

資料結構報告

梁詠琳

october 22,2024

CONTENTS

CHAPTER 1 解題說明	3
Figure1.1HW1.cpp	3
CHAPTER 1 解題說明	4
問題 2: 集合的冪集	4
Figure1.2HW2.cpp	4
CHAPTER 2 演算法設計與實作	5
CHAPTER 3 效能分析	6
CHAPTER 4 測試與過程	7
CHAPTER 5 申論及心得	2

CHAPTER 1 解題說明

問題 1: Ackermann 函數

Ackermann 函數是一個著名的遞迴函數。它是一個極為特別且有趣的數學遞迴函數，它並不是普通的遞迴，而是能夠成倍成倍地增長，這種增長速度令人驚嘆。

簡單來說，Ackermann 函數就像數學家眼中的「無底洞」，計算起來無窮無盡。其定義如下：

Ackermann's function $A(m, n)$ is defined as follows:

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

This function is studied because it grows very fast for small values of m and n . Write a recursive function for computing this function. Then write a nonrecursive algorithm for computing Ackermann's function.

此問題要求我們實作遞迴版本的 Ackermann 函數，並設計一個非遞迴演算法。

實作參見檔案HW1.cpp，其遞迴函式：

```
5  ✓  int acker(int m, int n)
6      {
7          // 當 m 等於 0, 返回 n + 1
8          if (m == 0)
9              return n + 1;
10         else if (n == 0) // 當 n 等於 0, 遞迴計算 acker(m-1, 1)
11         {
12             return acker(m - 1, 1);
13         }
14         else // 否則, 遞迴計算 acker(m-1, acker(m, n-1))
15         {
16             return acker(m - 1, acker(m, n - 1));
17         }
18     }
```

Figure1.1HW1.cpp

這個函數的美妙之處在於它的遞迴深度，對於較小的 m 和 n ，它的結果還能輕鬆計算，但一旦數字變大，電腦也會面臨「燒腦」的挑戰。

(這點TA教我算，數字在個位數就已然很煩了!)

CHAPTER 1 解題說明

問題 2: 集合的冪集

If S is a set of n elements, the *powerset* of S is the set of all possible subsets of S . For example, if $S = (a, b, c)$, then $\text{powerset}(S) = \{(), (a), (b), (c), (a, b), (a, c), (b, c), (a, b, c)\}$. Write a recursive function to compute $\text{powerset}(S)$.

冪集是某個集合的所有子集的集合。我會視它為集合中的所有可能性。

此問題要求我們使用遞迴來生成一個集合的冪集，而每次遞迴可以理解為選擇是否把某個元素加入當前子集的「二元選擇」。

實作參見檔案HW1.cpp, 其遞迴函式:

```
7  int powerset(string a, int b, string c)
8  {
9      // 當 b 等於字串長度時，輸出當前生成的子集
10     if (b == a.size()) {
11         cout << c << " "; // 輸出子集
12         return 0; // 結束當前遞迴
13     }
14     // 遞迴不選擇當前字符
15     powerset(a, b + 1, c);
16     // 遞迴選擇當前字符
17     powerset(a, b + 1, c + a[b]);
18 }
```

Figure1.2HW2.cpp

CHAPTER 2 演算法設計與實作

問題 1: 遞迴 Ackermann 函數, 用遞迴函數來計算 Ackermann 函數。

以下是程式碼的詳細設計:

```
1    #include <iostream>
2    using namespace std;
3
4    // 遞迴計算 Ackermann 函數
5    ✓ int acker(int m, int n)
6    {
7        // 當 m 等於 0, 返回 n + 1
8        if (m == 0)
9            return n + 1;
10       else if (n == 0) // 當 n 等於 0, 遞迴計算 acker(m-1, 1)
11       {
12           return acker(m - 1, 1);
13       }
14       else // 否則, 遞迴計算 acker(m-1, acker(m, n-1))
15       {
16           return acker(m - 1, acker(m, n - 1));
17       }
18   }
19
20   ✓ int main()
21   {
22       // 測試 Ackermann 函數, 輸出結果
23       cout << acker(1, 2); // 計算 A(1, 2)
24
25       return 0;
26   }
```

Figure2.1HW1.cpp

CHAPTER 2 演算法設計與實作

問題 2: 遞迴生成冪集

以下是程式碼的設計:

```
1      #include <iostream>
2      #include <string>
3
4      using namespace std;
5
6      // 遞迴生成冪集的函式
7  ✓ int powerset(string a, int b, string c)
8      {
9          // 當 b 等於字串長度時，輸出當前生成的子集
10         if (b == a.size()) {
11             cout << c << " "; // 輸出子集
12             return 0; // 結束當前遞迴
13         }
14         // 遞迴不選擇當前字符
15         powerset(a, b + 1, c);
16         // 遞迴選擇當前字符
17         powerset(a, b + 1, c + a[b]);
18     }
19
20  ✓ int main()
21     {
22         // 定義字符串 a 和空字符串 c
23         string a, c;
24         a = "abc"; // 設定字串為 "abc"
25         c = ""; // 初始化空字符串
26         powerset(a, 0, c); // 開始遞迴生成冪集
27         return 0;
28     }
```

