

问题1：稀疏多项式求导

1. 独立完成

算法设计思想

采用循环链表存储稀疏多项式。逐项遍历，若指数为 0（常数项），则删除该节点；否则执行求导操作：系数乘指数，指数减一。

代码 (C++)

```
void differentiate(PolyNode* head) {
    if (!head || head->next == head) return;

    PolyNode* trail = head;
    PolyNode* current = head->next;

    while (current != head) {
        if (current->exp == 0) {
            trail->next = current->next;
            delete current;
            current = trail->next;
        } else {
            current->coef *= current->exp;
            current->exp -= 1;
            trail = current;
            current = current->next;
        }
    }
}
```

复杂度分析

时间复杂度 $O(n)$ ，空间复杂度 $O(1)$ 。

2. LLM 完成

用户提问

用 C++ 实现循环链表存储稀疏多项式，并写一个函数就地完成求导。

模型回答

```
void derivative(PolyNode* head) {
    if (!head || head->next == head) return;
    PolyNode* prev = head;
    PolyNode* curr = head->next;
    while (curr != head) {
        if (curr->exp == 0) {
            prev->next = curr->next;
        }
    }
}
```

```
        delete curr;
        curr = prev->next;
    } else {
        curr->coef *= curr->exp;
        curr->exp--;
        prev = curr;
        curr = curr->next;
    }
}
}
```

复杂度：O(n) 时间，O(1) 空间。

3. 对比分析

两者思路 and 实现几乎一致，算法效率相同。

问题2：按指数奇偶分解稀疏多项式

1. 独立完成

算法设计思想

遍历多项式链表，将指数为奇数的项放入 oddList，偶数的项放入 evenList，最后形成两个循环链表。

代码 (C++)

```
void splitPolynomial(PolyNode* head, PolyNode*& oddList, PolyNode*&
evenList) {
    oddList = new PolyNode(); oddList->next = oddList;
    evenList = new PolyNode(); evenList->next = evenList;

    PolyNode* oddTail = oddList;
    PolyNode* evenTail = evenList;
    PolyNode* current = head->next;

    while (current != head) {
        PolyNode* next = current->next;
        if (current->exp % 2 != 0) {
            oddTail = oddTail->next = current;
        } else {
            evenTail = evenTail->next = current;
        }
        current = next;
    }

    oddTail->next = oddList;
    evenTail->next = evenList;
    head->next = head;
}
```

复杂度分析

时间复杂度 $O(n)$ ，空间复杂度 $O(1)$ 。

2. LLM 完成

用户提问

设计一个算法，将稀疏多项式链表按指数奇偶性拆分为两部分。

模型回答

```
void splitPoly(PolyNode* head, PolyNode*& odd, PolyNode*& even) {
    odd = new PolyNode(); odd->next = odd;
    even = new PolyNode(); even->next = even;
    PolyNode* cur = head->next;
    while (cur != head) {
        PolyNode* nxt = cur->next;
        if (cur->exp % 2) {
            cur->next = odd->next;
            odd->next = cur;
        } else {
            cur->next = even->next;
            even->next = cur;
        }
        cur = nxt;
    }
}
```

复杂度： $O(n)$ 时间， $O(1)$ 空间。

3. 对比分析

两种写法核心逻辑相同。我使用尾插法保持原有顺序，LLM 采用头插法，导致最终顺序可能不同，但正确性不影响。

问题3：大整数加法

1. 独立完成

算法设计思想

使用链表存储大整数，每个节点表示一位数字，低位在前。逐位相加并处理进位，最终构建新链表作为结果。

代码 (C++)

```
BigInt add(const BigInt& rhs) const {
    BigInt result("0");
    result._clear();

    BigIntNode *p1 = this->head, *p2 = rhs.head, *res_tail = nullptr;
    int carry = 0;

    while (p1 || p2 || carry) {
        int sum = (p1 ? p1->digit : 0) + (p2 ? p2->digit : 0) + carry;
        BigIntNode* newNode = new BigIntNode{sum % 10, nullptr};
        carry = sum / 10;
        if (!result.head) {
            result.head = res_tail = newNode;
        } else {
            res_tail = res_tail->next = newNode;
        }
        if (p1) p1 = p1->next;
        if (p2) p2 = p2->next;
    }
    return result;
}
```

