如何設計遞迴函式

Pei-yih Ting

遞迴函式

- 遞迴 (recursive) 函式是自己呼叫自己的函式
  - 在呼叫自己之前一定有 if-else 一類的選擇判斷敘述, 以避免無窮盡的遞迴呼叫
  - 函式中至少有一條控制路徑不包含遞迴呼叫, 稱為 base case
  - 函式一定有輸入參數, 用來指定所解決問題的規模

例如:

```
int sum(int n) {
    if (n==0)
        return  0;
    else
        return sum(n-1) + n;
}
```

1+2+…+n

- 運用遞迴方式撰寫程式的好處:

  對於適合用遞迴方式解決的問題來說

  ① 程式碼的表達性比較高, 容易看懂, 容易偵錯

  ② 程式比較精簡, 不需要額外實作堆疊來輔助運算

遞迴函式 (cont'd)

Recursive structure

- 適合用遞迴方式解答的問題特徵
  - 一個大問題的解可以藉由一或多個子問題的解來合成出來; 問題本身能夠用分治法 (divide-and-conquer, 各個擊破) 解而且解答的形式一致的; 另外問題的拆解必須是有效率的
  - 配合樹狀資料結構的處理函式 (包含樹狀結構的搜尋), 尤其是需要運用堆疊來輔助解決的問題
- 遞迴函式的缺點:

需要藉由 memoization 來加速

  - 執行時需要比較多的記憶體 (系統堆疊), 只有在所解決的問題大小不會用完系統堆疊時程式才能正確運作; 一個簡單的迴圈需要使用 O(1) 的常數記憶體, 寫成等效的遞迴函式在執行時會用掉 O(n) 的系統堆疊空間
  - 執行時有可能需要很多的時間: 例如一個計算第 n 個 Fibonacci 數列元素的遞迴程式需要 $O(2^n)$ 的運算時間, 相對地一個迴圈版本的程式 (動態規劃) 只需要 O(n) 的運算時間

遞迴函式 (cont'd)

- 遞迴是一種重複 (repetition) 的程式架構
  - 迴圈程式架構可以轉成遞迴函式
  - 可以用遞迴解的問題, 一定也可以用迴圈解, 至少可以自己用輔助的堆疊資料結構配合迴圈來完成, 不過有很多運用遞迴解很直覺的問題, 運用迴圈解需要特別的觀察以及演算法
- 交互遞迴 (mutually recursive)
  - 函式 f() 呼叫函式 g(), g() 再呼叫 f() 也是一種遞迴, f() 間接地呼叫 f() 自己
- 尾端遞迴 (tail-recursive)
  - 每一個遞迴呼叫完成後沒有任何其它動作, 只回傳該數值
    例如: return tail_rec_fun(p1, p2);
    下列三個例子都不是
    ① return rec_fun(p1, p2) + 10;
    ② r = rec_fun(p1, p2); printf("%d\n", p1); return r;

遞迴函式 (cont'd)

③ rec_func(p1,p2); return rec_fun(p2,p1);

- 尾端遞迴的函式在設計時不需要特別注意遞迴函式呼叫的深度, 很多編譯器都可以最佳化程式碼 (VC: /O1 /O2), 概念上在呼叫下一層遞迴函式時除了更換呼叫引數的內容外, return address 以及區域變數所用到的空間是完全一樣的, return address 的內容也是完全不變的, 呼叫的時候沿用原先的堆疊框就可以, 不需要像呼叫一般的函式一樣建立新的堆疊框, 維持 return address 不變相當於最後處理到 base case 時 return 一次就回到最上層的呼叫端, 請注意最佳化程式並不會把尾端遞迴直接換成迴圈
- 尾端遞迴函式有特別的設計方法 (對相同問題來說, 參數比一般遞迴函式多)、與問題拆解方法, 和一般遞迴函式直覺的表示方法有一些差別, 不過有時候為了解除遞迴深度對程式設計的限制, 反而犧牲掉了遞迴函式精簡易讀的特性

遞迴函式的設計方法

- 基本步驟

a. 定義出遞迴函式以及其參數 (遞迴函式一定需要)
b. 想清楚這個遞迴函式在某一參數時能夠解的問題是什麼
c. 把需要解決的問題拆解為較小的問題
d. 呼叫遞迴函式解決這個小問題, 並且用這個答案組合出原來問題的答案
e. 問題縮到最小時 (base case) 可以直接寫出答案

- 例如: 計算陣列 data[] 裡 n 個元素 data[0]~data[n-1] 的總和

a. int sum(int data[], int n)
b. 這個函式能夠計算並回傳 data[0]+data[1]+…+data[n-1]
c. 拆解為小一點的問題: n-1 個元素的總和
d. sum(data, n-1) 可以算出 data[0]+…+data[n-2], 所以 sum(data, n) 的結果應該是 sum(data, n-1) + data[n-1]
e. sum(data, 0) 的結果是 0

```
int arraySum01(int data[], int n) {
    if (n==0)
        return 0;
    else
        return arraySum01(data, n-1) + data[n-1];
}
```

- 前面這個範例裡，呼叫 sum(data, n) 以後會依序呼叫 sum(data, n-1), sum(data, n-2), ..., sum(data, 0) 共 n+1 次遞迴函式呼叫，遞迴深度 n+1，才能夠計算出答案
- 在撰寫遞迴函式時，前面這樣的問題拆解方式是比較沒有效率的，需要 O(n) 次的函式呼叫，在許可的情況下應該**盡量尋找遞迴深度淺一些的問題拆解方法**，例如在計算陣列 data[] 裡 n 個元素 data[0]~data[n-1] 的總和時，可以考慮拆成下面兩個子問題：計算 **data[0]~data[(n-1)/2]** 的總和以及
  計算 **data[(n+1)/2]~data[n-1]** 的總和
  最後把兩者加總起來; 這個例子裡**遞迴函式呼叫的次數**沒有減少, 更理想的狀況是函數呼叫的次數也盡量降低

- a. int sum(int data[], int istart, int iend)
  b. 這個函式能夠計算並回傳 **data[istart]+…+data[iend]**
  c. 拆解為兩個小一點的問題:
     計算 (iend-istart+2)/2 及 (iend-istart+1)/2 個元素的總和
     sum(data, istart, istart+(iend-istart)/2) 算出
       data[istart]+data[istart+1]+…+data[istart+(iend-istart)/2]
     sum(data, istart+(iend-istart)/2+1, iend) 算出
       data[istart+(iend-istart)/2+1]+…+data[iend]
  d. 因此 sum(data, istart, iend) 的結果應該是上述兩者之和
  e. sum(data, i, i) 的結果是 data[i]

```
int arraySum02(int data[], int istart, int iend) {
    if (istart==iend)
        return data[istart];
    else
        return arraySum02(data, istart, istart+(iend-istart)/2) +
                   arraySum02(data, istart+(iend-istart)/2+1, iend);
}
```

- 函式 int sum(int data[], int istart, int iend) 有三個參數，目的是希望函式裡處理 data[istart]~data[iend] 這 iend-istart+1 筆整數資料，data[0]~data[istart-1]是沒有用到的，所以前兩個參數其實可以合併在一起，只傳遞 data[istart] 的位址當作是陣列的起始記憶體位址，可以更進一步**簡化為兩個參數的版本**：**int sum(int *startData, int ndata)**，其中 startData 存放 data[istart] 陣列元素的記憶體位址，代表由 &data[istart] 開始的後半段資料陣列，ndata 則為資料的筆數，遞迴呼叫時這些參數會保存在堆疊框上，少一個參數也降低一點點需要的堆疊記憶體，同時程式的描述會比較簡潔和直覺一些

```
int arraySum03(int data[], int n) {        int *data
    if (n==0)
        return 0;
    else if (n==1)
        return data[0];
    else
        return arraySum03(data, (n+1)/2) +
                 arraySum03(&data[(n+1)/2], n/2);
}
```

Fibonacci

- a. int fibonacci(int n)
  b. **這個函式計算並回傳 f(n)=f(n-1)+f(n-2), f(1)=f(2)=1**
  c. 拆解為兩個小一點的問題: 計算 f(n-1) 與 f(n-2)
  d. 由 fibonacci(n-1) 及 fibonacci(n-2) 合成出 fibonacci(n) 的結果，也就是 fibonacci(n)=fibonacci(n-2)+fibonacci(n-1)
  e. fibonacci(2) 以及 fibonacci(1) 都是 1

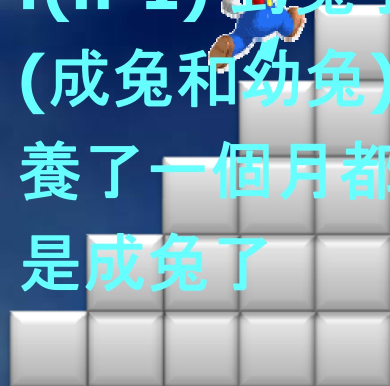
```
int fibonacci(int n) {
    if ((n==2)||(n==1))
        return 1;
    else
        return fibonacci(n-2) + fibonacci(n-1);
}
```

- 這樣子設計出來的程式很簡潔，但是執行起來需要呼叫 fibonacci 函式 O(2^n) 次，其中很多次的參數是一樣的，所以執行效率很不好，如果配合使用額外的陣列的話，就很有效率了, memoization，如果用陣列配合迴圈來寫的話… dynamic programming

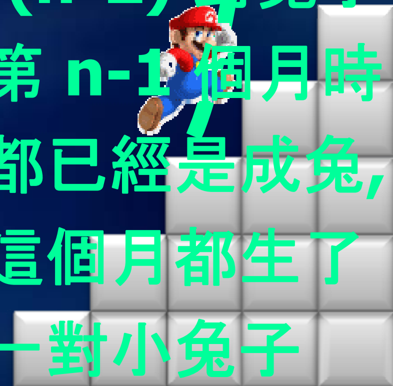
Fibonacci Applications

- Rabbit breeding $f(n) = f(n-1) + f(n-2)$
- Stair climbing
- Rays through glasses

第 1 個月買了一對幼兔
第 2 個月長大為成兔
第 3 個月生下一對幼兔
...