Figure2.2HW2.cpp

CHAPTER 3 效能分析

Ackermann 函數

Ackermann 函數的增長速度極快，甚至超過了常規的遞迴函數。這使得它的

時間複雜度難以直接計算。

$$T(m, n) = O(A(m, n))$$

時間複雜度:

$O(A(m, n))$

可以說，對於較小的輸入值，它的計算還是可以接受的，但隨著m和n值的增大，計算的時間會成倍增長。因此，這裡的A(m,n)是 Ackermann 函數本身，這意味著其時間複雜度與輸出結果直接相關。

$$S(m, n) = O(\text{遞迴深度})$$

空間複雜度:

$O(\text{遞迴深度})$

Ackermann 函數有著非常高的時間複雜度，並且在大數值下其資源消耗也非常驚人。因此，當 m 增加時，遞迴深度的增長非常迅速。對於固定的m，遞迴深度通常是隨n指數增長。

CHAPTER 3 效能分析

冪集函數

對於一個大小為 n 的集合，其冪集總共有 2^n 個子集，這是由於每個元素都有兩種選擇：可以被包含在子集中，也可以不被包含。因此，

生成冪集的時間複雜度為

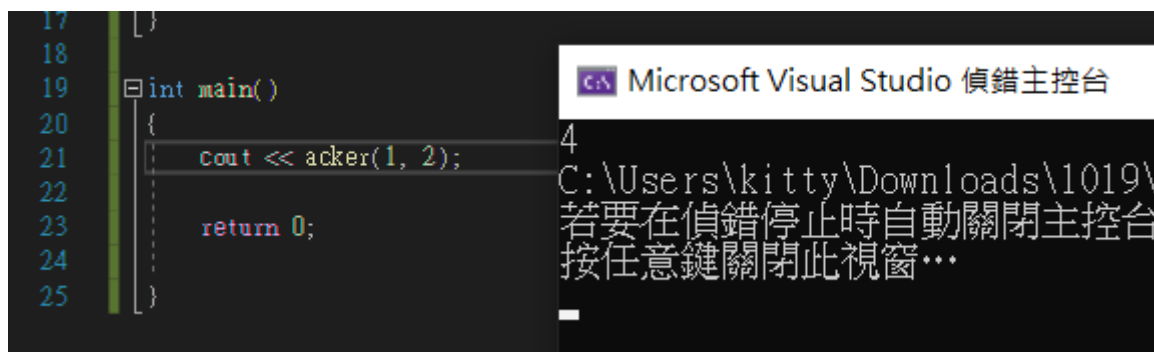
時間複雜度：
$$T(n) = O(2^n)$$
$$T(n) = O(2^n)$$

空間複雜度：
$$S(n) = O(n)$$
$$S(n) = O(n)$$

，這是一個隨著集合大小呈指數增長的複雜度。不過，這也是冪集算法的一個有趣之處，隨著集合變大，生成的子集數量急劇增多。

CHAPTER 4 測試與過程

問題 1 測試：



```
17 }
18
19 int main()
20 {
21     cout << acker(1, 2);
22
23     return 0;
24 }
25
```

Microsoft Visual Studio 偵錯主控台

4

C:\Users\kitty\Downloads\1019\

若要在偵錯停止時自動關閉主控台
按任意鍵關閉此視窗...

Figure4.1HW1.cpp

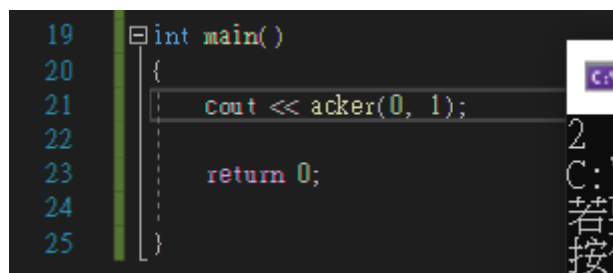
先用了A(1,2)來測試程式,
 $A(1,2)=A(0,A(1,1))$
 $A(1,1)=A(0,A(1,0))=A(0,2)=3$
 $A(0,3)=4$

這驗證了我們的遞迴函數能夠正確處理小規模的 Ackermann 函數運算。

以下改成A(0,1)=2。

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

(我覺得m或者n其中一個為0, 比較好寫)



```
19 int main()
20 {
21     cout << acker(0, 1);
22
23     return 0;
24 }
25
```

