

## 问题1：线性表逆置

### 1. 独立完成

#### 算法设计思想

顺序表：通过首尾交换实现逆置。链表：利用三个指针，逐个反转节点指向。

#### 代码 (C++)

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

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(NULL) {}
};

void reverseVector(vector<int>& arr) {
    int i = 0, j = arr.size() - 1;
    while (i < j) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
        i++;
        j--;
    }
}

ListNode* reverseList(ListNode* head) {
    ListNode* prev = NULL;
    ListNode* curr = head;
    while (curr != NULL) {
        ListNode* nxt = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nxt;
    }
    return prev;
}
```

#### 复杂度分析

顺序表和链表均为  $O(n)$  时间， $O(1)$  空间。

---

### 2. LLM 完成

#### 用户提问

帮我用C++解决一个算法题：“编写代码分别实现一个顺序表和一个单链表的就地逆置。给出解题思路、代码和时间复杂度分析。”

### 模型回答

顺序表：双指针法，代码如下：

```
#include <vector>
#include <utility>
using namespace std;

void reverseVector(vector<int>& nums) {
    int start = 0, end = nums.size() - 1;
    while (start < end) {
        swap(nums[start], nums[end]);
        start++;
        end--;
    }
}
```

单链表：三指针迭代法，代码如下：

```
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(nullptr) {}
};

ListNode* reverseList(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* current = head;
    while (current != nullptr) {
        ListNode* nextNode = current->next;
        current->next = prev;
        prev = current;
        current = nextNode;
    }
    return prev;
}
```

复杂度：两者均为  $O(n)$  时间， $O(1)$  空间。

---

### 3. 对比分析

我的代码简洁，能直接运行，但缺乏更详细的思路说明。LLM 的答案更完整、规范，并解释了每一步的原理。两者在算法效率上一致。

---

### 问题2：删除线性表中重复元素

## 1. 独立完成

### 算法设计思想

顺序表：逐个扫描并检查前面是否出现过，若没有则保留。链表：双重循环，外层遍历，内层删除重复。

### 代码 (C++)

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

void removeDupVector(vector<int>& arr) {
    int k = 0;
    for (int i = 0; i < arr.size(); i++) {
        bool found = false;
        for (int j = 0; j < k; j++) {
            if (arr[i] == arr[j]) {
                found = true;
                break;
            }
        }
        if (!found) {
            arr[k] = arr[i];
            k++;
        }
    }
    arr.resize(k);
}

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(NULL) {}
};

void removeDupList(ListNode* head) {
    for (ListNode* p = head; p != NULL; p = p->next) {
        ListNode* q = p;
        while (q->next != NULL) {
            if (q->next->val == p->val) {
                ListNode* tmp = q->next;
                q->next = tmp->next;
                delete tmp;
            } else {
                q = q->next;
            }
        }
    }
}
```

### 复杂度分析

顺序表和链表均为  $O(n^2)$  时间， $O(1)$  空间。

## 2. LLM 完成

### 用户提问

帮我用C++解决一个算法题：“编写代码分别删除一个顺序表和一个单链表中所有重复的元素，并保持剩余元素的相对顺序不变。”

### 模型回答

顺序表：使用哈希集合提高效率。

```
#include <vector>
#include <unordered_set>
using namespace std;

void removeDuplicatesFromVector(vector<int>& nums) {
    unordered_set<int> seen;
    vector<int> result;
    for (int num : nums) {
        if (seen.find(num) == seen.end()) {
            seen.insert(num);
            result.push_back(num);
        }
    }
    nums = result;
}
```

单链表：同样利用哈希集合。

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

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(nullptr) {}
};

void deleteDuplicatesInList(ListNode* head) {
    if (!head) return;
    unordered_set<int> seen;
    seen.insert(head->val);
    ListNode* curr = head;
    while (curr->next != nullptr) {
        if (seen.count(curr->next->val)) {
            ListNode* temp = curr->next;
            curr->next = temp->next;
            delete temp;
        } else {
            seen.insert(curr->next->val);
        }
    }
}
```

```
        curr = curr->next;
    }
}
```

复杂度：O(n) 时间，O(n) 空间。

---

### 3. 对比分析

独立完成方案省空间，但时间复杂度较高。LLM 方案时间复杂度更优，代价是增加了哈希表存储。两者差异体现了典型的空间与时间权衡。

---

## 问题3：合并有序线性表

### 1. 独立完成

#### 算法设计思想

设置两个指针分别扫描两个有序链表，每次取较小的节点插入结果，最后将未处理部分整体接入。

#### 代码 (C++)

```
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(NULL) {}
};

ListNode* mergeList(ListNode* a, ListNode* b) {
    ListNode dummy(0);
    ListNode* tail = &dummy;
    while (a && b) {
        if (a->val <= b->val) {
            tail->next = a;
            a = a->next;
        } else {
            tail->next = b;
            b = b->next;
        }
        tail = tail->next;
    }
    tail->next = (a ? a : b);
    return dummy.next;
}
```

#### 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

帮我用C++写一个函数，将两个已按递增顺序排列的单链表合并为一个新的递增顺序单链表。

### 模型回答