复杂度分析

时间复杂度 $O(\max(m, n))$ ，空间复杂度 $O(\max(m, n))$ 。

2. LLM 完成

用户提问

用链表存储大整数，实现加法运算。

模型回答

```
BigInt add(const BigInt& other) const {
    BigInt result;
    ListNode* tail = nullptr;
    int carry = 0;
    ListNode *p = this->head, *q = other.head;
    while (p || q || carry) {
        int sum = (p ? p->digit : 0) + (q ? q->digit : 0) + carry;
        carry = sum / 10;
        ListNode* node = new ListNode{sum % 10, nullptr};
        if (!result.head) result.head = tail = node;
        else tail = tail->next = node;
        if (p) p = p->next;
        if (q) q = q->next;
    }
    return result;
}
```

复杂度： $O(\max(m,n))$ 时间， $O(\max(m,n))$ 空间。

3. 对比分析

两者几乎相同，主要区别在封装方式和成员变量命名。我代码更贴合课堂要求，LLM 更偏向工程化。

问题4：查找两个链表的共同后缀

1. 独立完成

算法设计思想

先分别计算两个链表的长度，让长链表先走差值步数，然后同步前进，直到找到相同节点或走到末尾。

代码 (C++)

```
ListNode* findIntersection(ListNode* headA, ListNode* headB) {
    int lenA = 0, lenB = 0;
    for (ListNode* p = headA->next; p; p = p->next) lenA++;
    for (ListNode* p = headB->next; p; p = p->next) lenB++;

    ListNode *ptrA = headA->next, *ptrB = headB->next;
    if (lenA > lenB) {
        for (int i = 0; i < lenA - lenB; ++i) ptrA = ptrA->next;
    } else {
        for (int i = 0; i < lenB - lenA; ++i) ptrB = ptrB->next;
    }

    while (ptrA != ptrB) {
        ptrA = ptrA->next;
        ptrB = ptrB->next;
    }
    return ptrA;
}
```

复杂度分析

时间复杂度 $O(m+n)$ ，空间复杂度 $O(1)$ 。

2. LLM 完成

用户提问

给定两个链表，找出它们的公共后缀部分。

模型回答

```
ListNode* getIntersection(ListNode* a, ListNode* b) {
    int lenA = 0, lenB = 0;
    for (auto p = a; p; p = p->next) lenA++;
    for (auto p = b; p; p = p->next) lenB++;
    while (lenA > lenB) { a = a->next; lenA--; }
    while (lenB > lenA) { b = b->next; lenB--; }
    while (a && b && a != b) {
        a = a->next;
        b = b->next;
    }
    return a;
}
```

复杂度：O(m+n) 时间，O(1) 空间。

3. 对比分析

两者完全一致，唯一差异是是否保留头结点。逻辑和效率相同。

附录：C++完整代码实现

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cmath>

// 为节点类型定义别名，减少代码冗余
using PolyNode = struct PolyNode;
using BigIntNode = struct BigIntNode;
using ListNode = struct ListNode;

/*
 * 问题 1 & 2：稀疏多项式
 */
struct PolyNode {
    float coef = 0.0f;
    int exp = -1;
    PolyNode* next = nullptr;
};

// 在多项式尾部添加一项
void appendTerm(PolyNode* head, float coef, int exp) {
    PolyNode* tail = head;
    while (tail->next != head) {
        tail = tail->next;
    }
    tail->next = new PolyNode{coef, exp, head};
}
```

```
}

// 格式化打印多项式, 处理正负号和系数为1的情况
void printPolynomial(const PolyNode* head) {
    if (!head || head->next == head) {
        std::cout << "0";
        return;
    }
    const PolyNode* iter = head->next;
    bool isFirstTerm = true;
    while (iter != head) {
        if (!isFirstTerm) {
            if (iter->coef > 0) std::cout << " + ";
            else std::cout << " - ";
        } else if (iter->coef < 0) {
            std::cout << "-";
        }

        float absCoef = std::abs(iter->coef);
        if (absCoef != 1.0f || iter->exp == 0) {
            std::cout << absCoef;
        }

        if (iter->exp > 0) {
            std::cout << "x";
            if (iter->exp > 1) std::cout << "^" << iter->exp;
        }

        isFirstTerm = false;
        iter = iter->next;
    }
}

// 问题1: 就地求导
void differentiate(PolyNode* head) {
    if (!head || head->next == head) return;

