，再由二進位轉成十六進位即可。

$$(365.5)_8 = (\underline{011}\ \underline{110}\ \underline{101}\ .\ \underline{101})_2$$
$$= (0\ \underline{1111}\ \underline{0101}\ .\ \underline{1010})_2$$
$$F \quad 5 \quad A$$
$$= (F5.A)_{16}$$

② 先將十六進位轉成二進位，之後再轉成八進位即可。

$$(293.AC)_{16} = (\underline{0010}\ \underline{1001}\ \underline{0011}\ .\ \underline{1010}\ \underline{1100})_2$$
$$= (\underline{001}\ \underline{010}\ \underline{010}\ \underline{011}\ .\ \underline{101}\ \underline{011})_2$$
$$1 \quad 2 \quad 2 \quad 3 \quad 5 \quad 3$$
$$= (1223.53)_8$$

練習:

1.計算 0.828125_{10}=__________$_{16}$

2.計算 BCD_{16}=______$_2$=_________$_{10}$

3.計算 1023_{10}=__________$_{16}$

4.計算 10010.101111_2=______$_8$=______$_{16}$

5.計算 571.2438=________$_{10}$

6.某資料存於某記憶體位址由 $B08D_{16}$ 到 $BCBB_{16}$，問其容量為幾 KB? _______

⌑ 補數的運算

二進位的補數分下列兩種

- 1's 補數:即為原數的相反(0 變 1，1 變 0)
- 2's 補數(2 進位的負數表示法)= 該數的 1's 補數+1

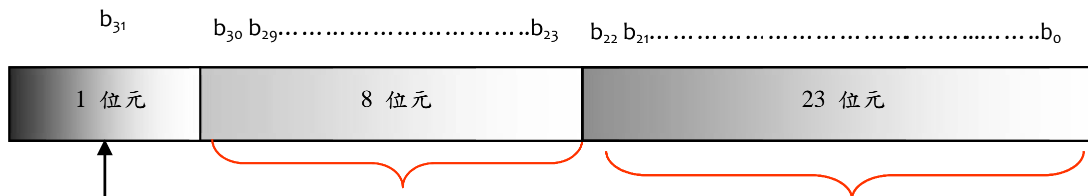
2's 補數表示法:若為 n bits，可表示的範圍為$-(2)^n \sim +(2)^n-1$

例:請寫出答案

	1's 補數	2's 補數
(1) 0011		
(2)10101100		
(3)11010011		

⌑ 浮點數表示法

- 表示實數資料
  - ❑ 單倍精確浮點數：32 位元。
  - ❑ 雙倍精確浮點數：64 位元。
  - ❑ 延伸精確浮點數：80 位元。
- 表示法



- 說明
  - ❑ 正/負符號
    - ➢ b31=0 表示此實數為正數；b31=1 表示此實數為負數。
  - ❑ 偏差指數
    - ➢ 8 位元表示的非負整數值為 0~255。
    - ➢ 實數可由很小至很大，故需要正、負二種指數，因此以 127 為指數偏差值，實際的指數值=偏差指數-127。
      - ◆ 偏差指數的範圍為 127～255，則代表真正指數值為 0～128。
      - ◆ 偏差指數介於 126～0 之間，則代表真正指數值介於-1～-127 之間。
      - ◆ 偏差指數是 132，其真正指數則為 5；偏差指數是 120，其真正指數即是-7。
  - ❑ 小數部分

- 此處的小數部分是經過正規化(normalization)後的小數。由於它有 23 位元，所以可準確到小數點後 23 位。

範例 19：-0.111×2^{-4}之浮點表示法爲何?

解答:

因此數是負值，故 $b_{31}=1$。並將-0.111×2^{-4}正規化成-1.11×2^{-5}，故其指數爲−5。但因指數偏差值爲 127，故偏差指數是-5 + 127 = 122，表示成八位元的二進數字爲 01111010。有效小數部分則爲 11$\underline{0\cdots0}$(21 個 0)。所以此數被儲存爲 1 01111010 11000000000000000000000。

範例 20： ±0 之浮點表示法爲何?

解答:

因 0 無法寫成正規化形式，故需以特殊情形待之。最簡單的方法即是令三部分皆爲 0。故±0 是以 0 0000000 000000000000000000000000。

數碼系統

種類

8-4-2-1 碼、二五碼、環形計數碼、五取二碼、超三碼、Gray Code(格雷碼)、BCD 碼、標準 BCD 碼、ASCII 碼、EBCDIC 碼、其它。

BCD 碼:每個十進位數可用 4bit 一組的二進數來表示。

十進數	BCD 碼
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

練習:

1. 轉換 BCD 碼為十進位數:

100101_{BCD}=__________$_{10}$ 11001_{BCD}=__________$_{10}$ 011000011001_{BCD}=__________$_{10}$

2. 轉換十進位為 BCD 碼:

104_{10}=__________$_{BCD}$ 547_{10}=__________$_{BCD}$ 359_{10}=__________$_{BCD}$

⌖ **ASCII: 美國資訊標準交換碼**

一般使用 7 個 bit 來表示一個字元(character)。是目前使用最廣泛的通信碼。

⌖ **Gray Code:** 通常用於資料的傳輸，不適合用於算術運算。(每一次只有一個位元異動)。

- ❖ 任何連續的兩個二進位表示法，只有一個位元不相同；其餘相同。
- ❖ 用二個位元來表示整數 0，1，2，3，

  方法一：

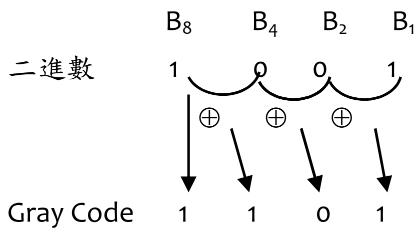
  即 G1=｛0=00，1=01，2=11，3=10｝

  方法二：

  即 G2=｛00=10，1=11，2=01，3=00｝。
- ❖ 學者研究出一種二進碼，稱為反射葛雷碼（Reflected Gray code），其編碼方式唯一而且有系統，故廣泛應用在計算機領域。

十進位	**Gray Code**	二進位
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111

⌖ **二進位化為 Gray Code:**



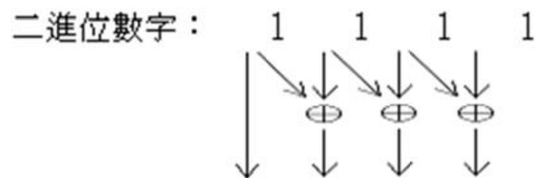
⊕：代表 XOR (即二者相異才為 1)

範例17：試將15以4位元反射葛雷碼表示之。

解答：

1. 先將15轉成4位元的二進位數字 = $(1111)_2$
2. 將二進位數字轉成反射葛雷碼：
   將反射葛雷碼的位元由左到右依次產生。
   最左邊的位元 = 二進位數字最左邊的位元。
   左邊第二個位元 = 二進位數字左邊第二個位元 ⊕ 最左邊的位元
   左邊第三個位元 = 二進位數字左邊第三個位元 ⊕ 左邊第二個位元
   最右邊的位元 = 二進位數字左邊第四個位元 ⊕ 左邊第三個位元

二進位數字：　1　1　1　1



反射葛雷碼：　1　0　0　0

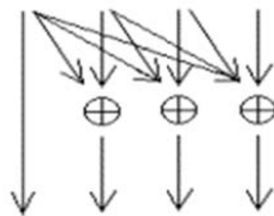
故得15的反射葛雷碼為 1000

範例18：試問4位元反射葛雷碼1100，其相對應的十進位數字為何？

解答：

1. 由 N_{RG} 先轉成 N_2，其方法如下：

N_{RG} = 1　1　0　0



N_2 = 1　0　0　0

2. 將二進位數字轉成十進位數字
   $N_2 = (1000)_2 = 8$
   故反射葛雷碼1100代表十進位數字8。

練習:

1. 十進位 5 的 Gray Code 為________
2. 二進位 101101 的 Gray Code 為__________

霍夫曼碼(Huffman Code)

- 不固定長度的編碼方式，符號編碼長度與出現頻率成反比。
- 編碼步驟
  - 找出所有符號出現頻率。
  - 將頻率最低的兩者相加得出另一個頻率。

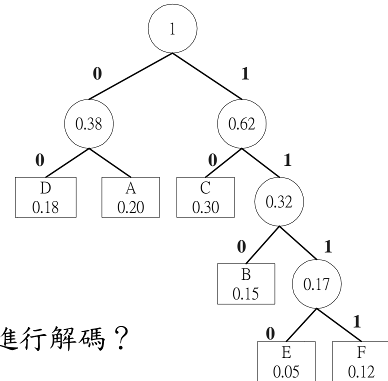
- 重覆以上第二步驟，將最低兩個頻率相加，直到只剩下一個頻率為止。
- 根據合併關係分配 0 與 1，而形成一棵編碼樹。

實例—編碼

- 假設編碼系統有 A, B,C, D, E, F 等六個符號，期出現頻率依序為 0.2, 0.15, 0.3, 0.18, 0.05, 0.12，試設計霍夫曼碼？
- 編碼結果

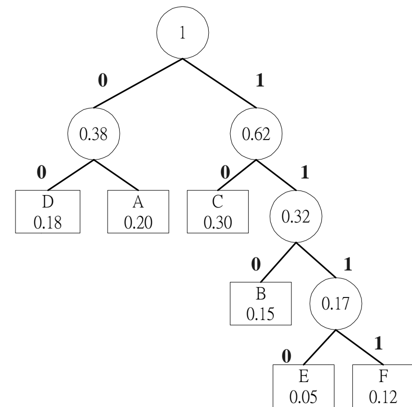
  A: 01; B:110; C:10

  D:00; E:1110;F:1111
- 總共所需位元：17 bits。



實例—解碼

- 請依照上一題所設計霍夫曼碼，將 111110010000110 進行解碼？
- 解碼結果
  - **FCADDB**



數碼檢查與更正

方法:

- 同位元檢查(Parity check)—又分為奇數同位元檢查和偶數同位元檢查
- 定數檢查(fixed-count check)
- Cyclic Redundancy Check(CRC)
- 漢明碼(Hamming Code)

同位元檢查

主要原理是於資料碼後自動加一個 parity bit (同位元)，使此資料碼內"1"的個數形成奇數或偶數。

若採奇同位元數碼檢查法，則 parity bit 和資料碼內 1 的個數總和應為奇數。

反之，則為偶同位元數碼檢查法。

例 1:使用奇同位:

資料碼	同位位元
1101101	0
1011001	1

例 2:使用偶同位:

資料碼	同位位元
1101101	1
1011001	0

【練習】:

1. 採用奇同位元檢查法傳送 7 位元資料，以下為接收端收到的各筆資料，何者可確知在傳送中有錯誤發生？ (A)11100000 (B)10110000 (C)10001111 (D)10101010。

2. 經加上偶同位元檢查後，下列代碼何者正確？ (A)1010111 (B)0110111 (C)1111001 (D)1011001。

⌖ Hamming Code

漢明碼為一個兼具自動錯誤偵測與更正一個 bit 的雙種功能，若用 7 個 bit，4 個代表 data，3 個代表檢查碼。

bit position	7	6	5	4	3	2	1	其中
Data	M_4	M_3	M_2		M_1			$c_1=M_4 \oplus M_2 \oplus M_1$ $c_2=M_4 \oplus M_3 \oplus M_1$ $c_3=M_4 \oplus M_3 \oplus M_2$
Check bit				C_3		C_2	C_1	

Hamming Distance:

代表兩組信號不同 bit 個數之和。

如:$A=(10101)_2$　$B=(11110)_2$， 因為有 3 個位元不同，所以 Hamming Distance=3.

Unit 2：布林代數與邏輯電路

Section 1: 布林代數

- 基本定理

(1) 單一律(Law of Tautology)

$\mathbf{A \cdot A = A}$

$\mathbf{A \cdot 1 = A}$

$\mathbf{A \cdot 0 = 0}$

$\mathbf{A + 0 = A}$

$\mathbf{A + 1 = 1}$

$\mathbf{A + A = A}$

(2) 交換律(Commutative Law)

$\mathbf{A \cdot B = B \cdot A}$

$\mathbf{A + B = B + A}$

(3) 結合律(Associative Law)

$\mathbf{A + (B + C) = (A + B) + C}$

$\mathbf{A \cdot (B \cdot C) = (A \cdot B) \cdot C}$

(4) 分配律(Distributive Law)

$\mathbf{A \cdot (B + C) = (A \cdot B) + (A \cdot C)}$

$\mathbf{A + (B \cdot C) = (A + B) \cdot (A + C)}$

(5) 補數(Complement)

$\mathbf{A = \overline{\overline{A}}}$

$\mathbf{A + \overline{A} = 1}$

$\mathbf{A \cdot \overline{A} = 0}$

(6) 第摩根定理(DeMorgan's Theorem)

$\mathbf{\overline{A \cdot B} = \overline{A} + \overline{B}}$

$\mathbf{\overline{A \cdot B \cdot C} = \overline{A} + \overline{B} + \overline{C}}$

$\mathbf{\overline{A + B} = \overline{A} \cdot \overline{B}}$

$\mathbf{\overline{A + B + C} = \overline{A} \cdot \overline{B} \cdot \overline{C}}$

- 真值表

A	B	A・B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

A	B	A→B
0	0	1
0	1	1
1	0	0
1	1	1

Section 2: 基本邏輯電路

⌖ 邏輯電路分類

1. 組合邏輯電路:
2. 基本邏輯閘
3. 半加器
4. 全加器
5. 比較器
6. 解碼器
7. 編碼器
8. 多工器

⌖ 基本邏輯閘

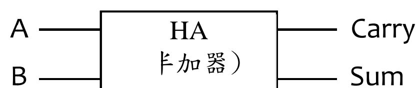
AND gate
OR gate
NOT gate
NAND gate=(NOT AND gate)
NOR gate=(NOT OR gate)
XOR gate=(Exclusive OR gate)
XNOR gate=(Exclusive NOR gate)

⌖ 半加器(Half Adder, HA):能處理 2 個 bit 的相加

A	B	Carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

CARRY = AB

SUM = $\overline{A}B + AB = \overline{A} \oplus B$



線路圖:

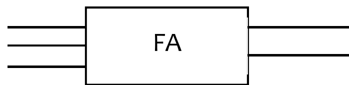
⌖ 全加器

能處理 3 個 bit 的相加

A	B	C	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Carry=$(A \oplus B)C + AB$

Sum=$(A \oplus B) \oplus C$



資料來源：http://www.ltivs.ilc.edu.tw/kocp/logic/ch3/3-4.htm

Section 3:布林代數的基本定理與定律

■ 基本定理

定理(1) $X \cdot 0 = 0$

說明：AND 的運算，在變數條件均為 1 時結果方為 1 ，此定理中的一個變數已經固定為 0，所以不管 X 為 1 或 0 其結果必為 0。

定理(2) $X \cdot 1 = X$

說明：AND 的運算，在變數條件均為 1 時結果方為 1 ，此定理中的一個變數已經固定為 1，若 X 為 1 則結果為 1，若 X 為 0 則結果為 0，所以 $X \cdot 1 = X$。

定理(3) $X \cdot X = X$

說明：AND 的運算，在變數條件均為 1 時結果方為 1 ，若 X 為 1 則 $1 \cdot 1 = 1$，若 X 為 0 則 $0 \cdot 0 = 0$，所以 $X \cdot X = X$。

定理(4) $X \cdot X' = 0$

說明：AND 的運算，在變數條件均為 1 時結果方為 1 ，而 X 與 X'總是相反的，亦即 $1 \cdot 0 = 0$ 或 $0 \cdot 1 = 0$，所以 $X \cdot X' = 0$。

定理(5) $X + 0 = X$

說明：OR 的運算，在變數條件任何一者為 1 時結果為 1 ，若 X 為 1 則 $1+0 = 1$，若 X 為 0 則 $0 + 0 = 0$，所以 $X + 0 = X$。

定理(6) $X + 1 = 1$

說明：OR 的運算，在變數條件任何一者為 1 時結果為 1 ，此定理中的一個變數已經固定為 1，所以 $X+1 = 1$。

定理(7) $X + X = X$

說明：OR 的運算，在變數條件任何一者為 1 時結果為 1 ，若 X 為 1 則 $1+1 = 1$，若 X 為 0 則 $0 + 0 = 0$，所以 $X + X = X$。

定理(8) $X + X' = 1$

說明：OR 的運算，在變數條件任何一者為 1 時結果為 1 ，而 X 與 X'總是相反的，亦即 1 + 0 = 1 或 0 + 1 = 1，所以 X + X'= 1。

布林代數除了以上的定理可以作為運算時的法則之外，在多變數的運算式中尚有一些定律可以運用

■ 基本定律

定律(1) 交換律：

a. $X+Y = Y+X$

b. $X \cdot Y = Y \cdot X$

定律(2) 結合律：

a. $(X+Y)+Z = X+(Y+Z)$

b. $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$

定律(3) 分配律：

a. $X \cdot (Y+Z) = XY+XZ$

b. $X+(Y \cdot Z) = (X+Y) \cdot (X+Z)$

c. $(W+X) \cdot (Y+Z) = WY+WZ+XY+XZ$

定律(4) 吸收律：

a. $X+XY = X$

b. $X+X'Y = X+Y$

証明：$X+XY = X$

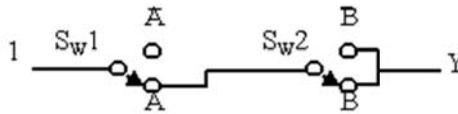
$X+XY$	$= X(1+Y)$	(分配律)
	$= X \cdot 1$	($\because 1+Y = 1$)
	$= X$	

証明：$X+X'Y = X+Y$

$X+X'Y$	$= X(1+Y)+X'Y$	($\because 1+Y = 1$)
	$= X+XY+X'Y$	(分配律)
	$= X+(X+X')Y$	(分配律)
	$= X+Y$	($\because X+X' = 1$)

Section 4: 布林代數的化簡

布林代數式可以描述一件事成功的途徑及條件，例如圖 3.4-1 中欲使 Y=1 的途徑有兩條，一條為開關 SW1 的 A 及 SW2 的 B 均接通，另一條為開關 SW1 的 A 及 SW2 的 B'均接通，布林代數式就為 Y=AB+AB'，但是我們發現只要開關 SW1 的 A 接通時，無論開關 SW2 在 B 或 B'的位置，Y 均等於 1，所以 Y=AB+AB'=A 是絕對成立的，這種代數式的簡化可以更清楚的知道事件成功的關鍵，減少了考慮的項目或變數。因此布林代數的化簡(simplification)是為了消除不必考慮的項或變數，化簡後邏輯結果不變。



(圖 3.4-1)Y=AB+AB 開關示意圖

■ 利用定理定律化簡

例 3.4-1-1

試化簡 Y =ABC+AB'C+ABC'+AB'C'

Y = AC(B+B')+ AC'(B+B') (∵分配律)

= AC+AC' (∵B+B'＝1)

= A(C+C') (∵分配律)

= A (∵C+C'＝1)

例 3.4-1-2

試化簡 Y=AB'C+A'B'C+BC

Y= AB'C+A'B'C+BC

= B'C(A+A')+BC

= B'C + BC

= C(B'+B)

= C

例 3.4-1-3

試化簡 Y =A(B+C)A'+D

Y= (AB+AC)A'+D

= (A'AB+A'AC)+D

= (0+0)+D (∵A'A＝0，0+0=0)

= D

例 3.4-1-4

試化簡 Y = A + A' + B

Y = A +A' + B

= (A + A') +B (結合律)

= 1+ B (∵A+A'=1)

= 1

例 3.4-1-5

試化簡 Y = AB'+ABC+A'B'C'

Y = AB'C+AB'C'+ABC+A'B'C' (∵AB'=AB'(C+C'))

= AB'C+ABC+ AB'C'+A'B'C' (交換律)

= AC(B+B')+B'C'(A+A')

= AC+B'C'

例 3.4-1-6

試化簡 Y = AB+A'BC

Y= B(A+A'C)

= B(A+C) (吸收律)

= AB+BC (分配律)

或 Y = AB+A'BC

= ABC+ABC'+A'BC (∵AB=AB(C+C')')

= ABC+ABC'+ABC+A'BC (∵ABC=ABC+ABC)

= AB+BC

■ **利用卡諾圖化簡**

卡諾圖(Karnaugh map)是化簡布林代數式的工具圖，它利用了 A+A'=1 的原理將相鄰的兩項得以消除，快速的得到最簡的布林代數式，化簡的方法請跟著例題學習。

例 3.4-2-1 試化簡 Y =AB+A'B

步驟 1 ：繪出兩個變數的卡諾圖

	B'	B
A'		
A		

步驟 2 ：將 Y 為 1 的項目先填入卡諾圖，其餘填 0。

本例 Y =AB+A'B 中有兩項，AB 或 A'B 均可令 Y=1

故卡諾圖應為：

	B'	B	
A'	0	1	A'B=1
A	0	1	AB=1

步驟 3 ：將相鄰為 1 的項圈起來。

(注意：所圈之項 N 的多少除了必須是相鄰為 1 的項之外，還必須滿足 N=2n，n 為正整數，例如 2、4、8、16 個)

	B'	B
A'	0	(1)
A	0	(1)

爲 1 的項 N=2
滿足 $N=2=2^1$

步驟 4：圈起來的項有互補變數者視為可消除之變數，再重新列出布林代數式。

本例中 A 與 A'互補，亦即 Y =AB+A'B=B(A+A')=B，所以今後卡諾圖中相鄰且互補的的變數可以得以消除，因此本例 Y=B 即為化簡結果。

(注意：圈中有兩個"1"，必然可以消去一個變數。)

例 3.4-2-2　試化簡 Y = AB'C+A'B'C+BC

步驟 1 繪出三個變數的卡諾圖

(注意：B'C'、B'C、BC、BC'的排列順序，而非 B'C'、B'C、BC'、BC，這是為了相鄰的項均有互為補數的變數以利消除化簡。)

	B'C'	B'C	BC	BC'
A'				
A				

步驟 2 將 Y 為 1 的項目先填入卡諾圖，其餘填 0。

因為 Y =AB'C+A'B'C+BC

= AB'C+A'B'C+ABC+A'BC

所以卡諾圖為：

	B'C'	B'C	BC	BC'
A'	0	1	1	0
A	0	1	1	0

步驟 3：將相鄰為 1 的項(N)圈起來，且 N=4 滿足=2n，n 為正整數 2。

	B'C'	B'C	BC	BC'
A'	0	1	1	0
A	0	1	1	0

爲 1 的項 N=4
滿足 $N=4=2^2$

步驟 4：圈起來的項有互補變數者視為被消除之變數，再重新列出布林代數式。

本例中 A 與 A'互補、B 與 B'互補，所以 Y=C 即為化簡結果。

(注意：圈中有四個"1"，必然可以消去兩個變數。)

例 3.4-2-3 試化簡 Y = A'B' + AB'C'D'+ AB'CD'

步驟 1 繪出四個變數的卡諾圖

	C'D'	C'D	CD	CD'
A'B'				
A'B				
AB				
AB'				

步驟 2 將 Y 為 1 的項目先填入卡諾圖，其餘填 0。

因為 A'B'=A'B'C+A'B'C'

=A'B'CD+A'B'CD'+A'B'C'D+A'B'C'D'

所以 Y = A'B'CD+A'B'CD'+A'B'C'D+A'B'C'D'+ AB'C'D'+ AB'CD'

則卡諾圖為：

	C'D'	C'D	CD	CD'
A'B'	1	1	1	1
A'B	0	0	0	0
AB	0	0	0	0
AB'	1	0	0	1

步驟 3 將相鄰為 1 的項圈起來。

	C'D'	C'D	CD	CD'
A'B'	1	1	1	1
A'B	0	0	0	0
AB	0	0	0	0
AB'	1	0	0	1

圈一：Y=A'B'

圈二：Y=B'D'
(四個角落也相鄰)

步驟 4 圈起來的項有互補變數者視為被消除之變數，再重新列出布林代數式。

圈一中 C'D'+C'D 可消去 D，C'D+CD 可消去 C'，CD+CD'可消去 D，CD'+C'D'可消去 C，化簡後剩下 A'B'。

圈二中直的看 A'B'+AB'消去 A 剩下 B'，橫的看 C'D'+CD'消去 C 剩下 D'，此圈化簡後為 B'D'。

故 Y= A'B'+ B'D' (注意：有幾個圈，化簡後的布林代數式就有幾個項，本例中有兩個圈，所以有 A'B'及 B'D'兩項。)

例 3.4-2-4

試用卡諾圖化簡真值表的輸出布林代數式。

由於真值表已經直接列舉了為 1 的輸出項，所以直接將 7 個為 1 的輸出項為 1 的輸出項填入卡諾圖化簡即可。

(注意：真值表內的變數排列為 DCBA，卡諾圖的排列左邊的變數就放 DC，上方則為 BA，而非 AB、CD，這樣的安排比較容易比對真值表的項。)

Y=1 的項包括：

D'C'BA'
D'CB'A'
D'CBA'
DC'B'A'
DC'BA'
DCB'A'

D	C	B	A	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

將為 1 的項填入卡諾圖後化簡：

	B'A'	B'A	BA	BA'
D'C'	0	0	0	1
D'C	1	0	0	1
DC	1	0	0	1
DC'	1	0	0	1

圈一
圈二
圈三

圈一：Y=BA'　　圈二：Y=CA'　圈三：Y=DA'

故化簡前有 7 個為 1 的輸出項，化簡後為 Y=BA'+CA'+DA'，僅有 3 個項而且每個項中的變數都減少了兩個。

Unit5: 數位影像簡介

資料來源：http://www.mustek.com.tw/Support/newer.htm#w16

何謂圖像解析度？

列印長度每單位的顯示像素數目來描述影像，通常解析度是每英吋的點數或是每英寸像素來測量。影像解析度決定於影像被建立且能被大多的影像軟體來改變的。如果此影像來自於掃描器，此掃描解析度將變成影像解析度，對於同樣的列印尺寸，一個高解析度的影像將比低解析度的影像包涵更多更小的像素對於一個影像輸入裝置的掃描器，掃描一個高解析度通常比較低解析度衍生出更詳細的影像。

影像解析度

影像解析度，是以 ppi 每英吋當中的像素點數（pixels per inch）來表示，其解析度、長度以及像素點數之間的關係如下：

像素點數 / 解析度 = 長度

長度 × 解析度 = 像素點數

例如：一張 3 × 5 吋的影像，解析度為 300 ppi，

則長寬點為：(3 × 300) × (5 × 300) = 900 × 1500 點。

何謂像素尺寸？

像素的數目沿著影像的寬度與高度 當由掃描器建立一影像，像素尺寸的效果為掃描解析度乘上掃描區域，舉例說明，假如我們掃描 4X6，100dpi 解析度的照片，其結果必為 400X600 像素，而其檔案大小比例根據其像素尺寸，然而掃描較高解析度將會製造出大檔案，然而，顯示器的大小對於影像尺寸或是列印紙的大小，像素的尺寸只是增加輸出裝置的解析度罷了。請參考掃描解析度，螢幕解希度和列表機解析度來了解更多知識。

何謂色階？

在圖像中在每個不同的顏色中顯示每個像素的最大數目，有時後可以用位元數來量測，通常表現的色彩模示為黑白模式、灰階與 RGB 三原色，其色階如下:

黑白模式 **1 bit** (2 to the 1st = 2 colors)

灰階模式 **8 bits** (2 to the 8th = 256 different gray levels**)**

全彩模式 **24 bits** (2 to the 24th = 16.7 millions colors)

何謂檔案大小？

檔案大小 = **像素尺寸 x 色階**，

舉例：

100 x 80 像素的圖像中，以黑白(1bit)呈像的的檔案大小為

100 x 80 x 1 bit = 8000 bits = 1000 bytes

100 x 80 像素的圖像中，以灰階(8bits) 顯示的大小為

100 x 80 x 8 bit = 64000 bits = 8000 bytes

100 x 80 像素的圖像中，以 RGB 三原色呈像的影像將需要

100 x 80 x (24 bit) = 100 x 80 x (3 bytes) = 24000 bytes

舉例說明，如果你掃描一個 letter-size(8.5x11inch)的尺寸 300dpi 彩色模式， 你將會轉換成一個

8.5 x 300 x 11 x 300 x 24 bits = 24MB

請完成下表：以 24bit 呈像的影像的檔案大小

Resolution (DPI)	Image Dimension		
	1" x 1"	4" x 6"	8.5" x 11"
72	15 KB	365 KB	1,420 KB
100	29 KB	703 KB	2,739 KB
300	264 KB	6,328 KB	24,653 KB
400	469 KB	11,250 KB	43,828 KB
500	732 KB	17,578 KB	68,481 KB
600	1,055 KB	25,313 KB	98,613 KB
1,200	4,219 KB	101,250 KB	394,453 KB

資料加密技術簡介

蘇俊銘 (Jun Ming Su)jmsu@csie.nctu.edu.tw

加密技術體系

- **按照密鑰的公開與否，可以分為 ：**
  - **對稱式加密體系(Symmetric Key)：**
    - 加密金鑰和解密金鑰是相同的。
    - 對稱演算法速度快，所以在處理大量資料的時候被廣泛使用，其關鍵是保證密鑰的安全。
    - **典型的演算法：**
      - 有 DES 及其各種變形(如 Triple DES)，IDEA，RC4、RC5
      - 對稱式加密法中最具影響的是 DES 密碼 。
  - **非對稱式金鑰(Asymmetric Key)體系：**
    - 分別存在一個公鑰和私鑰，公鑰公開，私鑰保密。
    - 公鑰和私鑰具有一一對應的關係，用公鑰加密的資料只有用私鑰才能解開。
    - 其效率低於對稱密鑰體系，
    - **典型的演算法：**
      - RSA、背包密碼，Elliptic Curve 等等。
    - 最有影響的公鑰加密演算法是 RSA：
      - 足夠位元數的 RSA 能夠抵抗到目前為止已知的所有密碼攻擊。

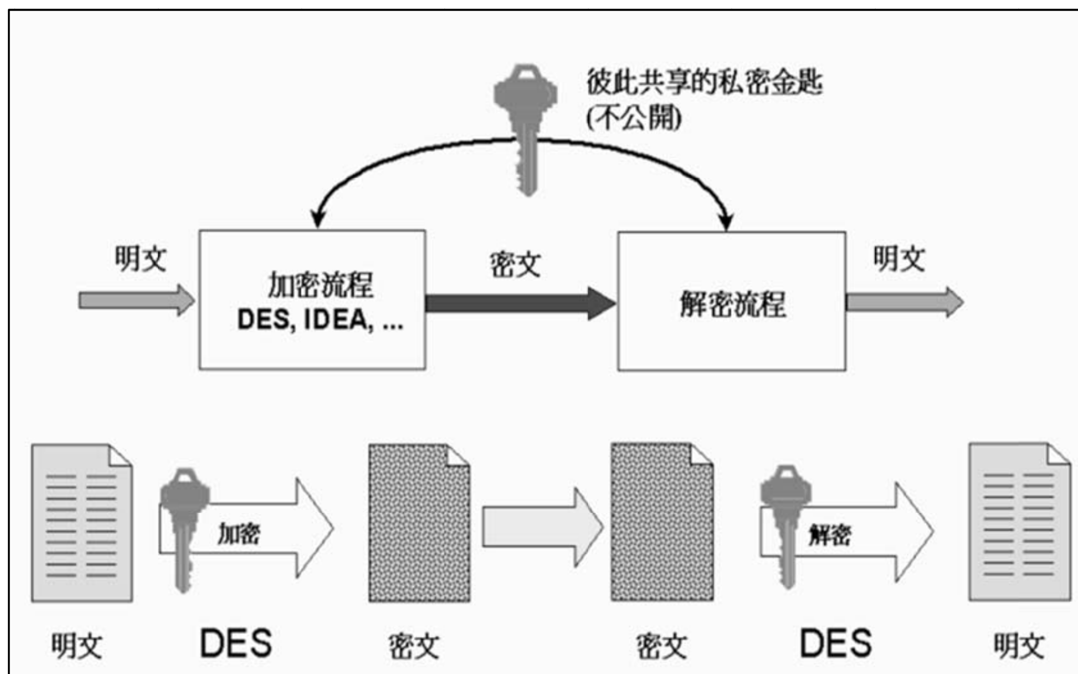


Figure 1-對稱式加密技術

DES (Data Encryption Standard)簡介

- **DES**：
  - 為目前最常用的對稱式金鑰密碼系統之一。
  - 由 IBM 公司在 1970 年代所發展出的加密演算法並在 1977 年經美國國家標準局(NBS）採用為聯邦標準（FIPS PUB 46-2）。
- **基本原理**：
  - 利用 Shannon 的多重加密的觀念，並利用 Confusion（混淆）與 Diffusion（散佈）等方式：將明文**(PlainText)**轉換成其他格式，並散佈明文的每一個小部分擴散到密文的各部分以達到加密效果。
  - **保密技巧即是**：
    - 將原始資料「明文」弄得非常散亂，讓破解者無法利用統計方式或其他數學分析技巧將加密後的「密文(Ciphertext)」還原成原來的明文。
  - DES 的加密方法是透過 16 回合的運算(位移)所組成：
    - 每一回合的運算目的，在於將上一回合所打散的明文在弄得更亂一些。
    - 就是指每一次的運算相當於在明文中多加了一道鎖，因此經過 DES 運算之後，其原始資料已被 16 把鎖給保護住了。
- DES 為一對稱式加密演算法
- **對稱式保密系統**：
  - 已知 *K*1 即知 *K*2。
- **非對稱式保密系統**：
  - 已知 *K*1，但無法得知 *K*2。
  - *K*1 稱為公開金鑰(Public Key)， *K*2 稱為私有金鑰(Private Key)。