2

C:\V

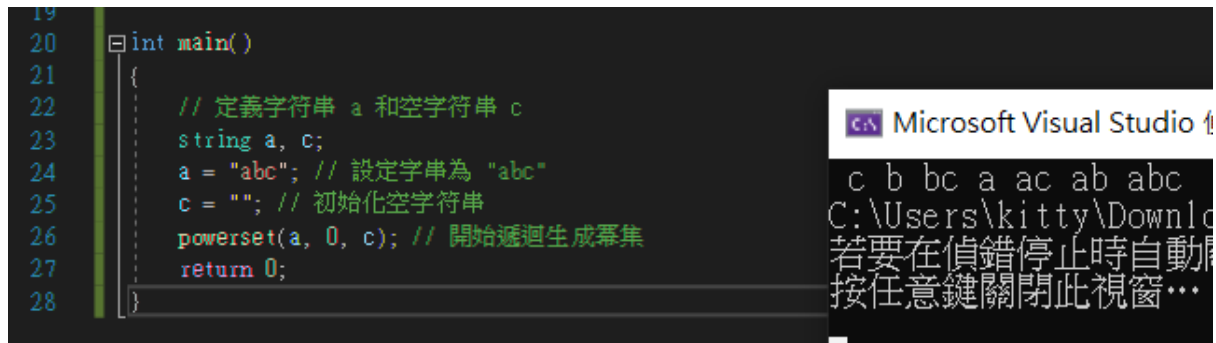
若

按

Figure4.2HW1.cpp

CHAPTER 4 測試與過程

問題 2 測試：



```
19
20 int main()
21 {
22     // 定義字符串 a 和空字符串 c
23     string a, c;
24     a = "abc"; // 設定字符串為 "abc"
25     c = ""; // 初始化空字符串
26     powerset(a, 0, c); // 開始遞迴生成冪集
27     return 0;
28 }
```

Microsoft Visual Studio 17.10.4

c b bc a ac ab abc
C:\Users\kitty\Downl
若要在偵錯停止時自動
按任意鍵關閉此視窗...

Figure4.3HW2.cpp

我使用了"a,b,c"，通過 powerset(a, 0, c) 遞迴生成冪集：

1. 空集("")
2. 單元素("a","b","c")
3. 雙元素("ab","bc","ac")
4. 全部("abc")

驗證通過。這證明了遞迴生成冪集的函數在這個測試範例下運行良好。

CHAPTER 5 申論

申論：遞迴的無限力量與挑戰

遞迴 (recursion) 是一個計算機科學中的強大工具，通過讓一個函數呼叫自己來解決問題，它表現出了驚人的解題能力。無論是像 Ackermann 函數這樣極具挑戰的數學遞迴問題，還是生成冪集這樣看似簡單但充滿數學深度的操作，遞迴都做得好。

Ackermann 函數的遞迴特性

Ackermann 函數是遞迴世界中的一個特例，因為它不是一個「簡單」的遞迴函數。其遞迴深度和計算複雜度隨著輸入參數 m, n 增加而呈現出爆炸性的增長。這樣的遞迴函數不僅挑戰了我們對計算複雜度的理解，也考驗了我們在編寫程式時的資源管理能力。當 m, n 稍微大一些時，這個函數的計算會耗費極多的記憶體和時間，因此在實際應用中，我們很少直接使用這類遞迴來解決具體問題，但它的學術價值和理論意義是無可替代的。Ackermann 函數的魅力不僅僅在於它的遞迴深度，還在於它揭示了遞迴的無限可能性。

冪集與遞迴的結合

冪集問題雖然在數學中相對簡單，但當我們利用遞迴來解決時，這個問題變得更加有趣。遞迴的核心概念是「拆解」問題，將一個大問題分解為若干個更小的子問題。在冪集生成的過程中，每次遞迴都需要做出兩個選擇：是否將當前元素加入子集中。這種「選擇」的過程恰恰就是冪集生成的精髓。遞迴的結構讓我們能夠輕鬆地列舉出所有可能的子集，這是一個在有限步驟內無限探索可能性的過程。

CHAPTER 6心得

心得: 遞迴如人生, 無窮無盡的選擇

遞迴的學習過程讓我深刻地感受到, 它就像人生一樣充滿了選擇和挑戰。每次遞迴呼叫時, 我們都在面臨一個新的選擇: 是繼續向前還是停下? 這與我們生活中的選擇何其相似! 正如在 Ackermann 函數中, 隨著遞迴層數的增加, 問題變得越來越複雜, 人生中的選擇也總是讓我們越來越難以抉擇。我們時常需要在生活的不同方面做出判斷, 或前進, 或退縮, 而每一次的決策都影響著未來的結果。

此外, 遞迴的強大在於它讓我們可以以簡單的規則解決看似無限複雜的問題。這提醒我, 無論我們面對多大的困難, 只要我們能夠冷靜分析, 將複雜的問題逐層拆解, 就一定能找到解決之道。

冪集問題則教會我另一個道理: 人生中每一個選擇都會產生不同的結果, 這些結果或許有時微小到難以察覺, 但當所有可能的選擇集合在一起時, 卻能構建出我們完整的人生。遞迴讓我們看到, 每一次選擇都會帶來一個新的分支, 而這些分支正是未來無限可能性的來源。

總結來說, 學習遞迴不僅增強了我的程式設計能力, 更讓我對生活中的選擇和挑戰有了全新的思考方式。正如遞迴無限循環一樣, 人生的每一刻也都是無限延伸的可能性, 只要我們勇於面對, 總能找到屬於自己的答案。