資料結構作業 2

賴明賢
41043244

解題說明

為了實現多項式類別 Polynomial，我們首先需要設計一個表示單項式的類別 Term，其中包含係數和指數。然後，我們在 Polynomial 類別中使用一個 Term 陣列來存儲多項式的各項。對於加法和乘法運算，我們需要遍歷和操作這些陣列中的項，並將結果存儲在新的多項式對象中。

舉例說明： 假設有兩個多項式：

- a(x)=3x^2+2x+1
- b(x)=x^3+4x+5

我們的加法運算會將相同指數的項進行相加，得到新多項式 c(x)=x^3+3x^2+6x+6。而乘法運算會通過展開所有項，得到結果 d(x)=3x^5+2x^4+13x^3+23x^2+14x+5。

參考 polynomial - 演算法筆記 (ntnu.edu.tw)

2. Algorithm Design & Programming - 40%

```cpp
#include <iostream>
#include <algorithm>
#include <cmath>
using namespace std;

class Polynomial;

class Term {
    friend Polynomial;
public:
    float coef;  // 係數
    int exp;     // 指數
};

class Polynomial {
    friend ostream& operator<<(ostream& o, const Polynomial& poly);

public:
    Polynomial(); // 建構子
    Polynomial(const Polynomial& poly); // 複製建構子
    ~Polynomial(); // 解構子
    Polynomial add(const Polynomial& poly); // 加法運算
    Polynomial mult(const Polynomial& poly); // 乘法運算
    float Eval(float x); // 代入 x 計算多項式的值
    void NewTerm(float coef, int exp); // 新增項
private:
    void insertTerm(const Term& term); // 插入項至多項式
private:
    Term* termArray; // 用於存儲多項式項的陣列
    int capacity; // 陣列的目前容量
    int terms; // 多項式中目前的項數
};

Polynomial::Polynomial() {
    this->terms = 0;
    this->capacity = 10;
    termArray = new Term[this->capacity];
```

```
}

Polynomial::Polynomial(const Polynomial& b) {
    this->terms = b.terms;
    this->capacity = b.capacity;
    termArray = new Term[this->capacity];
    copy(b.termArray, b.termArray + b.terms, termArray);
}

Polynomial::~Polynomial() {
    delete[] termArray; // 釋放記憶體
}

Polynomial Polynomial::add(const Polynomial& b) {
    Polynomial c;
    int ac = 0;
    int bc = 0;
    while (ac < terms && bc < b.terms) {
        if (termArray[ac].exp == b.termArray[bc].exp) {
            float coef = termArray[ac].coef + b.termArray[bc].coef;
            if (coef != 0) c.NewTerm(coef, termArray[ac].exp);
            ac++; bc++;
        }
        else if (termArray[ac].exp < b.termArray[bc].exp) {
            c.NewTerm(b.termArray[bc].coef, b.termArray[bc].exp);
            bc++;
        }
        else {
            c.NewTerm(termArray[ac].coef, termArray[ac].exp);
            ac++;
        }
    }
    while (ac < terms) {
        c.NewTerm(termArray[ac].coef, termArray[ac].exp);
        ac++;
    }
```

```
    while (bc < b.terms) {
        c.NewTerm(b.termArray[bc].coef, b.termArray[bc].exp);
        bc++;
    }
    return c;
}

Polynomial Polynomial::mult(const Polynomial& b) {
    Polynomial c;
    for (int i = 0; i < terms; i++) {
        for (int j = 0; j < b.terms; j++) {
            float coef = termArray[i].coef * b.termArray[j].coef;
            int exp = termArray[i].exp + b.termArray[j].exp;
            c.NewTerm(coef, exp);
        }
    }
    return c;
}

void Polynomial::NewTerm(float coef, int exp) {
    if (terms == capacity) {
        capacity *= 2;
        Term* tmp = new Term[capacity];
        copy(termArray, termArray + terms, tmp);
        delete[] termArray;
        termArray = tmp;
    }
    Term ATerm;
    ATerm.coef = coef;
    ATerm.exp = exp;
    insertTerm(ATerm);
}

void Polynomial::insertTerm(const Term& term) {
    int i;
    for (i = 0; i < terms && term.exp < termArray[i].exp; i++) {
    }
    if (i < terms && term.exp == termArray[i].exp) {
        termArray[i].coef += term.coef;
        if (termArray[i].coef == 0) {
            for (int j = i; j < terms - 1; j++) {
                termArray[j] = termArray[j + 1];
            }
            terms--;
        }
    } else {
        for (int j = terms; j > i; j--) {
            termArray[j] = termArray[j - 1];
        }
        termArray[i] = term;
        terms++;
    }
}

float Polynomial::Eval(float x) {
    float sum = 0;
    for (int i = 0; i < terms; i++) {
        sum += termArray[i].coef * pow(x, termArray[i].exp);
    }
    return sum;
}

ostream& operator<<(ostream& o, const Polynomial& poly) {
    for (int i = 0; i < poly.terms - 1; i++) {
        o << poly.termArray[i].coef << "x^" << poly.termArray[i].exp << " + ";
    }
    o << poly.termArray[poly.terms - 1].coef << "x^" << poly.termArray[poly.terms - 1].exp;
    return o;
}
```

```cpp

int main() {
    Polynomial a;
    int coef;
    float exp;
    cout << "請輸入多項式a的係數與指數（如 3 2 表示 3x^2，輸入 0 0 結束輸入）: " << endl;
    cin >> coef >> exp;
    while (coef != 0 || exp != 0) {
        a.NewTerm(coef, exp);
        cin >> coef >> exp;
    }

    Polynomial b;
    cout << "請輸入多項式b的係數與指數（如 3 2 表示 3x^2，輸入 0 0 結束輸入）: " << endl;
    cin >> coef >> exp;
    while (coef != 0 || exp != 0) {
        b.NewTerm(coef, exp);
        cin >> coef >> exp;
    }

    cout << a << " + " << b << " = " << a.add(b) << endl;
    cout << a << " * " << b << " = " << a.mult(b) << endl;

    cout << "請輸入欲求值的x: " << endl;
    float x;
    cin >> x;
    cout << "在x=" << x << "時，a+b的值為: " << a.add(b).Eval(x) << endl;

    return 0;
}