```
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(nullptr) {}
};

ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    ListNode dummy(0);
    ListNode* current = &dummy;
    while (l1 && l2) {
        if (l1->val < l2->val) {
            current->next = l1;
            l1 = l1->next;
        } else {
            current->next = l2;
            l2 = l2->next;
        }
        current = current->next;
    }
    current->next = (l1 ? l1 : l2);
    return dummy.next;
}
```

复杂度：时间  $O(m+n)$ ，空间  $O(1)$ 。

---

## 3. 对比分析

两者算法完全一致，差别在细节：LLM 使用了更清晰的命名与注释，更具可读性；独立方案更接近课堂作业风格。整体来看，LLM 更系统，我的方案更简洁。

---

## 问题4：线性表的循环左移

### 1. 独立完成

#### 算法设计思想

顺序表：使用辅助数组保存前  $k$  个元素，再拼接到末尾。链表：先形成环，再找到新尾节点，断开形成新的表头。

#### 代码 (C++)

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

void rotateLeftVector(vector<int>& arr, int k) {
    int n = arr.size();
    if (n == 0) return;
    k %= n;
    vector<int> tmp(arr.begin(), arr.begin() + k);
    for (int i = k; i < n; i++) arr[i-k] = arr[i];
    for (int i = 0; i < k; i++) arr[n-k+i] = tmp[i];
}

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(NULL) {}
};

ListNode* rotateLeftList(ListNode* head, int k) {
    if (!head) return head;
    int n = 1;
    ListNode* tail = head;
    while (tail->next) {
        tail = tail->next;
        n++;
    }
    k %= n;
    if (k == 0) return head;
    tail->next = head;
    ListNode* newTail = head;
    for (int i = 0; i < k-1; i++) newTail = newTail->next;
    ListNode* newHead = newTail->next;
    newTail->next = NULL;
    return newHead;
}
```

### 复杂度分析

顺序表：时间  $O(n)$ ，空间  $O(k)$ 。链表：时间  $O(n)$ ，空间  $O(1)$ 。

---

## 2. LLM 完成

### 用户提问

为顺序表和单链表设计一个算法，将其中的元素循环左移  $k$  位。要求空间复杂度为  $O(1)$ 。

### 模型回答

顺序表：使用三次反转。

```
#include <vector>
#include <algorithm>
using namespace std;

void rotate(vector<int>& nums, int k) {
    int n = nums.size();
    if (n == 0) return;
    k %= n;
    reverse(nums.begin(), nums.begin() + k);
    reverse(nums.begin() + k, nums.end());
    reverse(nums.begin(), nums.end());
}
```

单链表：通过环化处理。

```
ListNode* rotateRight(ListNode* head, int k) {
    if (!head || !head->next || k == 0) return head;
    int n = 1;
    ListNode* old_tail = head;
    while (old_tail->next) {
        old_tail = old_tail->next;
        n++;
    }
    old_tail->next = head;
    ListNode* new_tail = head;
    for (int i = 0; i < n - k % n - 1; i++) {
        new_tail = new_tail->next;
    }
    ListNode* new_head = new_tail->next;
    new_tail->next = nullptr;
    return new_head;
}
```

---

### 3. 对比分析

独立方案在顺序表部分使用额外空间，不够优化。LLM 使用三次反转，在空间效率上更佳。链表部分两种方法基本一致。LLM 的方案更精炼，而独立方案更直观。

---

## 问题5：约瑟夫环问题

### 1. 独立完成

#### 算法设计思想

构造循环链表模拟报数，每轮前进  $m-1$  步删除对应节点，直至剩下最后一个节点。

#### 代码 (C++)



```
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(NULL) {}
};

int josephus(int n, int m) {
    if (n <= 0) return -1;
    ListNode* head = new ListNode(1);
    ListNode* p = head;
    for (int i = 2; i <= n; i++) {
        p->next = new ListNode(i);
        p = p->next;
    }
    p->next = head;
    ListNode* prev = p;
    while (prev->next != prev) {
        for (int i = 0; i < m-1; i++) prev = prev->next;
        ListNode* t = prev->next;
        prev->next = t->next;
        delete t;
    }
    int ans = prev->val;
    delete prev;
    return ans;
}
```

### 复杂度分析

时间复杂度  $O(nm)$ ，空间复杂度  $O(n)$ 。

---

## 2. LLM 完成

### 用户提问

用C++和一个循环链表来解决约瑟夫环问题。函数输入  $n$  和  $m$ ，输出最后剩下的人的编号。

### 模型回答

```
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(nullptr) {}
};

int josephus(int n, int m) {
    if (n < 1 || m < 1) return -1;
    if (n == 1) return 1;

    ListNode* head = new ListNode(1);
```

```
ListNode* tail = head;
for (int i = 2; i <= n; ++i) {
    tail->next = new ListNode(i);
    tail = tail->next;
}
tail->next = head;

ListNode* current = tail;
while (current->next != current) {
    for (int i = 0; i < m-1; ++i) {
        current = current->next;
    }
    ListNode* node_to_remove = current->next;
    current->next = node_to_remove->next;
    delete node_to_remove;
}
int winner = current->val;
delete current;
return winner;
}
```