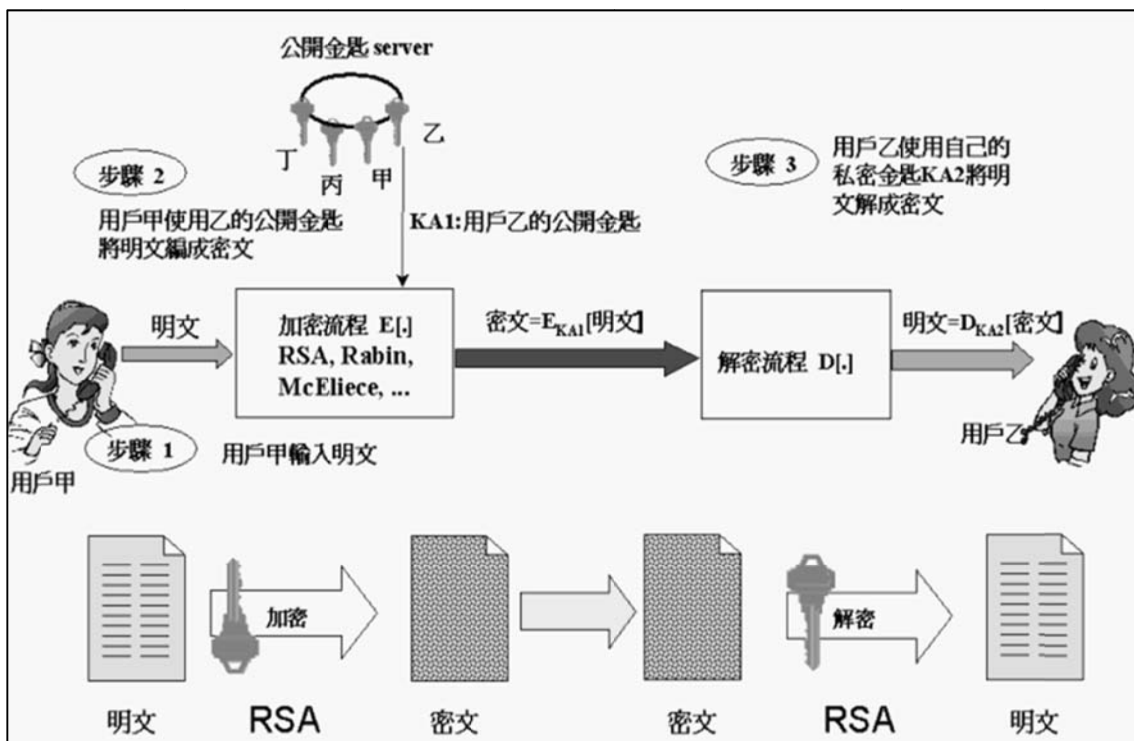
DES 之缺點

- **加密與解密需用同一個 key**：
  - 所以在傳輸時，要如何使雙方都使用同一個 key，為一重要問題。
- **DES 所使用的金鑰只有 56 bits**：
  - **DES 的區塊為 64 位元** ：
  - 因其中每個位元組(8 bits)皆取 1 位元作為同位 (parity) 核對故有效鍵長 56 位元 ：
    - 在其第 8、16、24、32、40、48、56、64 為同位元檢查碼，在做加密或解密動作時，其同位元檢查碼沒有真正使用。
  - 要破解 DES 金鑰相當於猜測 2^{56} 種數據資料，
  - 以今日的電腦硬體技術而言，該金鑰很容易破解。
    - 若以晶片處理速度可在一個 µsec(10^{-6})中分析一金鑰，
      - 則一天可測試 $8.64*10^{10}$ 把金鑰，大約 10^6 個晶片，在一天內就可以將所有可能的金鑰全部測試出來。
  - 因此專家認為金鑰應該考慮採用 **Triple-DES**。
    - 把原有的 DES 加解密位元擴充，為 56*3=168 位元，經過三次加解密程序，增加原有單一 DES 的安全性。
    - 但是 DES 目前仍然是一套相當方便的加密標準。

RSA 演算法 (非對稱式加密技術)

- **RSA 算取自于它的創始人的名字 ：**
  - 於 1977 年，美國麻省理工學院 Rivest，Shamir，Adelman 3 位教授所發表。
  - RSA 基於數學難題，具有大數因數分解。
  - RSA 使用兩個密鑰：公鑰(Public Key, PK)與私鑰(Private Key, SK)
    - 加密時把明文分成塊，塊的大小可變，但不超過密鑰的長度。RSA 把明文塊轉化為與密鑰長度相同的密文

- **RSA 運作方式**
  - 假設資料(Data)要由 **A 機器**傳至 **B 機器**：
    - 由 **B 機器**用亂數決定一個 key,我們稱之為私鑰 Private Key (SK) 。
    - 這個 key 都只留在 B 機器裡不送出來。
  - 由這個 SK 計算出另一個 key
    - 稱之為公開金鑰 Public Key (PK) 。
    - 這個 Public Key 的特性是幾乎不可能反演算出 Private Key
    - 然後將這個 Public Key 透過網路丟給 **A 機器**.
  - **A 機器**將資料(明文)用這個 Public Key 編碼：
    - 這個編碼過的資料(密文)一定得使用 Private Key 才解得開。
    - 然後 **A 機器**將**密文**透過網路傳給 **B 機器。**
  - B 再用 Private Key 將資料解碼。
  - 如果有第三者竊聽資料時：
    - 他只能得到 B 傳給 A 的 Public Key，以及 A 用這個 Public Key 編碼後的密文。
    - 沒有 SK，第三者無法解碼，所以 RSA 方法確實能防止第三者的竊聽。



- **RSA 的安全性依賴於大數分解：**
  - 公鑰和私鑰都是兩個大質數（ 大於 100 個十進制位）的函數。
  - 從一個密鑰(SK)和密文推斷出明文的難度等同於分解兩個大質數的積。
  - **密鑰對的產生：**
    - 選擇兩個大質數：p 和 q 。
    - 計算： $n = p \times q$
    - 然後隨機選擇<u>**加密密鑰 e**</u>，
      - 要求 e 和 $(p-1)\times(q-1)$ 互質，即 GCD(e, (p - 1) × (q - 1))=1。
    - 最後，利用 Euclid 算法計算<u>**解密密鑰 d**</u>：
      - 滿足: $e \times d = 1 \mod ((p-1)\times(q-1))$
      - 其中 n 和 d 也要<u>互質</u>。
      - 數 e 和 n 是<u>**公鑰**</u>，d 是<u>**私鑰**</u>。
      - 兩個質數 p 和 q 不再需要，應該丟棄，不要讓任何人知道。
  - **加密信息 m（二進制表示）時：**
    - 首先把 m 分成等長數據塊 m1, m2, ..., mi ，塊長 s ，其中 s≦ n， s 盡可能的大。
      - 0, 1, 2,..., 25 與 a, b, c,..., z 一對一對應，把訊息中的文字轉變成數字 M：
        - » 例如 YES→$M=24q^2+4q^1+18q^0$ 。
      - 對應的<u>密文</u>是：
        - » $c_i = (m_i^{\wedge e}) \mod n$ …… (a)
      - <u>解密</u>時作如下計算：
        - » $m_i = (c_i^{\wedge d}) \mod n$ …… (b)
  - **RSA 可用於數位簽章(Digital Signature)：**
    - 用（a）式簽名，用（b)式驗證。操作時考慮到安全性和 m 信息量較大等因素，一般是先作 HASH(雜湊) 運算。

RSA 與 DES 之比較

	RSA	DES
發表年代	1977	1976
發明人	美國麻省理工學院三位教授 Rivest、Shamir、 Adlemam 。	IBM 及美國國家安全局
基本特徵	加密 Key ≠解密 Key	加密 KEY = 解密 Key
主要優點	Public Key 可以公開，而且可以提供數位簽章	加密解密速度快
主要缺點	解密速度慢，Key 生成耗時， 初期系統成本高	Key 傳送困難並且 Key 必須共享

現今的加密技術

- **要破解 128 位元對稱加密的數位金鑰：**
  - 需要花上自現在至太陽再度循環為新星、吞噬地球所需的時間。
- **要破解 1024 位元非對稱加密的數位金鑰：**
  - 亦需要相同的時間。
- **比較可能的危險：**
  - 可能因不小心遺失了金鑰密碼提示句而使您的金鑰遭到盜取的機率，要遠高於金鑰遭到破解的機率。
- **SSL(Secure Socket Layer)：**
  - SSL 是由 Netscape 所發展，它是介於 Application Protocol 和 TCP/IP 間一個公開的、公用的資料安全性之通訊協定。
  - SSL 之功能有為**傳輸資料加密、連結伺服器之認證、確保傳送信息**之完整性等。
  - **即為與另一方在通訊之前先講好的一套方法。**
    - 此方法可以在通訊雙方之間建立一個秘密通道。
    - 凡是一些不希望被他人知道的機密資料都可以透過此通道傳送給對方

這樣一來即使資料必須要通過公開通路(如 Internet) ，也不用擔心資料會被別人偷窺。

電子交易安全 蘇偉慶(財金公司安控部高級工程師)

非對稱性系統

所謂非對稱性系統就是加密與解密分別使用不同的金鑰：公開金鑰(Public Key)與私密金鑰(Private Key)，因此亦稱為公開金鑰加解密系統。此系統具有以下特性：

・通訊雙方只需交換公開金鑰，因無隱密之特性，故無須以秘密通道進行交換。
・可經由認證中心(Certificate Authority，簡稱 CA)簡化金鑰之交換作業與信賴關係之建立。
・目前主要應用包括 FEDI、SET、FXML、VISA 3D-Secure 發卡行授權資料之簽章及部分網路銀行業務等。
・常見演算法包括 Diffi-Hellmen(主要用來做秘密金鑰之協商交換)、DSA (NIST 公告之簽章標準，只適用於數位簽章)及 RSA(可同時做加密與數位簽章)等。
・可應用於確保資料的隱密性、來源性、完整性及不可否認性。

由於 RSA 乃目前國內金融應用最廣的非對稱性加解密技術，以下將說明其在數位簽章與數位信封之應用。

一、數位簽章

數位簽章(Digital Signature)類似資料押碼，主要目的在防止非法第三者冒名傳送或竄改傳輸中的資料，以確保資料之來源辨識性與完整性。不同的是，若配合 CA 做金鑰之認證，數位簽章在法

律上已直接賦予其合法性。

雜湊(Hashing)是數位簽章進行資料前置處理不可或缺之技術，它具有以下特性：

1.將任意長度之輸入資料轉換成固定長度之資料。

2.具單向不可逆之特性，故又稱為單向函數(One Way Function)。

3.目前常見之演算法有 MD2、MD5(輸出長度均為 128 bits)、SHA-1(輸出長度為 160 bits)等。

4.單獨使用無法防範惡意之資料竄改，通常搭配數位簽章使用。

5.若加入金鑰運算(如 RFC2104 定義的 HMAC: Keyed-Hashing for Message Authentication)，可確保資料的正確性及來源辨識性。

傳送端產生數位簽章前，先將欲簽章之本文經由雜湊運算取得訊息摘要(Message Digest)，再以私密金鑰對訊息摘要做解密(簽章)運算。(訊息摘要在簽章前可能需做 Padding 處理，細節在此略過。)最後將數位簽章連同本文一併送至接收端。

接收端取得訊息後，先取出本文，以相同雜湊運算取得訊息摘要，然後再以傳送者的公開金鑰對數位簽章做加密(驗章)運算，將結果與訊息摘要比對，即可判斷訊息之正確性。

在此作業下，因私密金鑰只有傳送者才知道，也只有傳送者可產生對應之簽章，故可確保訊息的來源辨識(唯一)性及正確性，另若公開金鑰經 CA 認證，更可確保其不可否認性。

二、數位信封

RSA 公開金鑰系統同樣可運用於資料之加密，然而基於效能考量，通常不以 RSA 演算法直接對本文做加密，而是採用數位信封(Digital Envelop)技術，其特色是以快速的對稱性加解密演算法進行大量資料之加密運算，而以非對稱性加解密演算法解決對稱性金鑰基碼交換之棘手問題。

傳送端首先任意產生用以加密此次本文資料之秘密金鑰(Session Key)，對欲傳送之本文做加密，再以接收端之公開金鑰對此 Session Key 做加密，最後將密文與加密保護後之 Session Key 一併傳送至接收端。

接收端在取得訊息時，先以其私密金鑰解密取得正確的 Session Key，再以此對密文做解密，即可取得完整之明文資料。

在此作業下，只有接收者知道 Session Key 解密所需之私密金鑰，因此可確保傳送資料之隱密性。

結語

密碼學是提供電子銀行交易安全的關鍵技術，根據業務性質、交易對象、安全需求及相關法令規定，選擇適當之安全技術，將可有效降低交易之安全威脅，進而提供便捷且安全之電子銀行交易平台。

Unit 11：動態規劃法(Dynamic Programming, DP)

By 黃易學姐

+ 概念：將問題分割成許多子問題(subproblem)，利用已解決之子問題求出原問題的解。想法類似遞迴(recursion)，但以表格記錄的方式，避免重複計算相同的子問題。一般用於求問題之最佳解。

+ 做法：1. 列出問題之最佳解(optimal solution)的遞迴式。2. 從最小的子問題開始依序計算其最佳解(bottom-up)並以表格記錄結果。3. 最後得到問題之最佳解。

+ 舉例：費式數列(Fibonacci Sequence)

若費式數列之第一、二項為1，而其它每一項都為其前兩項之和，則遞迴式可表示如下：

$$f(n)=\begin{cases}1, 0\le n<2\\ f(n-1)+f(n-2), n\ge 2\end{cases}$$

其中，以 n=0 為數列之第一項。

+ 相關應用題型：最短路徑問題(Shortest Path)、最長遞增子序列(Longest Increasing Subsequence, LIS)、最長共同子序列(Longest Common Subsequence, LCS)、背包問題(Knapsack)、零錢問題、最大連續和(Maximum Consecutive Sum)、最大子矩陣、最大矩形、矩陣相乘(Matrix-Chain Multiplication)、拿石頭、旅行推銷員問題(Traveling Salesman Problem, TSP)、爬樓梯問題、貼磁磚問題(Tiling)……

+ 範例程式碼：最長遞增子序列

```
//  最長遞增子序列(LIS)
//  求 s 之最長遞增子序列的長度
//  2008.05.26    Celia
#include <stdio.h>
#define N 6                 //  s 的大小

int s[N] = {1, 2, 9, 3, 4, 10},
    len[N];                 //  len[i]:  以 s[i] 為結尾之子序列的最大長度

int main() {
   int i, j,
      max_len = 0;          //  s 之子序列的最大長度(LIS 的長度)

   for(i = 0; i < N; i++)
      len[i] = 1;           //  以 s[i] 為結尾之子序列長度至少是1

   for(i = 0; i < N; i++) {
      for(j = 0; j < i; j++)    //  如果 s[i] 可以接在 s[j] 後面，而且接在 s[j] 後面所形成之 LIS 長度比之前長
         if((s[i] > s[j]) && (len[i] <= len[j]))
            len[i] = len[j] + 1;

      if(max_len < len[i])      //  更新 LIS 最大長度
         max_len = len[i];
   }

   printf("%d\n", max_len);
   return 0;
}
```

+ 範例程式碼：最長共同子序列

```
//  最長相同子序列 (LCS)
//  求 s1和 s2之 LCS 長度
//  2008.06.07     Celia
#include <stdio.h>
#define N1 6              //  s1序列的大小
#define N2 8              //  s2序列的大小

char s1[N1+1] = {' ', 'A', 'U', 'G', 'E', 'C', W'},
     s2[N2+1] = {' ', 'X', 'A', 'I', 'E', 'C', 'U', 'W', 'G'};

int main() {
    int i, j,
       v[N1+1][N2+1];      //  v[i][j]: s1[1]~s1[i]和 s2[1]~s2[j]之 LCS 長度

    for(i = 0; i < N1; i++)  //  初始化
       v[i][0] = 0;
    for(j = 0; j < N2; j++)
       v[0][j] = 0;

    for(i = 1; i < N1; i++) {
       for(j = 1; j < N2; j++) {
          if(s1[i] == s2[j])
             v[i][j] = v[i-1][j-1] + 1;
          else
             v[i][j] = (v[i][j-1] > v[i-1][j])? v[i][j-1] : v[i-1][j];
       }
    }

    printf("%d\n", v[i][j]);
    return 0;
}
```

+ 範例程式碼：最大連續和

```
// 最大連續和
// 求一數列之連續子序列的最大和
// 2008.05.31     Celia
#include <stdio.h>
#define N 9              // 數列長度

int main() {
    int i,
        a[N] = {-3, 9, 0, -2, 8, 3, -9, -1, 4},
        sum = 0,          // (從某位置開始)到目前為止的和
        max_sum = 0;      // 最大之和

    for(i = 0; i < N; i++) {
        if(sum < 0)
            sum = a[i];     // 不加之前的和會比較大
        else
            sum += a[i];    // 加之前的和比較大

        if(max_sum < sum)
            max_sum = sum;
    }

    printf("%d\n", max_sum);

    return 0;
}
```

+ 範例程式碼：0-1 背包問題

```
//  背包問題(0-1 knapsack)
//  每項物品只有一個，求可裝入背包的最大價值
//  2008.05.30     Celia
#include <stdio.h>
#include <string.h>
#define N 6              //  物品數量
#define M 30             //  背包最大承重

int main() {
    int i, j,
       value[N+1] = {0, 100, 50, 5, 250, 120, 10},   //  物品價值
       weight[N+1] = {0, 10, 9, 1, 15, 20, 5},       //  物品重量
       v[N+1][M+1];       //  v[i][j]:  最多放 i 個物品，且物品總重不超過 j 時，背包中物品的最大價值

    memset(v, 0, sizeof(v)); //  將 v 歸零

    for(i = 1; i <= N; i++) {
       for(j = 1; j <= M; j++) {
          v[i][j] = (v[i-1][j] > v[i][j-1])? v[i-1][j] : v[i][j-1];

          if((weight[i] <= j) && (v[i][j] < v[i-1][j-weight[i]] + value[i]))
             v[i][j] = v[i-1][j-weight[i]] + value[i];
       }
    }

    printf("%d\n", v[N][M]);

    return 0;
}
```

+ 附表：

最短路徑問題(Floyd-Warshall Algorithm)	**ACM** 534 544 567 10048 10099 11015 *//869 10000 10342* **Zero** b117(**TOI** 2008-4)
最短路徑問題(Dijkstra's Algorithm)	**TIOJ** 1290
最長遞增子序列	**ACM** 481 497 *//437 10131 10534 11240* **TIOJ** 1175
最長共同子序列	**ACM** 111 10066 10192 10405 10635 10949
最長共同遞增子序列	**TIOJ** 1051(**NPSC** 2003-final-G)
0-1 背包問題	**ACM** 562 624 10664 **Zero** b116(**TOI** 2008-3)
零錢問題	**ACM** 147 166 357 674
最大連續和	**ACM** 10684
最大子矩陣	**ACM** 108 836 10074
最大矩形	**TIOJ** 1063(北市95-5)
矩陣相乘	**ACM** 348 *//10003*
拿石頭	**ACM** 10404 10891
爬樓梯問題	**ACM** 825 *//10157*
貼磁磚問題	**ACM** 900 10359 10918
未分類	**ACM** 104 116 435 568 884 10154 10237 10254 10271 10497 10564 10603 10702 10912 11000 11003 11401 **TIOJ** 1019 1288 1291

Celia 學姊說：

我覺得一升二的學妹們應該要先看一下排列組合和機率，因為去年北市賽加一加這兩樣份量好像挺重的。
還有就是 C++一些內建的函式和類別，如果熟悉使用的話在比賽中可以省下不少時間。但是大部分人都是學 C，而且就算是學 C++的其實也很少人會去接觸這些東西。我大部分都是利用 C/C++ Reference 上面的資料加上自己摸索學會的…

附上我跟鮪魚的 Blog(我沒有徵詢她的同意，不過我想她不會介意的)
基本上裡面只有滿滿的 Code…

Anny 老師說：

真的想不出來時；或寫出來了，但覺得不夠好時，可以參考學姐們的想法…同時也可以學習學姐們整理程式競賽題的方法哦!

http://www.wretch.cc/blog/celiaailec

http://bluefintuna.wordpress.com/

這真是『無價之寶』哦!!

Unit 12：動態規劃法(Dynamic Programming, DP)例題與解法

By 黃易學姐

celia781011@gmail.com

Section I - DP 例題列表

Problem 1：最大矩形(Area)......2
- Problem 1-1 : Take the Land (ACM 10074)......3

Problem 2：正直 DE (2007 NPSC 高中組決賽 b090: D. 正直 DE)......**4**
- Problem 2-1 : Matrix Chain Multiplication (ACM Q442)......6
- Problem 2-2 : Optimal Array Multiplication Sequence (ACM Q348)......8
  - Problem 2-2-1 : Cutting Sticks (ACM Q10003)......10
  - Problem 2-2-2 : Mixtures (SPOJ Problem Set 345. Mixtures)......11

Problem 3 : Walking on the Safe Side (ACM Q825)......**12**
- Problem 3-1 : Expressions (ACM Q10157)......14
  - Problem 3-1-1 : How Many Trees ? (ACM Q10303)......16
  - Problem 3-1-2 : Count the Trees (ACM Q10007)......17
  - Problem 3-1-3 : Safe Salutations (ACM 991)......18

Problem 4 : Tiling (ACM Q10359)......**19**
- Problem 4-1 : Brick Wall Patterns (ACM 900 - Brick Wall Pattern)......20
- Problem 4-2 : Tri Tiling (ACM Q10918: Tri Tiling)......21
- Problem 4-3：鋪磁磚問題(94 北市賽-prob 3)......22

縮排縮比較遠代表和原本題目較大，但是都是類似的想法。

*是要用到 Backtracking 的題目。照難度排是 4312~

Section II 題目與解析

Problem 1：最大矩形(Area)

Description

在一個 M x N 的區域內，散落了許多不同的障礙物，我們想要知道的是，在這個 M x N 的區域內，最大的矩形空地面積是多少？倘若我們用 0 與 1 表示這個區域內的空地狀況：0 代表這個子區域已被障礙物覆蓋，1 代表這個子區域仍為空地，我們假設每一個 0 或 1 所代表的子區域面積為 1，那麼在下面這個例子中（M=4，N=5），最大的矩形空地為陰影所覆蓋的區域，其面積為 8。

```
0  0  1  1  0
0  1  1  1  1
0  1  1  1  1
0  0  1  0  0
```

在本題中，請依據輸入輸出的規定，針對輸入的地圖，輸出其最大的矩形空地面積。

Input

輸入檔第一行有兩個整數，依序為 M 和 N, M≦200, N≦200；接下來的 M 行中，每一行有 N 個 0 或 1 的數字。這 N 個數字彼此間用一個空白隔開。

Output

請將最大矩形空地面積寫出至輸出檔。

Sample Input

```
4 5
0 0 1 1 0
0 1 1 1 1
0 1 1 1 1
0 0 1 0 0
```

Sample Output

```
8
```

From: 95 北市賽(prob 5)

解析：

Problem 1-1 : Take the Land (ACM 10074)

Description

有一個窮人來到國王那裡對國王說：「國王呀！景氣太差了，我已經無法養家活口了！請給我一點錢吧！」國王回答說：「現在國庫空虛，無多餘的錢給你。但是我可以給你一塊土地讓你去耕種謀生。在南部有一大片矩形的森林，那裡的樹被很整齊的種著。其中有些樹已經被砍下來了，現在我允許你可以佔有森林中一塊矩形的土地，條件是：這塊矩形土地上的樹都必須已經被砍掉了。來人啊！把森林的地圖拿來！你看，地圖上標示為 1 的地方代表種有一棵樹，標示為 0 的地方代表那棵樹已經被砍掉了。」

你的任務就是幫助這位窮人找出地圖中最大且上面無樹的矩形面積(長度以樹為單位)。

Input

輸入含有多組測試資料。每組測試資料的第一列有 2 個整數 M,N（1 <= M,N <= 100）。代表森林中樹的列數與行數。接下來的 M 列每列有 N 個字元（0 或 1），相鄰的字元中間以一空白字元相隔。這 M 列就是森林的地圖。

當 M、N 都為 0 時，代表輸入結束。請參考 Sample Input。

Output

對於每一組測試資料輸出一列，輸出地圖中最大且上面無樹的矩形面積。

Sample Input

```
6 7
0 1 1 0 1 1 0
0 0 0 0 0 1 0
1 0 0 0 0 0 1
0 1 0 0 0 0 1
1 1 0 0 0 1 0
1 1 0 1 1 0 0
3 5
0 1 0 1 0
1 0 1 0 1
0 1 0 1 0
0 0
```

Sample Output

```
12
1
```

From: ACM 10074

解析：

Problem 2： 正直 DE (2007 NPSC 高中組決賽 b090: D. 正直 DE)

Description

日本的貴族學校「Agnes」要來台開設分校了！

這間在日本以優秀研究成果享譽國際的大學，五年前在 Agnes 學園校長 DE（學園高層充滿著不為人知的機密，故校長的名稱用神秘的代號 DE 稱之）熟嫻的外交手段經營下，向業界招募了五年五千億的大筆資金，並在極短的時間內超英趕美，登上全世界大學的 TOP10。

在經過多年考核後，董事會決定在台灣設立國外第一所分校，台大校長在聽聞此消息後，準備從今年高中生中挑出間諜，潛入該學校收集情報，而你就是那位被挑中的天才高中生。

為了確保你可以通過「Agnes」的入學考試（雖然你是台灣最天才的高中生，但是要面對世界各地的高中生仍是一件不容易的事情！），台大派出校園內馬尾綁最好的學妹向「Agnes」的教學長蜜蜂騙取得了入學考試的題目（沒想到教學長喜歡馬尾女孩的傳聞是真的！）。

在看過入學考試的題目後，你對於拿滿分非常有自信，但為了降低間諜任務失敗的機率（任務失敗你就…嘿嘿嘿…），你決定在考前事先研究出計算該題目最快的方法（因為答案實在太難記了）。

<!--[if !supportEmptyParas]--> <!--[endif]-->

以下便是「Agnes」的入學考試題目：

> 2007 年 Agnes 學園入學考考題
> 試題卷共 1 頁，1 題，滿分 100 分
> <!--[if !supportEmptyParas]--> <!--[endif]-->
> **第一題： (共 100%)**
> ①定義二元數學運算子∮（A,B）=C，(此符號稱為 tera operator）
> A,B,C 各為一矩陣，其中 $C(i,j)=\sum_{k} A(i,k)\Theta B(k,j)$
> ②定義二元數學運算子 <!--[if !vml]--> $\Theta(X,Y)=\begin{pmatrix} X*Y & if & Y>X \\ Y*X & if & Y\le X \end{pmatrix}$ <!--[endif]-->（此符號稱為彼彼運算子），其中 X,Y 各為一非負實數。
> <!--[if !supportEmptyParas]--> <!--[endif]-->
> 請按照背面的數值，計算每小題 A_1∮A_2∮A_3… ∮A_{N-1}∮A_N 之結果。
> （請翻頁繼續作答）

聰明的你很快就發現了幾件重要的事實：

<!--[if !supportLists]-->1. <!--[endif]-->A 矩陣的 Column 數目一定要等於 B 矩陣的 Row 數目才可以做∮運算。

<!--[if !supportLists]-->2. <!--[endif]-->C 矩陣的大小一定會是 Row(A)*Column(B)，也就是 Row(C)=Row(A) 且 Column(C)=Column(B)。

<!--[if !supportLists]-->3. <!--[endif]-->彼彼運算需要大量的計算時間，計算一次 C=∮（A,B）需要 Row(A)* Column(A)*Column(B) 個彼彼運算。

<!--[if !supportLists]-->4. <!--[endif]-->要計算超過 2 個以上的 tera operation 時，依照不同的順序會需要不同的彼彼運算次數。

例如：要計算 A∮B∮C 的話，有兩種選擇：

（(A∮B) ∮C） 或者 (A∮(B∮C))。

<!--[if !supportEmptyParas]--> <!--[endif]-->

假設 A 是 7x10 的矩陣，B 是 10x22 的矩陣，C 是 22x37 的矩陣，則

(A∮B) ∮C 需要(7*10*22) + (7*22*37) = 7238 次彼彼運算

A∮(B∮C) 需要(10*22*37)+(7*10*37) = 10730 次彼彼運算

<!--[if !supportLists]-->5. <!--[endif]-->雖然利用其它的特殊技巧可以更快速的計算出答案，但為了防止自己算錯，你決定只利用添加括號的方式來加快運算速度（也就是改變運算的順序）。

<!--[if !supportEmptyParas]--> <!--[endif]-->

根據你不平凡的智力，計算 500 次彼彼運算需要一張 A4 白紙，因此一張雙面 A4 總共可計算 1000 次，因為此次入學考試由正直的 DE 親自監考，正直的 DE 常常懷疑學生不正直，故你最好事先計算出最低所需的計算紙數量，以免當天考試因為帶太多計算紙而被正直的 DE 懷疑你的身份！
<!--[if !supportEmptyParas]--> <!--[endif]-->

Input

<!--[if !supportEmptyParas]--> <!--[endif]-->

輸入檔以一個數字 T 為開頭，代表題目卷共有 T 個小題（1<=T<=5）。

每個小題以一個整數 N（1<=N<= 1000），代表有幾個矩陣需要做運算，接著有 N 行輸入，R_i 和 C_i 分別代表矩陣 A_i 的 Row 及 Column 大小（1<=R_i<=1000， 1<=C_i<=1000，1<=i<=N）。

Output

對每一組小題，你應該輸出一個非負整數 H，代表若單獨計算該小題最少需要幾張計算紙，且在最後一行輸出整場考試最少需使用幾張計算紙。

Sample Input

```
2
3
7 10
10 22
22 37
6
30 35
35 15
15 5
5 10
10 20
20 25
```

Sample Output

```
8
16
23
```

From : 2007 NPSC 高中組決賽

解析：

Problem 2-1 : Matrix Chain Multiplication (ACM Q442)

Description

假設你必須做 A*B*C*D*E 的運算，在這裡 A,B,C,D,E 都是矩陣（matrix）。由於矩陣相乘具有連結性（associative），所以相乘的順序可以是任意的。然而所需要的基本乘法數卻與不盡相同。

例如：A 是個 50*10 的矩陣，B 是個 10*20 的矩陣，C 是個 20*5 的矩陣。那麼就有 2 種不同的表示式來求出 A*B*C。分別是(A*B)*C 和 A*(B*C)。而其所需用到的基本乘法數前者為 15000 次，後者為 3500 次。

