資料結構報告

梁詠琳

November 19,2024

資料結構 課堂作業 November 22,2024

CONTENTS

梁詠琳

-CHAPTER 1 解題說明

問題 **1**：實作 **Polynomial** 類別

根據 **Figure1** 和 **Figure2** 提供的抽象資料型別（**ADT**）和私有數據成員，實現一個 **Polynomial 類別**。

Polynomial 類別代表多項式，它需要有適當的成員來儲存多項式的係數與指數。

舉例：$3x^2 + 2x + 1$

它的指數：[2, 1, 0]

而係數 ：[3, 2, 1]

設計目標

1. 使用私有成員儲存係數和指數。
2. 提供接口來添加、刪除或操作多項式項目。
3. 支援多項式的輸入與輸出。

class *Polynomial* {

// $p(x) = a_0 x^{e_0} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $<e_i, a_i>$,

// where a_i is a nonzero **float** coefficient and e_i is a non-negative integer exponent.

public:

Polynomial();

// Construct the polynomial $p(x) = 0$.

Polynomial Add(*Polynomial poly*);

// Return the sum of the polynomials ***this** and *poly*.

Polynomial Mult(*Polynomial poly*);

// Return the product of the polynomials ***this** and *poly*.

float *Eval*(**float** *f*);

// Evaluate the polynomial ***this** at *f* and return the result.

};

Figure 1. Abstract data type of *Polynomial* class

資料結構　　　　課堂作業　　　　November 22,2024

要完成這份作業，我們需要實現一個 **Polynomial** 類別，並提供多項式的輸入、輸出和基本運算功能（如加法、減法等）。以下是詳細步驟和思路：

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

梁詠琳

CHAPTER 1 解題說明

問題 **2**：撰寫 **C++** 函數來輸入和輸出多項式

這些函數需要支持多項式的輸入與輸出。

函數需要重載 **<<** 和 **>>** 運算符。

需要設計兩個重載運算符：

>> 運算符：用於從輸入(例如鍵盤或檔案)讀取多項式資料。

<< 運算符：用於輸出多項式的格式化表示。

```
class Polynomial ; // forward declaration

class Term {
friend Polynomial;
private:
    float coef;   // coefficient
    int exp;      // exponent
};


The private data members of Polynomial are defined as follows:

private:
      Term *termArray;   // array of nonzero terms
      int capacity;      // size of termArray
      int terms;         // number of nonzero terms
```

Figure 2. The private data members of *Polynomial* class

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

資料結構 課堂作業 November 22,2024

-CHAPTER 2　演算法設計與實作

問題**1**：設計了基本的多項式類別，實現了添加和顯示功能。
以下是程式碼的設計：

問題1 的程式應包含：

Polynomial 類別的基本結構。

添加和顯示多項式項目的方法（如 addTerm 和 display）。

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

// 定義 Polynomial 類別
class Polynomial {
private:
    vector<int> coefficients; // 儲存係數
    vector<int> exponents;    // 儲存指數

public:
    Polynomial() {}

    // 添加新項目
    void addTerm(int coeff, int exp) {
        coefficients.push_back(coeff);
        exponents.push_back(exp);
    }

    // 輸出多項式
    void display() const {
        for (size_t i = 0; i < coefficients.size(); ++i) {
            cout << coefficients[i] << "x^" << exponents[i];
            if (i < coefficients.size() - 1) cout << " + ";
        }
        cout << endl;
    }
};

int main() {
    Polynomial p;
    p.addTerm(3, 2); // 添加 3x^2
    p.addTerm(2, 1); // 添加 2x
    p.addTerm(1, 0); // 添加常數 1