第 n-1 個月有 f(n-1) 對兔子 (成兔和幼兔) 養了一個月都是成兔了



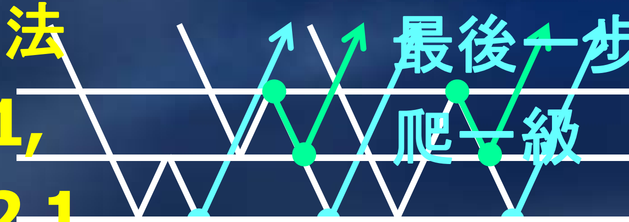
第 n-2 個月有 f(n-2) 對兔子 第 n-1 個月時都已經是成兔, 這個月都生了一對小兔子

Mario 每一步可以爬一級或兩級, 爬到最上面總共有多少種方法

$f(5) = f(4) + f(3)$

1 1 1 1 1, 1 1 1 2, 1 1 2 1, ..., 2 1 1 1, 2 1 2, 2 2 1

最後一步爬一級 $f(3) = f(2) + f(1)$ 最後一步爬二級



reflect once

$f(0)$ $f(1)$ $f(2) = f(1) + f(0)$

Subset Sum

- a. int subsetSum(int data[], int n, int target)
  b. **判斷是否 data[0]~data[n-1] 中某些數字的和是 target**
  c. 拆解為兩個小一點的問題: data[n-1] 不在內, data[0]~data[n-2] 中某些數字的和是 target; data[n-1] 在內, data[0]~data[n-2] 中某些數字的和是 target-data[n-1]
  d. data[n-1] 在或不在集合內這兩種狀況中只要有一種可以找到答案, target 就可以表示成這 n 個數字的子集合的和, 所以是 || 兩個結果
  e. target 如果是 0 當然空集合就是答案; n 如果是 0 且 target != 0 則一定沒有答案

```
int subsetSum(int data[], int n, int target) {
    if (target==0) return 1;
    else if (n==0) return 0;
    return subsetSum(data, n-1, target) ||
                subsetSum(data, n-1, target-data[n-1]);
}
```

- 程式執行起來需要呼叫 subsetSum 函式 $O(2^n)$ 次，雖然概念表達的很清楚, 但是執行效率很不好 (本來就是 NP-complete 的問題)

GCD and LCM

- a. int gcd(int m, int n) // precondition: m>=n>0
  b. 這個函式計算並回傳 m 與 n 的最大公因數
  c. 拆解為小一點的問題: gcd(n, m%n)
  d. gcd(m, n) = gcd(n, m%n)
  e. 如 m 為 n 的整數倍, gcd(m, n) = n

```
// assume m >= n > 0
int gcd(int m, int n) {
    int ans;
    if (m % n == 0)
        ans = n;
    else
        ans = gcd(n, m % n);
    return ans;
}
```

```
int lcm(int m, int n) {
    return m * n / gcd(m, n);
}
```

二分搜尋法 (Binary Search)

- 假設資料陣列 data[] 中的資料已經由小到大依序排列

  a. int binarySearch(int data[], int start, int end, int target)

  b. 這個函式在整數陣列元素 data[start] 到元素 data[end] 之間尋找 target, 找到了回傳 target 在陣列中的位置, 找不到回傳 -1

  c. 拆解為小一點的問題: 原來是在 start 到 end 之間, 拆成更小的區間

  d. center=(start+end)/2, 如果 target 比 data[center] 大, 就在 [center+1,end] 區間中尋找, 反之在 [start,center-1] 間尋找

  e. 如果 target == data[center] 就找到了

  如果 end < start 就沒有任何元素在區間裡, 一定找不到

```
int binarySearch(int data[], int start, int end, int target) {
   int center = (start+end)/2;
   if (start>end) return -1;
   if (target == data[center]) return center;
   else if (target > data[center])
      return binarySearch(data, center+1, end, target);
   else
      return binarySearch(data, start, center-1, target);
}
```

Tail-recursive

最大乘積的整數拆解方法

- 一個正整數 n 可以分解成較小正整數的和, $n = a_1+a_2+\ldots+a_k$, 請寫一個程式找到一種分解方法能夠使得乘積 $a_1a_2\ldots a_k$ 最大
- 遞迴設計
  - a. int findMaxProd(int n)
  - b. 這個函式計算並回傳最大乘積 $a_1a_2\ldots a_k$ 滿足 $n = a_1+a_2+\ldots+a_k$
  - c. 拆解為兩個小一點的問題:
    - ① $n_1=a_1+\ldots+a_k$, 最大乘積 $a_1\ldots a_k$
    - ② $n_2=b_1+\ldots+b_\ell$, 最大乘積 $b_1\ldots b_\ell$

    滿足 $n=n_1+n_2$, 在這種分解法下最大乘積為 $a_1a_2\ldots a_k\, b_1b_2\ldots b_\ell$
  - d. n 不拆開的話, 乘積就是 n, 將 n 拆為 n_1+n_2 的方法總共有 n/2 種, 在這些分解方法下找到乘積的最大值就是我們的答案
  - e. n <= 4 時不拆解就可以得到最大值, 也就是 n

```
int findMaxProd(int n) {
    int i, max=n, prod;
    if (n<=4) return n;
    for (i=1; i<=n/2; i++) {
        prod = findMaxProd(i) *
                   findMaxProd(n-i));
        if (prod > max) max = prod;
    }
    return max;
}
```

- 這個程式很短很容易了解, 但是最大的問題就在效率, 你可以想像呼叫 findMaxProd(20) 時, findMaxProd(5) 被呼叫了16384 次嗎?

最大乘積的整數拆解方法 (cont'd)

- 該怎麼辦? 又需要 **memoization** 了, 每個**正整數至少有一種最大乘積的拆解方法**, 走過就要留下痕跡, 算過就要記錄下來, 不要浪費時間和能量重複計算相同的東西
- 就像是費氏序列或是組合數的計算一樣, 運用**遞迴/陣列**, 或用**迴圈/陣列**來設計, 是標準的動態規劃最佳化方法
- 將每個正整數拆解開的最大乘積記錄在陣列 **P(i)** 裡:

i	1	2	3	4	5	6	7	8	9	10	11	
P(i)	1	2	3	4	6	9	12	18	27	36	54	...
x*	0	0	0	0	2	3	2	2	3	2	2	...

$$P(i) = \max_{1 \le x \le i} \{ x\, P(i-x) \}$$

$$x^* = \operatorname{argmax}_{1 \le x \le i} \{ x\, P(i-x) \}$$

P(11) = 54 = 2 * 3 * 3 * 3

i	1	2	3	4	5	6	7	8	9	10	11	
P(i)	1	2	3	4	6	9	12	18	27	36	54	. . .
x*	0	0	0	0	2	3	2	2	3	2	2	. . .

$$P(i) = \max_{1 \le x \le i} \{ x\, P(i-x) \}$$

$$x^* = \operatorname{argmax}_{1 \le x \le i} \{ x\, P(i-x) \}$$

- 陣列

```
int P[12]={1,2,3,4}, xstar[12]={0};
int i, j, n=11;
```

- 迴圈 … 順向DP

```
for (i=5; i<=n; i++)
    for (P[i]=1*P[i-1],xstar[i]=1,j=2; j<i; j++)
        if (j*P[i-j]>P[i])
            P[i] = j*P[i-j], xstar[i] = j;
```

- 遞迴 … 逆向DP

這個遞迴和前面的遞迴不同

```
int maxProd(int P[], int xstar[], int i) {
    if (P[n]>0) return P[n];
    int j, tmp;
    for (P[i]=1*maxProd(P, xstar, i-1),
            xstar[i]=1,j=2; j<i; j++)
        if (j*(tmp=maxProd(P, xstar, i-j))>P[i])
            P[i] = j*tmp, xstar[i] = j;
}
```

- Rod cutting, String Partition, Unbounded Knapsack, …

迴文判別 Palindrome

- "A man, a plan, a canal, Panama!" "Amor, Roma" "race car" "taco cat" "Was it a car or a cat I saw?" "No 'x' in Nixon"
- 一個字串去掉標點符號和空白以後由前面看和後面看一模一樣
- a. int isPalindrome(int len, char buf[]) //已去除標點符號和空白
  b. 這個函式檢查字元陣列 **buf[]** 中的字串是否為迴文
  c. 拆解為長度短一點的問題: 檢查去除第一個字元和最後一個字元以後的長度 len-2 字串是否為迴文
  d. 第一個字和最後一個字相同且 isPalindrome(len-2,buf+1)為真則原字串是一個迴文
  e. 長度為 1 或是 0 的字串都是迴文

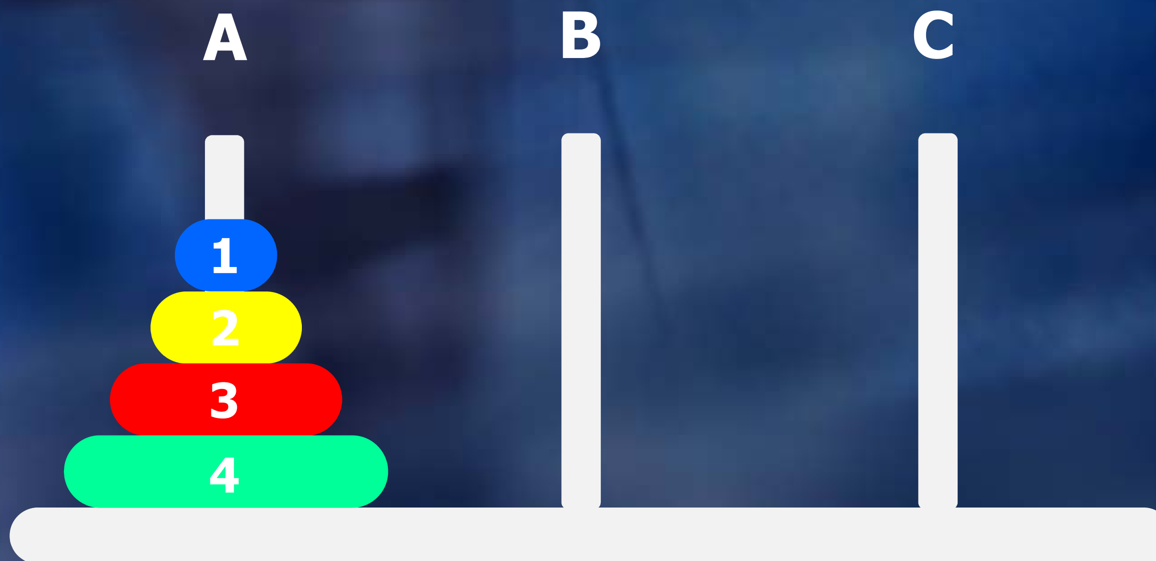
The Sator Square

S	A	T	O	R
A	R	E	P	O
T	E	N	E	T
O	P	E	R	A
R	O	T	A	S

```
int isPalindrome(int len, char buf[]) {
    if (len <= 1)
        return 1;
    if (buf[0] != buf[len-1])
        return 0;
    else
        return isPalindrome(len-2, buf+1);
}
```

河內塔 Towers of Hanoi

- 將一疊由小到大的的盤子由一個柱子上移到另外一個柱子上，並且維持原來的順序：
  - 每一個步驟只能移動一個盤子
  - 任何時候，大的盤子不可以放在比它小的盤子之上
  - 有一個輔助的柱子可以使用