    PolyNode* trail = head; // trail 指针始终指向 current 的前一个节点
    PolyNode* current = head->next;

    while (current != head) {
        if (current->exp == 0) { // 常数项求导为0, 删除节点
            trail->next = current->next;
            delete current;
            current = trail->next;
            // trail 指针不动, 因为它已经是新 current 的前驱
        } else {
            current->coef *= current->exp;
            current->exp -= 1;
            trail = current;
            current = current->next;
        }
    }
}
```

```
// 问题2: 按指数奇偶拆分链表
void splitPolynomial(PolyNode* head, PolyNode*& oddList, PolyNode*&
evenList) {
    oddList = new PolyNode();
    oddList->next = oddList;
    evenList = new PolyNode();
    evenList->next = evenList;

    PolyNode* oddTail = oddList;
    PolyNode* evenTail = evenList;
    PolyNode* current = head->next;

    while (current != head) {
        PolyNode* next = current->next; // 必须先保存下一个节点, 否则链会断
        if (current->exp % 2 != 0) {
            oddTail = oddTail->next = current;
        } else {
            evenTail = evenTail->next = current;
        }
        current = next;
    }

    oddTail->next = oddList;
    evenTail->next = evenList;
    head->next = head; // 清空原链表
}

/*
 * 问题 3: 大整数运算
 */
struct BigIntNode {
    int digit = 0;
    BigIntNode* next = nullptr;
};

class BigInt {
    BigIntNode* head;

    // 辅助函数: 递归打印, 优雅地逆序输出
    void _print_recursive(BigIntNode* node) const {
        if (!node) return;
        _print_recursive(node->next);
        std::cout << node->digit;
    }

    // 释放链表所有节点的内存
    void _clear() {
        while (head) {
            BigIntNode* temp = head;
            head = head->next;
            delete temp;
        }
    }
}
```



```

    }

public:
    // 构造函数，支持从字符串创建
    BigInt(const std::string& numStr = "0") : head(nullptr) {
        for (char c : numStr) {
            head = new BigIntNode{c - '0', head};
        }
    }

    // 析构函数
    ~BigInt() { _clear(); }

    // 拷贝构造函数：实现真正的深拷贝
    BigInt(const BigInt& other) : head(nullptr) {
        if (!other.head) return;
        BigIntNode* other_p = other.head;
        BigIntNode* this_tail = nullptr;
        while (other_p) {
            BigIntNode* newNode = new BigIntNode{other_p->digit, nullptr};
            if (!head) {
                head = this_tail = newNode;
            } else {
                this_tail = this_tail->next = newNode;
            }
            other_p = other_p->next;
        }
    }

    // 赋值运算符：典型的 copy-and-swap 写法
    BigInt& operator=(BigInt other) {
        std::swap(head, other.head);
        return *this;
    }

    // 输出
    void print() const {
        if (!head || (!head->next && head->digit == 0)) {
            std::cout << 0;
            return;
        }
        _print_recursive(head);
    }

    // 加法
    BigInt add(const BigInt& rhs) const {
        BigInt result("0");
        result._clear();

        BigIntNode *p1 = this->head, *p2 = rhs.head, *res_tail = nullptr;
        int carry = 0;

        while (p1 || p2 || carry) {
            int sum = (p1 ? p1->digit : 0) + (p2 ? p2->digit : 0) + carry;

```

```

        BigIntNode* newNode = new BigIntNode{sum % 10, nullptr};
        carry = sum / 10;
        if (!result.head) {
            result.head = res_tail = newNode;
        } else {
            res_tail = res_tail->next = newNode;
        }
        if (p1) p1 = p1->next;
        if (p2) p2 = p2->next;
    }
    return result;
}

};