    cout << "多項式為：";
    p.display();
    return 0;
}
```

Figure2.1HW1.cpp

梁詠琳

CHAPTER 2 演算法設計與實作

問題**2**：實現多項式的輸入和輸出，並重載運算符。
以下是程式碼的設計：
設計 Polynomial 類別的完整功能：

- 支持添加多項式項目。
- 實現多項式的加法、減法(還有乘法)等基本操作。
- 支持輸入與輸出多項式。

基本功能：

- 添加項目（addTerm 方法）。
- 顯示多項式（display 方法）。

輸入與輸出：

- 使用重載的 **>>** 和 **<<** 運算符。

多項式加法：

- 將兩個多項式相加，合併同類項。

多項式減法：

- 將一個多項式從另一個多項式中減去。

多項式乘法：

- 計算兩個多項式的乘積。

資料結構 課堂作業 November 22,2024

HW2_2.cpp

HW2 (全域範圍)

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

// 定義 Polynomial 類別
class Polynomial {
private:
    vector<int> coefficients; // 儲存係數
    vector<int> exponents;    // 儲存指數

    // 合併同類項
    void simplify() {
        for (size_t i = 0; i < exponents.size(); ++i) {
            for (size_t j = i + 1; j < exponents.size(); ++j) {
                if (exponents[i] == exponents[j]) {
                    coefficients[i] += coefficients[j];
                    coefficients.erase(coefficients.begin() + j);
                    exponents.erase(exponents.begin() + j);
                    --j;
                }
            }
        }
    }

public:
    Polynomial() {}

    // 添加新項目
    void addTerm(int coeff, int exp) {
        coefficients.push_back(coeff);
        exponents.push_back(exp);
    }

    // 重載輸入運算符 >>
    friend istream& operator>>(istream& in, Polynomial& poly) {
        int coeff, exp;
        cout << "輸入多項式項目 (係數 指數) ，輸入 -1 -1 結束：" << endl;
        while (true) {
            in >> coeff >> exp;
            if (coeff == -1 && exp == -1) break;
            poly.addTerm(coeff, exp);
        }
        poly.simplify(); // 自動合併同類項
        return in;
    }

```

梁詠琳

```cpp
    }

    // 重載輸出運算符 <<
    friend ostream& operator<<(ostream& out, const Polynomial& poly) {
        for (size_t i = 0; i < poly.coefficients.size(); ++i) {
            if (poly.coefficients[i] != 0) {
                out << poly.coefficients[i] << "x^" << poly.exponents[i];
                if (i < poly.coefficients.size() - 1) out << " + ";
            }
        }
        return out;
    }

    // 多項式加法
    Polynomial operator+(const Polynomial& other) const {
        Polynomial result = *this;
        for (size_t i = 0; i < other.coefficients.size(); ++i) {
            result.addTerm(other.coefficients[i], other.exponents[i]);
        }
        result.simplify(); // 合併同類項
        return result;
    }

    // 多項式減法
    Polynomial operator-(const Polynomial& other) const {
        Polynomial result = *this;
        for (size_t i = 0; i < other.coefficients.size(); ++i) {
            result.addTerm(-other.coefficients[i], other.exponents[i]);
        }
        result.simplify(); // 合併同類項
        return result;
    }
```

HW2_2.cpp

HW2 (全域範圍)

```cpp

    // 多項式乘法
    Polynomial operator*(const Polynomial& other) const {
        Polynomial result;
        for (size_t i = 0; i < coefficients.size(); ++i) {
            for (size_t j = 0; j < other.coefficients.size(); ++j) {
                int newCoeff = coefficients[i] * other.coefficients[j];
                int newExp = exponents[i] + other.exponents[j];
                result.addTerm(newCoeff, newExp);
            }
        }
        result.simplify(); // 合併同類項
        return result;
    }
};

int main() {
    Polynomial p1, p2;

    // 輸入第一個多項式
    cout << "輸入第一個多項式：" << endl;
    cin >> p1;

    // 輸入第二個多項式
    cout << "輸入第二個多項式：" << endl;
    cin >> p2;

    // 輸出兩個多項式
    cout << "第一個多項式：" << p1 << endl;
    cout << "第二個多項式：" << p2 << endl;

    // 加法運算
    Polynomial sum = p1 + p2;
    cout << "加法結果：" << sum << endl;

    // 減法運算
    Polynomial diff = p1 - p2;
    cout << "減法結果：" << diff << endl;

    // 乘法運算
    Polynomial prod = p1 * p2;
    cout << "乘法結果：" << prod << endl;

    return 0;
}
```

Figure2&3&4.2HW2.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。

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)