給你某一種矩陣相乘的表示式，你的任務是寫一個程式算出需要多少個基本乘法。

Input

輸入分為 2 部分。第一部份為矩陣的資料，第二部分為矩陣相乘的表示式。

第一列有一個整數 n（1<= n <= 26），代表在輸入的第一部份有多少個矩陣。接下來的 n 列每列為一矩陣的資料。內容為 1 個大寫英文字母及 2 個整數，分別代表矩陣的名字，欄數（rows）及列數（columns）。

輸入的第二部分每列為一測試資料，內容為矩陣相乘的表示式。表示式嚴格的遵守以下的語法（以 EBNF 的形式）

SecondPart = Line { Line } <EOF>

Line = Expression <CR>

Expression = Matrix | "(" Expression Expression ")"

Matrix = "A" | "B" | "C" | ... | "X" | "Y" | "Z"

請參考 Sample Input。

Output

對輸入第二部分的每組測試資料輸出一列。如果該表示式為不合法的矩陣相乘，則輸出 error。否則請輸出此表示式所需的乘法次數。請參考 Sample Output。

Sample Input

```
9
A 50 10
B 10 20
C 20 5
D 30 35
E 35 15
F 15 5
G 5 10
H 10 20
I 20 25
A
B
```

```
C
(AA)
(AB)
(AC)
(A(BC))
((AB)C)
(((((DE)F)G)H)I)
(D(E(F(G(HI)))))
((D(EF))((GH)I))
```

Sample Output

```
0
0
0
error
10000
error
3500
15000
40500
47500
15125
```

From ACM Q442

解析：

Problem 2-2 : Optimal Array Multiplication Sequence (ACM Q348)

Description

給你 2 個矩陣 A、B，我們使用標準的矩陣相乘定義 C=AB 如下：

$$C_{i,j} = \sum_k A_{i,k} \times B_{k,j}$$

A 陣列中欄（column）的數目一定要等於 B 陣列中列（row）的數目才可以做此 2 陣列的相乘。若我們以 rows(A)，columns(A)分別代表 A 陣列中列及欄的數目，要計算 C 陣列共需要的乘法的數目為：rows(A)*columns(B)*columns(A)。例如：A 陣列是一個 10x20 的矩陣，B 陣列是個 20x15 的矩陣，那麼要算出 C 陣列需要做 10*15*20，也就是 3000 次乘法。

要計算超過 2 個以上的矩陣相乘就得決定要用怎樣的順序來做。例如：X、Y、Z 都是矩陣，要計算 XYZ 的話可以有 2 種選擇：(XY)Z 或者 X(YZ)。假設 X 是 5x10 的陣列，Y 是 10x20 的陣列，Z 是 20x35 的陣列，那個不同的運算順序所需的乘法數會有不同：

(XY)Z

- 5*20*10 = 1000 次乘法完成(XY)，並得到一 5x20 的陣列。
- 5*35*20 = 3500 次乘法得到最後的結果。
- 總共需要的乘法的次數：1000+3500=4500。

X(YZ)

- 10*35*20 = 7000 次乘法完成(YZ)，並得到一 10x35 的陣列。
- 5*35*10 = 1750 次乘法得到最後的結果。
- 總共需要的乘法的次數：7000+1750=8750。

很明顯的，我們可以知道計算(XY)Z 會使用較少次的乘法。

這個問題是：給你一些矩陣，你要寫一個程式來決定該如何相乘的順序，使得用到乘法的次數會最少。

Input

含有多組測試資料，每組測試資料的第一列，含有 1 個整數 N（N <= 10）代表有多少個陣列要相乘。接下來有 N 對整數，代表一陣列的列數及欄數。這 N 個陣列的順序與要你相乘的陣列順序是一樣的。

N=0 代表輸入結束。請參考 Sample Input。

Output

每組測試資料輸出一列，內容為矩陣相乘的順序（以刮號來表示）使得所用的乘法次數最小。如果有不只一組答案，輸出任一組均可。請參考 Sample Output。

Sample Input

```
3
1 5
5 20
20 1
3
5 10
10 20
20 35
6
30 35
35 15
15 5
5 10
10 20
20 25
0
```

Sample Output

```
Case 1: (A1 x (A2 x A3))
Case 2: ((A1 x A2) x A3)
Case 3: ((A1 x (A2 x A3)) x ((A4 x A5) x A6))
```

From ACM Q348

解析：

Problem 2-2-1 : Cutting Sticks (ACM Q10003)

Description

你的任務是替一家叫 Analog Cutting Machinery (ACM)的公司切割木棍。切割木棍的成本是根據木棍的長度而定。而且切割木棍的時候每次只切一段。

很顯然的，不同切割的順序會有不同的成本。例如：有一根長 10 公尺的木棍必須在第 2、4、7 公尺的地方切割。這個時候就有幾種選擇了。你可以選擇先切 2 公尺的地方，然後切 4 公尺的地方，最後切 7 公尺的地方。這樣的選擇其成本為：10+8+6=24。因為第一次切時木棍長 10 公尺，第二次切時木棍長 8 公尺，第三次切時木棍長 6 公尺。但是如果你選擇先切 4 公尺的地方，然後切 2 公尺的地方，最後切 7 公尺的地方，其成本為：10+4+6=20，這成本就是一個較好的選擇。

你的老闆相信你的電腦能力一定可以找出切割一木棍所需最小的成本。

Input

每組測試資料 3 列，第一列有 1 個整數 L（L<1000），代表需要切割的木棍的長度。第二列有一個整數 N（N<50），代表需要切的次數。第三列有 N 個正整數 C_i（$0 < C_i < L$）代表木棍需被切割的地方。這 N 個整數均不相同，且由小到大排列好。

L=0 代表輸入結束。請參考 Sample Input。

Output

對每一組測試資料，輸出最小的切割成本。請參考 Sample Output。

Sample Input

```
100
3
25 50 75
10
4
4 5 7 8
0
```

Sample Output

```
The minimum cutting is 200.
The minimum cutting is 22.
```

From ACM Q10003

解析：

Problem 2-2-2 : Mixtures (SPOJ Problem Set 345. Mixtures)

Description

Harry Potter has n mixtures in front of him, arranged in a row. Each mixture has one of 100 different colors (colors have numbers from 0 to 99).

He wants to mix all these mixtures together. At each step, he is going to take two mixtures that stand next to each other and mix them together, and put the resulting mixture in their place.

When mixing two mixtures of colors a and b, the resulting mixture will have the color (a+b) mod 100.

Also, there will be some smoke in the process. The amount of smoke generated when mixing two mixtures of colors a and b is a*b.

Find out what is the minimum amount of smoke that Harry can get when mixing all the mixtures together.

Input

There will be a number of test cases in the input.

The first line of each test case will contain n, the number of mixtures, 1 <= n <= 100.

The second line will contain n integers between 0 and 99 - the initial colors of the mixtures.

Output

For each test case, output the minimum amount of smoke.

Sample Input

```
2
18 19
3
40 60 20
```

Sample Output

```
342
2400
```

In the second test case, there are two possibilities:

- first mix 40 and 60 (smoke: 2400), getting 0, then mix 0 and 20 (smoke: 0); total amount of smoke is 2400
- first mix 60 and 20 (smoke: 1200), getting 80, then mix 40 and 80 (smoke: 3200); total amount of smoke is 4400

The first scenario is a much better way to proceed.

From SPOJ Problem Set 345. Mixtures

解析：

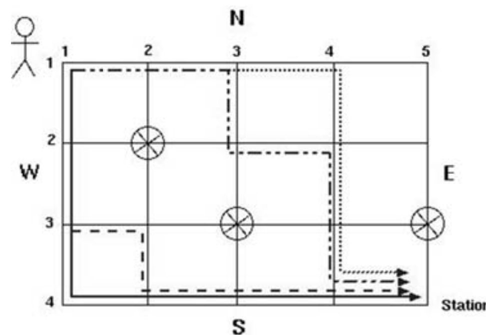
Problem 3 : Walking on the Safe Side (ACM Q825)

Descritption

在正方城這個城市走路是件相當容易的事，因為所有的道路都是像棋盤的線一樣，把城市切割成一塊一塊的正方形。大部分的十字路口都是安全的，行人可以直接通過。然而也有少數的十字路口比較危險，所以建有地下道或天橋供行人通過。

現在有一個人想要從位於城市西北方（也就是左上角）的公園十字路口到位於東南方（也就是右下角）的車站十字路口去。由於他是個懶惰的人，他不想要走比需要多一點點的路，也就是說他一定是往下或往右走，絕對不會往上或往左走。另外，他也不喜歡走天橋或地下道，所以他會避開這些危險的十字路口。你的任務就是幫他算一下從左上角走到右下角有多少種不同的走法。

以下的圖顯示出有 4 條東西向的道路，有 5 條南北向的道路，有 3 個十字路口是危險的。所以從左上角走到右下角要走(N-1)+(W-1) = 3+4 = 7 格的距離，並且總共有 4 種不同的走法。



Input

輸入的第一列有一個正整數，代表以下有多少組測試資料。每組測試資料的第一列有 2 個整數 W,N（均不大於 100），W 代表東西向道路的數目，N 代表南北向道路的數目，編號如上圖所示。接下來的 W 列代表這 W 條東西向道路，每列的第一個數為這是第幾條東西向道路，接下來有 0 個或多個不等的數，代表某些南北向道路與這條東西向道路相交的十字路口是危險的。

輸入的第一列與第一組測試資料之間，以及各組測試資料之間均有一空白列，第一組 Sample Input 表示的路如上圖所示，請參考。

Output

每組測試資料輸出一列，為一個整數。代表這個人從左上角走到右下角有多少種不同的走法。

測試資料間亦請空一列。

Sample Input

```
2

4 5
1
2 2
3 3 5
4

5 5
1
2 1 2 3 4
```

3 1 2 3
4 1 2 3 5
5 1 2 3

Sample Output

4

0

From ACM Q825

解析：

Problem 3-1 : Expressions (ACM Q10157)

Description

令 X 表示正確的括號表示式。X 的內容為僅含 '(' 和 ')' 的字串。定義如下：

- 空字串屬於 X。
- 假如 A 屬於 X ，那麼 (A) 也屬於 X。
- 假如 A 和 B 都屬於 X ，那麼 AB 也屬於 X。

例如：以下的字串都是正確的括號表示式（所以當然屬於 X）

()(())()
(()(()))

而以下的字串不是正確的括號表示式（所以當然不屬於 X）

(()))(()

())(()

令 E 是一個正確的括號表示式（因此 E 屬於 X）。

E 的長度為字串的長度。

而 E 的深度 D(E) 被定義如下：

0 如果 E 是空字串

D(E)= D(A)+1 如果 E = (A), 並且 A 屬於 X

max(D(A),D(B)) 如果 E = AB, 並且 A, B 都屬於 X

例如："()(())()" 的長度是 8，而深度為 2 。

給你 n 和 d ，你的任務是找出長度為 n，且深度為 d 的所有正確的括號表示式共有多少個。

例如： n=6, d=2 共有以下 3 種正確的括號表示式。

(())()
()(())
(()())

Input

每組測試資料一列，含有 2 個整數 n 和 d（2 <= n <= 300，1 <= d <= 150）。

輸入不會超過 20 列，但可能含有空白列，你不用理會這些空白列。

Output

對每一組測試資料輸出一列。長度為 n，且深度為 d 的所有正確的括號表示式共有多少個。

Sample Input

```
6 2
300 150
100 2
100 4
100 19
10 1
10 2
```

```
10 3
10 4
10 5
```

Sample Output

```
3
1
562949953421311
1194307416194742096260l6
104413485759480877889l9740
1
15
18
7
1
```

From ACM Q10157

解析：

Problem 3-1-1 : How Many Trees ? (ACM Q10303)

Description

在資料結構中有一個很有名的主題就是二元樹。給你節點的數目 n，請你求出最多可以形成多少種不同的二元樹。

Input

每筆測試資料一列。每列有 1 個整數 n（1 <= n <= 1000），代表有多少個節點。

Output

對每一列輸入，請輸出可以形成多少種不同的二元樹。

Sample Input

```
1
2
3
9
50
99
```

Sample Output

```
1
2
5
4862
1978261657756160653623774456
22750883079422934966181954039568885395604168260154104734000
```

From ACM Q10303

解析：

Problem 3-1-2 : Count the Trees (ACM Q10007)

Description

有一種社會文明病叫做"變態性強制冥想症"（ACM，Abnormally Compulsive Meditation）。這種心理狀態失序的症狀在程式設計師中有時候會常見到。這種症狀會導致暫時性的失去言語的能力，因為病人的腦中充滿了極度有趣或極富挑戰性的事情。

Juan 是一個傑出的程式設計師。但是最近他的家人非常的擔心他，因為他找到一個很有挑戰性的問題，並且他已經有幾個星期沒有講話了。這個問題是：給你 n 個節點（每個節點有唯一的標籤在上面），請問用這些節點可以建構到多少種不同的二元樹。例如：若給你一個節點 A，那只可以有一種二元樹（A 當作根節點）。若給你節點 A 及 B，那你可以有 4 種不同的二元樹。如下圖所示：

如果你能夠提供一個解題方法，Juan 將可以再度說話，他的家人將永遠的感激你。

Input

每組測試資料一列，含有一個整數 n（1 <= n <= 300）。代表用來建構二元樹的節點數目。n=0 代表輸入結束。

Output

對每一組測試資料，輸出可以建構的二元樹的數目。

Sample Input

```
1
2
10
25
0
```

Sample Output

```
1
4
60949324800
75414671852339208296275849248768000000
```

From ACM Q10007

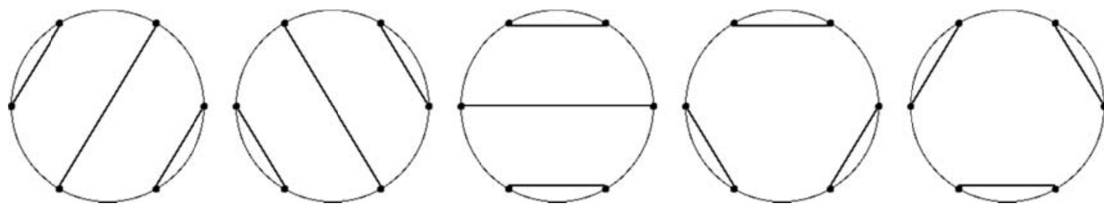
解析：

Problem 3-1-3 : Safe Salutations (ACM 991)

Description

As any minimally superstitious person knows all too well, terrible things will happen when four persons do a crossed handshake.

You, an intrepid computer scientist, are given the task of easing the burden of these people by providing them with the feasible set of handshakes that include everyone in the group while avoiding any such crossings. The following figure illustrates the case for 3 pairs of persons:



Input

The input to this problem contains several datasets separated by a blank line. Each dataset is simply an integer *n*, the number of **pairs** of people in the group, with $1 \leq n \leq 10$.

Output

The output is equally simple. For each dataset, print a single integer indicating the number of possible crossless handshakes that involve everyone in a group with *n* pairs of people. Print a blank line between datasets.

Sample Input

4

Sample Output

14

From ACM Q991

解析：

Problem 4 : Tiling (ACM Q10359)

Description

在 2*n 的矩形區域中，以 2*1 或 2*2 的小磁磚來貼總共有多少種方法？

以下為 2*17 矩形區域一種貼磁磚的方式。

Input

每組測試資料一列，有 1 個整數 n（0 <= n <= 250）。

Output

對每組測試資料請輸出一列，在 2*n 的矩形區域中共有多少種貼磁磚的方法。

Sample Input

```
0
2
8
12
100
200
```

Sample Output

```
1
3
171
2731
845100400152152934331135470251
1071292029505993517027974728227441735014801995855195223534251
```

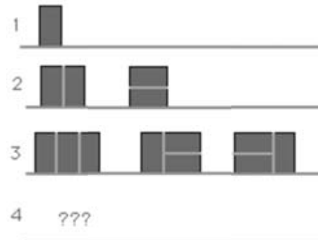
From ACM Q10359

解析：

Problem 4-1 : Brick Wall Patterns (ACM 900 - Brick Wall Pattern)

Description

If we want to build a brick wall out of the usual size of brick which has a length twice as long as its height, and if our wall is to be two units tall, we can make our wall in a number of patterns, depending on how long we want it. From the figure one observe that:



- There is just one wall pattern which is 1 unit wide - made by putting the brick on its end.
- There are 2 patterns for a wall of length 2: two side-ways bricks laid on top of each other and two bricks long-ways up put next to each other.
- There are three patterns for walls of length 3.

How many patterns can you find for a wall of length 4? And, for a wall of length 5?

Problem

Your task is to write a program that given the length of a wall, determines how many patterns there may be for a wall of that length.

Intput

Your program receives a sequence of positive integers, one per line, each representing the length of a wall. The maximum value for the wall is length 50. The input terminates with a 0.

Output

For each wall length given in the input, your program must output the corresponding number of different patterns for such a wall in a separate line.

Sample Intput

```
1
2
3
0
```

Sample Output

```
1
2
3
```

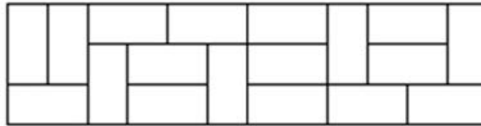
From ACM 900 - Brick Wall Pattern

解析：

Problem 4-2 : Tri Tiling (ACM Q10918: Tri Tiling)

Description

要用大小為 **2*1** 的磁磚貼滿面積 **3*n** 的矩形共有多少種方法？以下是 n=12 的一種貼法。



Input

輸入含有多組測試資料。

每組測試資料一列有一個整數 n（0 <= n <= 30）。

當 n=-1 代表輸入結束。請參考 Sample Input。

Output

對每一組測試資料輸出一列，輸出貼磁磚的方法共有多少種。

Sample Input

```
2
3
8
12
-1
```

Sample Output

```
3
0
153
2131
```

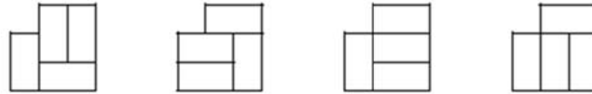
From ACM Q10918: Tri Tiling

解析：

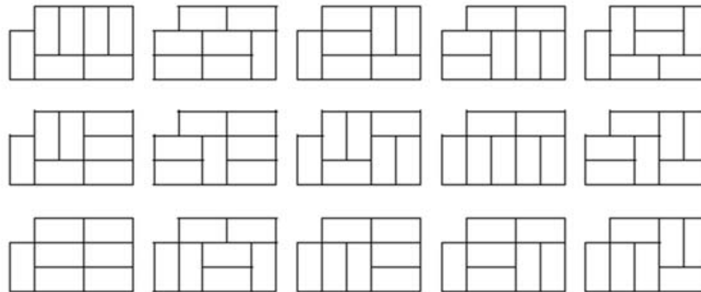
Problem 4-3：鋪磁磚問題(94 北市賽-prob 3)

Description

某學校有一片狹長形狀的畸零地，其寬度、長度分別為 30 公分及 n×10 公分(其中 n 為輸入之值，n 為奇數，n>=3)，但在西北角缺了寬度、長度均為 10 公分的一角。現在我們要使用 (3×n-1)/2 塊磁磚將此片畸零地鋪滿，每塊磁磚的寬度、長度均為 10 公分及 20 公分，我們想知道共有多少種鋪法。請你撰寫一個程式來求出答案。以下圖為例，當 n=3 時，可看出共有 4 種不同的鋪法。



當 n=5 時，由下圖，可看出共有 15 種不同的鋪法。



Constraints

n 為奇數，3<=n<=41。

Input

輸入檔可能包含多筆測試資料。每筆測試資料佔一列，包含一個正整數 n。

Output

以螢幕輸出資料為不同鋪法的次數。注意：輸出之整數值可能多達 12 位數。

Sample Input

```
3
5
```

Sample Output

```
4
15
```

From 94 北市賽(prob 3)

解析：

Unit 21：排列組合

Algorithm Gossip: 排列組合 **http://caterpillar.onlyfun.net/Gossip/AlgorithmGossip/AlgorithmGossip.htm**

說明

將一組數字、字母或符號進行排列，以得到不同的組合順序，例如 1 2 3 這三個數的排列組合有：1 2 3、1 3 2、2 1 3、2 3 1、3 1 2、3 2 1。

解法

可以使用遞迴將問題切割為較小的單元進行排列組合，例如 1 2 3 4 的排列可以分為 1 [2 3 4]、2 [1 3 4]、3 [1 2 4]、4 [1 2 3]進行排列，這邊利用旋轉法，先將旋轉間隔設為 0，將最右邊的數字旋轉至最左邊，並逐步增加旋轉的間隔，例如：

1 2 3 4 -> 旋轉 1 -> 繼續將右邊 2 3 4 進行遞迴處理
2 1 3 4 -> 旋轉 1 2 變為 2 1-> 繼續將右邊 1 3 4 進行遞迴處理
3 1 2 4 -> 旋轉 1 2 3 變為 3 1 2 -> 繼續將右邊 1 2 4 進行遞迴處理
4 1 2 3 -> 旋轉 1 2 3 4 變為 4 1 2 3 -> 繼續將右邊 1 2 3 進行遞迴處理

實作

```
#include <stdio.h>
#include <stdlib.h>
#define N 4

void perm(int*, int);

int main(void) {
    int num[N+1], i;

    for(i = 1; i <= N; i++)
        num[i] = i;

    perm(num, 1);

    return 0;
}

void perm(int* num, int i) {
    int j, k, tmp;

    if(i < N) {
        for(j = i; j <= N; j++) {
            tmp = num[j];
            // 旋轉該區段最右邊數字至最左邊
            for(k = j; k > i; k--)
                num[k] = num[k-1];
            num[i] = tmp;

            perm(num, i+1);

            // 還原
            for(k = i; k < j; k++)
                num[k] = num[k+1];
            num[j] = tmp;
        }
    }
    else {   // 顯示此次排列
        for(j = 1; j <= N; j++)
            printf("%d ", num[j]);
        printf("\n");
    }
}
```

【討論】：上列解法中所產生的排列並不是按字典式的順序，如要將其改成字典順序排列，則須將第 i 項與第 n 項的交換作業換成 *i~i*、*i~i+1*、*i~i+2*、...、*i~n* 逐一向右移動的作業。

實作--排列組合的產生（字典順序

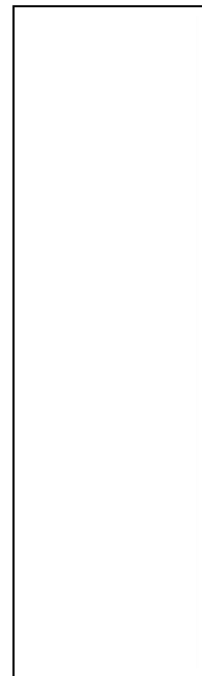
```
#include <stdio.h>

#define N 4
int p[N+1];

void perm(int);

void main(void)
{
      int i;

      for (i=1;i<=N;i++)            /*  初始設定  */
            p[i]=i;
      perm(1);
}
void perm(int i)
{
      int j,k,t;

      if (i<N){
            for (j=i;j<=N;j++){
                  t=p[j];                          /* p[i]～p[j]的向右移動  */
                  for (k=j;k>i;k--)
                        p[k]=p[k-1];
                  p[i]=t;

                  perm(i+1);                       /*  遞迴呼叫  */

                  for (k=i;k<j;k++)             /*  將陣列排列還原成遞迴呼叫前的狀況  */
                        p[k]=p[k+1];
                  p[j]=t;
            }
      }
      else {
            for (j=1;j<=N;j++)              /*  顯示排列組合  */
                  printf("%d ",p[j]);
            printf("\n");
      }
}
```

【參考】：單字的產生

如將上列程式中的資料改成，

p[1]='a'; p[2]='c'; p[3]='h'; p[4]='t';

並以排列的前 3 個單字來顯示，則會產生下列的英文單字。



Unit 22：產生可能的集合

說明

給定一組數字或符號，產生所有可能的集合(包括空集合)，例如給定 1 2 3，則可能的集合為：{}、{1}、{1,2}、{1,2,3}、{1,3}、{2}、{2,3}、{3}。

解法

如果不考慮字典順序，則有個簡單的方法可以產生所有的集合，思考二進位數字加法，並注意 1 出現的位置，如果每個位置都對應一個數字，則由 1 所對應的數字所產生的就是一個集合，例如：

000	{}
001	{3}
010	{2}
011	{2,3}
100	{1}
101	{1,3}
110	{1,2}
111	{1,2,3}

瞭解這個方法之後，剩下的就是如何產生二進位數？有許多方法可以使用，您可以使用 unsigned 型別加上&位元運算來產生，這邊則是使用陣列搜尋，首先陣列內容全為 0，找第一個 1，在還沒找到之前將走訪過的內容變為 0，而第一個找到的 0 則變為 1，如此重複直到所有的陣列元素都變為 1 為止，例如：

000 => 100 => 010 => 110 => 001 => 101 => 011 => 111

如果要產生字典順序，例如若有 4 個元素，則：

{} => {1} => {1,2} => {1,2,3} => {1,2,3,4} =>
{1,2,4} =>
{1,3} => {1,3,4} =>
{1,4} =>
{2} => {2,3} => {2,3,4} =>
{2,4} =>
{3} => {3,4} =>
{4}

簡單的說，如果有 n 個元素要產生可能的集合，當依序產生集合時，如果最後一個元素是 n，而倒數第二個元素是 m 的話，例如：

{a b c d e n}

則下一個集合就是{a b c d e+1}，再依序加入後續的元素。

例如有四個元素，而當產生{1 2 3 4}集合時，則下一個集合就是{1 2 3+1}，也就是{1 2 4}，由於最後一個元素還是 4，所以下一個集合就是{1 2+1}，也就是{1 3}，接下來再加入後續元素 4，也就是{1 3 4}，由於又遇到元素 4，所以下一個集合是{1 3+1}，也就是{1 4}。

實作

- 產生可能的集合--無字典順序

```
#include <stdio.h>
#include <stdlib.h>

#define MAXSIZE 20

int main(void) {
	char digit[MAXSIZE];
	int i, j;
	int n;

	printf("輸入集合個數：");
	scanf("%d", &n);

	for(i = 0; i < n; i++)
		digit[i] = '0';

	printf("\n{}"); //  空集合

	while(1) {
		// 找第一個 0，並將找到前所經過的元素變為 0
		for(i = 0; i < n && digit[i] == '1'; digit[i] = '0', i++);

		if(i == n)   // 找不到 0
			break;
		else              // 將第一個找到的 0 變為 1
			digit[i] = '1';

		// 找第一個 1，並記錄對應位置
		for(i = 0; i < n && digit[i] == '0'; i++);

		printf("\n{%d", i+1);

		for(j = i + 1; j < n; j++)
			if(digit[j] == '1')
				printf(",%d", j + 1);

		printf("}");
	}

	printf("\n");

	return 0;
}
```

• 產生可能的集合--字典順序

```
#include <stdio.h>
#include <stdlib.h>

#define MAXSIZE 20

int main(void) {
    int set[MAXSIZE];
    int i, n, position = 0;

    printf("輸入集合個數：");
    scanf("%d", &n);
    printf("\n{}");
    set[position] = 1;

    while(1) {
        printf("\n{%d", set[0]);   // 印第一個數
        for(i = 1; i <= position; i++)
            printf(",%d", set[i]);
        printf("}");

        if(set[position] < n) {   // 遞增集合個數
            set[position+1] = set[position] + 1;
            position++;
        }
        else if(position != 0) {   // 如果不是第一個位置
            position--;            // 倒退
            set[position]++;   // 下一個集合尾數
        }
        else   // 已倒退至第一個位置
            break;
    }

    printf("\n");

    return 0;
}
```

Unit 23：m 元素集合的 n 個元素子集

Algorithm Gossip: m 元素集合的 n 個元素子集

說明

假設有個集合擁有 m 個元素，任意的從集合中取出 n 個元素，則這 n 個元素所形成的可能子集有那些？

解法

假設有 5 個元素的集點，取出 3 個元素的可能子集如下：
{1 2 3}、{1 2 4 }、{1 2 5}、{1 3 4}、{1 3 5}、{1 4 5}、{2 3 4}、{2 3 5}、{2 4 5}、{3 4 5}

這些子集已經使用字典順序排列，如此才可以觀察出一些規則：

1. 如果最右一個元素小於 m，則如同碼錶一樣的不斷加 1
2. 如果右邊一位已至最大值，則加 1 的位置往左移
3. 每次加 1 的位置往左移後，必須重新調整右邊的元素為遞減順序

所以關鍵點就在於哪一個位置必須進行加 1 的動作，到底是最右一個位置要加 1？還是其它的位置？

在實際撰寫程式時，可以使用一個變數 positon 來記錄加 1 的位置，position 的初值設定為 n-1，因為我們要使用陣列，而最右邊的索引值為最大的 n-1，在 position 位置的值若小於 m 就不斷加 1，如果大於 m 了，position 就減 1，也就是往左移一個位置；由於位置左移後，右邊的元素會經過調整，所以我們必須檢查最右邊的元素是否小於 m，如果是，則 position 調整回 n-1，如果不是，則 positon 維持不變。

實作

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 20

int main(void) {
    int set[MAX];
    int m, n, position;
    int i;

    printf("輸入集合個數 m：");
    scanf("%d", &m);
    printf("輸入取出元素 n：");
    scanf("%d", &n);

    for(i = 0; i < n; i++)
        set[i] = i + 1;

    // 顯示第一個集合
    for(i = 0; i < n; i++)
        printf("%d ", set[i]);
    putchar('\n');

    position = n - 1;
```

```
    while(1) {
        if(set[n-1] == m)
            position--;
        else
            position = n - 1;

        set[position]++;

        // 調整右邊元素
        for(i = position + 1; i < n; i++)
            set[i] = set[i-1] + 1;

        for(i = 0; i < n; i++)
            printf("%d ", set[i]);
        putchar('\n');

        if(set[0] >= m - n + 1)
            break;
    }

    return 0;
}
```

Unit 24：數字拆解

Algorithm Gossip: 數字拆解

說明

3 = 2+1 = 1+1+1 所以 3 有三種拆法
4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1 共五種
5 = 4 + 1 = 3 + 2 = 3 + 1 + 1 = 2 + 2 + 1 = 2 + 1 + 1 + 1 = 1 + 1 +1 +1 +1

共七種
依此類推，請問一個指定數字 NUM 的拆解方法個數有多少個？

解法

我們以上例中最後一個數字 5 的拆解為例，假設 f(n)為數字 n 的可拆解方式個數，而 f(x, y)為使用 y 以下的數字來拆解 x 的方法個數，則觀察：

5 = 4 + 1 = 3 + 2 = 3 + 1 + 1 = 2 + 2 + 1 = 2 + 1 + 1 + 1 = 1 + 1 +1 +1 +1

使用函式來表示的話：

f(5) = f(4, 1) + f(3,2) + f(2,3) + f(1,4) + f(0,5)

其中 f(1, 4) = f(1, 3) + f(1, 2) + f(1, 1)，但是使用大於 1 的數字來拆解 1 沒有意義，所以 f(1, 4) = f(1, 1)，而同樣的，f(0, 5)會等於 f(0, 0)，所以：

f(5) = f(4, 1) + f(3,2) + f(2,3) + f(1,1) + f(0,0)

依照以上的說明，使用動態程式規畫（Dynamic programming）來進行求解，其中 f(4,1)其實就是 f(5-1, min(5-1,1))，f(x, y)就等於 f(n-y, min(n-x, y))，其中 n 為要拆解的數字，而 min()表示取兩者中較小的數。

使用一個二維陣列表格 table[x][y]來表示 f(x, y)，剛開始時，將每列的索引 0 與索引 1 元素值設定為 1，因為任何數以 0 以下的數拆解必只有 1 種，而任何數以 1 以下的數拆解也必只有 1 種：

```
for(i = 0; i < NUM +1; i++){
   table[i][0] = 1;  // 任何數以 0 以下的數拆解必只有 1 種
   table[i][1] = 1;  // 任何數以 1 以下的數拆解必只有 1 種
}
```

接下來就開始一個一個進行拆解了，如果數字為 NUM，則我們的陣列維度大小必須為 NUM x (NUM/2+1)，以數字 10 為例，其維度為 10 x 6 我們的表格將會如下所示：

1 1 0 0 0 0
1 1 0 0 0 0
1 1 2 0 0 0
1 1 2 3 0 0
1 1 3 4 5 0
1 1 3 5 6 7
1 1 4 7 9 0
1 1 4 8 0 0
1 1 5 0 0 0
1 1 0 0 0 0

實作