程式輸出

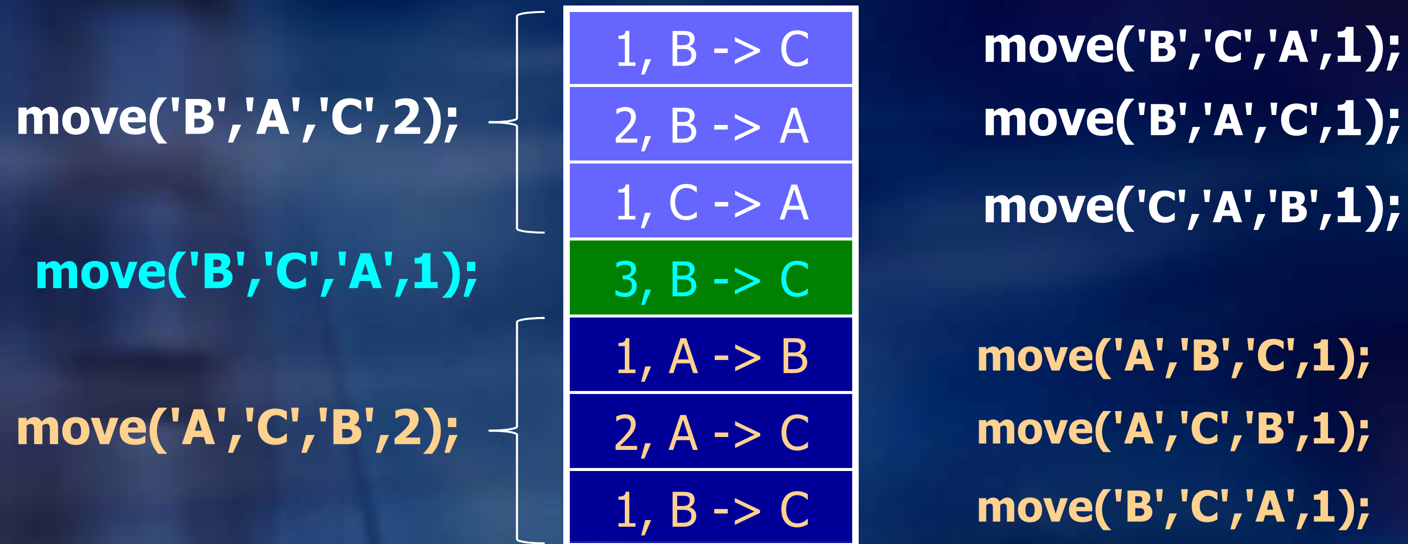
1, A -> B
2, A -> C
1, B -> C
3, A -> B
1, C -> A
2, C -> B
1, A -> B
4, A -> C
1, B -> C
2, B -> A
1, C -> A
3, B -> C
1, A -> B
2, A -> C
1, B -> C

- a. void move(char from_peg, char to_peg, char aux_peg, int nDisks)
  b. 印出由 from_peg 柱子搬 nDisks 個盤子到 to_peg 柱子的步驟, 過程中以 aux_peg 柱子為輔助, 避免大盤子在小盤子之上
  c. 拆解為三個小一點的問題: 2 個 nDisks-1 以及 1 個直接移動
  d. 由子問題的解合成整個問題的解:
  運作條件: aux_peg 柱子上如果有盤子的話, 需要大於 from_peg 柱子上最上面的 nDisks-1 個盤子, to_peg 柱子上如果有盤子的話,需要大於 from_peg 柱子上最上面的 nDisks 個盤子
  ① 由 from_peg 柱子搬 nDisks-1 個盤子到 aux_peg 柱子
  ② 由 from_peg 柱子搬 1 個盤子到 to_peg 柱子
  ③ 由 aux_peg 柱子搬 nDisks-1 個盤子到 to_peg 柱子
  假設原來 to_peg 柱子以及 aux_peg 柱子上沒有比 from_peg 柱子最上面算起第 nDisks 個盤子小的盤子, 所以可以用上面的三個步驟來完成; 接下來考慮過程中兩次移動 nDisks-1 個盤子, 在三個柱子上也沒有比這 nDisks-1 個盤子小的, 所以可以再用上面的三個步驟來完成
  e. nDisks=1 時: to_peg 柱子上的盤子比 from_peg 柱子上第一個盤子大的話, 直接搬移

程式輸出

Move top 3 disks from peg B to peg C using peg A as auxiliary peg

move('B', 'C', 'A', 3);



Tower of Hanoi (cont'd)

```
01 void move(char from_peg,  /* input - characters naming         */
02               char to_peg,     /*           the problem's            */
03               char aux_peg,    /*           three pegs               */
04               int  nDisks)  {  /* input - number of disks to move */
05    if (nDisks == 1) {
06       printf("Move disk 1 from peg %c to peg %c\n", from_peg, to_peg);
07    } else {
08       move(from_peg, aux_peg, to_peg, nDisks - 1);
09       printf("Move disk %d from peg %c to peg %c\n", nDisks, from_peg, to_peg);
10       move(aux_peg, to_peg, from_peg, nDisks - 1);
11    }
12 }
```

請注意這樣子移動的次數一定是最少的, 如果是只有一個碟子時, 只要移動 a_1 = 1 次, 如果只有兩個碟子時, 只要移動 a_2 = **3** 次, **...**, 如果有 **n** 個碟子時, 你知道最大的碟子至少要移動 **1** 次, 但是在移動它之前, 要把上面 **n-1** 個碟子都先移到輔助的柱子上 (由 **from_peg** 到 **aux_peg**), 那麼最少要移動 a_{n-1} 次, 然後移動最大的碟子 (由 **from_peg** 到 **to_peg**), 然後再把輔助的柱子上面的 **n-1** 個碟子移到目標柱子上, 又需要 a_{n-1} 次, 總共需要移動 $a_n = 2 a_{n-1} + 1$ 次

Recursively Counting Upward

```
int main() {
    int data[10];
    countUp(4, data, 0);
    return 0;
}
void countUp(int ndata, int data[], int pivot) {
    if (pivot==ndata)
        print(ndata, data);
    else
        for (data[pivot]=1; data[pivot]<=ndata; data[pivot]++)
            countUp(ndata, data, pivot+1);
}
void print(int ndata, int data[]) {
    printf("%d", data[0]); // ndata>0
    for (int i=1; i<ndata; i++)
        printf(" %d", data[i]); printf("\n");
}
```

```
int main() {
    int data[4], ndata=4;
    for (i=0; i<ndata; i++)
        data[i] = 1;
    for (; data[0]<=ndata; next(ndata, data))
        print(ndata, data);
    return 0;
}
void next(int ndata, int data[]) {
    int i;
    for (i=ndata-1; i>0; i--)
        if (data[i]<n)
            { data[i]++; return; }
        else
            data[i] = 1;
    data[0]++;
}
```

a b c d e

data[pivot]~data[ndata-1]

1 1 1 1
1 1 1 2
1 1 1 3
1 1 1 4
1 1 2 1
…
1 1 4 4
1 2 1 1
1 2 1 2
1 2 1 3
1 2 1 4
1 2 2 1
1 2 4 4
1 3 1 1
…
2 1 1 1
…
4 4 4 3
4 4 4 4

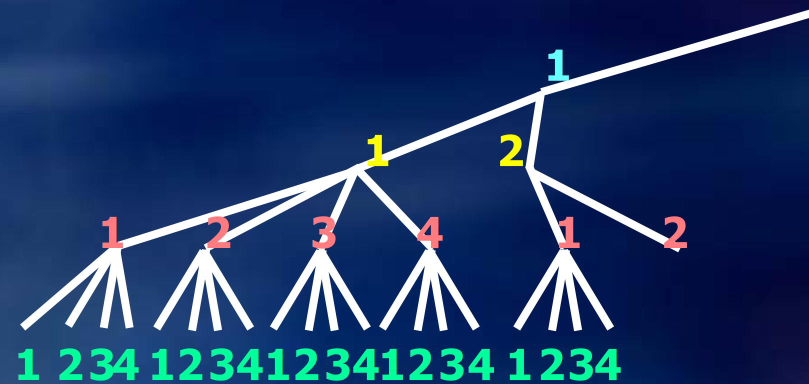
countUp(0)

countUp(1)

countUp(2)

countUp(3)

countUp(4)




```
void countUp (int ndata, int data[], int pivot ) {
    if ( pivot==ndata  )
        print(ndata, data);
    else
        for (data[pivot]=1 ; data[pivot]<=ndata; data[pivot]++ )
            countUp(ndata, data, pivot+1) ;
}
```

on and on ...

Some saving on the last level of recursive function call

```
void countUp(int ndata, int data[], int pivot) {
   for (data[pivot]=1; data[pivot]<=ndata; data[pivot]++)
      if (pivot<ndata-1)
         countUp(ndata, data, pivot+1);
      else
         print(ndata, data);
}
```

列印出 n 個數字的所有排列方法

- 問題拆解

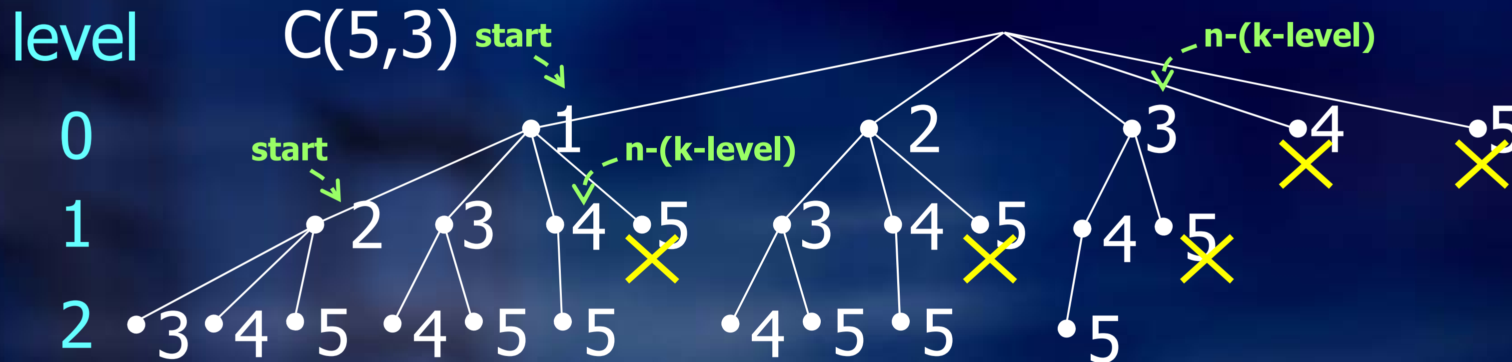
  1 || permutations of {2, 3, 4}
  2 || permutations of {1, 3, 4}
  3 || permutations of {2, 1, 4}
  4 || permutations of {2, 3, 1}

```
for (i=0; i<n; i++) a[i] = i+1;
permutation(a, 0, n-1);
```

```
void permutation(int perm[], int start, int end) {
    if (start == end) printPerm(perm, end+1);
    for (int i=start; i<=end; i++) {
        swap(&perm[start], &perm[i]);
        permutation(perm, start+1, end);
        swap(&perm[start], &perm[i]);
    }
}
```

1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 3 2
1 4 2 3
2 1 3 4
2 1 4 3
…
3 2 **1** 4
3 2 4 1
…
4 2 3 **1**
4 2 1 3
…
4 1 2 3

Recursive (n,k)-Combinations



- select level-k elements from seq[start]~seq[n-(k-level)] and put the result to result[level]~result[k-1]
- **comb(n,k,seq,0,0,result);**
- void **comb**(int n, int k, const int seq[], int start, int level, int result[]) {

```
    if (level==k) print(result, k); // result[0]~result[k-1]
    else for (int i=start; i<=n-(k-level); i++)
            result[level] = seq[i],
            comb(n, k, seq, i+1, level+1, result);
}
```

快速排序法 (Quick Sort)

- 範例: 9, 5, 12, 19, 2, 6, 4, 15, 8, 3, 7
- 目標: 2, 3, 4, 5, 6, 7, 8, 9, 12, 15, 19
- 遞迴函式設計:
  - 分治法 (Divide and Conquer): 切割問題, 各個擊破
  - 每一步驟挑選一個元素放在正確的位置上, 將其它數字分為兩群, 比這個元素大或是等於的放右邊, 小於的放左邊

    例如: 將第一個元素 9 放在正確位置上

{5, 2, 6, 4, 8, 3, 7}　　　　{12, 19, 15}

  - 如此就可以得到兩個元素個數比較少的排序問題
  - 問題: ①怎麼把 9 放在正確的位置上? 把 9 和其它數字一個一個比較就可以; ②怎樣才能夠有效率地分成兩群?