```

效能分析

1. add 函式 (多項式相加)

- **時間複雜度**: O(m + n)，其中 m 和 n 分別是兩個多項式的項數。這是因為相加操作需要逐項遍歷兩個多項式的所有項。
- **空間複雜度**: O(m + n)，因為結果多項式最多包含 m + n 項，儲存在新的多項式中。

2. mult 函式 (多項式相乘)

- **時間複雜度**: O(m * n)，因為每個多項式中的每一項都需要與另一個多項式中的每一項相乘，這是一個雙重迴圈。
- **空間複雜度**: 最壞情況下是 O(m * n)，因為結果多項式最多會有 m * n 項，雖然合併同類項後實際項數可能會少於此數值。

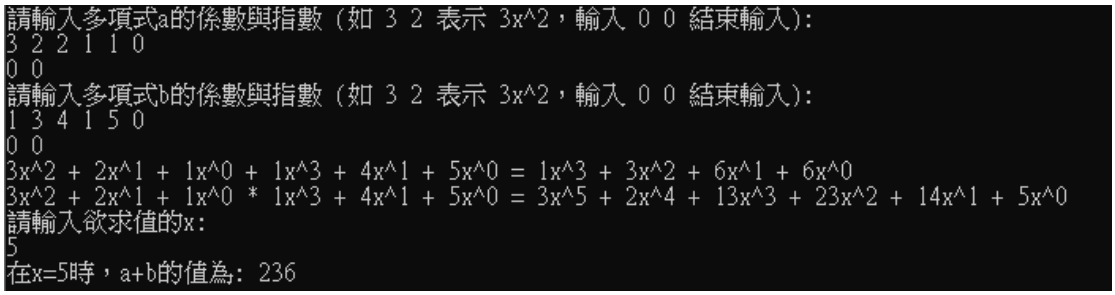
4. 空間使用量 (一般情況下)

- 程式在操作過程中主要使用的空間來自於多項式的儲存（Term 陣列）和結果多項式的儲存，因此在最壞情況下的空間複雜度會依賴於這些操作結果。

5. 整體空間複雜度

- 在多項式的操作中，對於兩個多項式各自的項數為 m 和 n，最壞情況下的空間複雜度會是 O(m + n) (相加) 或 O(m * n) (相乘)，具體取決於操作的種類。

測試與驗證(Testing and Proving)



心得討論:

由於是第二次修這堂課，所以做這項作業時不像第一次一樣毫無頭緒，這次比較快就做出來了，有參考 polynomial - 演算法筆記 (ntnu.edu.tw)此網頁。
多項式的相加與相乘涉及到對不同項的處理，尤其是在處理不同指數的項時，需要仔細考慮如何有效地組合和簡化。這種運算的複雜度會隨著多項式項數的增加而顯著增大。
對於大規模多項式，直接使用簡單的演算法（如雙重迴圈進行乘法）會導致運行時間顯著增加，這也凸顯出選擇適當的資料結構和演算法來處理數據的重要性。。