```
#include <stdio.h>
#include <stdlib.h>
#define NUM 10      //  要拆解的數字
#define DEBUG 0

int main(void) {
    int table[NUM][NUM/2+1] = {0}; // 動態規畫表格
    int count = 0;
    int result = 0;
    int i, j, k;

    printf("數字拆解\n");
    printf("3 = 2+1 = 1+1+1  所以 3 有三種拆法\n");
    printf("4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1");
    printf("共五種\n");
    printf("5 = 4 + 1 = 3 + 2 = 3 + 1 + 1");
    printf(" = 2 + 2 + 1 = 2 + 1 + 1 + 1 = 1 + 1 +1 +1 +1");
    printf("共七種\n");
    printf("依此類推，求 %d 有幾種拆法？", NUM);

    // 初始化
    for(i = 0; i < NUM; i++){
        table[i][0] = 1;   // 任何數以 0 以下的數拆解必只有 1 種
        table[i][1] = 1;   // 任何數以 1 以下的數拆解必只有 1 種
    }

    // 動態規劃
    for(i = 2; i <= NUM; i++){
       for(j = 2; j <= i; j++){
            if(i + j > NUM) // 大於 NUM
                continue;

            count = 0;
            for(k = 1 ; k <= j; k++){
                count += table[i-k][(i-k >= k) ? k : i-k];
            }
            table[i][j] = count;
        }
    }

    // 計算並顯示結果
    for(k = 1 ; k <= NUM; k++)
        result += table[NUM-k][(NUM-k >= k) ? k : NUM-k];
    printf("\n\nresult: %d\n", result);

    if(DEBUG) {
        printf("\n 除錯資訊\n");
        for(i = 0; i < NUM; i++) {
            for(j = 0; j < NUM/2+1; j++)
                printf("%2d", table[i][j]);
            printf("\n");
        }
    }

    return 0;
}
```

Unit 25：選擇、插入、氣泡排序

Algorithm Gossip: 選擇、插入、氣泡排序

說明

選擇排序（Selection sort）、插入排序（Insertion sort）與氣泡排序（Bubble sort）這三個排序方式是初學排序所必須知道的三個基本排序方式，它們由於速度不快而不實用（平均與最快的時間複雜度都是 $O(n^2)$），然而它們排序的方式確是值得觀察與探討的。

解法

(一) 選擇排序

將要排序的對象分作兩部份，一個是已排序的，一個是未排序的，從後端未排序部份選擇一個最小值，並放入前端已排序部份的最後一個，例如：

排序前：70 80 31 37 10 1 48 60 33 80

1. [1] 80 31 37 10 70 48 60 33 80 選出最小值 1
2. [1 10] 31 37 80 70 48 60 33 80 選出最小值 10
3. [1 10 31] 37 80 70 48 60 33 80 選出最小值 31
4. [1 10 31 33] 80 70 48 60 37 80 ......
5. [1 10 31 33 37] 70 48 60 80 80 ......
6. [1 10 31 33 37 48] 70 60 80 80 ......
7. [1 10 31 33 37 48 60] 70 80 80 ......
8. [1 10 31 33 37 48 60 70] 80 80 ......
9. [1 10 31 33 37 48 60 70 80] 80 ......

(二) 插入排序

像是玩樸克一樣，我們將牌分作兩堆，每次從後面一堆的牌抽出最前端的牌，然後插入前面一堆牌的適當位置，例如：

排序前：92 77 67 8 6 84 55 85 43 67

1. [77 92] 67 8 6 84 55 85 43 67 將 77 插入 92 前
2. [67 77 92] 8 6 84 55 85 43 67 將 67 插入 77 前
3. [8 67 77 92] 6 84 55 85 43 67 將 8 插入 67 前
4. [6 8 67 77 92] 84 55 85 43 67 將 6 插入 8 前
5. [6 8 67 77 84 92] 55 85 43 67 將 84 插入 92 前
6. [6 8 55 67 77 84 92] 85 43 67 將 55 插入 67 前
7. [6 8 55 67 77 84 85 92] 43 67 ......
8. [6 8 43 55 67 77 84 85 92] 67 ......
9. [6 8 43 55 67 67 77 84 85 92] ......

(三) 氣泡排序法

顧名思義，就是排序時，最大的元素會如同氣泡一樣移至右端，其利用比較相鄰元素的方法，將大的元素交換至右端，所以大的元素會不斷的往右移動，直到適當的位置為止。

基本的氣泡排序法可以利用旗標的方式稍微減少一些比較的時間，當尋訪完陣列後都沒有發生任何的交換動作，表示排序已經完成，而無需再進行之後的迴圈比較與交換動作，例如：
排序前：95 27 90 49 80 58 6 9 18 50

1. 27 90 49 80 58 6 9 18 50 [95] 95 浮出
2. 27 49 80 58 6 9 18 50 [90 95] 90 浮出
3. 27 49 58 6 9 18 50 [80 90 95] 80 浮出
4. 27 49 6 9 18 50 [58 80 90 95] ......
5. 27 6 9 18 49 [50 58 80 90 95] ......
6. 6 9 18 27 [49 50 58 80 90 95] ......
7. 6 9 18 [27 49 50 58 80 90 95] 由於接下來不會再發生交換動作，排序提早結束

在上面的例子當中，還加入了一個觀念，就是當進行至 i 與 i+1 時沒有交換的動作，表示接下來的 i+2 至 n 已經排序完畢，這也增進了氣泡排序的效率。

實作

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void selsort(int[]);   // 選擇排序
void insort(int[]);    // 插入排序
void bubsort(int[]);   // 氣泡排序

int main(void) {
    int number[MAX] = {0};
    int i;

    srand(time(NULL));

    printf("排序前：");
    for(i = 0; i < MAX; i++) {
        number[i] = rand() % 100;
        printf("%d ", number[i]);
    }

    printf("\n 請選擇排序方式：\n");
    printf("(1)選擇排序\n(2)插入排序\n(3)氣泡排序\n:");
    scanf("%d", &i);

    switch(i) {
        case 1:
            selsort(number); break;
        case 2:
            insort(number); break;
        case 3:
            bubsort(number); break;
```

```
            default:
                printf("選項錯誤(1..3)\n");
        }

        return 0;
}
```

```
void selsort(int number[]) {
        int i, j, k, m;

        for(i = 0; i < MAX-1; i++) {
                m = i;
                for(j = i+1; j < MAX; j++)
                        if(number[j] < number[m])
                                m = j;

                if( i != m)
                        SWAP(number[i], number[m])

                printf("第 %d 次排序：", i+1);
                for(k = 0; k < MAX; k++)
                        printf("%d ", number[k]);
                printf("\n");
        }
 }
```

```
void insort(int number[]) {
        int i, j, k, tmp;

        for(j = 1; j < MAX; j++) {
                tmp = number[j];
                i = j - 1;
                while(tmp < number[i]) {
                        number[i+1] = number[i];
                        i--;
                        if(i == -1)
                                break;
                }
                number[i+1] = tmp;

                printf("第 %d 次排序：", j);
                for(k = 0; k < MAX; k++)
                        printf("%d ", number[k]);
                printf("\n");
        }
}
```

```
void bubsort(int number[]) {
    int i, j, k, flag = 1;

    for(i = 0; i < MAX-1 && flag == 1; i++) {
        flag = 0;
        for(j = 0; j < MAX-i-1; j++) {
            if(number[j+1] < number[j]) {
                SWAP(number[j+1], number[j]);
                flag = 1;
            }
        }

        printf("第 %d 次排序：", i+1);
        for(k = 0; k < MAX; k++)
            printf("%d ", number[k]);
        printf("\n");
    }
}
```

Unit 26：Shell 排序法 - 改良的插入排序

Algorithm Gossip: Shell 排序法 - 改良的插入排序

說明

插入排序法由未排序的後半部前端取出一個值，插入已排序前半部的適當位置，概念簡單但速度不快。

排序要加快的基本原則之一，是讓後一次的排序進行時，儘量利用前一次排序後的結果，以加快排序的速度，Shell 排序法即是基於此一概念來改良插入排序法。

解法

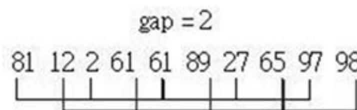
Shell 排序法最初是 D.L Shell 於 1959 所提出，假設要排序的元素有 n 個，則每次進行插入排序時並不是所有的元素同時進行時，而是取一段間隔。

Shell 首先將間隔設定為 n/2，然後跳躍進行插入排序，再來將間隔 n/4，跳躍進行排序動作，再來間隔設定為 n/8、n/16，直到間隔為 1 之後的最後一次排序終止，由於上一次的排序動作都會將固定間隔內的元素排序好，所以當間隔越來越小時，某些元素位於正確位置的機率越高，因此最後幾次的排序動作將可以大幅減低。

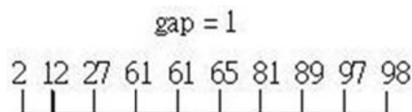
舉個例子來說，假設有一未排序的數字如右：89 12 65 97 61 81 27 2 61 98

數字的總數共有 10 個，所以第一次我們將間隔設定為 10 / 2 = 5，此時我們對間隔為 5 的數字進行排序，如下所示：

畫線連結的部份表示 要一起進行排序的部份，再來將間隔設定為 5 / 2 的商，也就是 2，則第二次的插入排序對象如下所示：



再來間隔設定為 2 / 2 = 1，此時就是單純的插入排序了，由於大部份的元素都已大致排序過了，所以最後一次的插入排序幾乎沒作什麼排序動作了：



將間隔設定為 n / 2 是 D.L Shell 最初所提出，在教科書中使用這個間隔比較好說明，然而 Shell 排序法的關鍵在於間隔的選定，例如 Sedgewick 證明選用以下的間隔可以加快 Shell 排序法的速度：

$$4*(2^j)^2 + 3*(2^j) + 1$$

$$j = \log_2 \left[\frac{-3 + \sqrt{16*n - 7}}{8}\right]$$

其中 4*(2j)2 + 3*(2j) + 1 不可超過元素總數 n 值，使用上式找出 j 後代入 4*(2j)2 + 3*(2j) + 1 求得第一個間隔，然後將 2j 除以 2 代入求得第二個間隔，再來依此類推。

後來還有人證明有其它的間隔選定法可以將 Shell 排序法的速度再加快；另外 Shell 排序法的概念也可以用來改良氣泡排序法。

實作

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void shellsort(int[]);

int main(void) {
	int number[MAX] = {0};
	int i;

	srand(time(NULL));

	printf("排序前：");
	for(i = 0; i < MAX; i++) {
		number[i] = rand() % 100;
		printf("%d ", number[i]);
	}

	shellsort(number);

	return 0;
}

void shellsort(int number[]) {
	int i, j, k, gap, t;

	gap = MAX / 2;

	while(gap > 0) {
		for(k = 0; k < gap; k++) {
			for(i = k+gap; i < MAX; i+=gap) {
				for(j = i - gap; j >= k; j-=gap) {
					if(number[j] > number[j+gap]) {
						SWAP(number[j], number[j+gap]);
					}
					else
						break;
				}
			}
		}

		printf("\ngap = %d：", gap);
		for(i = 0; i < MAX; i++)
			printf("%d ", number[i]);
		printf("\n");

		gap /= 2;
	}
}
```

Unit 27：Heap 排序法 - 改良的選擇排序

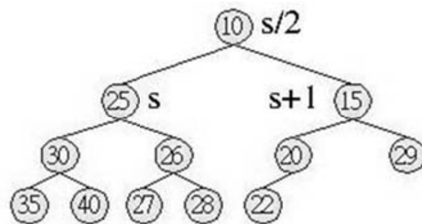
Algorithm Gossip: Heap 排序法 - 改良的選擇排序

說明

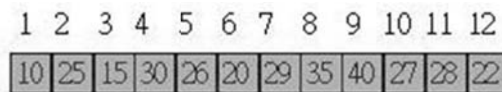
選擇排序法的概念簡單，每次從未排序部份選一最小值，插入已排序部份的後端，其時間主要花費於在整個未排序部份尋找最小值，如果能讓搜尋最小值的方式加快，選擇排序法的速率也就可以加快，Heap 排序法讓搜尋的路徑由樹根至最後一個樹葉，而不是整個未排序部份，因而稱之為改良的選擇排序法。

解法

Heap 排序法使用 Heap Tree（堆積樹），樹是一種資料結構，而堆積樹是一個二元樹，也就是每一個父節點最多只有兩個子節點（關於樹的詳細定義還請見資料結構書籍），堆積樹的父節點若小於子節點，則稱之為最小堆積（Min Heap），父節點若大於子節點，則稱之為最大堆積（Max Heap），而同一層的子節點則無需理會其大小關係，例如下面就是一個堆積樹：

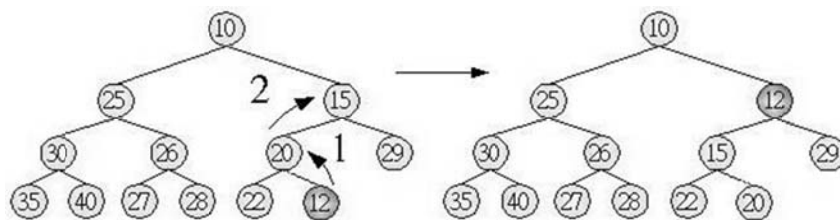


可以使用一維陣列來儲存堆積樹的所有元素與其順序，為了計算方便，使用的起始索引是 1 而不是 0，索引 1 是樹根位置，如果左子節點儲存在陣列中的索引為 s，則其父節點的索引為 s/2，而右子節點為 s+1，就如上圖所示，將上圖的堆積樹轉換為一維陣列之後如下所示：



1	2	3	4	5	6	7	8	9	10	11	12
10	25	15	30	26	20	29	35	40	27	28	22

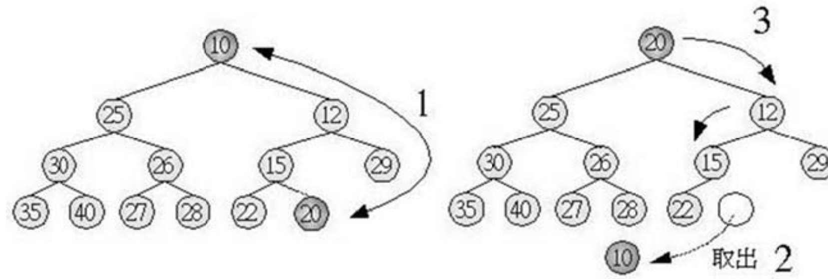
首先必須知道如何建立堆積樹，加至堆積樹的元素會先放置在最後一個樹葉節點位置，然後檢查父節點是否小於子節點（最小堆積），將小的元素不斷與父節點交換，直到滿足堆積樹的條件為止，例如在上圖的堆積加入一個元素 12，則堆積樹的調整方式如下所示：



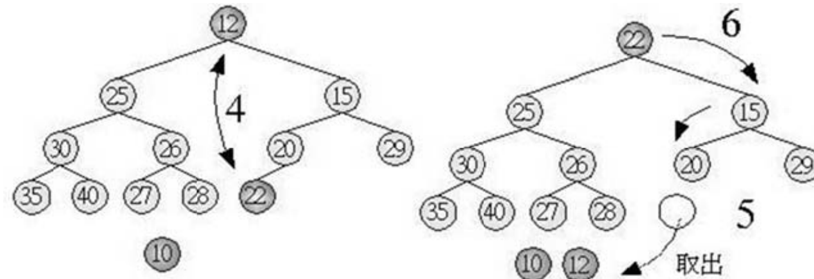
建立好堆積樹之後，樹根一定是所有元素的最小值，您的目的就是：

1. 將最小值取出
2. 然後調整樹為堆積樹

不斷重複以上的步驟，就可以達到排序的效果，最小值的取出方式是將樹根與最後一個樹葉節點交換，然後切下樹葉節點，重新調整樹為堆積樹，如下所示：



調整完畢後，樹根節點又是最小值了，於是我們可以重覆這個步驟，再取出最小值，並調整樹為堆積樹，如下所示：



如此重覆步驟之後，由於使用一維陣列來儲存堆積樹，每一次將樹葉與樹根交換的動作就是將最小值放至後端的陣列，所以最後陣列就是變為已排序的狀態。

其實堆積在調整的過程中，就是一個選擇的行為，每次將最小值選至樹根，而選擇的路徑並不是所有的元素，而是由樹根至樹葉的路徑，因而可以加快選擇的過程，所以 Heap 排序法才會被稱之為改良的選擇排序法。

實作

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void createheap(int[]);
void heapsort(int[]);

int main(void) {
    int number[MAX+1] = {-1};
    int i, num;

    srand(time(NULL));

    printf("排序前：");
    for(i = 1; i <= MAX; i++) {
        number[i] = rand() % 100;
        printf("%d ", number[i]);
    }

    printf("\n 建立堆積樹：");
    createheap(number);
    for(i = 1; i <= MAX; i++)
        printf("%d ", number[i]);
    printf("\n");

    heapsort(number);
```

```
	printf("\n");

	return 0;
}

void createheap(int number[]) {
	int i, s, p;
	int heap[MAX+1] = {-1};

	for(i = 1; i <= MAX; i++) {
		heap[i] = number[i];
		s = i;
		p = i / 2;
		while(s >= 2 && heap[p] > heap[s]) {
			SWAP(heap[p], heap[s]);
			s = p;
			p = s / 2;
		}
	}

	for(i = 1; i <= MAX; i++)
		number[i] = heap[i];

}

void heapsort(int number[]) {
	int i, m, p, s;

	m = MAX;
	while(m > 1) {
		SWAP(number[1], number[m]);
		m--;

		p = 1;
		s = 2 * p;

		while(s <= m) {
			if(s < m && number[s+1] < number[s])
				s++;
			if(number[p] <= number[s])
				break;
			SWAP(number[p], number[s]);
			p = s;
			s = 2 * p;
		}

		printf("\n 排序中：");
		for(i = MAX; i > 0; i--)
			printf("%d ", number[i]);
	}
}
```

Unit 28：快速排序法

Algorithm Gossip: 快速排序法

說明

快速排序法（quick sort）是目前所公認最快的排序方法之一（視解題的對象而定），雖然快速排序法在最差狀況下可以達 O(n^2)，但是在多數的情況下，快速排序法的效率表現是相當不錯的。

快速排序法的基本精神是在數列中找出適當的軸心，然後將數列一分為二，分別對左邊與右邊數列進行排序，而影響快速排序法效率的正是軸心的選擇。

這邊所介紹的第一個快速排序法版本，是在多數的教科書上所提及的版本，因為它最容易理解，也最符合軸心分割與左右進行排序的概念，適合對初學者進行講解。

解法

這邊所介紹的快速演算如下：

1. 將最左邊的數設定為軸，並記錄其值為 s

迴圈處理：

1. 令索引 i 從數列左方往右方找，直到找到大於 s 的數
2. 令索引 j 從數列右方往左方找，直到找到小於 s 的數
3. 如果 i >= j，則離開迴圈
4. 如果 i < j，則交換索引 i 與 j 兩處的值
5. 將左側的軸與 j 進行交換
6. 對軸左邊進行遞迴
7. 對軸右邊進行遞迴

透過以下演算法，則軸左邊的值都會小於 s，軸右邊的值都會大於 s，如此再對軸左右兩邊進行遞迴，就可以對完成排序的目的，例如下面的實例，*表示要交換的數，[]表示軸：

- [41] 24 76* 11 45 64 21 69 19 36*
- [41] 24 36 11 45* 64 21 69 19* 76
- [41] 24 36 11 19 64* 21* 69 45 76
- [41] 24 36 11 19 21 64 69 45 76
- 21 24 36 11 19 [41] 64 69 45 76

在上面的例子中，41 左邊的值都比它小，而右邊的值都比它大，如此左右再進行遞迴至排序完成。

實作

- C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void quicksort(int[], int, int);

int main(void) {
    int number[MAX] = {0};
    int i, num;

    srand(time(NULL));

    printf("排序前：");
    for(i = 0; i < MAX; i++) {
```

```
        number[i] = rand() % 100;
        printf("%d ", number[i]);
    }

    quicksort(number, 0, MAX-1);

    printf("\n 排序後：");
    for(i = 0; i < MAX; i++)
        printf("%d ", number[i]);

    printf("\n");

    return 0;
}

void quicksort(int number[], int left, int right) {
    int i, j, s;

    if(left < right) {
        s = number[left];
        i = left;
        j = right + 1;

        while(1) {
            // 向右找
            while(i + 1 < MAX && number[++i] < s) ;
            // 向左找
            while(j -1 > -1 && number[--j] > s) ;
            if(i >= j)
                break;
            SWAP(number[i], number[j]);
        }

        number[left] = number[j];
        number[j] = s;

        quicksort(number, left, j-1);    // 對左邊進行遞迴
        quicksort(number, j+1, right);   // 對右邊進行遞迴
    }
}
```

Unit 29：快速排序法（二）

Algorithm Gossip: 快速排序法（二）

說明

在 快速排序法（一） 中，每次將最左邊的元素設為軸，而之前曾經說過，快速排序法的加速在於軸的選擇，在這個例子中，只將軸設定為中間的元素，依這個元素作基準進行比較，這可以增加快速排序法的效率。

解法

在這個例子中，取中間的元素 s 作比較，同樣的先得右找比 s 大的索引 i，然後找比 s 小的索引 j，只要兩邊的索引還沒有交會，就交換 i 與 j 的元素值，這次不用再進行軸的交換了，因為在尋找交換的過程中，軸位置的元素也會參與交換的動作，例如：

41 24 76 11 45 64 21 69 19 36

首先 left 為 0，right 為 9，(left+right)/2 = 4（取整數的商），所以軸為索引 4 的位置，比較的元素是 45，您往右找比 45 大的，往左找比 45 小的進行交換：

- 41 24 76* 11 [45] 64 21 69 19 *36
- 41 24 36 11 45* 64 21 69 19* 76
- 41 24 36 11 19 64* 21* 69 45 76
- [41 24 36 11 19 21] [64 69 45 76]

完成以上之後，再初別對左邊括號與右邊括號的部份進行遞迴，如此就可以完成排序的目的。

實作

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void quicksort(int[], int, int);

int main(void) {
    int number[MAX] = {0};
    int i, num;

    srand(time(NULL));

    printf("排序前：");
    for(i = 0; i < MAX; i++) {
        number[i] = rand() % 100;
        printf("%d ", number[i]);
    }

    quicksort(number, 0, MAX-1);

    printf("\n 排序後：");
    for(i = 0; i < MAX; i++)
        printf("%d ", number[i]);

    printf("\n");

    return 0;
}
```

```
void quicksort(int number[], int left, int right) {
    int i, j, s;

    if(left < right) {
        s = number[(left+right)/2];
        i = left - 1;
        j = right + 1;

        while(1) {
            while(number[++i] < s) ;   // 向右找
            while(number[--j] > s) ;   // 向左找
            if(i >= j)
                break;
            SWAP(number[i], number[j]);
        }

        quicksort(number, left, i-1);     // 對左邊進行遞迴
        quicksort(number, j+1, right);    // 對右邊進行遞迴
    }
}
```

Unit 30：快速排序法（三）

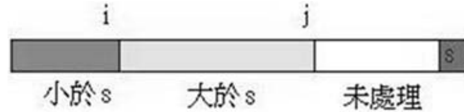
Algorithm Gossip: 快速排序法（三）

說明

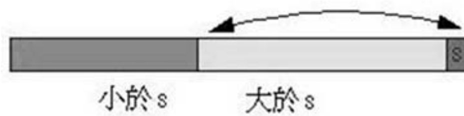
之前說過軸的選擇是快速排序法的效率關鍵之一，在這邊的快速排序法的軸選擇方式更加快了快速排序法的效率，它是來自演算法名書 Introduction to Algorithms 之中。

解法

先說明這個快速排序法的概念，它以最右邊的值 s 作比較的標準，將整個數列分為三個部份，一個是小於 s 的部份，一個是大於 s 的部份，一個是未處理的部份，如下所示 ：



在排序的過程中，i 與 j 都會不斷的往右進行比較與交換，最後數列會變為以下的狀態：

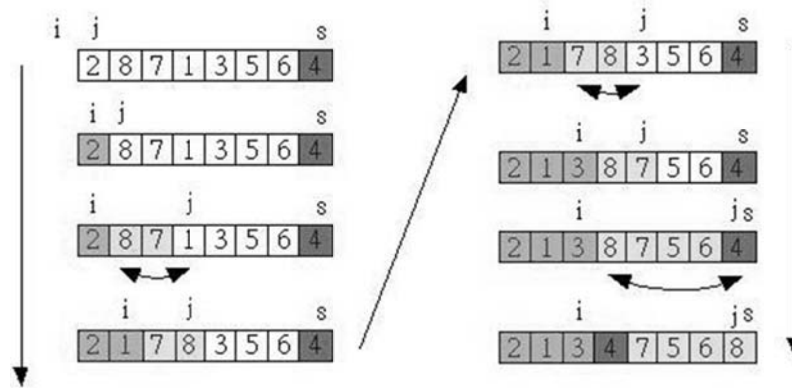


然後將 s 的值置於中間，接下來就以相同的步驟會左右兩邊的數列進行排序的動作，如下所示：

整個演算的過程，直接摘錄書中的虛擬碼來作說明：

```
QUICKSORT(A, p, r)
    if p < r
        then q <- PARTITION(A, p, r)
                 QUICKSORT(A, p, q-1)
                 QUICKSORT(A, q+1, r)
end QUICKSORT

PARTITION(A, p, r)
    x <- A[r]
    i <- p-1
    for j <- p to r-1
        do if A[j] <= x
                 then    i <- i+1
                         exchange A[i]<->A[j]
    exchange A[i+1]<->A[r]
    return i+1
end PARTITION
```

一個實際例子的演算如下所示：



實作

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

int partition(int[], int, int);
void quicksort(int[], int, int);

int main(void) {
    int number[MAX] = {0};
    int i, num;

    srand(time(NULL));

    printf("排序前：");
    for(i = 0; i < MAX; i++) {
        number[i] = rand() % 100;
        printf("%d ", number[i]);
    }

    quicksort(number, 0, MAX-1);

    printf("\n 排序後：");
    for(i = 0; i < MAX; i++)
        printf("%d ", number[i]);

    printf("\n");

    return 0;
}

int partition(int number[], int left, int right) {
    int i, j, s;

    s = number[right];
    i = left - 1;

    for(j = left; j < right; j++) {
        if(number[j] <= s) {
            i++;
            SWAP(number[i], number[j]);
        }
```

```
    }

    SWAP(number[i+1], number[right]);
    return i+1;
}

void quicksort(int number[], int left, int right) {
    int q;

    if(left < right) {
        q = partition(number, left, right);
        quicksort(number, left, q-1);
        quicksort(number, q+1, right);
    }
}
```

Unit 31：合併排序法(Merge Sort)

Algorithm Gossip: 合併排序法

說明

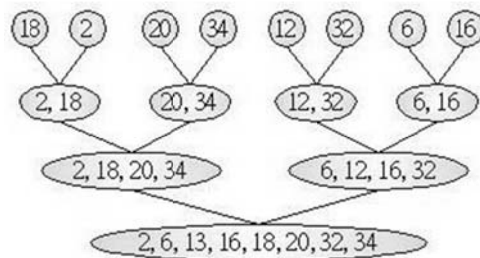
之前所介紹的排序法都是在同一個陣列中的排序，考慮今日有兩筆或兩筆以上的資料，它可能是不同陣列中的資料，或是不同檔案中的資料，如何為它們進行排序？

解法

可以使用合併排序法，合併排序法基本是將兩筆已排序的資料合併並進行排序，如果所讀入的資料尚未排序，可以先利用其它的排序方式來處理這兩筆資料，然後再將排序好的這兩筆資料合併。

有人問道，如果兩筆資料本身就無排序順序，何不將所有的資料讀入，再一次進行排序？排序的精神是儘量利用資料已排序的部份，來加快排序的效率，小筆資料的排序較為快速，如果小筆資料排序完成之後，再合併處理時，因為兩筆資料都有排序了，所有在合併排序時會比單純讀入所有的資料再一次排序來的有效率。

那麼可不可以直接使用合併排序法本身來處理整個排序的動作？而不動用到其它的排序方式？答案是肯定的，只要將所有的數字不斷的分為兩個等分，直到最後剩一個數字為止，然後再反過來不斷的合併，就如下圖所示：



不過基本上分割又會花去額外的時間，不如使用其它較好的排序法來排序小筆資料，再使用合併排序來的有效率。

下面這個程式範例，我們使用快速排序法來處理小筆資料排序，然後再使用合併排序法處理合併的動作。

實作

- C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX1 10
#define MAX2 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

int partition(int[], int, int);
void quicksort(int[], int, int);
void mergesort(int[], int, int[], int, int[]);

int main(void) {
    int number1[MAX1] = {0};
    int number2[MAX1] = {0};
    int number3[MAX1+MAX2] = {0};
    int i, num;

    srand(time(NULL));

    printf("排序前：");
    printf("\nnumber1[]：");
    for(i = 0; i < MAX1; i++) {
        number1[i] = rand() % 100;
```

```
        printf("%d ", number1[i]);
    }

    printf("\nnumber2[]：");
    for(i = 0; i < MAX2; i++) {
        number2[i] = rand() % 100;
        printf("%d ", number2[i]);
    }

    // 先排序兩筆資料
    quicksort(number1, 0, MAX1-1);
    quicksort(number2, 0, MAX2-1);

    printf("\n 排序後：");
    printf("\nnumber1[]：");
    for(i = 0; i < MAX1; i++)
        printf("%d ", number1[i]);
    printf("\nnumber2[]：");
    for(i = 0; i < MAX2; i++)
        printf("%d ", number2[i]);

    // 合併排序
    mergesort(number1, MAX1, number2, MAX2, number3);

    printf("\n 合併後：");
    for(i = 0; i < MAX1+MAX2; i++)
        printf("%d ", number3[i]);

    printf("\n");

    return 0;
}

int partition(int number[], int left, int right) {
    int i, j, s;

    s = number[right];
    i = left - 1;

    for(j = left; j < right; j++) {
        if(number[j] <= s) {
            i++;
            SWAP(number[i], number[j]);
        }
    }

    SWAP(number[i+1], number[right]);
    return i+1;
}

void quicksort(int number[], int left, int right) {
    int q;

    if(left < right) {
        q = partition(number, left, right);
        quicksort(number, left, q-1);
        quicksort(number, q+1, right);
    }
}
```

```
void mergesort(int number1[], int M, int number2[],
                        int N, int number3[]) {
    int i = 0, j = 0, k = 0;

    while(i < M && j < N) {
        if(number1[i] <= number2[j])
            number3[k++] = number1[i++];
        else
            number3[k++] = number2[j++];
    }

    while(i < M)
        number3[k++] = number1[i++];
    while(j < N)
        number3[k++] = number2[j++];
}
```

Unit 32：循序搜尋法（使用衛兵）

Algorithm Gossip: 循序搜尋法（使用衛兵）

說明

搜尋的目的，是在「**已排序的資料**」中尋找指定的資料，而當中循序搜尋是最基本的搜尋法，只要從資料開頭尋找到最後，看看是否找到資料即可。

解法

初學者看到循序搜尋，多數都會使用以下的方式來進行搜尋：

```
while(i < MAX) {
    if(number[i] == k) {
        printf("找到指定值");
        break;
    }
    i++;
}
```

這個方法基本上沒有錯，但是可以加以改善，可以利用設定衛兵的方式，省去 if 判斷式，衛兵通常設定在數列最後或是最前方，假設設定在列前方好了（索引 0 的位置），我們從數列後方向前找，如果找到指定的資料時，其索引值不是 0，表示在數列走訪完之前就找到了，在程式的撰寫上，只要使用一個 while 迴圈就可以了。