快速排序法 (cont'd)

istart

9, 5, 12, 19, 2, 6, 4, 15, 8, 3, 7

front　　iend　rear

步驟 1. while rear>istart &&
data[rear]>=data[istart]
rear--

9, 5, 12, 19, 2, 6, 4, 15, 8, 3, 7

front　rear

步驟 2. while front<iend &&
data[front]<data[istart]
front++

9, 5, 12, 19, 2, 6, 4, 15, 8, 3, 7

front　rear

步驟 3. if rear>front
交換 data[front++]及data[rear--]

9, 5, 7, 19, 2, 6, 4, 15, 8, 3, 12

front　rear

重複步驟 1 至步驟 3
直到 front > rear

(i.e. while (front < rear) { ... })

9, 5, 7, 3, 2, 6, 4, 15, 8, 19, 12

9, 5, 7, 3, 2, 6, 4, 8, 15, 19, 12

交換 data[istart] 及 data[rear],
以 data[rear] 為分割點
得到兩個長度比較短的排序問題

9, 5, 7, 3, 2, 6, 4, 8, 15, 19, 12

8, 5, 7, 3, 2, 6, 4, 9, 15, 19, 12

```
void quick_sort(int array[], int n) {
    if (n > 2) {
        int pivot = place_midst(array, n);
        quick_sort(array, pivot);
        quick_sort(&array[pivot+1], n - pivot - 1);
    }
    else if (n == 2)
        if (array[0]>array[1]) swap(array, 0, 1);
}
int place_midst(int array[], int n) { // 將 array[0] 放到正確位置 array[pivot]
    ...                               //而且比較小的放在前半, >= 的放在後半
}
void swap(int array[], int i, int j) { // 交換陣列元素 array[i] 及 array[j]
    ...
}
```

版本1

```
void quick_sort(int array[], int istart, int iend) {
    if (iend-istart > 1) {
        int pivot = place_midst(array, istart, iend);
        quick_sort(array, istart, pivot-1);
        quick_sort(array, pivot+1, iend);
    }
    else if (iend-istart == 1)
        if (array[istart]>array[iend]) swap(array, istart, iend);
}
int place_midst(int array[], int istart, int iend) { // 將 array[istart] 放到正確位置
    ...                                              // array[pivot], 而且比較小的放在
}                                                    // 前半, >= 的放在後半
```

版本2

快速選擇法 (Quick Select)

- 範例: 9, 5, 12, 19, 2, 6, 4, 15, 8, 3, 7
- 目標: **找到數列中第 k 小的數字**
- 遞迴函式設計:
  - Partition-based Selection algorithm
  - 每一步驟挑選一個元素放在正確的位置上, 將其它數字分為兩群, 比這個元素大的放左邊, 小的放右邊 (前面的 place_midst)

    例如: 將第一個元素 9 放在正確位置第 8 格內

							9			

{5, 2, 6, 4, 8, 3, 7}　　{12, 19, 15}

  - ❶ 如果 8 就是 k, 我們已經找到第 k 小的數字了

    ❷ 如果 8 比 k 大, 顯然第 k 小的數字在前半段

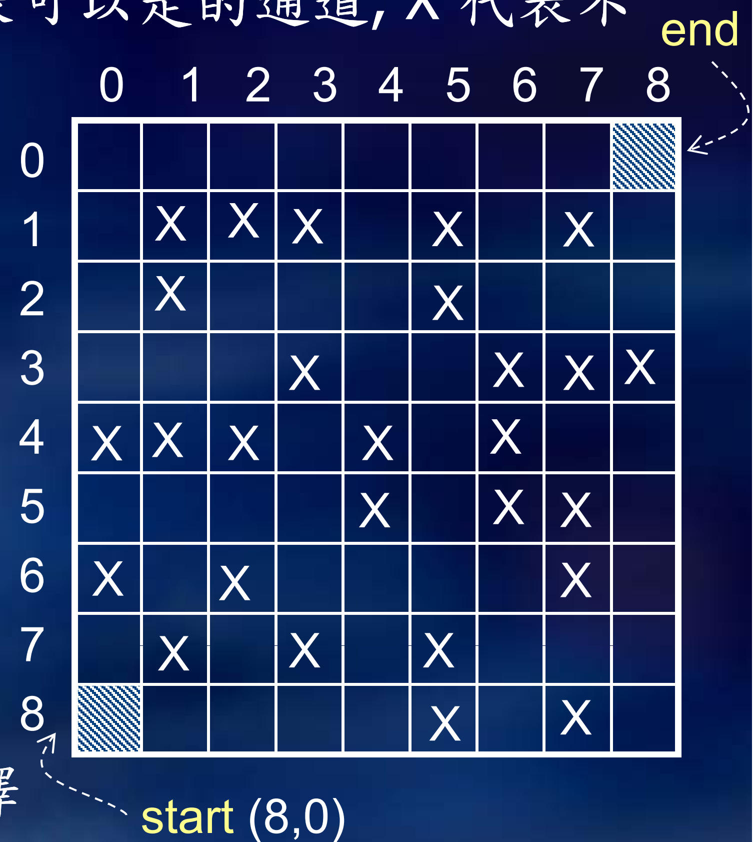
    ❸ 如果 8 比 k 小, 顯然第 k 小的數字在後半段

快速選擇法 (cont'd)

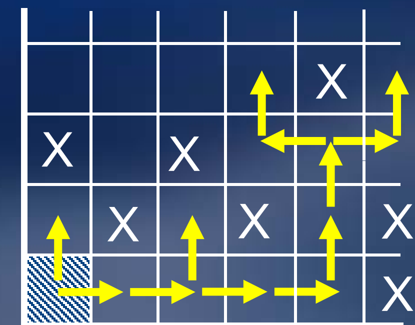
```
void quick_select(int kth, int array[], int istart, int iend) {
    if (iend-istart > 1) {
        int pivot = place_midst(array, istart, iend);
        if (pivot==kth-1)
            return pivot;
        else if (pivot>kth-1)
            return quick_select(kth, array, istart, pivot-1);
        else
            return quick_select(kth, array, pivot+1, iend);
    }
    else if (iend-istart == 1)
        if (array[istart]>array[iend]) swap(array, istart, iend);
    return kth-1;
}

int place_midst(int array[], int istart, int iend) { // 將 array[istart] 放到正確位置
    ...                                               // array[pivot], 而且比較小的
}                                                     // 放在前半, >= 的放在後半

void swap(int array[], int i, int j) { // 交換陣列元素 array[i] 及 array[j]
    ...
}
```

迷宮路徑搜尋

- 右圖二維陣列代表一個迷宮路徑，空格代表可以走的通道，X 代表不能走；由左下角座標 (8, 0) 的位置開始走，目標是左上角座標 (0, 8) 的位置。在每一個位置上，允許左上右下四個方向前進，除非無法前進了，否則不會後退，也不能走出邊界。



- 如右下圖，這是一個路徑搜尋的問題，一旦走到某一個位置，接下來最多有四種走法，程式需要一種一種嘗試，都無法前進的話 (有可能走出邊界，有可能是障礙物，有可能是由那個地方走過來的，也有可能先前已經經過了) 要倒退回去嘗試下一個選擇
- 我們會用**深度優先的搜尋方法** (Depth First Search, **DFS**)，遇見第一個可行的走法就一直走下去直到沒路為止，退回最接近而還沒有嘗試過的選擇，因此需要一個**堆疊**，記錄所有還沒有試過的選擇，由堆疊上拿出來的是最後放進去的



迷宮路徑搜尋 – 直覺的迴圈作法

```
int i = 0;                              // 0:左, 1:上, 2:右, 3:下
static const int dirs[][2] = {{-1,0}, {0,1}, {1,0}, {0,-1}};
while ((curX!=targetX) || (curY!=targetY)) { // 還沒走到目的地
    if (board[curX+dirs[i][0]][curY+dirs[i][1]]==-1) // 死路或是撞牆
        i = (i+1) % 4; // 下一個方向
    else
        curX += dirs[i][0], curY += dirs[i][1], i=0;
}
```

- 簡單的講這個程式的邏輯就是『**有路就往前走**』
- **問題**
  - ❶ 有路就一直往前走, 萬一繞圈圈怎麼偵測出來?
  - ❷ 四個方向順序嘗試, 需要避開先前走過來的路, 否則有機會變成不斷地來回走
  - ❸ 走到死路或是撞牆時, 該怎樣退回前面哪一個方向或是哪一格? 然後哪一個方向已經嘗試過, 該繼續試哪一個方向?