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

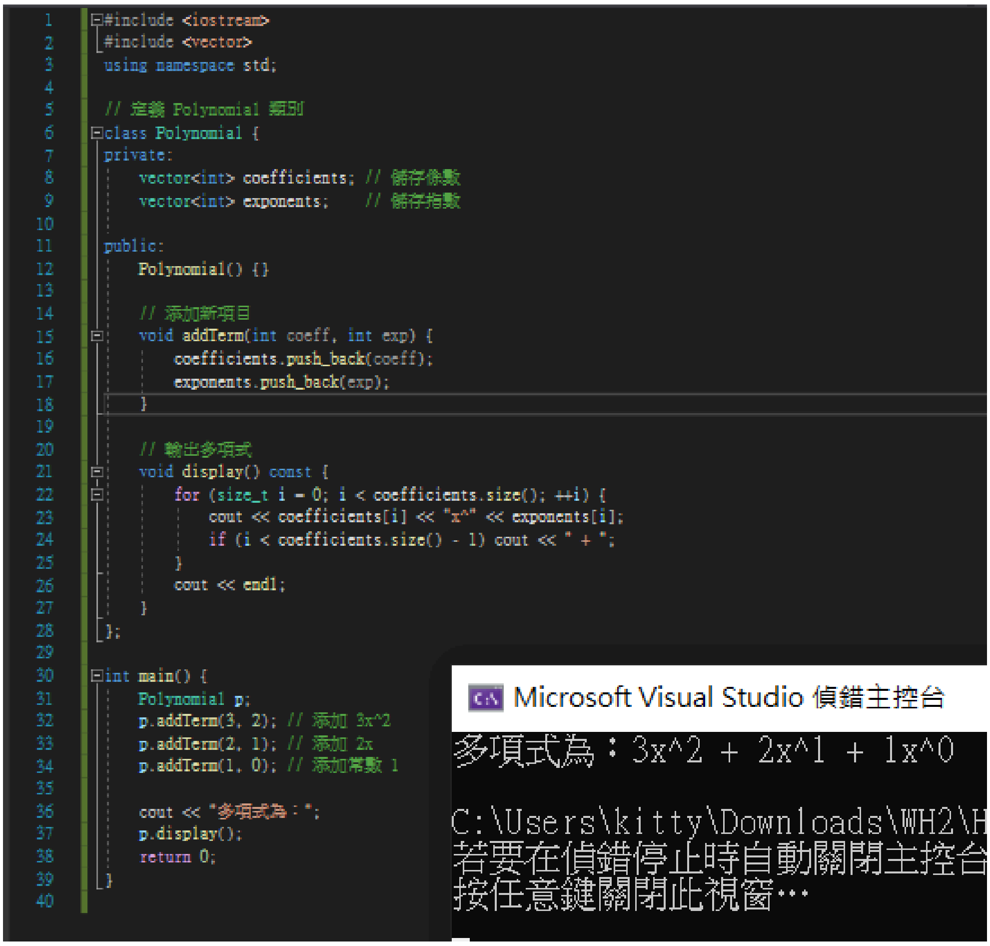
空間複雜度：
S(n)=O(n)

，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

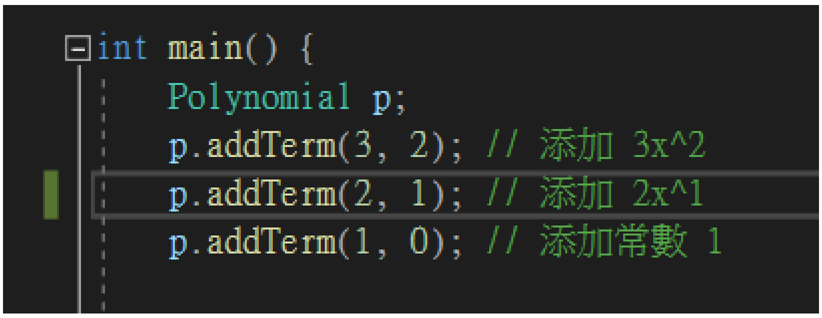
問題 **1** 測試:



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

// 定義 Polynomial 類別
class Polynomial {
private:
    vector<int> coefficients; // 儲存係數
    vector<int> exponents;    // 儲存指數

public:
    Polynomial() {}

    // 添加新項目
    void addTerm(int coeff, int exp) {
        coefficients.push_back(coeff);
        exponents.push_back(exp);
    }

    // 輸出多項式
    void display() const {
        for (size_t i = 0; i < coefficients.size(); ++i) {
            cout << coefficients[i] << "x^" << exponents[i];
            if (i < coefficients.size() - 1) cout << " + ";
        }
        cout << endl;
    }
};

int main() {
    Polynomial p;
    p.addTerm(3, 2); // 添加 3x^2
    p.addTerm(2, 1); // 添加 2x
    p.addTerm(1, 0); // 添加常數 1

    cout << "多項式為：";
    p.display();
    return 0;
}
```


Figure4.1HW1.cpp

以下我拿了 `3x^2 + 2x^1 + 1x^0` 去測試。有成功顯示答案。



```cpp
int main() {
    Polynomial p;
    p.addTerm(3, 2); // 添加 3x^2
    p.addTerm(2, 1); // 添加 2x^1
    p.addTerm(1, 0); // 添加常數 1
```

Figure4.2HW1.cpp

資料結構 課堂作業 November 22,2024

CHAPTER 4 測試與過程

問題 **2** 測試：

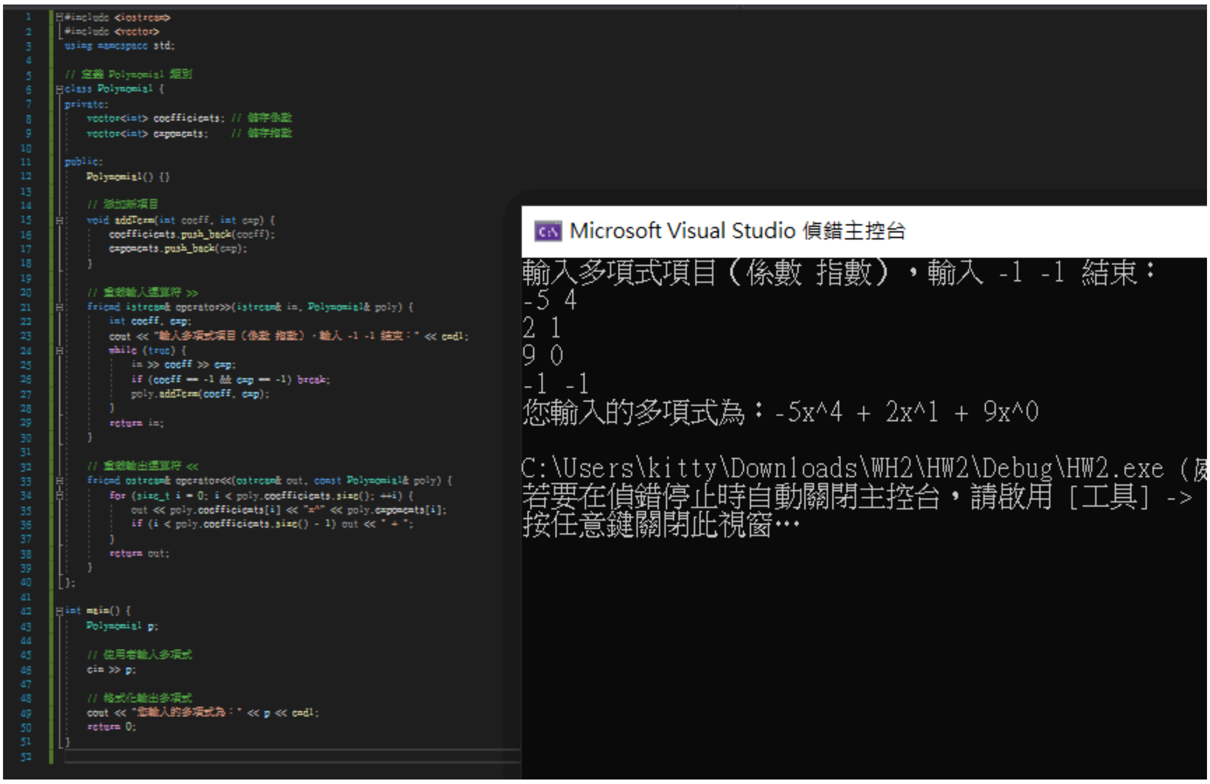


Figure4.3HW2.cpp

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。

梁詠琳

资料結構　　　　　　　　　課堂作業　　　　　November 22,2024

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

資料結構　　　　　　　　課堂作業　　　　　November 22,2024

