

遞迴函式 – Ackermann Function

丁培毅

實習目標：

1. 練習設計遞迴函式
2. 練習遞迴的思考方式
3. 追蹤遞迴函式的運作
4. 練習遞迴函式的偵錯方法

遞迴函式 – Ackermann Function

- 請撰寫一個程式，計算出下表中 Ackermann Function 的數值，Ackermann Function 的定義如下：

$$A(m, n) = \begin{cases} n+1, & m=0 \\ A(m-1, 1), & m>0, n=0 \\ A(m-1, A(m, n-1)), & m>0, n>0 \end{cases}$$

- 程式輸出範例如下：

$A(0, 0) = 1$
 $A(0, 1) = 2$

 $A(3, 9) = 4093$
 $A(4, 0) = 13$
 $A(4, 1) = 65533$
 $A(5, 0) = 65533$

m\n	0	1	2	3	4	5	6	7	8	9	n
0	1	2	3	4	5	6	7	8	9	10	n+1
1	2	3	4	5	6	7	8	9	10	11	n+2
2	3	5	7	9	11	13	15	17	19	21	2n+3
3	5	13	29	61	125	253	509	1021	2045	4093	$2^{n+3}-3$
4	13	65533									$2^{2^{n+2}}-3$
5	65533										

$13=2^{2^2}-3$, $65533=2^{2^{2^2}}-3$, 表格上其它沒有顯示的數值有數學上的定義，但是在計算上用現成的整數變數型態無法表示
 $A(4,2)=A(3,65533)=2^{65536}-3$, $A(5,1)=A(4,65533)=??$

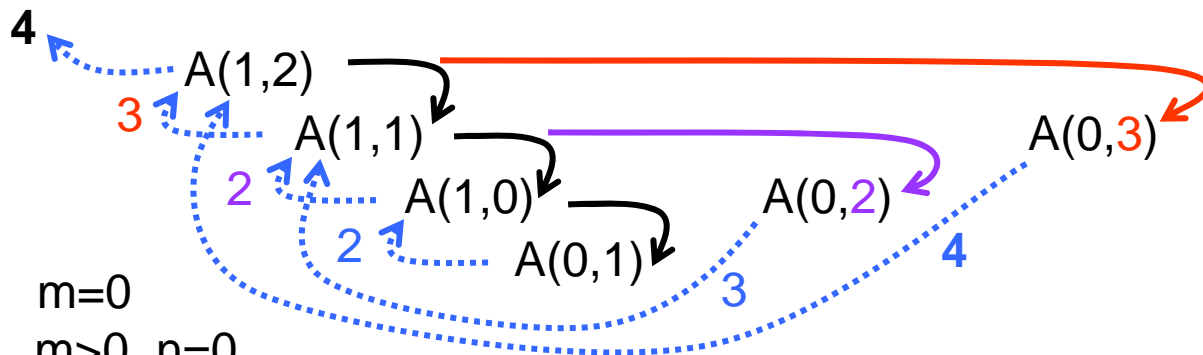
分析

1. 要用遞迴函式來計算 Ackermann Function 在某些參數 m, n 的數值時, 基本上還是遵循設計遞迴函式解的基本步驟:
 - a. 定義出遞迴函式及其參數 (遞迴函式一定需要, 不可以用全域變數取代)
 - 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

但是 c, d 拆解和合成兩個步驟是 Ackermann Function 的定義已經幫你做好了, 就好像在用遞迴函式計算 Fibonacci 數列的例子一樣, 只需要做 a, b, e 三個步驟就可以很快地完成這個程式, 我們先回憶一下計算 Fibonacci 數列的例子, $f(n)=f(n-1)+f(n-2)$, $f(1)=f(2)=1$