方法一

- a. int maze(int width, int height, char board[][22],
  int curX, int curY, int targetX, int targetY,
  char path[][2], int npath)

  b. 這個函式延伸 path[] 陣列內長度是 npath 的部份路徑, 搜尋由 (curX,curY) 開始到 (targetX,targetY) 的路徑, 如果找到一條路徑, 就回傳 1, 同時陣列 path[0]~path[npath-1] 就是完整路徑; 如果沒有找到任何路徑就回傳 0; board 陣列中除了記錄牆壁/障礙, 也標示所經過的路徑, 以免不斷地繞圈圈

  c. 拆解為小一點的問題: 每一次在某一條路徑上往前走一步時, 就靠近目的地一點, 又可以運用這個 maze 函式來搜尋接下去的路徑

  d. 每一次到達一個位置 (curX,curY), 有四個方向可以測試, 其中有一個是上一步的位置, board[][]==2; 不能走的, board[][]==-1; 可以走的, board[][]==0; 針對每一個可以走的, 嘗試往前走一步, 把這一步記錄在 path[] 陣列並且呼叫
  maze(width, height, board, curX+dirs[i][0],
  curY+dirs[i][1], targetX, targetY, path, npath)

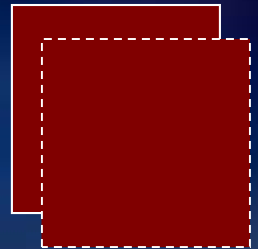
  e. (curX==targetX)&&(curY==targetY) 就表示走到目的地了

-1	0	0	0	0	-1	0
-1	-1	0	-1	2	2	0
-1	0	-1	0	-1	2	-1
-1	2	2	2	2	2	-1
-1	-1	-1	-1	-1	-1	-1

二維陣列

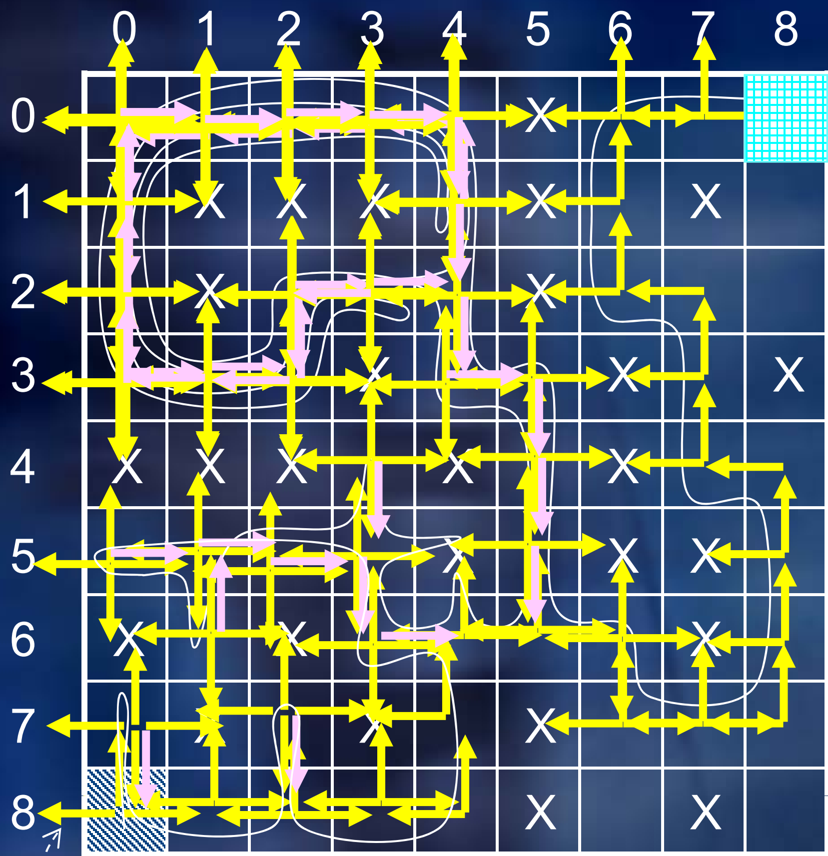
- int board[22][22]; // 打算處理 **20x20** 的迷宮, 周圍保留一列一行內容為 -1, 代表牆壁, 這樣子在程式內部測試時不需要特別去檢查有沒有超過邊界 -- 用 if 敘述去檢查陣列的註標是不是在範圍內 1≤row≤height, 1≤col≤width
- 不過這樣子宣告的話, 迷宮的實際座標就在 (1,1)-(width,height) 中間, 和平常陣列註標在 (0,0)-(width-1,height-1) 有一點點位移, 我們也可以在宣告的地方稍微動一點手腳, 讓我們保持一般的陣列使用方法, 但是又可以有希望有的邊界 ...... 能不能定義一個 (-1,-1)-(width,height) 的陣列?
- int board0[22][22];

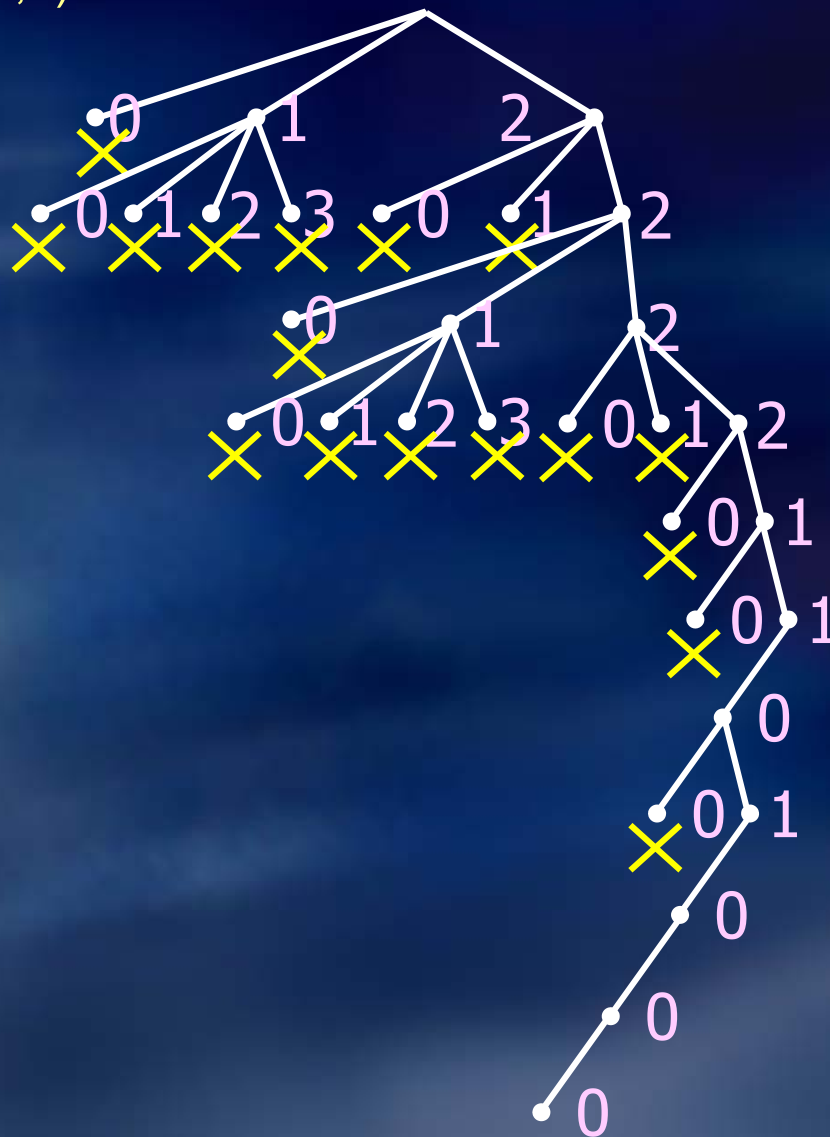
```
int board0[22][22];
int (*board)[22] = (int (*)[22]) &board0[1][1];
for (i=-1; i<=height; i++) board[i][-1] = board[i][width] = -1;
for (i=0; i<height; i++)
    for (j=0; j<width; j++) board[i][j] = 0;
for (j=0; j<width; j++) board[-1][j] = board[height][j] = -1;
```



```
int maze(int width, int height, char board[][22],
            int curX, int curY, int targetX, int targetY,
            char path[][2], int npath) { // 0:左,1:上,2:右,3:下
    static const int dirs[][2] = {{-1,0}, {0,1}, {1,0}, {0,-1}};
    if ((curX==targetX)&&(curY==targetY)) return 1;
    board[curX][curY] = 2; // 標示已經走過此格
    for (int i=0; i<4; i++)
        if (board[curX+dirs[i][0]][curY+dirs[i][1]]==0) {
            path[npath][0] = curX+dirs[i][0];
            path[npath][1] = curY+dirs[i][1];
            if (maze(width, height, board,
                        curX+dirs[i][0], curY+dirs[i][1],
                        targetX, targetY, path, npath+1)) return 1;
        }
    board[curX][curY] = 0; // 四個方向最後都是死路或已經走過, 退回前一層
    return 0;
}
```

迷宮路徑搜尋 — 執行範例





迷宮路徑搜尋 – 退回前一步

- 很多同學第一次看到前面的遞迴程式時, 可以接受每呼叫一次 maze() 就是往前走一步, 但是有點難想像到底怎麼退回前一步的, 例如下面這個主要程式段落裡, 如果某一次 maze(…,nextX, nextY, …) 的呼叫回傳 0, 代表如果下一步走 (nextX, nextY) 是沒有辦法走到目標點的, 所以要繼續嘗試 (curX, curY) 的下一個方向 (下一個 i), 這個時候其實就是退回前一步

```
for (i=0; i<4; i++)
    if (board[nextX=curX+dirs[i][0]][nextY=curY+dirs[i][1]]==0)
        if (maze(..., nextX, nextY, ...)) return 1;
```

- 從另外一種角度描述程式的表現, 前面我們數數字的程式, 當第一位填 2, 接下來由 (1,1,1) 數到 (n,n,n), 下一個數字就要退到最前面填 3; 現在這個迷宮程式如果你把遞迴函式呼叫時每一層迴圈裡的 i 值 (就是每一步是左上右下哪一個方向）列出，所謂退回前一步，就是當某一步試完四個方向以後，回前一格繼續試下一個方向

istart~iend

2	1	1	1
	~	~	~
	n	n	n

i

3	1	2	0	0	0
			~	~	~
			3	3	3

- 前面那個程式中 board[][] 陣列中只記錄了有沒有走過以及牆壁/障礙, 其實可以記錄更多的東西, 甚至走過的路徑都可以記下來, 不需要使用額外的 path[][2] 陣列

方法二

- a. int maze(int width, int height, char board[][22],
  int curX, int curY, int targetX, int targetY)

  b. 這個函式搜尋由 (curX,curY) 開始到 (targetX,targetY) 的路徑, 如果找到路徑, 就列印路徑並且回傳 1, 如果沒有找到任何路徑就回傳 0; board 陣列中除了記錄牆壁/障礙, 也標示由哪裡走過來的, 以免不斷繞圈圈, 同時找到路徑後還可以藉由所記錄的資料印出路徑)

  c. 拆解為小一點的問題: 每一次在某一條路徑上往前走一步時, 就靠近目的地一點, 又可以運用這個 maze 函式來搜尋接下去的路徑

  d. 每一次到達一個位置 (curX,curY), 有上下左右四個相鄰位置 (nextX, nextY) 可以測試, 其中有一個標示上兩步的位置, board[][]>0; 有的不能走, board[][]==-1; 有的可以走, board[][]==0; 針對每一個可以走的方向, 嘗試往前走一步, 把這一步的方向 (1:左,2:上,3:右,4:下) 記錄在 board[nextX][nextY] 中並且呼叫 maze(width, height, nextX =curX+dirs[i][0], nextY=curY+dirs[i][1], targetX, targetY)

  e. (curX==targetX)&&(curY==targetY) 就表示到達目的地了

```
    board[startX][startY] = 5; // visited
    maze(width, height, board, startX, startY, targetX, targetY);
```

```
int maze(int width, int height, char board[][22],
              int curX, int curY, int targetX, int targetY) {
  static const int dirs[][2] = {{-1,0}, {0,1}, {1,0}, {0,-1}};
  int nextX, nextY;
  if ((curX==targetX)&&(curY==targetY)) {
    printPath(width, height, board);
    return 1;
  }
  for (int i=0; i<4; i++) {
    nextX = curX+dirs[i][0], nextY = curY+dirs[i][1];
    if (board[nextX][nextY]==0) {
      board[nextX][nextY] = 1+i; // 1:左, 2:上, 3:右, 4:下
      if (maze(width, height, board, nextX, nextY, targetX, targetY))
        return 1;
      board[nextX][nextY] = 0;
    }
  }
  return 0; // 四個方向最後都是死路, 退回前一層
}
```