申論：多項式 ADT 與多載的趣味實作之旅 在程式設計的世界中，多項式（Polynomial）這個數學概念，總是與我們如影隨形。從學校數學課堂的「 3 𝑥 2 + 2 𝑥 + 1 3x 2 +2x+1」開始，到今天我們在 C++ 程式碼中實現它的「數位化身」，這是一段數學與程式的浪漫交織。而這次作業，讓我們踏上了一段充滿創意與挑戰的旅程，實現了多項式的抽象資料型態（ADT）與運算符多載。接下來，我會從技術實現到其趣味性，帶你探索這段奇妙的冒險。多項式 ADT：抽象的力量 首先，什麼是 ADT？如果把程式比作樂團，那 ADT 就是指揮，決定了整個多項式資料結構「能做什麼」，卻不拘泥於「怎麼做」。我們可以透過 ADT 對多項式進行資料抽象，把多項式拆解為「項」的集合，每一項有兩個屬性：係數（coefficient）與指數（exponent）。

接著，我們利用一組操作來操控這些項，例如添加新項目、顯示多項式內容，甚至進行多項式運算。這不僅僅是技術上的挑戰，更是程式設計思維的體現。我們選擇用陣列（或向量）來儲存項，類似於用書架來整理書本，而不是把它們隨便堆在地上。多項式 ADT 讓程式具有了條理性與可擴展性，比如日後我們要加入多項式加法、乘法，只需在既有架構上擴展即可。多載：讓程式更自然的魔法 接著來說說多載（Overloading）。

如果說 ADT 是多項式的靈魂，那多載就是讓它擁有靈活身體的魔法師。運算符多載讓我們可以重定義 C++ 的運算符行為，例如輸出運算符 << 與輸入運算符 >>，從而讓我們能用「直覺式」的方法與多項式互動。比如輸入多項式時，我們不需要額外撰寫難懂的函式名稱，直接使用 cin >> p1，程式會自動幫我們完成每一項的係數與指數的讀取；而輸出多項式時，用 cout << p1 就可以讓多項式以數學公式的形式呈現出來。這樣的程式設計，不僅讓使用者感覺流暢，更讓程式與數學的表達方式融為一體，真正實現了程式的「人性化」。實作中的趣味與挑戰 實作過程中，我們仿佛在組裝一部多項式運算的機器。每個程式碼片段都是一個齒輪，從 addTerm() 的基本操作到 << 的格式化輸出，最後拼裝成一個精密的結構。而這裡最大的樂趣莫過於挑戰遞迴函式，讓多項式能通過遞迴來進行求值。當你看到 3 𝑥 2 + 2 𝑥 + 1 3x 2 +2x+1 在 𝑥 = 2 x=2 時自動算出 17 17，這一刻，程式的威力讓人不禁拍案叫絕。 還記得剛開始的輸出格式嗎？我們試圖讓它更「數學化」，例如省略常數項的 ^0 和一次項的 ^1，這不僅是對程式細節的打磨，更是一種對使用者體驗的追求。畢竟，誰不希望自己的程式既強大又美觀呢？多項式的數學與程式魅力 總結來說，這次作業不僅僅是一次程式設計的實作，更像是一場與數學的跨界對話。多項式從黑板上的數學公式變成程式中的資料結構，ADTs 為它定義了「骨架」，多載為它注入了「靈魂」，遞迴函式則為它的運算賦予了「智慧」。

這段旅程讓我深刻體會到，程式設計的魅力不僅在於解決問題，更在於將抽象的概念具體化，讓枯燥的數學變得生動起來。未來的程式道路上，我將帶著這份熱情，繼續探尋數學與程式的交匯點，創造出更多有趣而實用的程式！

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