下面的程式為了配合衛兵的設置，自行使用快速排序法先將產生的數列排序，然後才進行搜尋，若只是數字的話，通常您可以使用程式語言函式庫所提供的搜尋函式。

實作

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

int search(int[]);
int partition(int[], int, int);
void quicksort(int[], int, int);

int main(void) {
    int number[MAX+1] = {0};
    int i, find;

    srand(time(NULL));

    for(i = 1; i <= MAX; i++)
        number[i] = rand() % 100;

    quicksort(number, 1, MAX);

    printf("數列：");
    for(i = 1; i <= MAX; i++)
        printf("%d ", number[i]);

    printf("\n 輸入搜尋值：");
    scanf("%d", &number[0]);

    if(find = search(number))
        printf("\n 找到數值於索引 %d ", find);
```

```
    else
        printf("\n 找不到數值");

    printf("\n");

    return 0;
}

int search(int number[]) {
    int i, k;

    k = number[0];
    i = MAX;

    while(number[i] != k)
        i--;

    return i;
}

int partition(int number[], int left, int right) {
    int i, j, s;

    s = number[right];
    i = left - 1;

    for(j = left; j < right; j++) {
        if(number[j] <= s) {
            i++;
            SWAP(number[i], number[j]);
        }
    }

    SWAP(number[i+1], number[right]);
    return i+1;
}

void quicksort(int number[], int left, int right) {
    int q;

    if(left < right) {
        q = partition(number, left, right);
        quicksort(number, left, q-1);
        quicksort(number, q+1, right);
    }
}
```

Unit 33：二分搜尋法 Binary Search

Algorithm Gossip: 二分搜尋法（搜尋原則的代表）

說明

如果搜尋的數列已經有排序，應該儘量利用它們已排序的特性，以減少搜尋比對的次數，這是搜尋的基本原則，二分搜尋法是這個基本原則的代表。

解法

在二分搜尋法中，從數列的中間開始搜尋，如果這個數小於我們所搜尋的數，由於數列已排序，則該數左邊的數一定都小於要搜尋的對象，所以無需浪費時間在左邊的數；如果搜尋的數大於所搜尋的對象，則右邊的數無需再搜尋，直接搜尋左邊的數。

所以在二分搜尋法中，將數列不斷的分為兩個部份，每次從分割的部份中取中間數比對，例如要搜尋 92 於以下的數列，首先中間數索引為(0+9)/2 = 4（索引由 0 開始）：

[3 24 57 57 **67** 68 83 90 92 95]

由於 67 小於 92，所以轉搜尋右邊的數列：
3 24 57 57 67 [68 83 **90** 92 95]

由於 90 小於 92，再搜尋右邊的數列，這次就找到所要的數了：
3 24 57 57 67 68 83 90 [**92** 95]

實作

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void quicksort(int[], int, int);
int bisearch(int[], int);

int main(void) {
    int number[MAX] = {0};
    int i, find;

    srand(time(NULL));

    for(i = 0; i < MAX; i++) {
        number[i] = rand() % 100;
    }

    quicksort(number, 0, MAX-1);

    printf("數列：");
    for(i = 0; i < MAX; i++)
        printf("%d ", number[i]);

    printf("\n 輸入尋找對象：");
    scanf("%d", &find);

    if((i = bisearch(number, find)) >= 0)
        printf("找到數字於索引 %d ", i);
    else
        printf("\n 找不到指定數");
```

```
    printf("\n");

    return 0;
}

int bisearch(int number[], int find) {
    int low, mid, upper;

    low = 0;
    upper = MAX - 1;

    while(low <= upper) {
        mid = (low+upper) / 2;
        if(number[mid] < find)
            low = mid+1;
        else if(number[mid] > find)
            upper = mid - 1;
        else
            return mid;
    }

    return -1;
}

void quicksort(int number[], int left, int right) {
    int i, j, k, s;

    if(left < right) {
        s = number[(left+right)/2];
        i = left - 1;
        j = right + 1;

        while(1) {
            while(number[++i] < s) ;   // 向右找
            while(number[--j] > s) ;   // 向左找
            if(i >= j)
                break;
            SWAP(number[i], number[j]);
        }

        quicksort(number, left, i-1);     // 對左邊進行遞迴
        quicksort(number, j+1, right);    // 對右邊進行遞迴
    }
}
```

Unit 34：費氏搜尋法

Algorithm Gossip: 費氏搜尋法

說明

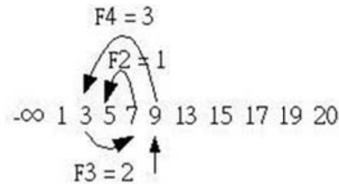
二分搜尋法每次搜尋時，都會將搜尋區間分為一半，所以其搜尋時間為 O(log(2)n)，log(2)表示以 2 為底的 log 值，這邊要介紹的費氏搜尋，其利用費氏數列作為間隔來搜尋下一個數，所以區間收斂的速度更快，搜尋時間為 O(logn)。

解法

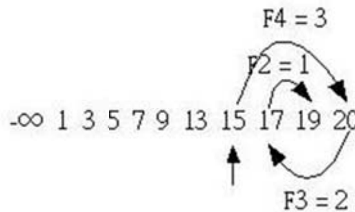
費氏搜尋使用費氏數列來決定下一個數的搜尋位置，所以必須先製作費氏數列，這在之前有提過；費氏搜尋會先透過公式計算求出第一個要搜尋數的位置，以及其代表的費氏數，以搜尋對象 10 個數字來說，第一個費氏數經計算後一定是 F5，而第一個要搜尋的位置有兩個可能，例如若在下面的數列搜尋的話（為了計算方便，通常會將索引 0 訂作無限小的數，而數列由索引 1 開始）：

-∞ 1 3 5 7 9 13 15 17 19 20

如果要搜尋 5 的話，則由索引 F5 = 5 開始搜尋，接下來如果數列中的數小於指定搜尋值時，就往左找，大於時就向右，每次找的間隔是 F4、F3、F2 來尋找，當費氏數為 0 時還沒找到，就表示尋找失敗，如下所示：



由於第一個搜尋值索引 F5 = 5 處的值小於 19，所以此時必須對齊數列右方，也就是將第一個搜尋值的索引改為 F5+2 = 7，然後如同上述的方式進行搜尋，如下所示：



至於第一個搜尋值是如何找到的？我們可以由以下這個公式來求得，其中 n 為搜尋對象的個數：

$F_x + m = n$
$F_x <= n$

也就是說 Fx 必須找到不大於 n 的費氏數，以 10 個搜尋對象來說：

$F_x + m = 10$

取 $F_x = 8, m = 2$，所以我們可以對照費氏數列得 x = 6，然而第一個數的可能位置之一並不是 F6，而是第 x-1 的費氏數，也就是 $F_5 = 5$。

如果數列 number 在索引 5 處的值小於指定的搜尋值，則第一個搜尋位置就是索引 5 的位置，如果大於指定的搜尋值，則第一個搜尋位置必須加上 m，也就是 $F_5 + m = 5 + 2 = 7$，也就是索引 7 的位置，其實加上 m 的原因，是為了要讓下一個搜尋值剛好是數列的最後一個位置。

費氏搜尋看來難懂，但只要掌握 $F_x + m = n$ 這個公式，自己找幾個實例算一次，很容易就可以理解；費氏搜尋除了收斂快速之外，由於其本身只會使用到加法與減法，在運算上也可以加快。

實作

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 15
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void createfib(void);        // 建立費氏數列
int findx(int, int);              // 找 x 值
int fibsearch(int[], int);   // 費氏搜尋
void quicksort(int[], int, int);   // 快速排序

int Fib[MAX] = {-999};

int main(void) {
    int number[MAX] = {0};
    int i, find;

    srand(time(NULL));

    for(i = 1; i <= MAX; i++) {
        number[i] = rand() % 100;
    }

    quicksort(number, 1, MAX);

    printf("數列：");
    for(i = 1; i <= MAX; i++)
        printf("%d ", number[i]);

    printf("\n 輸入尋找對象：");
    scanf("%d", &find);

    if((i = fibsearch(number, find)) >= 0)
        printf("找到數字於索引 %d ", i);
    else
        printf("\n 找不到指定數");

    printf("\n");

    return 0;
}

// 建立費氏數列
void createfib(void) {
    int i;

    Fib[0] = 0;
    Fib[1] = 1;

    for(i = 2; i < MAX; i++)
        Fib[i] = Fib[i-1] + Fib[i-2];
}

// 找 x 值
int findx(int n, int find) {
    int i = 0;
```

```
    while(Fib[i] <= n)
        i++;

    i--;
    return i;
}

// 費式搜尋
int fibsearch(int number[], int find) {
    int i, x, m;

    createfib();

    x   = findx(MAX+1,find);
    m = MAX - Fib[x];
    printf("\nx = %d, m = %d, Fib[x] = %d\n\n",
                                        x, m, Fib[x]);

    x--;
    i = x;

    if(number[i] < find)
        i += m;

    while(Fib[x] > 0) {
        if(number[i] < find)
            i += Fib[--x];
        else if(number[i] > find)
            i -= Fib[--x];
        else
            return i;
    }
    return -1;
}

void quicksort(int number[], int left, int right) {
    int i, j, k, s;

    if(left < right) {
        s = number[(left+right)/2];
        i = left - 1;
        j = right + 1;

        while(1) {
            while(number[++i] < s) ;   // 向右找
            while(number[--j] > s) ;   // 向左找
            if(i >= j)
                break;
            SWAP(number[i], number[j]);
        }

        quicksort(number, left, i-1);     // 對左邊進行遞迴
        quicksort(number, j+1, right);   // 對右邊進行遞迴
    }
}
```

Unit 35：老鼠走迷宮—堆疊與遞迴

Algorithm Gossip: 老鼠走迷宮（一）

說明

老鼠走迷宮是遞迴求解的基本題型，我們在二維陣列中使用 2 表示迷宮牆壁，使用 1 來表示老鼠的行走路徑，試以程式求出由入口至出口的路徑。

解法

老鼠的走法有上、左、下、右四個方向，在每前進一格之後就選一個方向前進，無法前進時退回選擇下一個可前進方向，如此在陣列中依序測試四個方向，直到走到出口為止，這是遞迴的基本題，請直接看程式應就可以理解。

演算法

```
Procedure GO(maze[]) [
    VISIT(maze, STARTI, STARTJ);
]

Procedure VISIT(maze[], i, j) [
    maze[i][j] = 1;

    IF(i == ENDI AND j == ENDJ)
        success = TRUE;

    IF(success != TRUE AND maze[i][j+1] == 0)
        VISIT(maze, i, j+1);
    IF(success != TRUE AND maze[i+1][j] == 0)
        VISIT(maze, i+1, j);
    IF(success != TRUE AND maze[i][j-1] == 0)
        VISIT(maze, i, j-1);
    if(success != TRUE AND maze[i-1][j] == 0)
        VISIT(maze, i-1, j);

    IF(success != TRUE)
        maze[i][j] = 0;
]
```

實作

```
#include <stdio.h>
#include <stdlib.h>

int visit(int, int);

int maze[7][7] = {{2, 2, 2, 2, 2, 2, 2},
                  {2, 0, 0, 0, 0, 0, 2},
                  {2, 0, 2, 0, 2, 0, 2},
                  {2, 0, 0, 2, 0, 2, 2},
                  {2, 2, 0, 2, 0, 2, 2},
                  {2, 0, 0, 0, 0, 0, 2},
                  {2, 2, 2, 2, 2, 2, 2}};

int startI = 1, startJ = 1;   // 入口
int endI = 5, endJ = 5;   // 出口
int success = 0;

int main(void) {
    int i, j;
```

```
    printf("顯示迷宮：\n");
    for(i = 0; i < 7; i++) {
        for(j = 0; j < 7; j++)
            if(maze[i][j] == 2)
                printf("█");
            else
                printf("  ");
        printf("\n");
    }

    if(visit(startI, startJ) == 0)
        printf("\n 沒有找到出口！\n");
    else {
        printf("\n 顯示路徑：\n");
        for(i = 0; i < 7; i++) {
            for(j = 0; j < 7; j++) {
                if(maze[i][j] == 2)
                    printf("█");
                else if(maze[i][j] == 1)
                    printf("◇");
                else
                    printf("  ");
            }
            printf("\n");
        }
    }

    return 0;
}

int visit(int i, int j) {
    maze[i][j] = 1;

    if(i == endI && j == endJ)
        success = 1;

    if(success != 1 && maze[i][j+1] == 0) visit(i, j+1);
    if(success != 1 && maze[i+1][j] == 0) visit(i+1, j);
    if(success != 1 && maze[i][j-1] == 0) visit(i, j-1);
    if(success != 1 && maze[i-1][j] == 0) visit(i-1, j);

    if(success != 1)
        maze[i][j] = 0;

    return success;
}
```

Unit 36：騎士走棋盤

Algorithm Gossip: 騎士走棋盤

說明

騎士旅遊（Knight tour）在十八世紀初倍受數學家與拼圖迷的注意，它什麼時候被提出已不可考，騎士的走法為西洋棋的走法，騎士可以由任一個位置出發，它要如何走完所有的位置？

解法

騎士的走法，基本上可以使用遞迴來解決，但是純粹的遞迴在維度大時相當沒有效率，一個聰明的解法由 J.C. Warnsdorff 在 1823 年提出，簡單的說，先將最難的位置走完，接下來的路就寬廣了，騎士所要走的下一步，「為下一步再選擇時，所能走的步數最少的一步。」，使用這個方法，在不使用遞迴的情況下，可以有較高的機率找出走法（找不到走法的機會也是有的）。

演算法

```
FOR(m = 2; m <=  總步數; m++) [
    測試下一步可以走的八個方向，記錄可停留的格數 count。

    IF(count == 0) [
        遊歷失敗
    ]
    ELSE IF(count == 1) [
        下一步只有一個可能
    ]
    ELSE [
        找出下一步的出路最少的格子
       如果出路值相同，則選第一個遇到的出路。
    ]

    走最少出路的格子，記錄騎士的新位置。
]
```

實作

```
#include <stdio.h>

int board[8][8] = {0};

int main(void) {
    int startx, starty;
    int i, j;

    printf("輸入起始點：");
    scanf("%d %d", &startx, &starty);

    if(travel(startx, starty)) {
        printf("遊歷完成！\n");
    }
    else {
        printf("遊歷失敗！\n");
    }

    for(i = 0; i < 8; i++) {
        for(j = 0; j < 8; j++) {
            printf("%2d ", board[i][j]);
        }
        putchar('\n');
```

```
    }

    return 0;
}

int travel(int x, int y) {
    // 對應騎士可走的八個方向
    int ktmove1[8] = {-2, -1, 1, 2, 2, 1, -1, -2};
    int ktmove2[8] = {1, 2, 2, 1, -1, -2, -2, -1};

    // 測試下一步的出路
    int nexti[8] = {0};
    int nextj[8] = {0};

    // 記錄出路的個數
    int exists[8] = {0};

    int i, j, k, m, l;
    int tmpi, tmpj;
    int count, min, tmp;

    i = x;
    j = y;

    board[i][j] = 1;

    for(m = 2; m <= 64; m++) {
        for(l = 0; l < 8; l++) {
            exists[l] = 0;
        }

        l = 0;

        // 試探八個方向
        for(k = 0; k < 8; k++) {
            tmpi = i + ktmove1[k];
            tmpj = j + ktmove2[k];

            // 如果是邊界了，不可走
            if(tmpi < 0 || tmpj < 0 || tmpi > 7 || tmpj > 7)
                continue;

            // 如果這個方向可走，記錄下來
            if(board[tmpi][tmpj] == 0) {
                nexti[l] = tmpi;
                nextj[l] = tmpj;
                // 可走的方向加一個
                l++;
            }
        }

        count = l;

        // 如果可走的方向為 0 個，返回
        if(count == 0) {
            return 0;
        }
        else if(count == 1) {
            // 只有一個可走的方向
```

```
            // 所以直接是最少出路的方向
            min = 0;
        }
        else {
            // 找出下一個位置的出路數
            for(l = 0; l < count; l++) {
                for(k = 0; k < 8; k++) {
                    tmpi = nexti[l] + ktmove1[k];
                    tmpj = nextj[l] + ktmove2[k];

                    if(tmpi < 0 || tmpj < 0 ||
                       tmpi > 7 || tmpj > 7) {
                        continue;
                    }

                    if(board[tmpi][tmpj] == 0)
                        exists[l]++;
                }
            }

            tmp = exists[0];
            min = 0;

            // 從可走的方向中尋找最少出路的方向
            for(l = 1; l < count; l++) {
                if(exists[l] < tmp) {
                    tmp = exists[l];
                    min = l;
                }
            }
        }

        // 走最少出路的方向
        i = nexti[min];
        j = nextj[min];
        board[i][j] = m;
    }

    return 1;
}
```

Unit 37：八個皇后

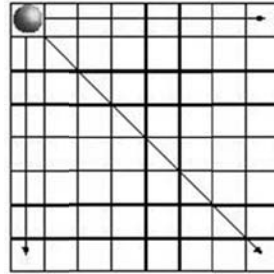
Algorithm Gossip: 八個皇后

說明

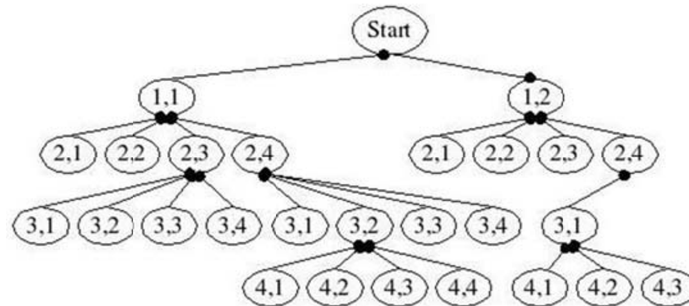
西洋棋中的皇后可以直線前進，吃掉遇到的所有棋子，如果棋盤上有八個皇后，則這八個皇后如何相安無事的放置在棋盤上，1970 年與 1971 年， E.W.Dijkstra 與 N.Wirth 曾經用這個問題來講解程式設計之技巧。

解法

關於棋盤的問題，都可以用遞迴求解，然而如何減少遞迴的次數？在八個皇后的問題中，不必要所有的格子都檢查過，例如若某列檢查過，該該列的其它格子就不用再檢查了，這個方法稱為分支修剪。



所以檢查時，先判斷是否在已放置皇后的可行進方向上，如果沒有再行放置下一個皇后，如此就可大大減少遞迴的次數，例如以下為修剪過後的遞迴檢查行進路徑：



八個皇后的話，會有 92 個解答，如果考慮棋盤的旋轉，則旋轉後扣去對稱的，會有 12 組基本解。

實作

```
#include <stdio.h>
#include <stdlib.h>
#define N 8

int column[N+1]; // 同欄是否有皇后，1 表示有
int rup[2*N+1]; // 右上至左下是否有皇后
int lup[2*N+1]; // 左上至右下是否有皇后
int queen[N+1] = {0};
int num; // 解答編號

void backtrack(int); // 遞迴求解

int main(void) {
    int i;
    num = 0;

    for(i = 1; i <= N; i++)
        column[i] = 1;
```

```
    for(i = 1; i <= 2*N; i++)
        rup[i] = lup[i] = 1;

    backtrack(1);

    return 0;
}

void showAnswer() {
    int x, y;
    printf("\n 解答 %d\n", ++num);
    for(y = 1; y <= N; y++) {
        for(x = 1; x <= N; x++) {
            if(queen[y] == x) {
                printf(" Q");
            }
            else {
                printf(" .");
            }
        }
        printf("\n");
    }
}

void backtrack(int i) {
    int j;

    if(i > N) {
        showAnswer();
    }
    else {
        for(j = 1; j <= N; j++) {
            if(column[j] == 1 &&
                rup[i+j] == 1 && lup[i-j+N] == 1) {
                queen[i] = j;
                // 設定為佔用
                column[j] = rup[i+j] = lup[i-j+N] = 0;
                backtrack(i+1);
                column[j] = rup[i+j] = lup[i-j+N] = 1;
            }
        }
    }
}
```

Unit 38：字串核對

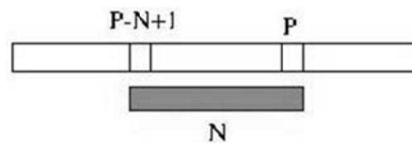
Algorithm Gossip: 字串核對

說明

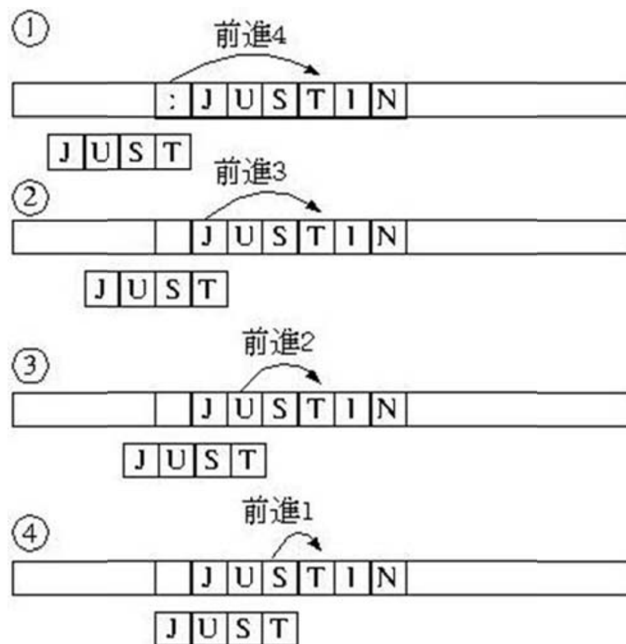
今日的一些高階程式語言對於字串的處理支援越來越強大（例如 Java、Perl 等），不過字串搜尋本身仍是個值得探討的課題，在這邊以 Boyer- Moore 法來說明如何進行字串說明，這個方法快且原理簡潔易懂。

解法

字串搜尋本身不難，使用暴力法也可以求解，但如何快速搜尋字串就不簡單了，傳統的字串搜尋是從關鍵字與字串的開頭開始比對，例如 Knuth-Morris-Pratt 演算法 字串搜尋，這個方法也不錯，不過要花時間在公式計算上；Boyer-Moore 字串核對改由關鍵字的後面開始核對字串，並製作前進表，如果比對不符合則依前進表中的值前進至下一個核對處，假設是 p 好了，然後比對字串中 p-n+1 至 p 的值是否與關鍵字相同。



那麼前進表該如何前進，舉個實際的例子，如果要在字串中搜尋 JUST 這個字串，則可能遇到的幾個情況如下所示：



依照這個例子，可以決定出我們的前進值表如下：

其它	J	U	S	T
4	3	2	1	4（match?）

如果關鍵字中有重複出現的字元，則前進值就會有兩個以上的值，此時則取前進值較小的值，如此就不會跳過可能的位置，例如 texture 這個關鍵字，t 的前進值應該取後面的 3 而不是取前面的 7。

實作

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void table(char*);    // 建立前進表
int search(int, char*, char*); // 搜尋關鍵字
void substring(char*, char*, int, int); // 取出子字串

int skip[256];

int main(void) {
    char str_input[80];
    char str_key[80];
    char tmp[80] = {'\0'};
    int m, n, p;

    printf("請輸入字串：");
    gets(str_input);
    printf("請輸入搜尋關鍵字：");
    gets(str_key);

    m = strlen(str_input); // 計算字串長度
    n = strlen(str_key);
    table(str_key);
    p = search(n-1, str_input, str_key);

    while(p != -1) {
        substring(str_input, tmp, p, m);
        printf("%s\n", tmp);
        p = search(p+n+1, str_input, str_key);
    }

    printf("\n");

    return 0;
}

void table(char *key) {
    int k, n;

    n = strlen(key);

    for(k = 0; k <= 255; k++)
        skip[k] = n;
    for(k = 0; k < n - 1; k++)
        skip[key[k]] = n - k - 1;
}

int search(int p, char* input, char* key) {
    int i, m, n;
    char tmp[80] = {'\0'};

    m = strlen(input);
    n = strlen(key);

    while(p < m) {
        substring(input, tmp, p-n+1, p);
```

```
            if(!strcmp(tmp, key))   // 比較兩字串是否相同
                return p-n+1;
            p += skip[input[p]];
      }

      return -1;
}

void substring(char *text, char* tmp, int s, int e) {
      int i, j;

      for(i = s, j = 0; i <= e; i++, j++)
            tmp[j] = text[i];

      tmp[j] = '\0';
}
```

Unit 39：背包問題（Knapsack Problem）

Algorithm Gossip: 背包問題（Knapsack Problem）

說明

假設有一個背包的負重最多可達 8 公斤，而希望在背包中裝入負重範圍內可得之總價物品，假設是水果好了，水果的編號、單價與重量如下所示：

0	李子	4KG	NT$4500
1	蘋果	5KG	NT$5700
2	橘子	2KG	NT$2250
3	草莓	1KG	NT$1100
4	甜瓜	6KG	NT$6700

解法

背包問題是關於最佳化的問題，要解最佳化問題可以使用「動態規劃」(Dynamic programming)，從空集合開始，每增加一個元素就先求出該階段的最佳解，直到所有的元素加入至集合中，最後得到的就是最佳解。

以背包問題為例，我們使用兩個陣列 value 與 item，value 表示目前的最佳解所得之總價，item 表示最後一個放至背包的水果，假設有負重量 1～8 的背包 8 個，並對每個背包求其最佳解。

逐步將水果放入背包中，並求該階段的最佳解：

- 放入李子

背包負重	1	2	3	4	5	6	7	8
value	0	0	0	4500	4500	4500	4500	9000
item	-	-	-	0	0	0	0	0

- 放入蘋果

背包負重	1	2	3	4	5	6	7	8
value	0	0	0	4500	5700	5700	5700	9000
item	-	-	-	0	1	1	1	0

- 放入橘子

背包負重	1	2	3	4	5	6	7	8
value	0	2250	2250	4500	5700	6750	7950	9000
item	-	2	2	0	1	2	2	0

- 放入草莓

背包負重	1	2	3	4	5	6	7	8
value	1100	2250	3350	4500	5700	6800	7950	9050
item	3	2	3	0	1	3	2	3

- 放入甜瓜

背包負重	1	2	3	4	5	6	7	8
value	1100	2250	3350	4500	5700	6800	7950	9050
item	3	2	3	0	1	3	2	3

由最後一個表格，可以得知在背包負重 8 公斤時，最多可以裝入 9050 元的水果，而最後一個裝入的水果是 3 號，也就是草莓，裝入了草莓，背包只能再放入 7 公斤（8-1）的水果，所以必須

看背包負重 7 公斤時的最佳解，最後一個放入的是 2 號，也就是橘子，現在背包剩下負重量 5 公斤（7-2），所以看負重 5 公斤的最佳解，最後放入的是 1 號，也就是蘋果，此時背包負重量剩下 0 公斤（5-5），無法再放入水果，所以求出最佳解為放入草莓、橘子與蘋果，而總價為 9050 元。

實作

```
#include <stdio.h>
#include <stdlib.h>

#define LIMIT 8     // 重量限制
#define N 5         // 物品種類
#define MIN 1       // 最小重量

struct body {
    char name[20];
    int size;
    int price;
};

typedef struct body object;

int main(void) {
    int item[LIMIT+1] = {0};
    int value[LIMIT+1] = {0};
    int newvalue, i, s, p;

    object a[] = {{"李子", 4, 4500},
                  {"蘋果", 5, 5700},
                  {"橘子", 2, 2250},
                  {"草莓", 1, 1100},
                  {"甜瓜", 6, 6700}};

    for(i = 0; i < N; i++) {
        for(s = a[i].size; s <= LIMIT; s++) {
            p = s - a[i].size;
            newvalue = value[p] + a[i].price;
            if(newvalue > value[s]) {// 找到階段最佳解
                value[s] = newvalue;
                item[s] = i;
            }
        }
    }

    printf("物品\t 價格\n");
    for(i = LIMIT; i >= MIN; i = i - a[item[i]].size) {
        printf("%s\t%d\n",
                a[item[i]].name, a[item[i]].price);
    }

    printf("合計\t%d\n", value[LIMIT]);

    return 0;
}
```

2010 選手訓練魔法書

By 阿南學姐

學姐團隊
陳南蓁 黃　易
郭盈妤 吳　雙
薛祐婷 邱意晴

Section IV: 2010 阿南學姐集訓篇
Unit 41-基礎字串函數 String (9 pages)
Unit 42-函數與遞迴 Function & Recursive (6 pages)
Unit 43-遞迴 Recursive (4 pages)
Unit 44-排序 Sorting (2 pages)
Unit 45-篩法 Sieve (1 pages)
Unit 46-樹狀結構 Tree (3 pages)
Unit 47-Advance C (7 pages)
Unit 48-圖形結構 Graph + DFS (4 pages)
Unit 49-動態演算法 Dynamic Programming (3 pages)
Unit 50-DPbySuhorng (4 pages)
Unit 51-貪婪演算法 Greedy Method (4 pages)

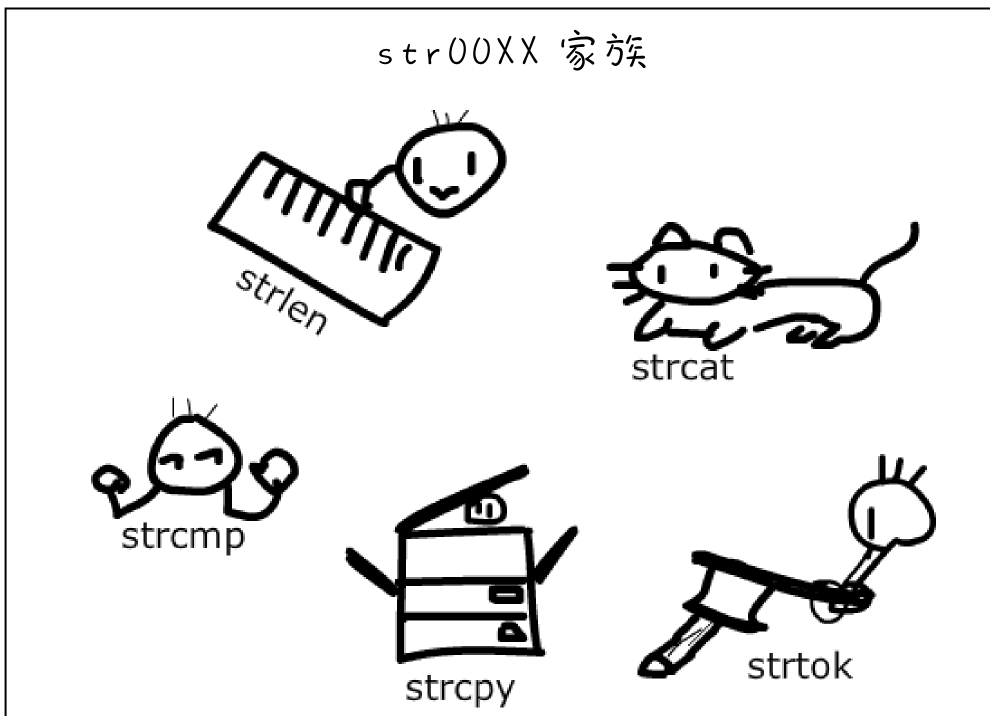
程式基礎之字串函數與處理篇

・前言・

無論是在現實世界或是網路世界，文字都是我們溝通的方式之一。在資訊的世界裡面，怎麼利用程式把文字照自己想要的方式呈現就是個很基本也很重要的技術。但我們可能會發現，許多文字處理要做的事情都很類似(例如把兩個字串接在一起，或是照特定格式輸出)，因此在 C 語言中提供了許多字串處理函數，讓程式設計師能夠省去做相同處理的時間。而今天我們就是要來介紹這些函數，並且要讓大家對 printf()和 scanf()這兩個格式化的輸入輸出有更深刻的理解喔！

・strOOXX 家族・

講到字串處理函數，首先要介紹的當然是 strOOXX 家族啦！你可能會問「什麼？我怎麼沒聽過這個家族！？」讓我們先介紹這個家族的成員，你就會知道他們是誰了！



strOOXX 家族隸屬於 string.h 這個標頭檔裡，所以如果我們要使用他，就必須要先 include 它：

```
#include <string.h>
```