路徑列印

- 如何由 board[][] 所記錄的資訊中列印路徑? **遞迴**
- 這是這種尋找路徑問題的標準方法 (很多動態規劃的問題裡用到)
- 為什麼需要用遞迴? 怎麼這麼麻煩?? 因為 board[][] 記錄的是如何走到這一個座標點的方法, 所以需要由 (targetX=0, targetY=8) 往前追蹤到 (8, 0), 這樣子找到的路徑順序就反過來了, 要**反序列印**如果**不用額外的陣列**的話, 最簡單的方法就是遞迴, 利用系統的堆疊來完成

```
void printPath0(int width, int height, int board[][22], int curX, int curY) {
    static const int dirs[][2] = {{-1,0}, {0,1}, {1,0}, {0,-1}};
    if ((curX==8)&&(curY==0))
        printf("(8,0)");
    else {
        int prev = board[curX][curY]-1;
        printPath0(width, height, board,
                   curX-dirs[prev][0], curY-dirs[prev][1]);
        printf(" (%d,%d)", curX, curY);
    }
}
```

-1	0	0	0	0	-1	2
-1	-1	0	-1	0	2	3
-1	0	-1	0	-1	2	-1
-1	0	3	3	3	3	-1
-1	-1	-1	-1	-1	-1	-1

```
void printPath(int width, int height, int board[][22]) {
    printPath0(width, height, board, 0, 8); printf("\n");
}
```

- 尋找所有路徑: 某一個方向走下一步達到終點時, 繼續搜尋下一個方向

```
board[startX][startY] = 5; // visited
maze(width, height, board, startX, startY, targetX, targetY);
```

```
void maze(int width, int height, char board[][22],
                   int curX, int curY, int targetX, int targetY) {
  static const int dirs[][2] = {{-1,0}, {0,1}, {1,0}, {0,-1}};
  int nextX, nextY;
  if ((curX==targetX)&&(curY==targetY)) {
    printPath(width, height, board);
    return;
  }
  for (int i=0; i<4; i++) {
    nextX = curX+dirs[i][0], nextY = curY+dirs[i][1];
    if (board[nextX][nextY]==0) {
      board[nextX][nextY] = 1+i; // 1:左, 2:上, 3:右, 4:下
      maze(width, height, board, nextX, nextY, targetX, targetY);
      board[nextX][nextY] = 0; //  以 (nextX,nextY) 為起點
    }                            // 的已經試完, 退回試下一種走法
  }
}
```

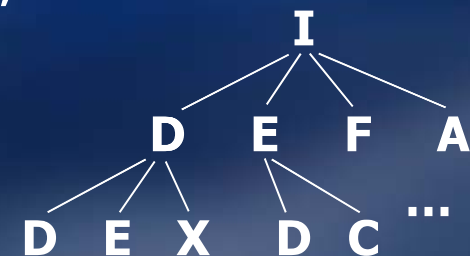
- 尋找最短路徑
  - DFS 可以尋找所有路徑並且記錄最短路徑長度, 不過最短路徑需要重新再執行一次搜尋
  - 在尋找的過程中也可以用目前最短的路徑作為上限, 如果某一條路徑的長度已經超過, 就退回測試下一個可能的選項
  - Dijkstra's shortest path 最短路徑演算法
  - 廣度優先搜尋法 (Breadth First Search, BFS)
  - Dynamic Programming

字串拼圖

- 給定一個如右圖的字串二維陣列, 請寫一個程式判斷目標字串是否出現在陣列中, 字串可以上下左右串聯, 也可以彎折, 不可以斜向串聯, 不可交叉, 例如 ABD, DBA, IAT, TAI, ERNM, MNRE 都在陣列中, BDAT, DIAXD, DIDX 都不在陣列中
- 假設二維字元陣列 board[7][7] 其中第 0 列, 第 0 行, 第 6 列, 第 6 行都填入空格來標示邊界; 一維字元陣列 target[] 裡有長度 len 的目標字串
- 如下圖先簡化這個問題成為 “target 目標字串與以 board[ix][iy] 為起點的所有字串比對”, 比對成功回傳 1, 否則回傳 0, 亦即與 I, ID, IE, IF, IA, IDD, IDE, IDX, IED, IEC, IFC, IFD, IAX, IAT, IAD, ... 由短而長逐步比對, 這個比對過程可以畫成樹狀圖以 DFS 搜尋, 可以用遞迴函式實作

A	B	D	E	C
S	E	D	I	F
B	V	X	A	D
E	R	Q	T	W
Y	N	M	P	Q

A	B	D	E	C
S	E	D	I	F
B	V	X	A	D
E	R	Q	T	W
Y	N	M	P	Q

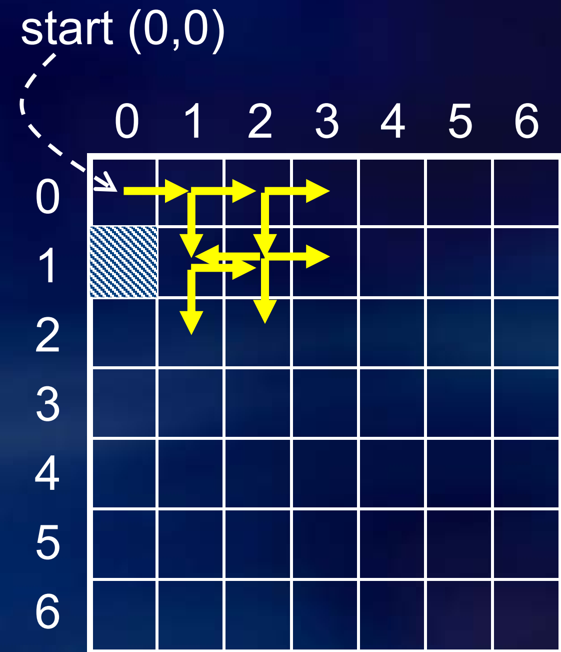


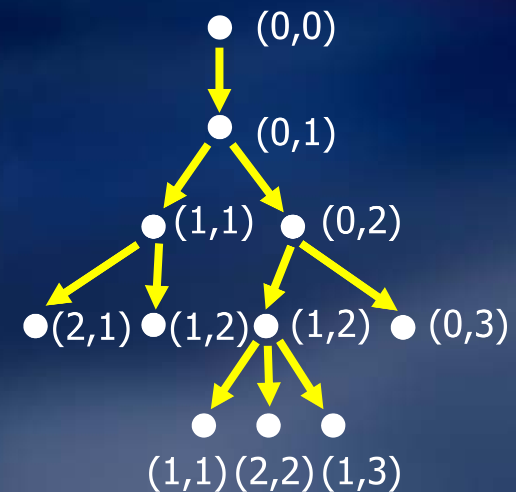
- a. int isMatched(char board[][7], int len, char target[], int x, int y)
  b. **這個函式檢查字元陣列 target[] 中的字串是否與以 board[x][y] 為中心的字串吻合**
  c. 長度短一點的問題: 如果第一個字吻合, 即 target[0]==board[x][y], 則進一步比對由 target[1] 開始的字串是否可以和由 board[x+1][y], board[x-1][y], board[x][y+1], 或是 board[x][y-1] 為中心的字串吻合, 注意需要避開折回或是交叉重疊的字串
  d. 如果任一子字串比對成功, 則表示完整的字串是吻合的, 回傳 1
  e. 長度為 1 且 board[x][y] == target[0] 時表示吻合, 回傳 1

```
int isMatched(char board[][7], int len, char target[], int x, int y) {
    if (board[x][y] == target[0]) {
        board[x][y] = -board[x][y];
        if (len == 1) return 1;
        else {
            if (isMatched(board, len-1, &target[1], x+1, y)) return 1;
            else if (isMatched(board, len-1, &target[1], x, y-1)) return 1;
            else if (isMatched(board, len-1, &target[1], x-1, y)) return 1;
            else if (isMatched(board, len-1, &target[1], x, y+1)) return 1;
        }
    }
    return 0;
}
```

Hamiltonian Path

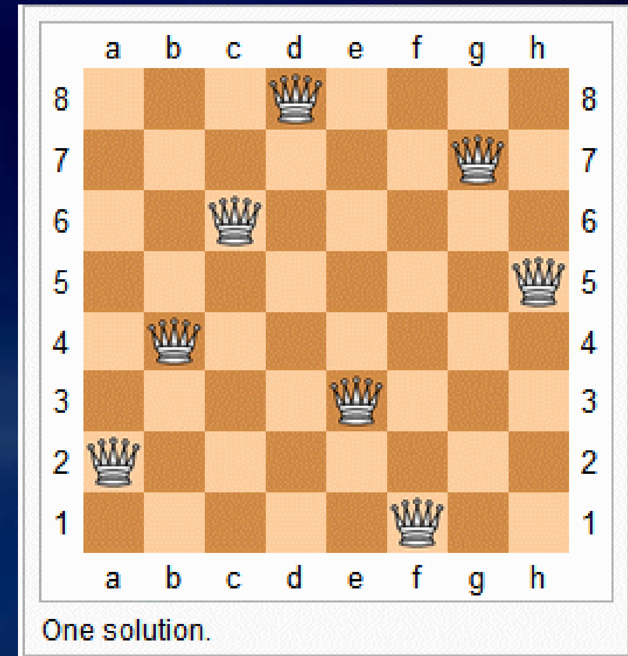
- 把右邊這樣的棋盤中每一格看成是一個圖形的節點，每一個節點和上下左右相鄰的四個節點連接在一起，請由 (0,0) 這一個節點開始，判斷是否存在一個路徑可以經過所有的節點恰好一次 (一筆畫過所有的節點)，這種路徑稱為 Hamiltonian Path
- 尋找 Hamiltonian Path 的問題是 NP-Hard
- 我們可以寫類似搜尋迷宮路徑的暴力演算法，程式效率會不好，但是對於參數比較小的問題來說，是可以找到答案的





```
int explore(int nSize, int iVisitStates[][15], int currentX, int currentY,
               int serialNo, long *lSolCount, long *lTrialCount) {
   int offset[4][2] = {-1,0, 0,-1, 1,0, 0,1};
   int i, nextX, nextY, lOldCount = *lSolCount;
   *lTrialCount += 1; // node counts (including all internal and leave nodes)
   if (serialNo == nSize*nSize-2) {
     *lSolCount += 1;
      return 1;
   }

   iVisitStates[currentX][currentY] = serialNo;

   for (i=0; i<4; i++) {
      nextX = currentX + offset[i][0];
      nextY = currentY + offset[i][1];
      if (iVisitStates[nextX][nextY] < 0)
         explore(nSize,iVisitStates, nextX,nextY, serialNo+1,
                  lSolCount, lTrialCount);
   }

   iVisitStates[currentX][currentY] = -1;
   return (*lSolCount>lOldCount ? 1 : 0);
}
```

8 Queen Problem



One solution.

