資料結構報告

梁詠琳

october 22,2024

CONTENTS

梁詠琳

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:

```cpp
int acker(int m, int n)
{
    // 當 m 等於 0，返回 n + 1
        if (m == 0)
                return n + 1;
        else if (n == 0) // 當 n 等於 0，遞迴計算 acker(m-1, 1)
        {
                return acker(m - 1, 1);
        }
        else // 否則，遞迴計算 acker(m-1, acker(m, n-1))
        {
                return acker(m - 1, acker(m, n - 1));
        }
}
```

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 *powerset* $(S) = \{(\,), (a), (b), (c), (a,b), (a,c), (b,c), (a,b,c)\}$. Write a recursive function to compute *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:

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

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：

```
#include <iostream>
using namespace std;

// 遞迴計算 Ackermann 函數
int acker(int m, int n)
{
    // 當 m 等於 0，返回 n + 1
        if (m == 0)
                return n + 1;
        else if (n == 0) // 當 n 等於 0，遞迴計算 acker(m-1, 1)
        {
                return acker(m - 1, 1);
        }
        else // 否則，遞迴計算 acker(m-1, acker(m, n-1))
        {
                return acker(m - 1, acker(m, n - 1));
        }
}

int main()
{
    // 測試 Ackermann 函數，輸出結果
        cout << acker(1, 2); // 計算 A(1, 2)

        return 0;
}
```

Figure2.1HW1.cpp

梁詠琳

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:

```cpp
#include <iostream>
#include <string>

using namespace std;

// 遞迴生成冪集的函式
int powerset(string a, int b, string c)
{
    // 當 b 等於字串長度時，輸出當前生成的子集
        if (b == a.size()) {
                cout << c << " "; // 輸出子集
                return 0; // 結束當前遞迴
        }
        // 遞迴不選擇當前字符
        powerset(a, b + 1, c);
        // 遞迴選擇當前字符
        powerset(a, b + 1, c + a[b]);
}

int main()
{
    // 定義字符串 a 和空字符串 c
        string a, c;
        a = "abc"; // 設定字串為 "abc"
        c = ""; // 初始化空字符串
        powerset(a, 0, c); // 開始遞迴生成冪集
        return 0;
}
```

Figure2.2HW2.cpp

梁詠琳

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(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。

梁詠琳

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

$$T(n) = O(2^n)$$

時間複雜度:
T(n)=O(2^n)

$$S(n) = O(n)$$

空間複雜度:
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。

梁詠琳

CHAPTER 4 測試與過程

問題 1 測試：

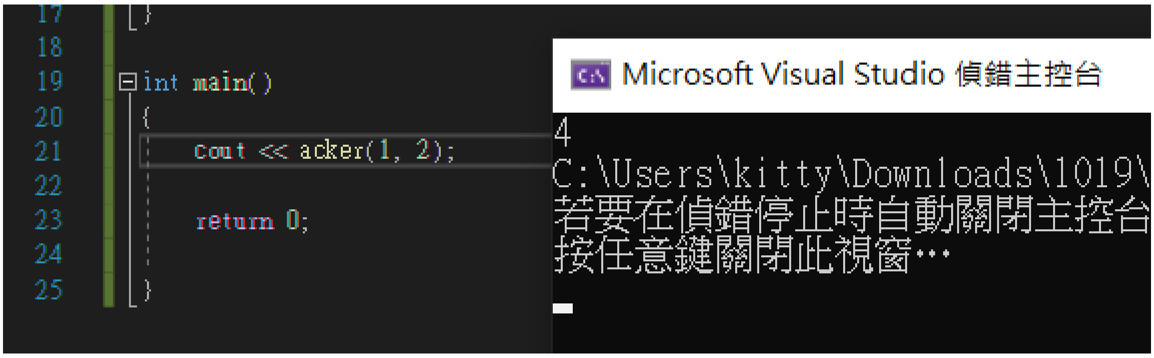

Figure4.1HW1.cpp

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)

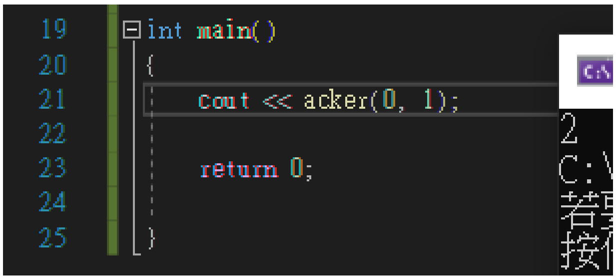

Figure4.2HW1.cpp

CHAPTER 4 測試與過程

問題 **2** 測試:

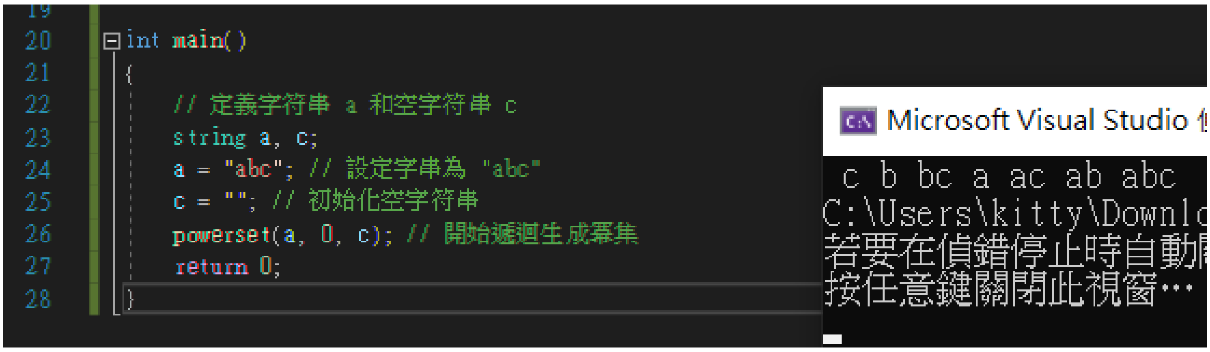

Figure4.3HW2.cpp

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，遞迴都做得到。

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。

梁詠琳

CHAPTER 6心得

心得:遞迴如人生，無窮無盡的選擇

Ackermann 函數讓我發現，看似簡單的程式碼，答案可能要算超難。

冪集問題則讓我變得很小心，一步一步把元素分好。

對我來說這種連電腦都有可能燒腦的程式十分少見，所以我覺得很讚。

梁詠琳