Work One

劉向榮

2024, 10 23

Problem 1:

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.

題目要求我使用兩種方法來實作 Ackermann 函數：一般遞迴方式和優化的方式

至於優化的方式我選擇使用矩陣來進行優化

Ackermann 函數以一般遞迴實作

建立一個函式，需要時就呼叫函式內有三個條件判斷式，就如上圖那般將三個條件列出或是呼叫下一個函式

矩陣優化 Ackermann 函數

簡單來說就是利用一個矩陣來儲存先前計算過的結果，避免重複計算。每次進行遞迴時，先檢查矩陣中是否已有結果，若有則直接回傳，若無則計算並存入矩陣。建立一個函式，需要時就呼叫函式內有三個條件判斷式，就如上圖那般將三個條件列出或是呼叫下一個函式

Problem 2:

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).

題目要求根據一個集合產生所有可能的子集。

子集合生成

對一個集合進行遞迴產生所有可能的子集。從第一個元素開始，遞迴地將每個元素加入子集並且回溯移除已加入的元素，繼續生成其他子集。最終輸出集合的所有子集。

我預先宣告了三個變數

vector<string> s;<- 宣告原始的集合 s，裡面存放著要生成子集的元素

vector<vector<string>> allsubsets; 這是一個二維向量，用來存儲所有生成的子集。每當生成一個完整的子集（nowSet），它就會被存入 allsubsets 中

vector<string> nowSet; 這是當前正在生成的子集。遞歸過程中，每次我們都會修改這個子集，逐步向它添加新元素或回溯刪除元素

除此之外，這邊我認為有兩個重點:

第一，元素可能是數字與字母單純的整數型態並不能容納字母，單單用字方式也顯得攏長，因此我使用了 auto 這個功能，可以叫系統自行判斷這是整數還是字元，然後逐字讀取，輸入例如 輸入 1 a b 3 c 就能將其判別為五個元素，

第二，要怎麼確定我生產出”所有”子集合，假設集合 s 為{a, b}，generateSubsets 函數的運行流程如下：

1. 初始狀態：
   - nowSet 為空。
   - index = 0，處理第一個元素。

2. 遞歸分支 1：選擇加入 a 到 nowSet 中，然後遞歸處理剩下的元素。
   - nowSet = {a}。
   - 接著遞歸處理 b，生成 {a, b}。
3. 遞歸分支 2：選擇不加入 a，直接遞歸處理剩下的元素。
   - nowwSet = {}。
   - 接著遞歸處理 b，生成 {b} 和 {}。

最終，所有子集會存入 allsubsets 中。

效能分析

1. 一般遞迴 **Ackermann** 函數
   - Ackermann 函數的遞迴深度極深，且計算量隨著 ***m*** 和 ***n*** 的增加呈現指數增長，導致效能瓶頸。由於重複計算較多，因此效能相對較低。計算次數可以透過變數 ***t*** 來跟蹤，通常計算次數會很大。
2. 矩陣優化 **Ackermann** 函數
   - 採用 記憶化，利用矩陣儲存先前計算過的結果，大大減少了重複遞迴的次數。對於大範圍的 m 和 n，效能明顯優於一般遞迴方法。時間複雜度也下降，隨著 m 和 n 的增加，優勢更加顯著，但基本上面對 m>4 以上的還是無力應對。
3. 集合子集生成
   - 子集生成的時間複雜度是 O(2^n)，因為一個集合有 n^2 種子集。對於小規模集合，這個計算量是可以接受的，但對於較大集合，生成子集的計算會明顯增加。

測試與驗證

1. **Ackermann** 函數測試
   - 測試了(3,1) =13 ，在兩種方式下進入遞迴的次數 一般比優化後的多了將近 70 幾的次數

2. 集合子集生成測試
    - 測試了一些簡單集合如 {a, b, c}，生成所有子集，驗證正確性。輸出結果如下：
        - { }
        - { a }
        - { b }
        - { c }
        - { a, b }
        - { a, c }
        - { b, c }
        - { a, b, c }
3. 驗證計算次數
    - 在兩種 Ackermann 函數實作中，利用計數器 t 來驗證函數執行的總次數。結果表明，矩陣優化方法所需的計算次數遠小於一般遞迴。

申論及心得

Ackermann 函數展示了計算科學中的遞迴深度與複雜度問題。一般遞迴實作會導致大量重複計算，而矩陣優化則很好地解決了這一問題。這表明在處理高深度遞迴問題時，記憶化(Memoization) 等技術是非常有效的。由於 Ackermann 函數的極端增長性，計算大的 m 和 n 是非常困難的，因此優化方案在實際應用中意義重大。

集合子集生成的問題也展示了回溯法的應用，通過遞迴生成所有子集，展示了回溯法的靈活性與有效性。這個問題可以進一步應用於解決排列組合問題，具有很好的應用價值。

在整個編程過程中，瞭解到了遞迴函數的特性，以及如何運用優化技術來提升演算法效能。通過這次實作，不僅加深了對 Ackermann 函數的理解，也學會了如何處理複雜遞迴問題，並有效提升程式的效能。