- 把 8 個皇后放在 8x8 的西洋棋棋盤上使得用基本的西洋棋規則，任意兩個皇后無法吃掉對方，也就是**兩個皇后不能在同一列、同一行、或是在對角線方向的直線上**
- 暴力解: DFS (深度優先搜尋), 需要使用堆疊來記錄路徑搜尋過程中的下一個選擇
- 一個完全不考慮限制的暴力搜尋法需要考慮 2^{8*8} 種可能的解; 如果考慮行與列的限制, 則有 **8!** 種可能的解; 如果再考慮對角線方向的限制, 可能解的數目又降低了
- 如果考慮旋轉與鏡射的重複性, 8后問題的 92 個解其實只剩下 12 個不同的解
- 由於是一個像是迷宮或是 Hamilton Path 這樣的深度優先搜尋, 我們可以撰寫遞迴函式來搜尋可能的解答
- n-queen 快速的經驗法則解法請參考 wiki

- a. int placeQueensForRemainingRows(int size, int cols[], int iRow)

  b. 這個函式將第 iRow 個到第 size-1 個皇后在不違反 cols[0]~cols[iRow-1] 的限制之下排好, 結果放在cols[iRow]~cols[size-1] 中, 如果能夠完成就回傳 1, 否則回傳值為 0

  c. 拆解為兩個小一點的問題: ❶ 排第 iRow 個皇后, 在第 iRow 列上, 測試第 0, 1, ..., size-1 行是否和先前放在 (0, cols[0])~(iRow-1,cols[iRow-1]) 上的皇后有衝突, ❷ 如果沒有的話, 再運用這個遞迴函式排好第 iRow+1 到 size-1 個皇后; 如果前者在所有可能性中找不到答案, 回傳 0; 如果後者找不到答案, 繼續 ❶ 中第 0, 1, ..., size-1 行未進行的測試 (backtracking)

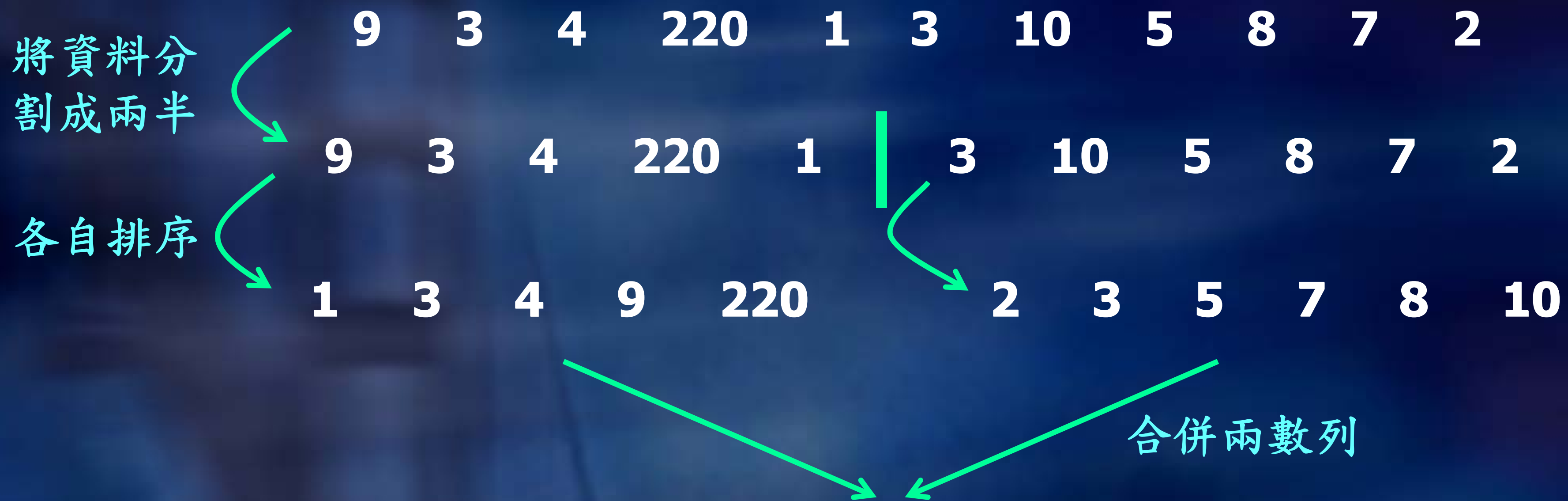
  d. 子問題的答案和原本的部份答案合併在一起就是答案了

  e. iRow 等於 size 時, cols[0]~cols[size-1] 已經完全排好, 列印

```
int placeQueensForRemainingRows(int size, int cols[], int iRow) {
    if (iRow == size) { printBoard(size, cols); return 1; }
    for (int iCol=0; iCol<size; iCol++)
        if (isSafeWithPrevious(size, cols, iRow, iCol)) {
            cols[iRow] = iCol;
            if (placeRemainingRows(size, cols, iRow+1)) return 1;
        }
    return 0;
}
```

合併排序法 (Merge Sort)

- 以一個範例來說明基本的演算法



- 運算複雜度: O(n log n), 比 quicksort 的平均 O(n log n) 好
- 需要額外的 O(n) 記憶體, 比 quicksort 多

```
void mergesort(int data[], int len) {
    int len1=len/2, len2=len-len1;
    if (len <= 1) return;
    mergesort(data, len1);  mergesort(&data[len1], len2);
    merge(data, len1, &data[len1], len2);
}
```

```
void merge(int data1[], int len1, int data2[], int len2) {
    int data1_idx=0, data2_idx=0, result_idx=0;
    int *buf = (int *) malloc((len1+len2) * sizeof(int));
    while (data1_idx < len1 || data2_idx < len2) {
        if (data1_idx < len1 && data2_idx < len2)
            buf[result_idx++] = (data1[data1_idx]<=data2[data2_idx]) ?
                                    data1[data1_idx++] : data2[data2_idx++];
        else if (data1_idx < len1)
            buf[result_idx++] = data1[data1_idx++];
        else
            buf[result_idx++] = data2[data2_idx++];
    }
    for (result_idx = 0; result_idx < len1+len2; result_idx++)
        data1[result_idx] = buf[result_idx];
    free(buf);
}
```

Determinant of an n-by-n Matrix

- Laplace's formula

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

- Recursion

$$\det(\mathbf{A}) = (-1)^{1+1}a_{1,1}M_{1,1} + (-1)^{1+2}a_{1,2}M_{1,2} + (-1)^{1+3}a_{1,3}M_{1,3}$$

minor

$$\mathbf{M_{1,3}} = \det\left(\begin{matrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{matrix}\right) \quad \textbf{i.e.} \quad \begin{pmatrix} a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{pmatrix}$$

- Termination

$\det(\mathbf{A}) = a_{1,1}a_{2,2} - a_{2,1}a_{1,2}$ for a 2-by-2 matrix **A**

迴圈 ⇒ 遞迴函式

- 迴圈

```
struct Local localVar; // contains persistent variables for <loop-body>
for (loopVar=1; loopVar<100; loopVar++)
    <loop-body>
```

- 遞迴函式

```
void recFun() {
    int loopVar = 1;
    struct Local localVar;
    recHelperFunc(loopVar, localVar);
}                              // 完成 <loop-body>, loopVar, loopVar+1, ..., 100
void recHelperFunc(int loopVar, struct Local &localVar) {
    if (loopVar<100) {
        <loop-body>
        loopVar++;
        recHelperFunc(loopVar, localVar);
    }
}
```

tail-recursive

範例

- 迴圈

```
for (i=0; i<n; i++)
    printf("%d\n", i);
```

- 遞迴函式

```
void recPrint() {
    int i=0;
    recPrintHelper(n, i); // prints i,i+1,...,n-1
}
void recPrintHelper(int n, int i) {
    if (i<n) {
        printf("%d\n", i);
        i++;
        recPrintHelper(n,i);
    }
}
```

```
void recPrint(int n) { // prints 1,...,n-1
    if (n==0)  printf("0");
    recPrint(n-1);
    printf("%d\n", n-1);
}
```

C++ default parameter

```
void recPrint(int n, int i=0) {
    if (i<n) {
        printf("%d\n", i);
        i++;
        recPrint(n,i);
    }
}
```

```
for (i=sum=0; i<n; i++) sum+=i;
---------------------------------------
int recSum(int n, int i=0) {
    if (i==n) return 0;
    return i+recSum(n,i+1);
}
--- or more commonly -------------
int recSum(int n) {
    if (i==0) return 0;
    return n+recSum(n-1);
}
```

範例 (cont'd)

> 這個範例其實並**不是**告訴你矩陣的乘法需要用遞迴才能寫，事實上迴圈的寫法就是非常簡潔的了，效率其實也沒有比最佳的演算法差多少，這個範例是希望讓你體會**迴圈和遞迴寫法的關聯性**

- 迴圈 矩陣乘法

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

- 遞迴函式 - 每次呼叫完成一個元素 c[i][j] / 一列 c[irow]

```
void multiplyMatrix(int m, int n, int a[][MAX],
                    int o, int p, int b[][MAX],
                    int c[][MAX], // 需要初始為 0
                    int idx) { // idx: 0~m*n*p-1, 但需要適當遞增機制
    int i=idx/(n*p), j=idx/n%p, k=idx%n;    // idx: 0 ~ m*n*p-1
    if (idx<m*p*o) {
        c[i][j] += a[i][k]*b[k][j];
        multiplyMatrix(m, n, a, o, p, b, c, idx+1);
} }
```

```
void multiplyMatrix(int m, int n, int a[][MAX],
                    int o, int p, int b[][MAX],
                    int c[][MAX], // 需要初始為 0
                    int irow) { // irow: 0~m-1
    if (irow<m) {
        for (j=0; j<p; j++)
            for (k=0,c[irow][j]=0; k<n; k++)
                c[irow][j] += a[irow][k]*b[k][j];
        multiplyMatrix(m, n, a, o, p, b, c, irow+1);
} }
}
```