如果是 C++，則爲

```
#include <cstring>
```

接著，就讓我們來一個個介紹他們吧！

首先要介紹的是 strlen，他的功能就是計算一個字串的長度。用法如下：

strlen()	計算字串長度
[標頭檔] string.h	
[格式] strlen(字串);	

[說明]

計算字串長度，回傳值為 int 型態，故可宣告 int 變數儲存。

[範例]

```
char str[]= {"Hi, how are you ? I'm fine."};
int n = strlen(str);
printf("%d\n", n);
```

執行結果為：

```
27
```

接著介紹 strcat，cat 是來自 concatenate，不過大部分的人都會暱稱這函數爲字串貓，其用法如下：

strcat()	將字串 2 接到字串 1 後
[標頭檔] string.h	
[格式] strcat(字串 1, 字串 2);	

[說明]

字串 1 必為字元陣列(字串)，字串 2 可為字串常數(即一般用""括住的文字)，亦可為字元陣列。

使用的時候必須注意字串 1 是否有足夠的空間，如果沒有可能會造成 segmentation fault，也就是常見的記憶體錯誤。

另外，strcat()函數執行時會從頭開始掃過整個字串，故重複做與長字串不適合用此函數，請用一般方法逐字串接。

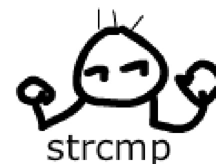
[範例]

```
char str[100]= {"Hi, how are you ?"};
strcat(str, "I'm fine.");
printf("%s\n", str);
```

執行結果為：

```
Hi, how are you ?I'm fine.
```

然後是 strcmp，顧名思義就是 str 的 compare 函數，可用來比較兩字串是否相等，用法如下：



strcmp()	比較兩個字串之 ASCII 碼排列順序
[標頭檔] string.h	
[格式] strcmp(字串 1, 字串 2);	

[說明]

比較字串 1 與字串 2 之 ASCII 碼排列順序，從頭逐漸比較，若相同就比第二個，直到其中一個字串的尾部。其回傳值如下：

字串 1 < 字串 2	回傳<0 的值
字串 1 == 字串 2	回傳 0
字串 1 > 字串 2	回傳>0 的值

註：這裡的字串 1 與字串 2 亦可為字串常數。

[範例]

```
printf("%d\n", strcmp("hi", "Hi"));
```

執行結果為：

```
1
```

[範例]

```
printf("%d\n", strcmp("Hi", "Hi"));
```

執行結果為：

```
0
```

[範例]

```
printf("%d\n", strcmp("a", "b"));
```

執行結果為：

```
-1
```

再來是 strcpy，也就是把 str copy 一份的函數，其用法如下：

strcpy()	將字串 2 複製到字串 1
[標頭檔] string.h	
[格式] strcpy(字串 1, 字串 2);	

[說明]

將字串 2 內的文字內容複製至字串 1(取代字串 1 原有內容)。

註：字串 1 必為字元陣列(字串)，字串 2 可為字串常數(即一般用""括住的文字)，亦可為字元陣列。

[範例]

```
char str[100];
strcpy(str, "Hello");
printf("%s\n", str);
```

執行結果為：

```
Hello
```

最後一個成員是 strtok，tok 指的是 **token**，指的是字串中一小段一小段的東西。用法如下：

strtok()	以區隔字串裡的字元區格被切字串
[標頭檔] string.h	
[格式] strtok(被切字串, 區隔字串);	

[說明]

使用後方區隔字串裡的字元切割字串，區隔字串內字元的排列順序與切格順序無關，切下來的字串中區隔字元則會被'\0'取代。例如：

```
char str[]= {"Hi, how are you ? I'm fine."};
strtok(str, "? ,.");
puts(str);
```

空白字元

執行結果為：

```
Hi
```

若是要繼續往下切，則必須改成

```
char str[]= {"Hi, how are you ? I'm fine."};
char *p;
strtok(str, "? ,.");
p = strtok(NULL, "? ,.");
puts(p);
```

執行結果為：

```
how
```

註記：上處所使用之*p 為指標，若是不清楚指標概念，亦可使用 strcpy()將之複製到另一個字元陣列裡。例如：

```
char str[]= {"Hi, how are you ? I'm fine."};
char str2[10];
strtok(str, "? ,.");
strcpy(str2, strtok(NULL, "? ,."));
puts(str2);
```

執行結果為：

```
how
```

・printf()和 scanf()：格式化輸入輸出・

printf 和 scanf 是我們最熟悉的兩個函數，但，你真的夠了解他們嗎？來看看吧！

printf()	基本輸出
[標頭檔] stdio.h	
[格式] printf("格式", 對應變數);	

[說明]

格式對應的是要輸出的樣式，如有變數要使用變數對照的格式：

int	%d	long long	%lld
float	%f	char	%c
double	%lf	char 陣列	%s
unsigned int	%u	16 進位整數	%x/%X

在整數的部份，我們會使用%[所佔格數]d 這樣的格式使得整個數字所佔隔數靠右固定，不足的部份會自動補空。例如：45 在[%5d]：

```
[   45]
```

如果使用[%05d]，那麼空格的部份就會補 0，變成：

```
[00045]
```

在浮點數的部份，一般我們會使用像

%[所佔格數(含小數)].[小數點後幾位]

這樣的格式來取小數點後幾位，前面的部份則是向右對齊，不足補空，總位數超過則直接印出。

例如：5.3354 在[%6.3lf]的格式下印出來長這樣：

```
[ 5.335]
```

在字串的部份，則是有(1)%-[格數]和(2)%.[格數] 兩種形式

前者是靠左對齊佔指定格數，字數不足則補空，超過則全部印出

後者是只印出指定格數，字數不足則全部印出不補空

(1)

```
char a[] = {"Hello, how are you?"};
printf("[%-28s]\n", a);
```

執行結果：

```
[Hello, how are you?         ]
```

(2)

```
char a[] = {"Hello, how are you?"};
printf("[%.8s]\n", a);
```

執行結果：

```
[Hello, h]
```

另外，上述用法所輸入的任何數字，可用*代替，並在逗號後方指定。例如：

```
char a[] = {"Hello, how are you?"};
int s = 7;
double c = 5.5344;
printf("[%.*s][%.*lf]\n", s, a, s, c);
```

執行結果為：

```
[Hello, ][5.5344000]
```

[補充]

如需在 DevC++使用 **long long** 型態，請使用%I64d。

有些特殊格式像是%g 是給浮點數用的，不會輸出多餘的零，太大會換成科學記號；%e/%E 是直接輸出成科學記號，兩者差別在 e 的大小寫。例如：

```
double a = 0.214743;
printf("%g\t%e\n", a, a);
```

執行結果：

```
0.214743        2.147430e-001
```

其他還有一些特殊的字元如下：

'\n'	換行	'\a'	嗶音	\'	'號	%%	%號
'\t'	跳格(tab)	'\b'	倒退一格	\"	"號	\\	\號

---

scanf()	基本讀入

[標頭檔] stdio.h

[格式] scanf("格式", 對應變數);

[說明]

格式對應的是要輸出的樣式，如有變數要使用變數對照的格式：

int	%d	**long long**	%lld
float	%f	**char**	%c
double	%lf	**char** 陣列	%s

除了%s 之外的對應變數，前方皆應加上一個&。

輸入中若含有空格(' ')，或是換行('\n')，皆視為一個部份(一個對應變數)已輸入結束。**(故若字串中有空白字元請用 gets())**

另外要注意的是，scanf()讀字串不管有多少連續空白、換行都會一次跳過，且在讀取變數時，若沒有輸入任何東西便按換行或空白 它會忽略而繼續讀取。

同樣地，scanf()裡的格式也可以加上數字，如此一來 scanf 只會讀取格式限定的資料，例如：

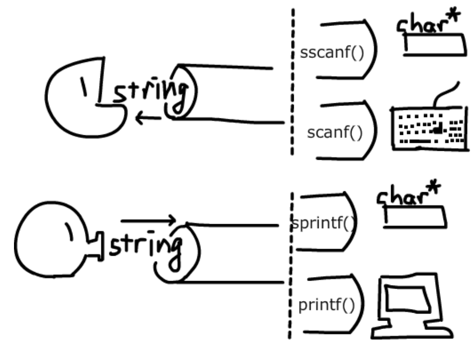
```
char str[100];
scanf("%5s", str);  /* 接著我們輸入 10 個 a */
printf("%s\n", str);
```

執行結果：

```
aaaaa
```

・sprintf()和 sscanf()：格式化輸入輸出到字串・

sprintf()/sscanf()跟 printf()/scanf()基本上是完全一樣的，差別只在後者輸出到螢幕上/從鍵盤讀入，前者則是輸出到字串中/從字串讀入。我們可以把這類函數想像成有一個管子做為中介，只差在管子的另外一端放的是什麼，如右圖所示，



sscanf()	將字串裡的資料依後面格式輸入
[標頭檔] stdio.h	
[格式] sscanf(字串, "格式", 對應變數);	

[說明]

類似 scanf()，唯一的不同在於並非給使用者輸入，而是把字串中的資料仿照一般輸入使用 scanf()方式存入變數。例如：

```
char str[]= {"3 4 0.56"};
int a, b;
double c;
sscanf(str, "%d %d %lf", &a, &b, &c);
printf("%d %d %.1lf\n", a, b, c);
```

執行結果為：

```
3 4 0.6
```

sprintf()	將後面資料輸入字串
[標頭檔] stdio.h	
[格式] sprintf(字串, "格式", 對應變數);	

[說明]

類似 printf()，唯一的不同是並非在螢幕上印出，而是將印出的東西當成字串存入指定字串中，例如：

```
char str[30];
int a = 4, b = 3;
double c = 0.84;
sprintf(str, "%d %d %.1lf", a, b, c);
puts(str);
```

執行結果為：

```
4 3 0.8
```

・gets()/puts()/getchar()/putchar()：字元/字串專用函數・

scanf()/printf()系列可以處理字元/字串以外的格式化輸入輸出，但在處理字串時常常不需要特殊格式，只需要把字串直接讀入/輸出即可，因此 C 裡面也提供了專用的函數，就讓我們一個個介紹他吧！

小知識：頻繁地呼叫輸入輸出函數會降低程式的效率，因此若能用字串一次輸入輸出，就不要使用字元輸入輸出。

gets()	字串讀入
[標頭檔] stdio.h	
[格式] gets(字串);	

[說明]

讀入一個字串存入指定的字串中，要注意的是，gets()常常會讀到上方 scanf()輸入時所讀入的換行字元，故須在上方先吸收掉該換行字元。(若上方有 scanf()可以在 gets()前加上 getchar();一行即可。)

另外，gets()之結束固定為換行字元，空白視為字串的一部分，且換行字元並不會被儲存。且 gets()碰到一次換行停一次，即使是連續的換行也會讀成空字串。

puts()	字串印出
[標頭檔] stdio.h	
[格式] puts(字串);	

[說明]

印出指定字串並換行。

除字元陣列外也可以使用如右格式：puts(“Hello”);

getchar()	字元讀入
[標頭檔] stdio.h	
[格式] getchar();	

[說明]

單行指令即可讀入，讀入的東西會放入緩衝區，若需儲存，則可宣告字元變數，執行：

字元變數 = getchar();

此函數可用來讀取任何字元，包括換行字元‘\n’與空白字元‘ ’

putchar()	字元印出
[標頭檔] stdio.h	
[格式] putchar(字元);	

[說明]

印出指定字元。

例如： putchar(‘a’);

印出結果為：

```
a
```

・memset()：填滿函數・

memset()	將陣列特定格數填入相同的字
[標頭檔] string.h	
[格式]　memset(陣列名, 填入字, 填入格數)	

[說明]

將一個字元陣列填入相同的字，亦可以填入二維陣列。

填入格數的單位是 bytes，故最好搭配 **sizeof**() 來使用。例如：

```
char p[12];
memset(p, '*', sizeof(p));
p[11] = '\0';
puts(p);
```

執行結果為：

```
***********
```

(註：字串結束一定要有結束字元'\0'，不然會有亂碼。)

另外，此函數也可用來填 **int** 陣列(一維二維皆可)，然而只能用來填數字 0&-1。(其餘的會出錯，因為它是每一個 byte 都會填一次，所以 int 的 4bytes 它會填四次，就會變成很大的數字了。)例如：

```
int n[2][2], i, j;
    memset(n, 0, sizeof(n));
    for(i=0; i<2; i++){
        for(j=0; j<2; j++)
         printf("%d", n[i][j]);
        printf("\n");
    }
    printf("\n");
```

執行結果為：

```
00
00
```

**這個函數可用於許多需全部歸零的陣列。*

函數與遞迴

函數

學過高一數學，相信你不會覺得這個名詞很陌生。C/C++之中也有函數，但是它可以做的事情不只數學計算，只要是你可以寫在 main()裡面的東西，通通都可以寫在函數裡。為什麼呢？因為，main()本身也是一個函數。

函數分成內建函數(Ex: main(), sqrt()…)和自訂函數(有的時候是別人寫的)。我們可以把函數想成一個小兵，有固定的任務，在交付任務時你可以給他一些固定格式的參考資料，最後也可以規定他要交固定格式的報告，或者不要交報告。因為他是小兵你是頭頭，所以你可以不斷地呼叫他。內建函數就是 C/C++之中內建的小兵，他要做的事情已經被固定了，就像 sqrt()的工作是開根號，你要交付給他的一個形態為 double 的數，最後他會回傳給你一份格式也是 double 的報告，上面寫著該數開根號後的值。

函數的執行流程

最簡單的函數的執行流程就是上面這兩種，第一行說明代表抽象流程，第二行說明代表程式執行流程(虛擬碼)

自訂函數的定義格式

講完了函數的大概念，現在來介紹一下自訂函數完整的定義方式：

第一種：定義在 main()之前

```
#include<stdio.h>
#include<stdlib.h>
回傳值資料型態  函數名稱(型態 1  變數 1,型態 2  變數 2……){
      /*函數處理動作*/
      return  值;
}

int main(){
      /*主程式碼*/
      system("Pause");
      return 0;
}
```

一開始就要先訂定回傳值的資料型態，然後是函數的名稱，後面的()裡面寫傳入值資料型態、名稱(等於是在規定接收到的東西得是什麼啦！)

第二種：定義在 main()之後

```
#include<stdio.h>
#include<stdlib.h>
回傳值資料型態  函數名稱(型態 1,型態 2……);
int main(){
      /*主程式碼*/
      system("Pause");
      return 0;
}
回傳值資料型態  函數名稱(型態 1  變數 1,型態 2  變數 2……){
      /*函數處理動作*/
      return  值;
}
```

跟第一種的差異在下述兩處：

1. 函數宣告的部分移到 main()之後
2. 在 main()之前要先宣告該函數的特徵：回傳值資料型態、函數名稱、傳入值資料型態(不需要名稱)，最後面要記得加分號。

寫在自訂函數大括號之中的程式碼就是你呼叫他時會執行的動作。我們可以在函數執行到某一個步驟時結束工作。結束工作的方法就是 return 某個東西，其型態必須是一開始規範的回傳值資料型態。如果型態是 void，那麼可以不用回傳任何東西回去。

呼叫函數

函數的標準呼叫格式如下：

函數名稱(傳入值 1,傳入值 2…);

如果有回傳值，我們可以這樣寫：

變數=函數名稱(傳入值 1,傳入值 2…);

等於是直接把函數的傳回值指派給變數。

要注意的是，**傳入的東西要跟我們在定義函數時設定的數量和型態一樣**。

函數的用途

函數最大的用途就是重複利用，像開根號這種很常用的功能，如果把它化成函數，我們就不需要再寫一段開根號的程式，而是呼叫 sqrt()就好了。如此一來便能夠加快寫程式的速度(絕大多數的大型程式都是由很多函數所組成的)。

區域變數與全域變數

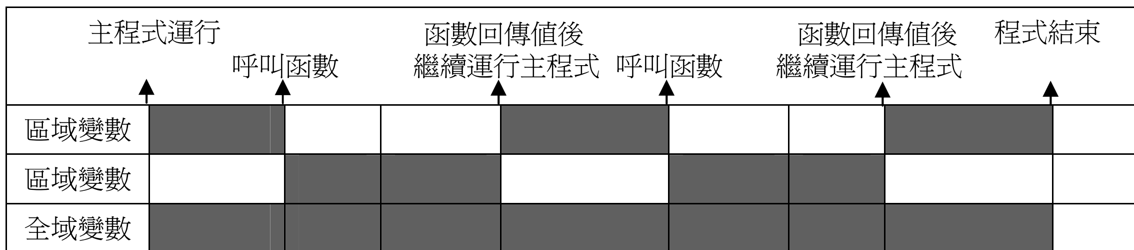
現在，我們要向大家介紹兩個名詞：**區域變數與全域變數**。

我們平常都只在 main()裡面宣告變數，但其實自訂函數裡也可以宣告變數，可是萬一我在兩個不同函數裡宣告了兩個名字一樣的變數會發生什麼事情嗎？

答案是：不會。

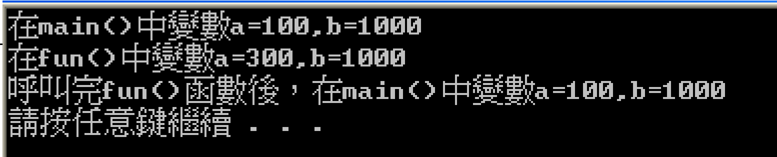
因爲**區域函數的生命週期只有在這個函數在運作的時候才會有用**，如果他沒有在運作(就是沒有在執行任何計算或是動作)，那麼這個區域變數就不會真的存在，所以你也就可以在兩個函數裡面宣告同樣的變數。

不過另外一種稱之爲全域變數的東西，就正好相反。他被宣告的時候是在所有函數之外，就是跟#include 那些同一層的東西。他的生命週期就是整支程式，無論你在哪個函數裡面都可以使用這個變數，但你也不能在任何一個函數內再宣告一個名字一樣的變數。我們可以舉個例子來畫時間軸：



第一個區域變數假定爲在 main()裡面宣告的區域變數，因爲 main()還沒結束，所以變數裡原有的資料並不會消失，只是你無法存取到他。第二個區域變數則假定爲一個自訂函數裡所宣告的變數，在呼叫時會活過來，然後傳回值後就會消失，等待下一次的呼叫；全域變數則是在整支程式運行的時候都還活著。整支程式結束後所有的變數就都會結束生命週期。

好，現在來看個例子：




```
#include<stdio.h>
#include<stdlib.h>
int b=1000;
void fun(){
	int a=300;
	printf("在 fun()中變數 a=%d,b=%d\n",a,b);
	return;
}
int main(){
	int a=100;
	printf("在 main()中變數 a=%d,b=%d\n",a,b);
	fun();
	printf("呼叫完 fun()函數後，在 main()中變數 a=%d,b=%d\n",a,b);
	system("Pause");
	return 0;
}
```

在這個例子中，我們分別在 fun()和 main()裡面都宣告了一個區域變數 a，在最外面的地方宣告了一個全域變數 b，可以由上面運行的結果發現，兩個 a 不相互影響，而 b 在整支程式裡面都是已經被宣告而且定義好了初值。

遞迴函數

有聽過遞迴函數嗎？也許你在數學上有看過這樣的函數：

$$f(x)=\begin{cases}1 & ,x=0\\ xf(x-1) & ,x>0\end{cases}$$

看的出來這個函數的功用嗎？沒錯就是階乘！這個函數自己呼叫自己，這就是所謂的遞迴函數(Recursive function)。在程式中，我們也可以讓函數呼叫自己，而且可以做的不只數學計算！現在讓我們來把上面那個數學函數用程式寫出來吧：

> 階乘：從 n 這個數字往下乘，一直乘到 1 的這個數值就稱為 n 階乘，數學符號寫成 n!，唯一例外是 0! 被定義成 1。Ex:3!=6,4!=24….
> 因為有這樣乘下去的特性，4!也會等於 4*3!甚至 4*3*2!...以此類推。

遞迴的思維方式其實和一般的直觀思考有些差異。要寫一個遞迴函數，首先我們要想清楚下面幾件事情：

1. 函數的用途
2. 函數自己和下一層之間的關係
3. 函數的終止條件

以上面階乘函數 f(n)的例子來說：

1. 函數的用途：計算 n!
2. 函數自己和下一層之間的關係：f(n) = n * f(n-1)
3. 函數的終止條件：f(0)≡1

另外一個經典的遞迴例子是費伯納基數列(Fibonacci Sequence)

> 費伯納基數列，簡稱費氏數列。是由義大利數學家費伯納基（Leonardo Fibonacci，西元 1170-1250 年）發現了這樣的一個數列：1、1、2、3、5、8、13、……；
>
> 費氏數列是是由一連串的數字所組成的（假設為 a1、a2、a3……an-1、an），而且這串數字之間具有一定的規則，就是：每一個數字必須是前兩個數字的和 (an = an-1 + an-2)。
>
> 後人也發現費氏數列有一些耐人尋味的有趣性質，例如：在大自然中，可以費氏數列來描述某些植物的生長規則，例如雛菊的花瓣的生長數目
>
> /*改自 http://hk.geocities.com/mathsworld2001/themes/fi.htm*/

我們也可以對費氏數列的遞迴函數做同樣的分析：

1. 函數的用途：

2. 函數自己和下一層之間的關係：

3. 函數的終止條件：

上面兩個例子都是數學的遞迴函數，那非數學的遞迴函數呢？最經典的就是大家都知道的河內塔(Hanoi Tower)，同樣地我們可以對河內塔進行分析：

1. 函數的用途：

2. 函數自己和下一層之間的關係：

3. 函數的終止條件：

遞迴可以處理的問題其實還有很多,以後學到 Depth First Search(DFS)和 Dynamic Programming 時，會需要很好的遞迴觀念，所以請大家遞迴的練習一定要做足，概念一定要弄清楚！

函數與遞迴 ACM 參考題單

Q271: Simply Syntax

Q374: Big Mod (可參考 Lucky 貓的提示)

Q10017: The Never Ending Towers of Hanoi

Q11121: Base -2

(這次這些都是作業了:P)

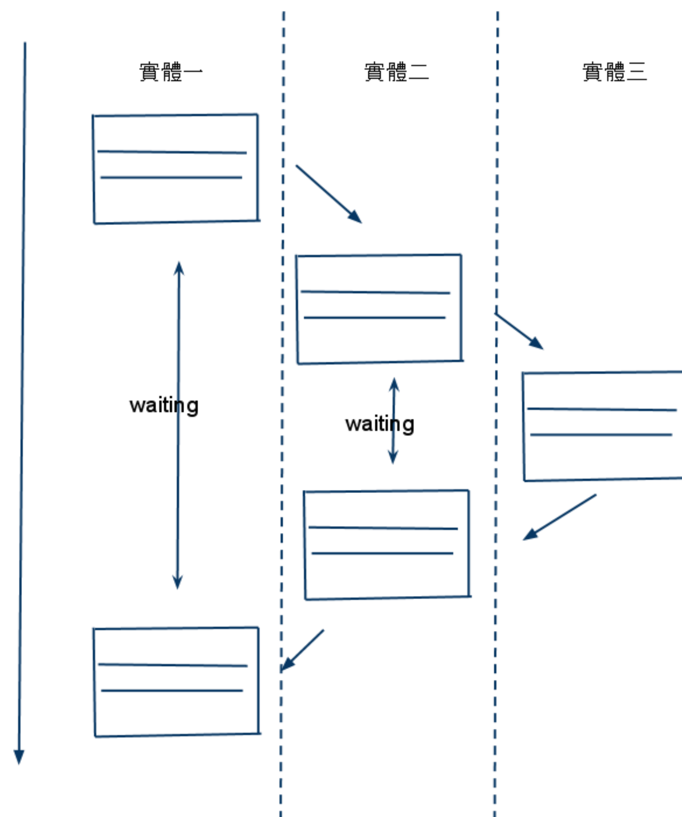
遞迴

前言

遞迴是程式設計中一個巧妙的技巧，有許許多多的問題都可以用遞迴在短短的幾行裡面解出。遞迴也在演算法中扮演著舉足輕重的地位，舉凡搜尋、動態規畫等等類型的演算法大多都有使用到遞迴，你有多了解遞迴呢？讓我們重新來檢視吧！

先談遞迴的定義

遞迴(recursion)的定義是：一個函數在自己的程式碼裡呼叫了自己。第一次看到可能會覺得很疑惑，爲什麼函數可以呼叫自己呢？首先我們可以這樣想：函數每被呼叫一次，就會新產生一個實體。所以第一次呼叫函數 A 和第二次呼叫函數 A，在跑的其實是不同的實體，因此就不會有衝突了。



一個巧妙的角度

在程式碼中，執行到呼叫遞迴函數時，應該要怎麼看比較直觀呢？一種簡單的方法是：把那行看成呼叫別的函數，而且要想成「每個函數都是在完成一個特定的任務，當函數結束 return 回來時，我已經可以得到我想要的東西了。」

最重要的兩個元素：遞迴關係和終止條件

要寫一個遞迴函數，最重要的就是要考慮兩件事情：遞迴關係和終止條件。所謂的遞迴關係是，我們在使用遞迴函數時，就像一般的函數一樣，我們不會跳著呼叫，也就是說如果函數 A 呼叫了函數 B，函數 B 又呼叫了函數 C，A 並不會知道 C 的存在，所以它也不關心 C 發生了什麼事情。A 只在乎自己告訴 B 什麼參數和 B 告訴他什麼樣的結果。在遞迴函數裡，我們稱這爲「兩者之間的遞迴關係」。定義了遞迴關係，整個遞迴函數的骨架就建好了。

終止條件則是另外一個很重要的元素，因爲遞迴不可能無窮無盡呼叫下去，所以一定要有終止的條件，也就是讓他不再繼續呼叫下去。終止條件決定了遞迴最後做出來的東西長什麼樣子，同樣遞迴關係的函數，只要終止條件不同，做出來的結果很可能也大不相同。

舉幾個簡單的例子

談到遞迴，最常舉的例子就是階乘、費柏納契(Fibonacci)數列、河內塔這幾個例子。

階乘

```
int fac(int n)
{
    if ( n <= 1 ) return 1;
    else return n * fac( n - 1 );
}
```

在這個例子裡，遞迴關係爲 `fac(n) = n * fac(n - 1)`，而終止條件爲當 `n <= 1` 時，就回傳 `1`。

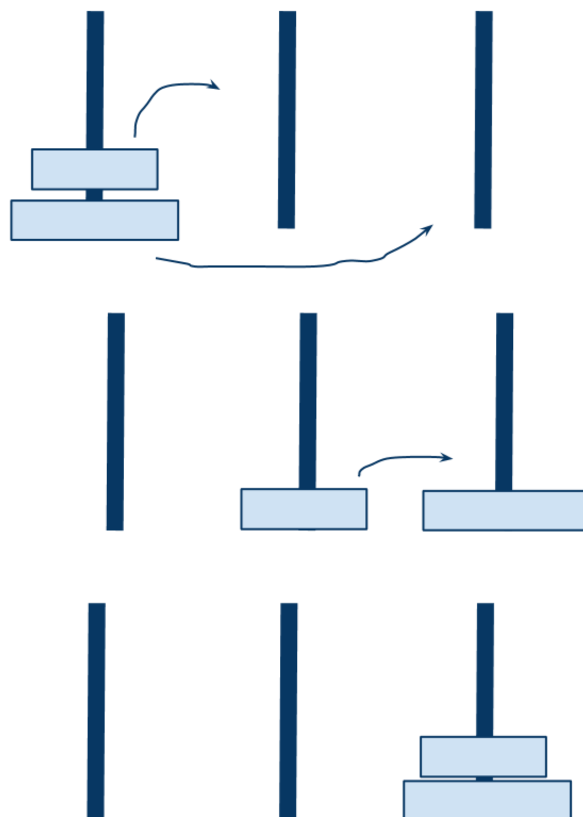
費柏納契數列

```
int fib(int n)
{
    if ( n <= 1 ) return 1;
    else return fib(n - 1) + fib(n - 2);
}
```

在這個例子裡，遞迴關係爲 `fib(n) = fib(n - 1) + fib(n - 2)`，而終止條件爲當 `n <= 1` 時，就回傳 1。

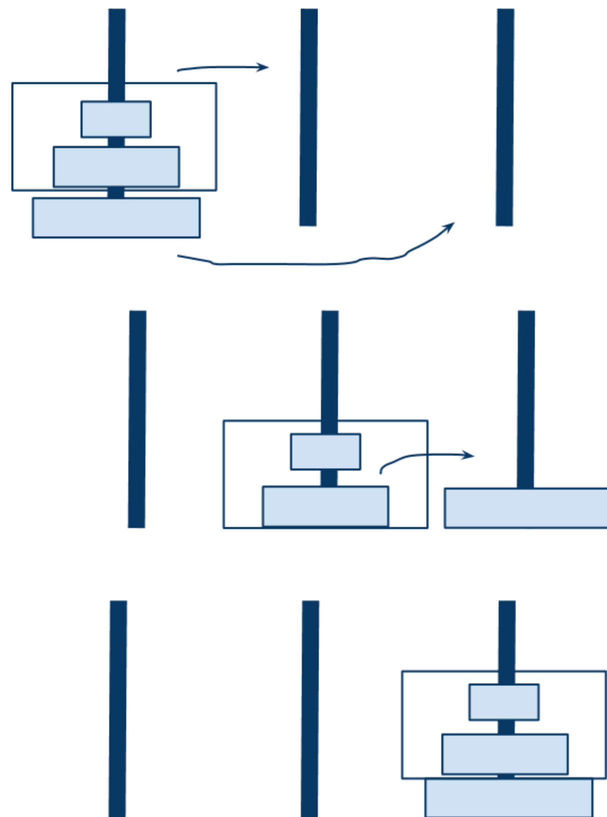
河內塔

河內塔是可以用遞迴輕鬆地解決的經典問題。河內塔的問題是這樣子的：給你三根柱子和一些盤子，從下往上盤子越來越小，每次只能移動一個盤子，最後要把所有盤子從第一根柱子移到第三根柱子上。在操作過程中，大盤子一定要在小盤子下面。以下是只有兩個盤子的操作示範：



當有三個盤子的時候，該怎麼辦呢？

答案是把整疊盤子看成兩個大盤子，也就是最後一個盤子是一個大盤子，上面剩下的其他盤子則是另外一個大盤子。我們對兩個大盤子進行兩個盤子的操作，不就可以達成我們要的目標了？



這個時候你一定會說：「嘿！這樣作弊！不是說一次只能移動一個盤子！」沒有錯，一次只能移動一個盤子，所以我們要用別的東西幫助我們移動上面那個大盤子，也就是我們要把上面那疊盤子，從第一根棍子移到第二根棍子上面。嗯？這個問題怎麼聽起來這麼熟悉？沒錯！這就是河內塔問題。只要用我們定義的遞迴函數，把盤子從第一根棍子移到第二根棍子，再把最底層的棍子移到最後一根棍子，再接著把剛剛的大盤子從第二根棍子移到第三根棍子，這個問題就解決了。什麼？你問程式該怎麼寫？這就要交給你啦！

Sorting Problem

問題描述

給你一些數字，請輸出從小到大(從大到小)排序之後的結果。

可用演算法及其平均時間複雜度

Bubble Sort: $O(n^2)$
Selection Sort: $O(n^2)$
Insertion Sort: $O(n^2)$
Merge Sort: $O(n\log n)$
Quick Sort: $O(n\log n)$
Heap Sort: $O(n\log n)$
細節可以參考良葛格的學習筆記：
http://caterpillar.onlyfun.net/Gossip/AlgorithmGossip/AlgorithmGossip.htm

C 語言內建排序函數——qsort()

qsort()是 C 語言內建(在 stdlib.h 裡)的排序函數(使用快速排序演算法)，只要傳入要排序的陣列、要被排序的元素個數、每個元素的大小(可用 sizeof()來計算)以及比較函數就可以將該陣列排序完成。其實際形式如下：

```
qsort(陣列, 元素個數, sizeof(其中一個元素), 比較函數的名稱);
```

這個函數最特別的一點是：你要自己寫比較函數，也就是第四個參數是一個「函數指標」，不過這不是我們今天要討論的重點。

比較函數的寫法如下：

```
int comp(const void *p, const void *q)
{    /*  假設我們今天要排的是 int 陣列，並且比較函數名叫 comp  */
int a = *(int *)p;
int b = *(int *)q;
if(a < b) return -1;
else if(a == b) return 0;
else if(a > b) return 1;
}
```