/*
 * 问题 4: 查找两个链表的共同后缀
 */
struct ListNode {
    char data;
    ListNode* next;
};

// 设计思路: 让长的链表先走几步, 再一起走, 判断节点是否相同
ListNode* findIntersection(ListNode* headA, ListNode* headB) {
    int lenA = 0, lenB = 0;
    for (ListNode* p = headA->next; p; p = p->next) lenA++;
    for (ListNode* p = headB->next; p; p = p->next) lenB++;

    ListNode *ptrA = headA->next, *ptrB = headB->next;

    if (lenA > lenB) {
        for (int i = 0; i < lenA - lenB; ++i) ptrA = ptrA->next;
    } else {
        for (int i = 0; i < lenB - lenA; ++i) ptrB = ptrB->next;
    }

    while (ptrA != ptrB) {
        ptrA = ptrA->next;
        ptrB = ptrB->next;
    }
    return ptrA;
}

//
=====
=====
// 测试入口
//
=====
=====
void test_problem1() {
    std::cout << "---- 问题1: 稀疏多项式求导 ----\n";
    PolyNode* poly = new PolyNode();

```

```
poly->next = poly;

appendTerm(poly, 3, 10);
appendTerm(poly, 5, 4);
appendTerm(poly, -2, 1);
appendTerm(poly, 7, 0);

std::cout << "原始多项式: ";
printPolynomial(poly);
std::cout << std::endl;

differentiate(poly);

std::cout << "求导后多项式: ";
printPolynomial(poly);
std::cout << std::endl;
}

void test_problem2() {
    std::cout << "\n--- 问题2: 按指数奇偶分解多项式 ---\n";
    PolyNode* poly = new PolyNode();
    poly->next = poly;

    appendTerm(poly, 10, 5);
    appendTerm(poly, 8, 4);
    appendTerm(poly, -1, 3);
    appendTerm(poly, 6, 2);
    appendTerm(poly, 9, 0);

    std::cout << "原始多项式: ";
    printPolynomial(poly);
    std::cout << std::endl;

    PolyNode *oddP = nullptr, *evenP = nullptr;
    splitPolynomial(poly, oddP, evenP);

    std::cout << "奇次项部分: ";
    printPolynomial(oddP);
    std::cout << std::endl;

    std::cout << "偶次项部分: ";
    printPolynomial(evenP);
    std::cout << std::endl;
}

void test_problem3() {
    std::cout << "\n--- 问题3: 大整数运算 ---\n";
    BigInt a("78326590");
    BigInt b("123456789");

    std::cout << "大整数 A = "; a.print(); std::cout << std::endl;
    std::cout << "大整数 B = "; b.print(); std::cout << std::endl;

    BigInt c = a.add(b);
```

```
        std::cout << "A + B   = "; c.print(); std::cout << std::endl;
    }

    void test_problem4() {
        std::cout << "\n--- 问题4: 查找两个链表的后缀 ---\n";
        // 构造题目中的例子: "loading" 和 "being"
        ListNode* str1 = new ListNode{' ', nullptr};
        ListNode* str2 = new ListNode{' ', nullptr};

        str1->next = new ListNode{'l', new ListNode{'o', new ListNode{'a', new
        ListNode{'d', nullptr}}}};
        str2->next = new ListNode{'b', new ListNode{'e', nullptr}};

        ListNode* commonPart = new ListNode{'i', new ListNode{'n', new
        ListNode{'g', nullptr}}};

        // 链接到公共部分
        ListNode* tail1 = str1;
        while(tail1->next) tail1 = tail1->next;
        tail1->next = commonPart;

        ListNode* tail2 = str2;
        while(tail2->next) tail2 = tail2->next;
        tail2->next = commonPart;

        ListNode* intersection = findIntersection(str1, str2);

        if (intersection) {
            std::cout << "两个单词 'loading' 和 'being' 的后缀起始于: '" <<
intersection->data << "'" << std::endl;
        } else {
            std::cout << "两个链表没有共同后缀。" << std::endl;
        }
    }

    int main() {
        test_problem1();
        test_problem2();
        test_problem3();
        test_problem4();
        return 0;
    }
}
```