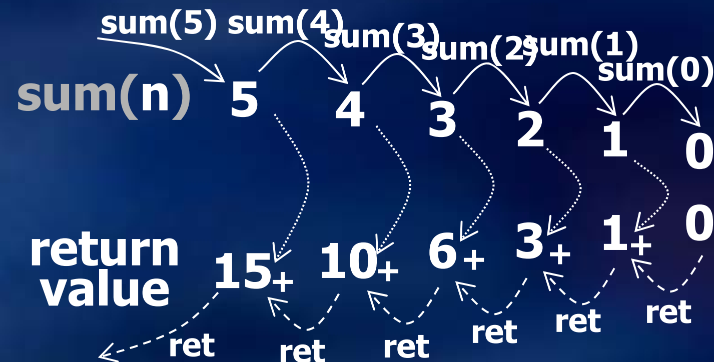
尾端遞迴函式

1. 尾端遞迴函式 (Tail Recursive Function, Tail Recursion) 也是一種遞迴函式，額外的特性是每次呼叫自己之後，必須立刻回傳結果，不能做任何其它事

例如：計算 f(n) = 0 + 1 + 2 + … + n 的遞迴函式

一般遞迴函式

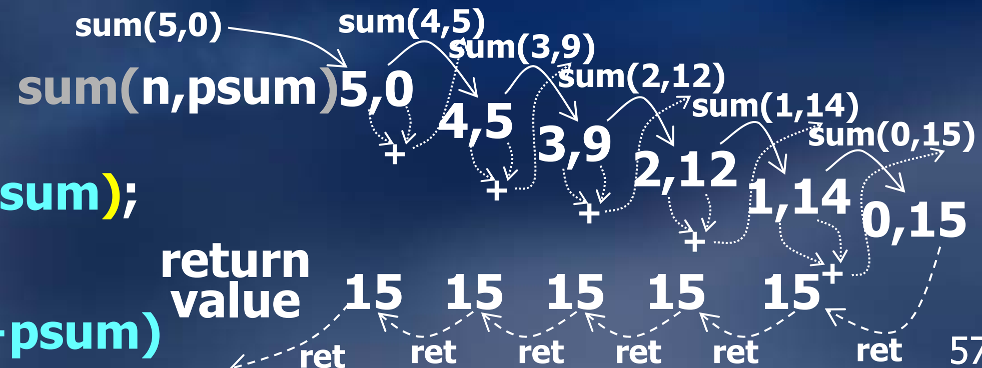
```
int sum(int n) {  // 計算 f(n)=0+1+2+...+n
    if (n==0)
        return 0;
    else
        return sum(n-1)+n;
}
```



C++ 預設參數

尾端遞迴函式

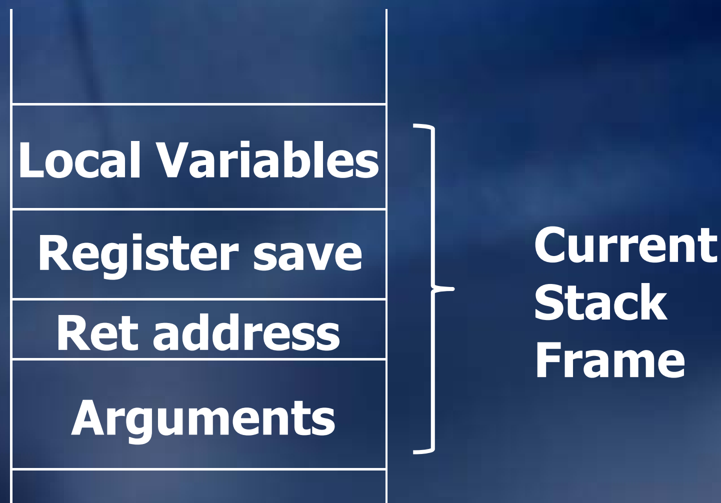
```
int sum(int n, int psum=0) { // 計算 f(n)=0+1+2+...+n+psum
    if (n==0)
        return psum;
    else
        return sum(n-1, n+psum);
}
```



計算 f(n)=1+2+…+(n-1)+(n+psum)

尾端遞迴函式 (2/4)

2. 尾端遞迴函式比一般的遞迴函式難寫難讀，為什麼要寫這種尾端遞迴函式呢? 因為 C++ 編譯器可以對這種函式最佳化，使得它就算呼叫自己非常多次，不會使用額外的堆疊空間 (因為尾端遞迴函式在下一層函式回傳後並不需要使用上一層函式裡面的任何參數、暫存器、或是變數的數值，所以概念上堆疊上的堆疊框可以直接在呼叫下一層函式時使用，上一層函式的 return address，直接作為下一層函式的 ret address，只需更換呼叫的引數值即可)



3. 計算 f(n)=1+...+n 的尾端遞迴函式的第二種設計方法：

a. 分解問題 : f(n)=sumTR(n,psum=0,i=1)=(0)+1+2+...+n=

sumTR(n-1,psum+i,i+1)=(0+1)+2+...+n=

...

sumTR(0,psum+n,n+1)=(0+1+...+n)

加總 psum 及 1~n 後 n 個數字

注意

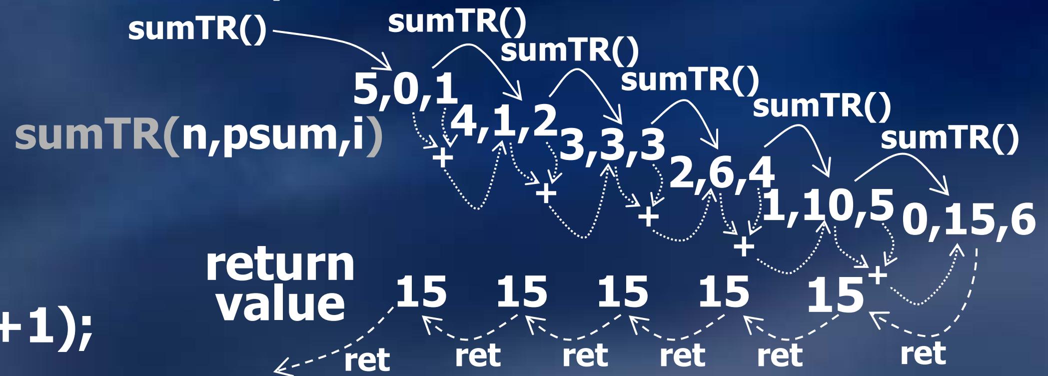
①一定有一個參數 **psum** 是部份答案，這個部份答案一層一層越來越接近想要計算的 **f(n)**，開始的呼叫是 **sumTR(n,psum=f(0),1)**

②下層函式 **sumTR(n',f(n-n'),n-n'+1)** 需要計算出原來的 **f(n)**，而不是子問題的答案，如此才能夠在每一層回傳最後結果，在呼叫 **sumTR(0, f(n),n+1)** 時基本上沒有計算的動作，第二個參數就是最後的答案 **f(n)**

b. 根據分解的問題定義出遞迴函式以及其參數

int sumTR(int n, int psum=0, int i)

```
int sumTR(int n, int psum, int i) {
   if (n==0)
      return psum;
   else
      return sumTR(n-1, psum+i, i+1);
}
```



4. 計算 Fibonacci f(n)=f(n-1)+f(n-2) 的尾端遞迴函式的設計方法:

a. 分解問題 : **f(n)=fibo(n,f1=f(0),f2=f(-1))**

f(n-1)=fibo(n-1,f1=f(1)=f(0)+f(-1),f2=f(0))

...

f(1)=fibo(1,f1=f(n-1),f2=f(n-2))

f(0)=fibo(0,f1=f(n)=f(n-1)+f(n-2),f2=f(n-1))

注意

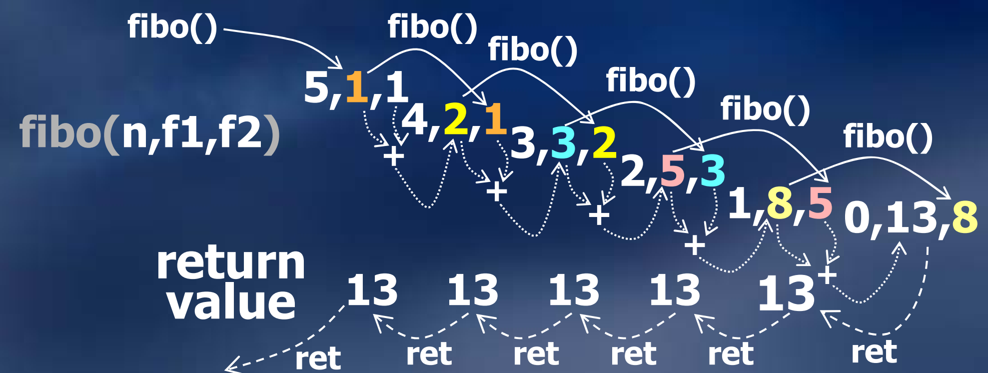
①一定有一個參數 **f1** 是部份答案，這個部份答案一層一層越來越接近想要計算的 **f(n)**，開始的呼叫是 **fibo(n,f1=f(0)=1,f2=f(-1)=1)**

②下層函式 **fibo(n',f(n-n'),f(n-n'-1))** 需要計算出原來的 **f(n)**，而不是子問題的答案，如此才能夠在每一層回傳最後結果，在呼叫 **fibo(0, f(n),f(n-1))** 時基本上沒有計算的動作，第二個參數就是答案 **f(n)**

b. 根據分解的問題定義出遞迴函式以及其參數

int fibo(int n, int f1=1, int f2=1)

```
int fibo(int n, int f1, int f2) {
   if (n==0)
      return f1;
   else
      return fibo(n-1, f1+f2, f1);
}
```



遞迴專屬的錯誤

- 全域/靜態變數

```
int sum(int data[], int n) {
    static int result = 0;
    if (n>0)  {
        result += data[n-1];
        sum(data, n-1);
    }
    return result;
}
```

- Side effect

```
 int recSum(int n, int i=0) {
    if (i==n) return 0;
    return i+recSum(n,i++);
}
```

系統堆疊大小控制

- platform default size

platform	default size
SunOS/Solaris	8172K bytes (Shared Version)
Linux	8172K bytes
Windows	1024K bytes (Release Version)
cygwin	2048K bytes

- Linux
  - ulimit -a # shows the current stack size
  - ulimit -s 32768 # sets the stack size to 32M bytes
- OSX
  - LD_FLAGS= -Wl,-stack_size,0x100000000
- Windows Visual Studio
  - Run "dumpbin /headers executable_file", and you can see the "size of stack reserve" information in "optional header values"
  - Run "editbin /STACK:size" to change the default stack size.
  - Visual Studio: **Properties -> Configuration Properties -> Linker -> System -> Stack Reserve Size**
  - #pragma comment(linker, "/STACK:36777216")
- g++ -Wl,--stack=256000000 test.cpp –o test.exe