這樣假設我們要排序一個有 N 個元素的 int 陣列 a，必須要這樣呼叫：

```
qsort(a, N, sizeof(int), comp);
```

這樣跑完 a 就會變成由小到大排列了，如果要由大到小，將-1 和 1 對調即可。

其實 comp 函數有比較短的寫法：

```
int comp(const void *p, const void *q)
{
return *(int *)p - *(int *)q;
}
```

這樣寫很方便，透過減法直接可以得到其關係式，但最大的問題是：當兩數前者是-2147483648，後者是任意正整數時的這種情況，就會造成overflow，所以使用這樣的方法時務必注意數值範圍。

練習題

11369 - Shopaholic

篩法建質數表

何謂篩法

我們這裡所說的篩法，是 Eratosthenes 篩法(Sieve)。它是一種由古希臘數學家 Eratosthenes 提出的，主要的功能是用來找出不大於 n 的所有質數。

篩法的原理

篩法的主要原理是建一個數字表，並且逐步把不大於 n 的所有質數的倍數都從表上砍掉，最後表上剩下的數就是質數。

篩法建質數表的實作

一般而言，我們會開一個夠大的陣列(到 n)，紀錄相對索引值是否已經被砍掉。之後，我們開始去窮舉每一個數，如果發現他是質數，便將其倍數通通從表上砍掉，並記錄下來，如果不是就跳過。這樣最後作完便能得到不大於 n 的所有質數了。順帶一提，我們通常會把 2 當成特例，從 3 開始處理。

較有效率的篩法

1. 假設現有一個質數 p，那我們可以從 p*p 這個數開始砍。
   因爲，若現有一個數 q 並非質數，則知道必有一個小於等於 sqrt(q)的因數
   若一數 q 小於 p*p 會被 p 砍到，則必有一個質因數小於 p。
   舉例來說，我們知道 5 是質數，那麼我們可以從 5 * 5 = 25 開始往上砍，因爲小於 25 的數像 15 = 3 * 5 會先被 3 砍到。
2. 跳過偶數
   把 2 作爲特殊處理之後，剩下的偶數便都是合數了。
   在砍質數倍數時，記得每次加兩倍，跳過偶數倍。
3. 有要砍東西的質數最高到 sqrt(n)，故 sqrt(n)以上的質數另外處理，不用砍，只需記錄即可。

參考題單

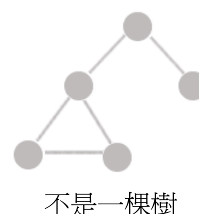
Q160: Factors and Factorials
Q294: Divisors
Q583: Prime Factors
Q10235: Simply Emirp
Q10789: Prime Frequency

樹 Tree

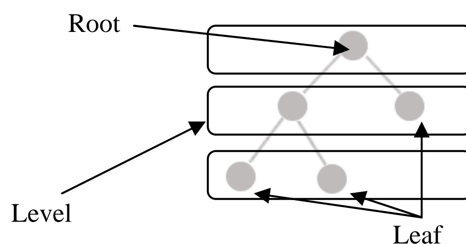
簡介

樹(Tree)是一種資料結構，屬於圖(Graph, 之後會細講)的一種，由節點(Node)組成和邊(Edge)，並具有以下特性：

1. 樹沒有環。
2. 任意兩點之間只有唯一一條路徑。
3. 樹上所有點之間都相連通。
4. 在樹上任意添加一條邊，就會產生環。
5. 在樹上任意刪除一條邊，一顆樹就裂成兩棵樹。
6. 邊數等於點數減一。

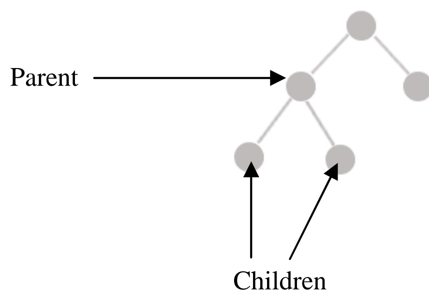

不是一棵樹

下面是範例：



一棵樹

我們稱最上面的點爲「根」(Root)，最下面的點爲「葉」(Leaf)，每個節點到根的距離爲「層」(Level)。



和節點連結且比節點上一層的節點，我們稱之爲父節點(Parent)，反之則稱爲子節點(Children)。

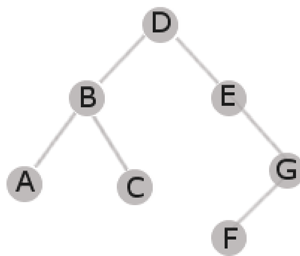
樹裡面最常見的是二元樹(Binary Tree)，其特性就是每個父節點最多只有兩個孩子。

二元樹實作

```
struct node
{
    int n;
    struct node *left, *right;
};
int main()
{
    struct node *root;
    root = (struct node*)malloc(sizeof(struct node));
    root->n = 0;
    root->right = root->left = NULL;
}
```

二元樹的表示式

二元樹有三種表示式：前序(preorder)、中序(inorder)、後序(postorder)，其記憶的口訣分別是「中左右」、「左中右」、「左右中」，以「中」字所在的位置可以快速記憶。



以這棵樹來說，它的三種表示法如下：

前序 → DBACEGF

中序 → ABCDEFG

後序 → ACBFGED

二元樹的復原

當我們知道一棵樹的中序表示式還有前序或後序表示式時，一棵樹的樣子就固定了。我們可以透過觀察法及遞迴的方式將樹重新建立起來。

以上面那棵樹爲例子，假設我們已經知道前序和中序表示式，那麼我們就可以按照前序逐步處理比且以中序作爲左右邊的相關依據。復原步驟如下：

1. D: 前序的第一個一定是樹根
2. B: 中序時他在 D 的左邊，故他爲 D 的左 child
3. A: 中序時他在 B 的左邊，故他爲 B 的左 child
4. C: 中序時他在 B 的右邊，故他爲 B 的右 child
5. E: 中序時他在 D 的右邊，故他爲 D 的右 child
6. G: 中序時他在 E 的右邊，故他爲 E 的右 chlid
7. F: 中序時他在 G 的左邊，故他爲 G 的左 child

通常我們會先依序找出前序裡每個節點其中序的位置，在逐步處理時可直接檢視兩者相對位置。

二分搜尋樹(Binary Search Tree, BST)

BST 是一種很特別的樹，就是將資料插入時，必須也同時要進行排序。這樣子的好處是，當建完樹時，資料就已經排序好了，在做字串處理上非常好用。簡單的 BST 可以這樣實作：

```
struct node
{
    int n;
    struct node *left, *right;
};
int main()
{
    int num[7] = {10, 30, 50, 11, 9, 20, 3};
    int i;
    struct node *root, *current, *tmp;
    root = (struct node*)malloc(sizeof(struct node));
    root->n = num[0];
    root->right = root->left = NULL;
    for(i=1; i<n; i++)
    {
        current = root;
        while(current != NULL)
        {
            if(current->n == num[i]) break;
            else if(current->n < num[i])
            {
                if(current->right == NULL)
                {
                    tmp = (struct node*)malloc(sizeof(struct node));
                    tmp->n = num[i];
                    tmp->left = tmp->right = NULL;
                    current->right = tmp;
                    break;
                }
                current = current->right;
            }
            else if(current->n > num[i])
            {
                if(current->left == NULL)
                {
                    tmp = (struct node*)malloc(sizeof(struct node));
                    tmp->n = num[i];
                    tmp->left = tmp->right = NULL;
                    current->left = tmp;
                    break;
                }
                current = current->left;
            }
        }
    }
}
```

練習題

536 - Tree Recovery
548 - Tree
10701 - Pre, in and post
10815 - Andy's First Dictionary (BST)

前置處理器－#define 和#include

一般的 C 語言中，程式指令是要給機器看然後來執行的，所以會需要經過「編譯」這個動作。而由#開頭的程式碼並不會編譯給機器執行，而是在編譯的過程中給編譯器看的，我們稱他們爲「前置處理器」。

先說我們平常用的#include，爲什麼每次我們都要先加那兩行呢？其實 stdio.h 和 stdlib.h 是兩個「函數庫」，也就是已經放了很多函數進去裡面所以可以直接使用那些函數。像 printf()和 scanf 這些基本的函數，就是被定義在 stdio.h 裡面的！至於 stdlib.h 則是我們寫讓他暫停的那行－system(“Pause”);system()所定義的地方。所以如果你沒有 include 好的話，程式執行上就會有一些問題。(不幸的是 DevC++有一點天兵，你就算沒寫它也不會怎樣，但實際上是去好一點的編譯器編譯是會怎樣的。)

再來是新的東西－#define(巨集指令)，他的格式如下：

```
#define 識別名稱 代換標記
```

記得最後面不用加分號。

所謂的識別名稱就是替換內容的縮寫，一般來說較喜歡用大寫標示較易辨認；代換標記的部份可以是常數、字串或是函數。下面舉些簡單的合法例子：

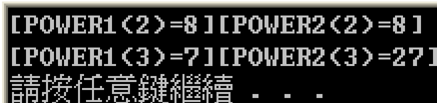
```
#define PI 3.14
#define SAY “CKEFGISC”
#define POWER(X) (X)*(X)*(X)
```

這些都是合法的定義。來看用這些寫成的例子：

```
#include<stdio.h>
#include<stdlib.h>
#define PI 3.14
#define SAY "CKEFGISC"
#define POWER(X) (X)*(X)*(X)
int main(){
    printf("圓半徑是 2，圓面積爲%f\n",2*2*PI);
    printf("We are %s\n",SAY);
    printf("2 的三次方=%d\n",POWER(2));

    system("Pause");
    return 0;
}
```

第三個定義的巨集其實就是一個簡單的小函數，不過要注意的是我們在後面每一個 X 都加上括號，爲什麼呢？來看一下有沒有加括號的差別：

```
#include<stdio.h>
#include<stdlib.h>
#define POWER1(X) X*X*X
#define POWER2(X) (X)*(X)*(X)
int main(){
    int i=2;
    printf("[POWER1(%d)=%d]",i,POWER1(i));
    printf("[POWER2(%d)=%d]\n",i,POWER2(i));

    printf("[POWER1(%d)=%d]",i+1,POWER1(i+1));
    printf("[POWER2(%d)=%d]\n",i+1,POWER2(i+1));

    system("Pause");
    return 0; }
```



有沒有發現了？沒有加括號的，在處理 i+1 的時候產生了錯誤。

因爲對他來說那只是一個代換，等於說現在我把 i+1 丟進去的時候，沒加括號的那一個會變成：i+1*i+1*i+1，而數學上又是先乘除後加減，所以就會變成 7。也是因爲這樣所以最好在定義巨集時要注意到這個問題，妥當運用括號。

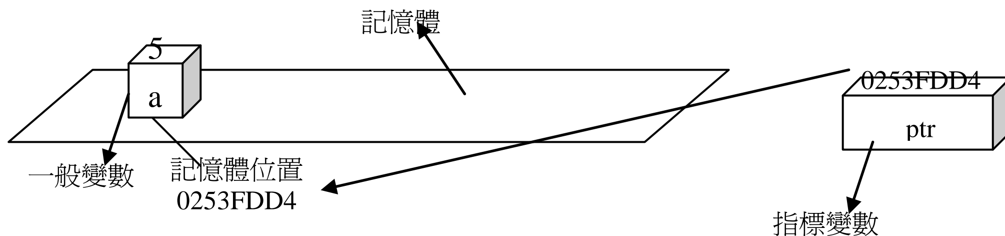
指標

指標，對於程式的學習是一個很大的關鍵，大型的程式、真正開始運用的許多東西都得運用到指標，而指標的概念說簡單，也不簡單，說難，也沒有那麼難(笑)。那麼，我們就正式來介紹一下指標吧！

指標(Pointer)其實是一種特殊的變數，用來存放的是***變數在記憶體中的位置***。我們在宣告變數的時候，記憶體上會產生一塊足夠的空間給這個變數；那麼我們要怎麼知道這個變數在哪？當然就是利用記憶體位址啦！位址，就好像我們的住家住址的感覺一樣，他是獨一無二。系統中也可以依位置來存取變數，就像郵差可以依住址來送信一樣。

利用指標變數，我們可以把變數在記憶體內的位址存入指標中，當我們需要用到這個變數時，便可以利用指標先找到該變數的位址，再由該位址取出位址內所儲存的變數值。這種方式我們稱之爲「***間接定址取值法***」。

畫個簡單的圖示來說：



等於說我們可以用指標變數裡存放的記憶體位置，找到他所指向的那個一般變數裡的數值。

指標變數的宣告

指標的變數所存放的是某個資料在記憶體中的位址，不是像數值、文字等資料內容，所以有特定的宣告方式：

```
資料型態 *指標變數名稱;
```

最重要的就是多了*這個符號，即是指標符號，這些變數有了指標符號就會變成指標變數，像是：

```
int *ptri;       /*宣告一個名爲 ptri 的整數型態指標變數*/
char *ptrch;   /*宣告一個名爲 ptrch 的字元型態指標變數*/
```

只要是 C 語言的資料型態，通通可以宣告成指標。

指標變數的使用

使用指標變數時，不是存取放在指標裡的位置，就是用指向位置內的那個資

料，這兩樣工作需要經由下列兩種指標運算子來完成：

1. ***位址運算子&***：用來求取變數的位置，像我們平常宣告變數 a，&a 就是他的記憶體位址。
2. ***依址取值運算子****：用來取得指標指向的位置內所放的資料。假設變數 a 的值是 100，有個指標 ptra 指向 a，*ptra 就是 a 的值 100。

像上面的例子，要怎樣把指標和變數連結起來呢？這時候就是要用位址運算子把記憶體位置存到指標內去了，如這樣的敘述：prta=&a;

化成程式來看看：

```
#include<stdio.h>
#include<stdlib.h>
int main(void){
    int a=100;
    int *ptra;
    ptra=&a;
    printf("a=%d,address of a=%p\n",a,&a);
    printf("*ptra=%d,ptra=%p\n",*ptra,ptra);
    system("Pause");
    return 0;
}
```

```
a=100,address of a=0022FF74
*ptra=100,ptra=0022FF74
請按任意鍵繼續 . . .
```

//剛剛沒有提到的是記憶體位置輸出的格式是%p。

ptra=&a 將 a 的記憶體位置放入 ptra，(&a 是 a 的記憶體位置)，*ptra 則是將 ptra 裡面指向的位置裡面的值取出。

剛剛所提到的兩個運算子，要注意的是因爲&是取址，所以像是&100 或是&(i++)都是不合法的！而*(依址取值運算子)跟在一開始宣告時的意義(指標符號)是有不同的意思唷！不要把兩個搞混了。

另外要注意的是，***指標指向的變數型態要跟他自己的型態一致***，不然並無法正確印出其內容。

指標的運算

C 語言中指標提供了三種運算：設定運算、加減法運算、差值運算。陷在來介紹一下：

1. ***設定運算***

   其實跟變數的指派很像，也一樣是使用了=(這裡稱爲設定運算子)。

   像 ptra=&a;這樣就是設定。而我們也可以假設宣告了兩個指標變數－ptra、ptrb;並且設定 ptra=&a;

   當我們寫 ptrb=ptra;時，ptrb 所指向的也是 a。而如果我們使用了*ptrb=*ptra 會發生什麼事呢？

   程式會先找到兩個的記憶體位置，並且將 ptrb 所指向的變數 b 裡面換成 ptra 所指向的變數 a 的值。

小重點：因爲指標的使用非常靈活，而且是直接控制到了記憶體，不小心可能會造成不可預期的情況發生，所以在宣告之後最好能馬上指向正確的變數，如果不行我們可以先將其設成 NULL。

Ex:ptr=NULL;。

把剛剛所敘述的化爲程式碼：

```
#include<stdio.h>
#include<stdlib.h>
int main(void){
    int a=100,b=200;
    int *ptra,*ptrb;
    ptra=&a;
    ptrb=&b;
    printf("一開始的 a=%d,b=%d\n",a,b);
    *ptrb=*ptra;
    printf("現在的 a=%d,b=%d\n",a,b);
    system("Pause");
    return 0;
}
```

```
一開始的a=100,b=200
現在的a=100,b=100
請按任意鍵繼續 . . .
```

有沒有注意到，因爲*ptrb=*ptra;的這行，b 的值改變了！注意這個改變是真的改變，原本 b 的值就被代換掉了唷！

2. ***指標的加法與減法運算***

在指標中我們一樣可以使用++跟--這兩個運算子，可是要記得它跟你在變數用的有不太一樣的意義－改變的是指向的記憶體位置。

比如說現在有個整數指標 ptra 指向整數變數 a 的記憶體位置(假設是 100)，如果執行 ptra++，那麼原本的 100 就會變成 104(爲什麼是 4！？因爲整數在記憶體中佔了 4 個位址長度，所以往後移一格便是 104。)

舉個例子來看：

```
#include<stdio.h>
#include<stdlib.h>
int main(void){
    int a=100,b=200;
    int *ptra,*ptrb;
    ptra=&a;
    ptrb=&b;
    printf("一開始的*ptra(%p)=%d,*ptrb(%p)=%d\n",ptra,*ptra,ptrb,*ptrb);
    ptra++;
    ptrb--;
    printf("現在的*ptra(%p)=%d,*ptrb(%p)=%d\n",ptra,*ptra,ptrb,*ptrb);
    system("Pause");
    return 0;
}
```

```
一開始的*ptra(0022FF74)=100,*ptrb(0022FF70)=200
現在的*ptra(0022FF78)=2293680,*ptrb(0022FF6C)=2293624
請按任意鍵繼續 . . .
```

有發現嗎？記憶體位置改變之後，因爲並沒有剛好變成對方的記憶體位置，兩個指標變數指向的值就既非 a 也非 b 了。

要注意如果在上面的例子我們想要把 b 的數值加上 1，那我們並不能寫「*ptrb++;」，而是得寫「*ptrb=*ptrb+1;」，前者的意思是去找 b 的下一格記憶體的數值而非把它指向的那格記憶體內的數字加上 1。

3. ***指標的差值運算***

C 語言中指標並不允許直接做加法運算，可是卻允許兩個相同資料型態的指標做差值運算，算出來的意義是什麼？答案是：在記憶體中兩個之間的距離，也就是相差的資料個數。我們來看下面這個例子：

```
#include<stdio.h>
#include<stdlib.h>
int main(void){
	int a=100,b=200;
	int *ptra,*ptrb;
	ptra=&a;
	ptrb=&b;
	printf("*ptra(%p)=%d,*ptrb(%p)=%d\n",ptra,*ptra,ptrb,*ptrb);
	printf("ptra-ptrb=%d\n",ptra-ptrb);
	system("Pause");
	return 0;
}
```

```
*ptra(0022FF74)=100,*ptrb(0022FF70)=200
ptra-ptrb=1
請按任意鍵繼續 . . .
```

差值運算的時候會自動把記憶體位置相減再除以該指標資料型態的記憶體位址長度，所運行的結果可以發現 a,b 之間差一格整數型態記憶體。

指標與函數

之前提過函數可以有傳回值，但是有沒有發現，傳進去的東西可以很多，傳出來卻只能有一個！這種時候我們就可以運用指標來幫我們解決問題囉～

把指標傳入函數，函數宣告時應該要長這樣

資料型態 函數名稱(指標變數型態 *名稱 1….)

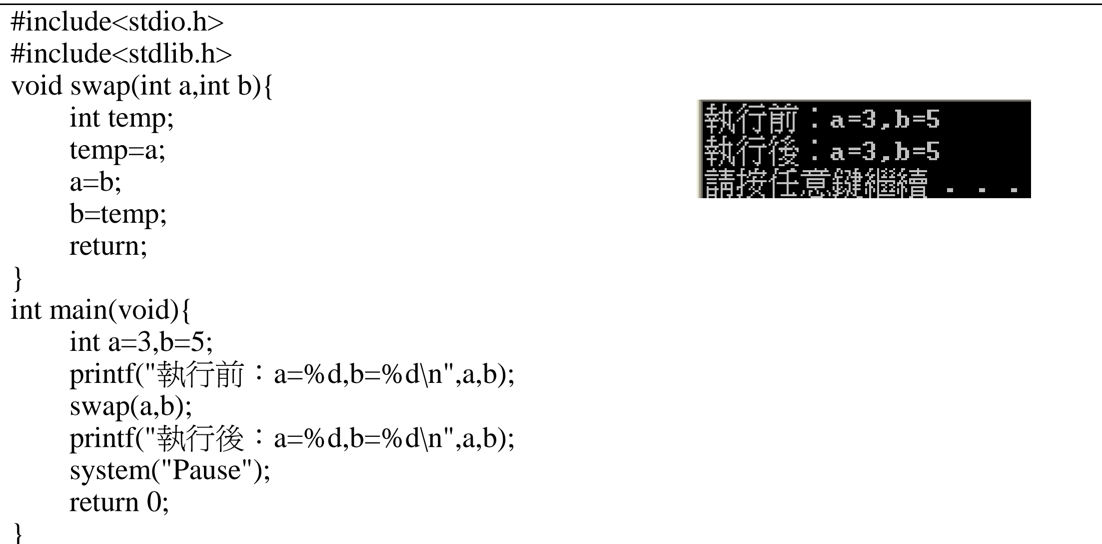
來看個例子吧。

```
#include<stdio.h>
#include<stdlib.h>
void fun(int *ptra,int *ptrb){
	int i=*ptrb;
	*ptrb=*ptra+*ptrb;
	*ptra=*ptrb-*ptra;
	return;
}
void list(int *ptra,int *ptrb){
	printf("*ptra(%p)=%d\n",ptra,*ptra);
	printf("*ptrb(%p)=%d\n",ptrb,*ptrb);
	return;
}
int main(void){
	int a=30,b=20;
	int *ptra,*ptrb;
	ptra=&a;
	ptrb=&b;
	list(ptra,ptrb);
	fun(ptra,ptrb);
	printf("處理後....\n");
	list(ptra,ptrb);
	system("Pause");
	return 0;
}
```

```
*ptra(0022FF74)=30
*ptrb(0022FF70)=20
處理後....
*ptra(0022FF74)=20
*ptrb(0022FF70)=50
請按任意鍵繼續 . . .
```

看到了嗎？list()這個函數的主要功用是印出 ptra 和 ptrb 所指向的位址以及其所含的值，而在執行了 fun()之後，我們把 ptra 跟 ptrb 兩個所指向的位址裡面的內容(也就是 a 和 b)都改變了，有沒有很神奇！活用可以很靈活，這就是指標爲什麼強大的原因之一了。

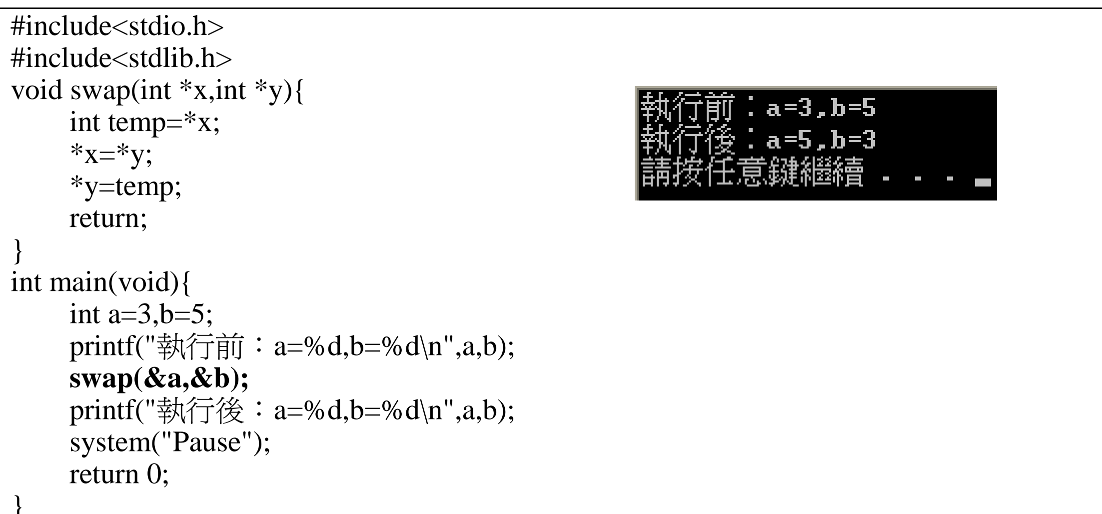
如果現在我們要寫一個函數把兩個變數裡的數值交換呢？這個時候指標也是好用的啦！先來看如果是本來不用指標，會變成……



```
#include<stdio.h>
#include<stdlib.h>
void swap(int a,int b){
	int temp;
	temp=a;
	a=b;
	b=temp;
	return;
}
int main(void){
	int a=3,b=5;
	printf("執行前：a=%d,b=%d\n",a,b);
	swap(a,b);
	printf("執行後：a=%d,b=%d\n",a,b);
	system("Pause");
	return 0;
}
```

```
執行前：a=3,b=5
執行後：a=3,b=5
請按任意鍵繼續 . . .
```

有沒有發現根本就沒差？用指標應該要這樣寫：



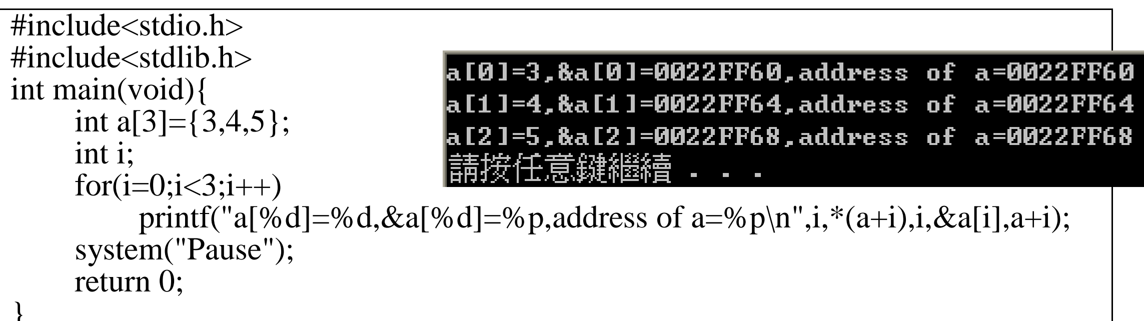
```
#include<stdio.h>
#include<stdlib.h>
void swap(int *x,int *y){
	int temp=*x;
	*x=*y;
	*y=temp;
	return;
}
int main(void){
	int a=3,b=5;
	printf("執行前：a=%d,b=%d\n",a,b);
	swap(&a,&b);
	printf("執行後：a=%d,b=%d\n",a,b);
	system("Pause");
	return 0;
}
```

```
執行前：a=3,b=5
執行後：a=5,b=3
請按任意鍵繼續 . . .
```

記得要傳入的是變數的位置，要用到先前提過的&運算子唷！

指標與陣列

　　陣列，其實我們可以把它當成是一種指標！不過，陣列是固定長度的記憶體區塊，指標則是一個變數。我們來看看他們之間的關係吧：



```
#include<stdio.h>
#include<stdlib.h>
int main(void){
	int a[3]={3,4,5};
	int i;
	for(i=0;i<3;i++)
		printf("a[%d]=%d,&a[%d]=%p,address of a=%p\n",i,*(a+i),i,&a[i],a+i);
	system("Pause");
	return 0;
}
```

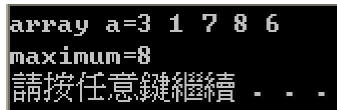
```
a[0]=3,&a[0]=0022FF60,address of a=0022FF60
a[1]=4,&a[1]=0022FF64,address of a=0022FF64
a[2]=5,&a[2]=0022FF68,address of a=0022FF68
請按任意鍵繼續 . . .
```

雖然上面的程式碼有點混亂，可是有沒有發現，我直接寫 a+i(其實就第一次就是 a，第二次是 a+1…)，跟陣列第 i 格的記憶體位置是一樣的——陣列的名稱，其

實在某個角度來說，就是一種指標。而基本上預設的陣列名稱，就是指向陣列第一格的，如果是陣列 a，a 一開始就是指向 a[0]，如果我們寫了 a++，那麼 a 現在就會指向 a[1]。

現在假設我們要用一個函數來對一個陣列來做處理，欸～有一點感覺嗎？沒錯！又是要用指標啦，就是把整個陣列的指標傳進去，然後利用函數去對指標指向的陣列位置作處理，來看個例子吧：(此例子爲找出陣列中的最大值)

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 5
int *maximum(int *m){
        int i,*max;
        max=m;
        for(i=1;i<SIZE;i++){
                if(*max<*(m+i))
                     max=m+i;
        }
        return max;

}
int main(void){
        int a[SIZE]={3,1,7,8,6};
        int i,*ptr;

        printf("array a=");
        for(i=0;i<SIZE;i++)
                printf("%d ",a[i]);
        ptr=maximum(a);
        printf("\nmaximum=%d\n",*ptr);
        system("Pause");
        return 0;
}
```



再上面的程式中我們宣告了一個傳回值爲指標的函數 maximum()，記得在呼叫這個函數時，並不需要在前面加上*符號。

字串陣列與指標陣列

一般如果我們宣告了字串陣列，那麼就會有每一個字串都配給一樣的記憶體空間的情況，假使最長的字串有 15 個字元，那麼就算其他的字串都只有 3 個字元，還是會配給到多餘的空間。幾筆資料可能不明顯，如果是數千萬筆，那個空間可就不一樣了！這時候如果使用指標陣列，就可以避免掉這個情況。指標陣列的宣告方式如下：

資料型態 *陣列名稱[個數];

記得陣列名稱前面那個*指標符號很重要。這樣宣告的意義其實就是宣告很多的指向陣列一開始的那格記憶體位置，所以不需要用到兩個中括號[]。

指標的部份大概就談到這了！其實他的運用以及和函數的搭配還有很多，只是現階段可能不太會用到。但是指標真的很重要，大家要盡量把觀念弄懂，對於「指向的概念」能夠越清楚越好唷～

圖 Graph

簡介

所謂的「圖」(Graph)乃是由「節點」(Node/Vertex)和「邊」(Edge)所組成，當邊沒有方向性時，我們稱這張圖為「無向圖」(Undirected Graph)，反之則稱為「有向圖」(Directed Graph)。當邊有特定數值時，我們稱之為「權重」，常見的權重有：花費、距離等。

兩種常見的圖資料結構

鄰接矩陣(Adjacency Matrix)和鄰接串列(Adjacency List)是最常見也最為單純，用來儲存一張圖的資料結構。鄰接矩陣就像建表，假設 1→5 中間有路，那麼 map[1][5]就會是 1 (這只是舉例的一種表示方式)。鄰接串列則是使用類似鏈結串列的東西來模擬圖，會更貼近圖的本質，但是因為存取不易，實際程式較少用到，然而當圖上的點過多時，鄰接串列將會是較好的選擇。

關於圖的細節

可以參考 DJWS 的網路日誌 http://www.csie.ntnu.edu.tw/~u91029/Graph.html

深度優先搜尋(Depth First Search, DFS)

DFS，深度優先搜尋(Depth First Search)的簡稱，又稱爲縱向搜尋法。

什麼是 DFS

顧名思義，是以 Depth，也就是深度爲優先考量的一種搜尋法。所謂的搜尋法，在圖論中，就是把所有的節點(node)走一遍的方法。

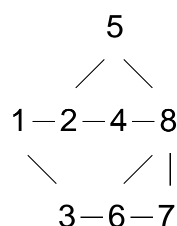
也就是我以走得深爲優先考量，當遇到末端時，才走向其他的路。
請注意所謂走得深是指「比這層深」，而不是在於「哪條比較深」，所以只要是比自己深的點就可以走。而且，在一個點上的時候，我們只會知道有跟它連接的路有哪些，而不知道這些點通往哪裡。

另外，DFS 通常也會有著「循序」、「漸增」的特質，也就是當我們要從 1 開始走，可以走的點有 3 4 5 三個， 那麼我們會先走到 3，然後等到三那條走到底了，才會回來走 4，同理走完 4 才會走 5 這樣。(不過其實這是一種習慣，並非絕對，如果你要漸減也是可以的。)

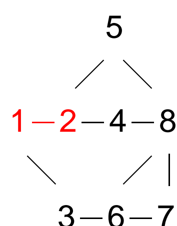
DFS 的用途非常廣泛，不一定要侷限用於「圖的搜尋」，一般像是排列組合、尤拉路徑、八皇后問題等等的，都可以用到 DFS。

從圖形來看

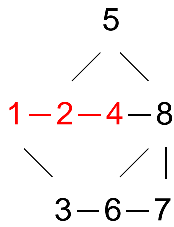
現在我們有一個圖型如下：



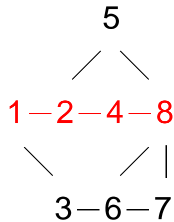
假設起點是 1，連接的點有 2 和 3，那麼以深度優先往下看會先走到 2



以 2 有連接的點來說：
1.走走過的點一定是繞路
2.以深度優先爲考量
3.循序漸增的特質
所以在 4.5 之中我們會先走到 4



4 就沒有什麼好疑惑的了，往下繼續便可以走到 8

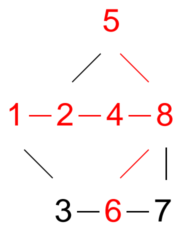


以 8 來說呢，他會在 5 6 7 之中先走到 5

可是走到 5 就可以發現：沒有其他未走過而且可以走的路可以走
這時我們便有一個 back 倒退的動作；也就是會回到上一個「仍有其他路可走」的點所以走過 5 之後，便又會走回 8。

＊ 順帶一提，back 回去的這個動作，一般來說我們稱爲 backtracking，也就是回溯，其意義就是搜尋所有可能解，並於失敗時回到上一步再找其它可能解。

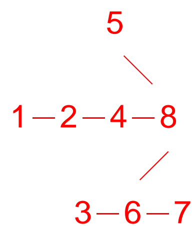
而這時 8，就可以選擇往 6 走去



這時 6 就可以走到 3 或 7

同樣地先走 3，發現 3 沒法走其他路，退回 6，然後走到 7。這時所有的點就都被走完了！

我們來看一下只保留走過路徑的圖形：



而其拜訪順序則爲 1 2 4 8 5 6 3 7