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

2.
 - a. `int ackermann(int m, int n)`
 - b. 這個函式計算並回傳 **Ackermann function $A(m,n)$** 的數值
 - c. 拆解為小一點的問題: 計算 $A(m-1,1)$, $A(m,n-1)$ 與 $A(m-1,A(m,n-1))$
 - d. 直接依照 **Ackermann function** 的定義, 在參數滿足不同條件的情況下用比較小的問題的結果直接作為最後的結果
 - e. $A(0,n)$ 的數值是 $n+1$
3. 請實作上列設計 (請注意遞迴函式一定需要設計函式的參數, 一定需要有 **base class** 的結束條件, 遞迴呼叫前一定需要測試參數是否符合結束條件, 不可以完全不測試就直接進行遞迴呼叫)
4. 設計完的遞迴函式裡面一般來說變數不多, 程式碼不多, 所以程式邏輯的複雜度不高 (相較於解同樣問題的迭代式程式來說), 程式看起來是很精簡的, 這是為什麼說遞迴函式中程式碼的表達性很高的原因
5. 有人說遞迴函式是很容易偵錯的, 這個說法需要稍微保留一點, 也許對於熟悉撰寫程式的人來說是很容易的, 但是對於初學者來說, 需要很清楚地掌握遞迴函式的運作過程: 需要知道在任何時候會有許多層函式一個呼叫一個, 執行下層函式時上層函式都還沒有結束, 每一層函式的參數以及部份執行結果也都還存在, 例如呼叫 $A(1,2)$ 時, 會引發一連串 $A(1,1)$, $A(1,0)$, $A(0,1)$, $A(0,2)$, $A(0,3)$ 的函式呼叫 (連同 $A(1,2)$ 的呼叫共 6 次函式呼叫, 最深的呼叫為第 4 層 $A(0,1)$)



$$A(m, n) = \begin{cases} n+1, & m=0 \\ A(m-1, 1), & m>0, n=0 \\ A(m-1, A(m,n-1)), & m>0, n>0 \end{cases}$$

6. 呼叫 $A(2,3)$ 時會引發 43 次呼叫 (其中最深的呼叫有 10 層), 呼叫 $A(3,4)$ 時會引發 10306 次呼叫 (其中最深的呼叫有 127 層); 如果你知道程式的運作方式, 看到程式沒有預期的表現時才能夠一步一步找出問題, 並且更正程式碼, 接下來你可以思考一件事, 就是怎樣修改程式讓它在執行過程中記錄上面這些資料 - 總共呼叫幾次, 最深的呼叫有幾層... 這個數值一般來說會盡量用理論推演得到, 而不見得是用程式執行得到; 總呼叫次數和程式執行時間直接相關, 最深有幾層呼叫則和系統堆疊須要準備多大有關係
7. 偵錯這種遞迴架構的程式時, 一開始可以先直接呼叫 **base case**, 檢查答案是不是對的, 然後呼叫大一點點的問題, 檢查答案是不是對的, 再大一點點, ..., 然後檢查大問題會分解成較小問題的過程 - 分解與合成是否正確? 列印函式每一次呼叫的**參數**和**回傳值**是有很大的幫助的, 可以驗證分解的過程是否正確, 合成的動作是否符合預期

8. 遞迴函式很大的缺點是執行的速度, 像前面呼叫 $A(3,4)$ 會引發 10306 次的函式呼叫, 就會使得執行效率很差, (如果你回想計算 fibonacci 數列的遞迴函式也會發現計算 $f(n)$ 需要呼叫函式 $O(2^n)$ 次), 解決的方法其實就是不要用遞迴...用迴圈來計算 fibonacci 數列的方法很簡單: $f(n)=f(n-1)+f(n-2)$, $f(1)=f(2)=1$

假設我們需要計算 $f(20)$, 就用一個陣列存放 $f(1), f(2), \dots, f(20)$

$f(1), f(2)$ 都是 1, $f(3)$ 則是前面兩個的和, $f(4)$ 是 $f(2), f(3)$ 的和, 算到 $f(20)$ 就好了, 也就是

```
int i, f[21] = {0, 1, 1};  
for (i=3; i<=20; i++) f[i] = f[i-1] + f[i-2];  
printf("f(20)=%d\n", f[20]);
```

迴圈只要執行 18 次, 使用 21 個元素的整數陣列, 存取記憶體 54 次, 所以要改進 Ackermann 函式的計算速率就需要用迴圈, 可以把 $A(m,n)$ 都記錄在陣列裡, 如此就可以用記憶體來換取時間, 參數相同的 $A(m,n)$ 只需要計算一次就好, 不需要重複計算很多次。以前面的表格來說, 如果需要計算 $m:0\sim3, n:0\sim9$, 以 $A(3,9)$ 來說, 需要 $A(2, A(3,8))=A(2,2045)$, 所以如果不用表格最後一行的公式的話, 需要的陣列維度是 $\text{int } A[4][2046]$, 如果要計算 $A(4,1)$ 的話, $A(4,1)=A(3, A(4,0))=A(3,13)=A(2, A(3,12))=A(2,32765)$, 需要的陣列宣告是 $\text{int } A[4][32766]$, 所以用迴圈計算 Ackermann function 需要大量記憶體