```
/*****************************************************************/
/*  練習時間
/*          5－7
/* (1)  ／|       ＼        其深度優先搜尋的拜訪順序爲何？
/*     1－3－4－6
/*          ＼    |
/*              8－9
/* (2) 1-2-3-4-8
/*        \    /
/*     7-5-6
/*****************************************************************/
```

實做的方式

一般而言，我們會使用遞迴或是堆疊、串列的方式來實做，但普遍來說以使用遞迴較爲簡單、乾淨，也能讓程式呈現簡潔有力的感覺。

至於路徑，也就是有沒有路，我們也可以使用二維陣列來表現

```
   1 2 3 4 5                     5
1  0 1 0 1 1                    / \
2  1 0 0 1 0                  1-2 3  像左邊那樣的一個二維陣列，
3  0 0 0 1 1                    \ | /  就可以表示這個圖
4  1 1 1 0 0                      4
5  1 0 1 0 0
```

補充：

遞迴其實在電腦裡面是在幕後幫你做好了很多事情。

早期的程式語言並不支援遞迴的概念，實際上遞迴有用到暫存區的堆疊。

練習題

195 - Anagram

291 - The House Of Santa Claus

441 - Lotto

10098 - Generating Fast, Sorted Permutation

Dynamic Programming 動態規劃

在開始談 DP 之前，我們要先探討一種問題——Search Problem。什麼是 Search Problem 呢？在聽到 Search 這個字的時候，你可能第一個會想到的是 Linear Search 或 Binary Search 的這種 Search。然而在這裡我們將這兩種 Search 視爲 Search Problem 的特例。爲了更精確的說明 Search，我們拿「數獨」作爲例子：

> **相信大家都玩過數獨，有一種策略就是對每一格空格，找目前為止能填進去的合法數字，接著填下一個空格，重複這個過程，如果都沒有發生填不下去的狀況，那麼就代表我們找到了一組解。如果中途發生填不下去的狀況，那麼就代表在某一格我們選的數字可能有問題，就要回去看看有沒有別種方法。**

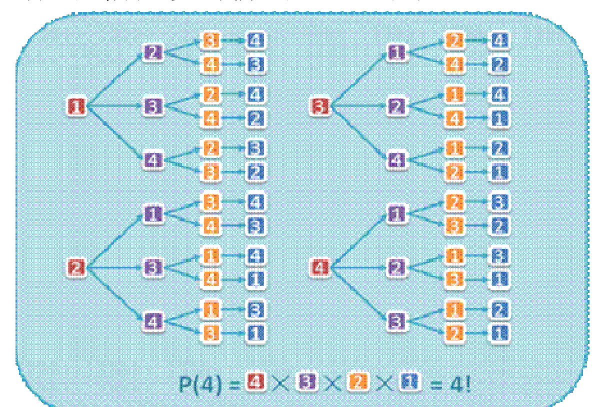
其實我們可以將整個過程，看成一棵樹(或者說是一張樹狀圖)。樹根是最開始的盤面，中間每一層都差一個數字，底層的葉子則是合法的完成盤面。Search 要做的事情，就是在這張樹狀圖上，從起點開始，找到一條能夠通往葉子的 path。但因爲最開始我們並不知道這棵樹長什麼樣，所以就要透過 search 來找出答案，而且解答有可能不只一種。

一般來說，要解決 search problem 要定義的有兩件事情：

> **第一件事情是，你怎麼去架構出這張樹狀圖？這個部分又分成兩個關鍵點：**
> **(1)樹狀圖的每一個節點代表什麼意思？**
> **(2)每層跟每層之間只能有一樣東西不同的話，那那樣東西該是什麼？**
>
> **第二件事情是，你要怎麼在這張樹狀圖上進行搜尋？要一路走到底再往回走，或者是一層走完才往下一層走？**

這兩件事情都會影響到效率、使用空間等問題。最常見的搜尋方式是 DFS 和 BFS，不過他們都只定義了後者，前者要依照不同的問題給出不同的定義，像數獨問題我們就可以定義每個節點代表盤面的長相，而每層節點之間只相差一個數字。

再舉另外一個例子來說：排列問題。排列問題就是給你一串不同的數字(有相同數字的情況也可以，但會比較複雜一點，可以自己想想看)，要你依照字典順序印出所有不同的排列。這個問題的樹狀節點代表的是目前長度的排列，root 就是長度爲 0 的排列。每一層之間設定成只能多一個數字，而且這個數字必須是之前還沒有用過的。像這樣的問題我們就可以用 DFS，並且定



義每層的分枝必須照數字大小來作。因此我們就可以利用 DFS 來 Search 到所有的解了。

好，在舉了兩個例子之後，我們可以發現樹狀圖上的節點扮演了關鍵的角色。在解題領域裡面，我們稱這個樹狀圖上的節點長相和在樹狀圖的位置爲「**狀態**」(**state**)，定義出好的狀態，就能夠讓搜尋的樹狀圖變得正確、簡單，像剛剛數獨問題，假設我定義的是「長度爲 n 時各個數字有幾個」作爲狀態，但我們會發現這個狀態並沒有辦法完整描述到我盤面的情況，而且其實早就可以很輕易地發現解答「各個數字有幾個」，像這樣的狀態就不是一個好的狀態。

而每層到每層之間的推演，也就是**這層的節點怎麼變到下一層的節點，我們稱爲「轉移方程」**。轉移方程扮演了「建邊」的角色，好的轉移方程可以大大減少樹狀圖的深度，讓問題可以更快速地找到解。

有了樹狀圖的概念之後，我們會發現一件事情：通常 search 的問題可以寫成一個遞迴式。

數獨問題的遞迴式是

S(n)=S(n-1)+某個合法數字　　　其中 n 代表第幾個空格
(我們要找的是合乎這個遞迴關係的一組解)

排列問題的遞迴式是

P(n)=P(n-1)+某個合法數字　　　其中 n 代表現在的長度
(我們要找的是合乎這個關係的所有解，並字典順序排好)

建立起 Search Problem 存在遞迴式這個概念後，現在我們要回到 DP。

DP 適合解決的，是一種特殊情況的 Search Problem，通常具有以下性質：

1. **狀態可以用簡單的方法來描述**
2. **樹狀圖存在不同的路徑可以到達相同的節點。(這個情況下樹狀圖並非真正的樹狀圖，但是為了方便我們還是稱之為樹狀圖。)**

我們可以想像一下，如果後者成立，直接用 DFS 來做會發生什麼事情呢？沒錯！就是重複運算。假設今天那個相同的節點，在樹狀圖比它下面的層數共做了一百萬次運算，另外又有 1000 種方式可以到達這個節點，那麼我們就會有 999*10^6 這麼多次重複的相同運算！這樣多麼不划算！所以 DP 用的方式，就是把算過的東西記起來！

也因此當我們遇到 DP 問題時，首先要決定的就是 search problem 的兩個重要因素：狀態和轉移方程，再來就是「我要怎麼表達這個狀態」。

接著我們以兔子跳鈴鐺爲例，兔子跳鈴鐺的解如果寫成遞迴式如下：

$$R(n) = \begin{cases} \min(R(n-1)+|x[n]-x[n-1]|, R(n-2),|x[n]-x[n-2]|) & \text{if } n>=3 \\ 0 & \text{else if}(n==1) \\ x[1] & \text{else if}(n==2) \end{cases}$$

所以我們就可以定義節點的長相就是目前已經跳的水平位移和，節點的位置就是目前是從下往上的第幾層(這是個比較簡單的例子)。也就是說，節點的長相通常就是我們要求的答案，而節點的位置就是在樹狀圖上的位置，以遞迴式來說就是參數，在 DP 中常用的方法就是 N 維陣列(一個參數是 1 維，兩個參數是 2 維，以此類推)。遞迴式如果寫清楚了，其實怎麼表達狀態自然也就清楚了。

接下來，就是 DP 的核心技巧了：從已知推到未知。也就是遞迴式我們把它變成從 n=1 一路往上做，做到我們要的 N。作的過程中我們要保證每一層都是最佳解。這樣的好處是，當我需要比我這層還要下面的解的時候，我已經知道他們是多少，而且一定是最佳，這樣就不需要再去算一次了。

以上就是 DP 這個方法的核心概念。雖然理解概念會幫助理解思考，但是 DP 這種問題還是非常的靈活、容易有變化，只能隨著經驗的累積，慢慢想到好的狀態和轉移方程了。

動態規劃 Dynamic Programming

一、簡介

先回憶一下分而治之法。在分而治之法當中，我們的核心思想是「把問題分解成規模較小、與原問題相似的子問題，解決子問題後，再合併得到原問題的答案」。考慮以下的程式：

```
Procedure Fibonacci( n )
1    If n = 0 Then Return 0
2    If n = 1 Then Return 1
3    Return Fibonacci( n – 1 ) + Fibonacci( n – 2 )
```

這是一個經典的分而治之問題。但是效率高嗎？Fibonacci(42)總共要跑多久？
仔細觀察，造成效率低落的原因似乎是因為同一個 Fibonacci (x)被重複呼叫了很多次。那我們改成以下：

```
Procedure Fibonacci( n )
1    Static Memoization[]
2    If n = 0 Then Return 0
3    If n = 1 Then Return 1
4    If Memoization [n] Exists Then Return Memoization [n]
5    Memoization [n] ← Fibonacci( n – 1 ) + Fibonacci( n – 2)
6    Return Memoization [n]
```

這就是動態規畫的核心思想；避免子問題的重複計算。其實，若換一種方式想，我們也可以寫成：

```
Procedure Fibonacci( n )
1    F[0] ← 0
2    F[1] ← 1
3    For i ← 2 To n Do
4        F[i] ← F[i – 1] + F[i – 2]
5    Return F[n]
```

事實上，第二段虛擬碼我們通常稱為記憶化搜索(Top-Down + Memoization)，第三段虛擬碼才是我們常稱的動態規劃(Bottom-Up)。最初，DP 用來解決最佳化問題，方式是透過建構最佳解的遞迴式，逐步算出每一項的值來求出所求問題的答案。

我們現在在競賽中，先暫時不那麼嚴謹，將組合計數類問題也納入 DP 的範疇，並將第二段虛擬碼那種形式也稱為 DP。這跟以後在資訊科學學到的定義可能會有差別。

二、特性與實做

能使用 DP 求解的問題通常有以下的特性：

1.最優子結構：問題的最優解可以(也必須)藉由子問題的最優解來推得。
2.重疊的子問題：因為子問題大量重疊，所以將子問題的答案記下來避免重複計算，可以節省時間。

此外，我們在討論 DP 問題時，常常以「狀態」(對當前問題的一種數學描述)來代表最優子結構，而已「狀態轉移」來代表從子問題算出原問題解答的過程。如果我們將一個狀態以一個節點表示，並將狀態之間的依存關係用有向邊代表，那我們會畫出一個 DAG。

在設計 DP 解法時，我們通常會從幾個步驟下手：

1.設計狀態：設計一種適合原問題的狀態，必須包含 DP 的前兩個特性。
2.找出狀態轉移方式：找出怎麼使用子問題來求得原問題的解答。
3.設定好邊界條件：找出最基本的問題的答案。
4.決定實做方式：使用 Top-Down 加上 Memoization 或是 Bottom-Up 填表格。
5.如果使用 Bottom-Up 的方式，需要決定好填表格的順序。

以費氏數列問題來說，我們的狀態就是費氏數列第 n 項，F_n。狀態轉移方式就是費氏數列的定義：$F_n = F_{n-1} + F_{n-2}$，邊界條件 $F_0 = 0$，$F_1 = 1$。如果使用 Bottom-Up 的方式，n 應該要從小填到大。

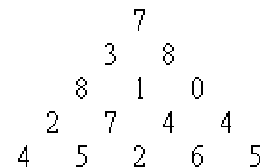
想一想：如果要印出一組解答，要怎麼寫 ?

三、思考問題

最大連續元素和 (UVa 507)
請在 A[1…n]當中找出一段 A[p…q]使得 $\sum_{i=p}^{q} A[i]$ 最大。請給出 $O(n)$ 解。

最大子矩陣 (UVa 108)
請在一個二維矩陣中找出一個子矩形，使得子矩形中的數值和最大。請給出 $O(n^3)$ 解。

三角旅行 (TIOJ 1288, also IOI '94 Day I, Problem I)
慮如右圖的三角形。現在你要從最上面走到最下面，每次可以走到左下、或右下的格子。請問經過的數字的和最大是多少 ?



池塘裡的青蛙 (TIOJ 1354)
池塘中有一隻青蛙在四塊石頭 A、B、C、D 之中跳來跳去。今青蛙由 A 起跳，每次跳到另一塊石頭，青蛙跳了 n 次後停在 A 的方法數有多少呢 ?

燈泡問題 (TIOJ 1007)
你有一個燈泡，這個燈泡最多可以連續點亮 n 個單位的時間不燒斷。請問如果總共有 m 單位的時間，那燈泡的明暗狀態有幾種可能的狀況 ? 例如 n = 2，m = 4 的答案是 13。請給出 $O(nm)$ 解。

A Graph Problem (UVa 11069)
現在有 n 個節點依序串成不分支的一長串。請算出符合以下條件的節點子集合共有幾種：
1.集合中不能有任兩個點相鄰(有邊直接連接)
2.這個集合是極大的。也就是說，你無法再加入任何一個節點，使得這個集合仍然滿足條件 1。
舉例來說，n = 5 時有{1, 3, 5} {2, 4} {2, 5} {1, 4}四種可能。

Cow Predigrees (USACO Section 2.3)
有一種神奇的樹，滿足：1.每個節點的度數不是 0 就是 2。2.這棵樹有 N 個節點，高度是 K。
請問，這種神奇的樹總共有幾種可能呢 ? 例如，N=5，H=3 的答案是 2 種。請給出 $O(KN^2)$ 的解。

背包問題 (TIOJ 1387 Striker 的秘密)
有 n 個物品，第 k 個物品重量為 W_k，價值為 V_k。如果你現在要取一些物品 S，滿足 $\sum_{i \in S} W_i \le W$，且 $\sum_{i \in S} V_i$ 的值最大。請設計一個 $O(nW)$ 解。

零錢問題 (UVa 357)
用 1, 5, 10, 25, 50 等面額的硬幣組合出 n 元有幾種方法 ? 如果是一般化的問題呢 ?

Longest Common Subsequence (UVa 10066)
請選出序列<A>的一個子序列<C>，使得<C>最長且<C>也是<B>的子序列。

Longest Increasing Subsequence (TIOJ 1175)
請找出 1~n 的某一種排列 π 與<1, 2, 3, …, n>的最長公共子序列。請給出 1.$O(n^2)$算法 2.$O(n \log n)$算法

Edit Distance (UVa 164 String Computer)
對於一個字串 s，你有以下幾個操作：
1.在 s 的某處插入一個字元 ch。
2.在 s 的某處刪除一個字元。
3.把 s 某處的一個字元 ch 替換為另一個字元 ch'。
請問要把字串 s 編輯成 s' 最少要幾個操作 ?

矩陣鍊連乘 (UVa 348)
給定一串相乘的矩陣 $A_1A_2…A_n$，請問要怎麼括括弧，才能讓乘法的運算次數最少 ? 請設計 $O(n^3)$ 解

最佳二元搜尋樹 (UVa 10304)
給定 n 個相異個鍵值 $e_1, e_2, …, e_n$ ($e_1 < e_2 < … < e_n$)，以及每個元素被查詢的次數 $f_1, f_2, …, f_n$，請問要怎麼安排二元搜尋樹，使得總花費最小 ? (花費為 $\sum_{i=1}^{n} f_i d_i$ 其中 d_i 代表元素 e_i 的深度。)

A 遊戲 (TIOJ 1029, also IOI '96 Day I, Problem I)
有一串由 n 個正整數組成的數列，兩個玩者輪流拿走一個最左邊或最右邊的數，直到所有的數都取完。而該玩家的得分就是他所取到的數字的和。請問兩個玩家都是最佳玩家的狀態下，得分分別是 ?

Counting Rectangles (UVa 10502)
請問在一個 m * n 的 0-1 矩陣中，有多少個矩形全部由 0 組成 ? 請給出 $O(nm)$ 解法。

最大正方形 (TIOJ 1097 營地)
請問在一個 m * n 的 0-1 矩陣中，全部由 0 組成的最大正方形面積是 ? ($1 \leqq m, n \leqq 5{,}000$)

Guitar Placing Area (NTUJ 0886)
請問在一個 m * n 的 0-1 矩陣中，包含 1 的個數不超過 K 的最大鎮方形面積是 ? ($1 \leqq m, n \leqq 1{,}000$)

Queue (UVa 10128)
現在有 n 位身高全不相同的人排成一列。若你從前面往後看能看到 p 個人，從後面往前看能看到 q 個人，則所有可能的排列方法共有幾種 ?

Strings (UVa 11081)
請問用字串 a、b 的子字串混和組成字串 c 總共有幾種方法 ? 1.請設計一個 $O(n^4)$ 解。 2.請優化到 $O(n^3)$

樹最遠距點對 (TIOJ 1213)
在一棵加權的無向樹中，距離最遠的兩個點距離是 ?

郵局設置問題 (TIOJ 1444, UVa 10459)
給定一棵無根樹，請問在你決定了它的樹根之後，樹的高度最大可能是 ? 最小可能是 ?

四、狀態壓縮

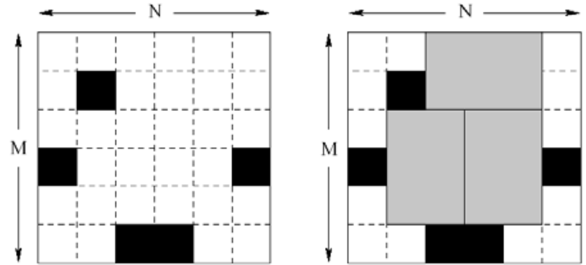
對於某些特殊的問題而言，我們的狀態可以用一個集合 S 來表示，而 S 是宇集合 U 的一個子集合。這個時候，狀態的數量總共有 $2^{|U|}$ 個。這個時候，我們可以利用二進位數的技巧，用 $0 \sim 2^{|U|}-1$ 等數字來代表每一個狀態。舉例如下：

Forming Quiz teams (UVa 10911)
現在有 2 * n 個人，第 i 個人與第 j 個人配對的花費是 C_{ij}。請將這 2 * n 個人配成 n 對(每兩個人一對)，使得所有對的花費和盡量小。

這題雖然屬於圖論的帶權匹配，但是在問題範圍不大的情況下，可以用 DP 解決，複雜度是 $O(n * 2^n)$。

打地鼠 (TIOJ 1014)
共有 n 個地鼠洞依序位在數線上 1, 2, ..., n 的位置，第 i 個地鼠洞的地鼠每 t_i 秒會出現一次。玩家一開始在原點，並隨時以 1 公尺/秒的速度移動。請問要打到所有地鼠最少一次，最少需要多久時間 ?

Traveling Salesperson Problem (TIOJ 1028, 旅遊規劃問題)
在一張帶權無向圖 G = (V, E)當中，請問從節點 s 出發，經過指定的節點(不限順序)$v_1, v_2, ..., v_k$ 的最短路徑長為何 ? 每條邊、每個點經過的次數不限。



晶片放置問題 (TIOJ 1468, Adapted from CEOI 2002)
你有許多 2x3 的晶片要塞在一個 MxN 的矩形晶圓中，但晶片不能放在標示黑色的位置。請問最多能放幾塊 ?

五、優化

對於一個 DP 算法，最基本的時間複雜度是 O(狀態數 * 狀態轉移的花費)。但是當該算法的時間效率不如我們要求時，可以考慮從幾個方面著手：

1.重新設計狀態：因為狀態數是影響 DP 效率最基本的因素
2.用單調性/四邊形不等式優化：透過單調性減少決策的花費。
3.在某些特殊的 DP 當中，還可以考慮使用矩陣加速狀態轉移。

國王烏龜的接駁車 (TIOJ 1623, Adapted from USACO 3.4, Raucous Rockers)
請在序列$<A_1, A_2, ..., A_n>$當中選出 k 段連續的、不重疊的段落，使得被選出的數字盡量多且每一段的數字和皆不超過 T。

最佳二元搜尋樹 (UVa 10304)
假設 root[i, j]代表 e[i...j]所建構出的最佳二元搜尋樹的根，Knuth 證明了對於任意 $1 \leqq i < j \leqq n$，我們皆有 $root[i, j-1] \leqq root[i, j] \leqq root[i+1, j]$。請將最佳二元搜尋樹 DP 的複雜度由 $O(n^3)$降到 $O(n^2)$。

山景 Skyline (TIOJ 1471, CSAPC'08)
山稜線的每一個片段都是 45 度向上或 45 度向下，且滿足 1.山峰從左到右的高度非遞減 2.山谷的高度不得低於地平線 3.左右兩端都是高度為 0 的山腳。請問一個長度為 n (2|n)的山稜線有幾種 ? ($n \leqq 3,000$)

腹黑、傲嬌 (NPSC '09 高中組初賽 pA)
定義 $A_i \equiv aA_{i-1} + bB_{i-1} (\text{mod } M)$，$B_i \equiv cA_{i-1} + dB_{i-1} (\text{mod } M)$，且 $A_0 = B_0 = 1$，請在 $O(\log n)$的時間內求出 A_n、B_n ($0 < n \leqq 10^7$)。

Chomiki (POI 17th Stage II, also NTUJ 0873)
對於給定的字串集合 $S = \{s_1, s_2, ..., s_n\}$，請求出最短的字串 s，使得 s 的子字串中最少有 M 個屬於 S。保證 $s_1, s_2, ..., s_n$ 當中任一個字串都不會是其他字串的子字串，且總長度不超過 10^5。

※更多資料請參考：《动态规划算法的优化技巧》(福州第三中學 毛子青)
：《动态规划加速原理之四边形不等式》(華中師大一附中 趙爽)

Greedy Method 貪婪法

Greedy Method(簡稱 Greedy)和 Dynamic Programming(DP)一樣，是屬於一種解題的策略。也就是說，他並非用來指稱一種特定的演算法，而是用來描述某一類的演算法。這一類的演算法都有一個特徵：在做選擇時，都盡可能地符合某個條件。要證明 Greedy 法是對的並不是一件容易的事情，相似的問題常常在數字變了之後就無法使用 Greedy，例如註明的換零錢問題：

> 把 n 個 1 元硬幣兌換成 50 元、10 元、5 元、1 元的硬幣，請問要如何換才能讓總硬幣數最少？

在這個例子裡，我們可以很自然地使用 Greedy：先盡可能地換成最大的，接著是次大的……這樣一路換到最小的，所得到的就會是最佳解。但如果硬幣的面額是 25 元、18 元、5 元、1 元，Greedy 找出來的就不會是最佳解。舉例來說：n = 41 時，Greedy 算出來的答案會是 25 元 1 個，18 元 0 個，5 元 3 個，1 元 1 個，但實際上的最佳解是 25 元 0 個，18 元 2 個，5 元 1 個，1 元 0 個。

我們無法一概而論究竟什麼時候可以使用 Greedy Method，但如果我們要使用 Greedy Method，就必須要有辦法說明他爲什麼對，但這往往是最困難的部分，就像人類的直覺(大多時候也是一種 Greedy)，你常常很難能去說清楚直覺到底是對的還是錯的。

以下我們講述兩個 Greedy Method 可解的問題。

Task Selecting Problem

Problem E

合作社叔叔傷腦筋

合作社叔叔一直都很忙碌，不但要幫同學尋找更多不同口味的巧克力，還要常常上 Plurk 和同學們互動。最近他遇到了一個困難：事情實在太多了，有些事情必須要請阿姨幫他處理。但是合作社叔叔還是很希望自己能盡量多做一點事情，因此他想請你這位最有潛力的程式設計師，幫他寫一支程式，決定哪些事情是他可以自己來的。他會告訴你每件事情必須要開始的時間點以及結束的時間點，請你幫他選出盡量多的事情，好讓他可以把剩下的工作分給阿姨。如果你能幫他寫出這支程式，說不定他下次就會送一盒巧克力給你喔！

Input

輸入只包含一筆測試資料，測試資料的第一行有一個整數 n (1<= n <=100000)，接下來 n 行每行有兩個 4 bytes 正整數 a_i, b_i($a_i < b_i$)，a_i 代表第 i 件任務開始的時間，b_i 代表第 i 件事情結束的時間。爲了方便你處理，叔叔已經把這些事情照 a_i 由小到大排好了，並且他告訴你，不會有 任何任務開始和結束的時間在另外一件任務中，也就是對於所有(a_i, b_i)，不存在(a_j, b_j)使得 $a_i < a_j < b_j < b_i$。你可以假設所有 a_i 都不相等，bi 亦同。

註：如果 b_i=a_j，不算重疊，也就是實際上事情所佔的時間區間爲[a_i,b_i)。

Output

輸出只有一個正整數，代表叔叔最多能自己做的事情數量。

Sample Input

```
5
1 3
2 4
3 8
5 10
8 11
```

Output for Sample Input

```
3
```

還記得這個熟悉的問題嗎？沒錯！這就是我們初賽的題目。這個題目大部分的人都非常直覺地用 Greedy 來解，但是爲什麼 Greedy 在這是對的呢？

我們先不要告訴大家爲什麼 Greedy 是對的，我們要反問大家，除了 Greedy 之外，還有什麼方法可以解呢？上個星期講了什麼？沒錯！這個題目也是一個 DP 可解的問題。DP 的遞迴式如下：

$$s(i) = \begin{cases} 1 & \text{if} \quad i == 1 \\ \max(\ s(i-1),\ 1 + s(pi)\ ) & \text{else} \end{cases}$$

s(i)代表最後一個任務可能是第 i 個任務的最佳解，pi 代表和最靠近第 i 個任務，但不和他重疊的那個個任務。

這個方法就是每新加入一個任務，我們就考慮看看到底要不要挑他，也就是考慮如果要做他，必須犧牲掉的任務數有沒有大於 1，如果沒有就挑，如果有就不挑。聰明的你仔細想想就會發現：如果犧牲掉的任務數是 1，那麼還不如不要挑的好，因爲這個新加的任務一定和之後的任務重疊的機會更高。如果犧牲掉的任務數是

0，那麼就絕對可以挑。在這樣的情況下，我們是不是就可以每次都挑當前剩下來的任務中，沒有跟之前任務重疊的最早的那個？也就是我們可以每次都從任務1開始挑，接著拿掉所有跟它有重疊的任務，再拿剩下來的任務中的第一個任務。重複此過程，直到全部的任務都考慮過爲止。這樣就可以找出最佳解了！有沒有很神奇？

* 想想看：如果今天給你的任務清單，(1) 沒有按照時間順序 (2)有 ai < bi < bj < aj 的情況，該怎麼辦？

Greedy + 二分搜尋法 = 暴力破解

有時候我們會遇到一些問題只問你最後的數字，不問你中間到底是怎麼解他的。這個時候我們就可以換一種角度來解決問題：我們先尋找可行的解，再尋找最佳解。這種問題我們就可以利用二分搜尋，搜尋的目標是我們要的解，每次搜尋到一個可能可行的解時，我們就用 Greedy 試試看這個解是不是可行，如果可以我們就將正在搜尋的目標解縮小試試看，如果不行就將正在搜尋的目標解放大。以裝東西問題爲例子：

> 給你許多物品的重量，物品編號從 1 到 n，現在我們要用 M 個箱子把所有東西裝起來，但有個前提是：物品一定要依序放好，同個箱子內的物品不能跳號。我們希望買來的所有箱子都是一樣大的，所以我們要先知道該買多大的箱子，現在要請你幫我們找出可用的最小箱子大小。

這個問題我們就可以以最大的物品重量做爲搜尋左界，總物品重量做爲搜尋右界，每次都取中間值 mid 做爲嘗試的大小。如果發現無法用 M 個大小爲 mid 的箱子裝下他，就要把搜尋左界變成 mid+1，再去做下一輪嘗試。如果發現可以裝下他，甚至不用到 M 個箱子就裝的下，那就要把搜尋右界變成 mid-1。重複此過程直到二分搜尋到底，即可找出最佳解。

練習題

714 - Copying Books

907 - Winterim Backpacking Trip

11413 - Fill the Containers

二分搜尋法 Binary Search

簡介

基礎的搜尋法有兩種：循序搜尋與二分搜尋。前者非常直覺，從頭到尾檢視每個元素，看看是否有我們要的元素。後者則是因為「二分搜尋」的特性，所以必須要在已排序的陣列才可使用。兩者最大的差異是搜尋的時間，如果整個程式之中必須要執行多次搜尋，那麼二分搜尋會比循序搜尋快上許多。

二分搜尋

在已排序好的陣列上(假設是由小到大排序)，我們可以確定：左邊的元素一定小於右邊的元素。也就是說，假設我們要找的是 4，陣列第五個位置的元素是 6，哪麼我們要找的元素一定在前四個位置(或是不存在)。

一般來說，我們使用二分搜尋法，都是從數列的最中間開始搜尋，如果要找的元素比最中間的元素小，那麼右邊那一半的元素都不需考慮，並且將左邊一半的元素視為新的數列，再將其最中間的元素和我們要尋找的元素進行比對，留下可能存在解的那一半。重複這個過程，直到找到我們要的元素或是數列只剩一個數字但卻又非我們所要的元素時，搜尋就結束了。

實作

我們通常會使用 while()迴圈來進行二分搜尋，並且宣告三個變數：left、right、mid，分別代表數列最左界的元素位置、最右界的元素位置以及最中間的元素位置。假設現有一陣列 array[]，內有 N 個元素，並且我們要找的元素叫做 element，那麼二分搜尋要這樣寫：

```
    left = 0;
    right = N - 1;
while( left <= right )
    {
        mid = (left + right) / 2;   /*  記得括號!  */
        if( array[mid] == element)
        {
            printf(“%d is at array[%d]\n”, element, mid);
            break; /*  array[mid] is what we want!  */
        }
        else if( array[mid] < element )
        {
            left = mid + 1;
        }
        else if( element < array[mid] )
        {
            right = mid - 1;
        }
    }
```

這樣最後 mid 為可行解，或者是最接近 element 的數值(可能大可能小)。