复杂度：时间  $O(nm)$ ，空间  $O(n)$ 。

---

### 3. 对比分析

两者均通过循环链表模拟，思路完全一致。区别在于变量选择与实现细节：LLM 更系统化，我的解法更贴近课堂作业风格。复杂度相同。

---

## 附录：C++完整代码实现

```
// 通用配置

#include <iostream>
#include <vector>
#include <algorithm> // swap 用

using namespace std;

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(nullptr) {}
};

void printVector(const vector<int>& a) {
    cout << "[";
    for (int i = 0; i < a.size(); i++) {
        cout << a[i];
    }
}
```

```
        if (i != a.size() - 1) cout << ", ";
    }
    cout << "]" << endl;
}

void printList(ListNode* head) {
    for (ListNode* p = head; p; p = p->next) {
        cout << p->val << " -> ";
    }
    cout << "nullptr" << endl;
}

void freeList(ListNode* head) {
    while (head) {
        ListNode* t = head;
        head = head->next;
        delete t;
    }
}

//问题1: 线性表逆置
void reverseVector(vector<int>& a) {
    int i = 0, j = a.size() - 1;
    while (i < j) {
        swap(a[i], a[j]);
        i++, j--;
    }
}

ListNode* reverseList(ListNode* head) {
    ListNode* pre = nullptr;
    ListNode* p = head;
    while (p) {
        ListNode* q = p->next;
        p->next = pre;
        pre = p;
        p = q;
    }
    return pre;
}

// 问题2: 删除重复元素
void removeDupVector(vector<int>& a) {
    int k = 0;
    for (int i = 0; i < a.size(); i++) {
        bool dup = false;
        for (int j = 0; j < k; j++) {
            if (a[i] == a[j]) {
                dup = true;
                break;
            }
        }
        if (!dup) a[k++] = a[i];
    }
}
```

```

    a.resize(k);
}

void removeDupList(ListNode* head) {
    for (ListNode* p = head; p; p = p->next) {
        ListNode* q = p;
        while (q->next) {
            if (q->next->val == p->val) {
                ListNode* t = q->next;
                q->next = t->next;
                delete t;
            } else {
                q = q->next;
            }
        }
    }
}

// 问题3: 合并两个有序链表
ListNode* mergeLists(ListNode* a, ListNode* b) {
    ListNode dummy(0);
    ListNode* tail = &dummy;
    while (a && b) {
        if (a->val <= b->val) {
            tail->next = a;
            a = a->next;
        } else {
            tail->next = b;
            b = b->next;
        }
        tail = tail->next;
    }
    tail->next = a ? a : b;
    return dummy.next;
}

// 问题4: 循环左移
void reverseRange(vector<int>& a, int l, int r) {
    while (l < r) swap(a[l++], a[r--]);
}

void rotateLeft(vector<int>& a, int k) {
    int n = a.size();
    if (n == 0) return;
    k %= n;
    if (k == 0) return;
    reverseRange(a, 0, k - 1);
    reverseRange(a, k, n - 1);
    reverseRange(a, 0, n - 1);
}

ListNode* rotateLeftList(ListNode* head, int k) {
    if (!head) return head;
    int n = 1;

```

```
ListNode* p = head;
while (p->next) {
    p = p->next;
    n++;
}
k %= n;
if (k == 0) return head;
p->next = head;
ListNode* newTail = head;
for (int i = 0; i < k - 1; i++) newTail = newTail->next;
ListNode* newHead = newTail->next;
newTail->next = nullptr;
return newHead;
}

// --- 问题5: 约瑟夫环 ---
int josephus(int n, int m) {
    if (n <= 0 || m <= 0) return -1;
    ListNode* head = new ListNode(1);
    ListNode* p = head;
    for (int i = 2; i <= n; i++) {
        p->next = new ListNode(i);
        p = p->next;
    }
    p->next = head; // 环
    ListNode* pre = p;
    while (pre->next != pre) {
        for (int i = 0; i < m - 1; i++) pre = pre->next;
        ListNode* t = pre->next;
        pre->next = t->next;
        cout << "淘汰: " << t->val << endl;
        delete t;
    }
    int ans = pre->val;
    delete pre;
    return ans;
}

// --- 主函数测试 ---
int main() {
    // 问题1
    vector<int> v1 = {4, 2, 7, 3, 0};
    cout << "原顺序表: "; printVector(v1);
    reverseVector(v1);
    cout << "逆置后: "; printVector(v1);

    ListNode* h1 = new ListNode(4);
    h1->next = new ListNode(2);
    h1->next->next = new ListNode(7);
    h1->next->next->next = new ListNode(3);
    h1->next->next->next->next = new ListNode(0);
    cout << "原链表: "; printList(h1);
    h1 = reverseList(h1);
    cout << "逆置后: "; printList(h1);
}
```

```
freeList(h1);

// 问题2
vector<int> v2 = {1,2,2,3,4,4,5};
cout << "\n去重前: "; printVector(v2);
removeDupVector(v2);
cout << "去重后: "; printVector(v2);

ListNode* h2 = new ListNode(1);
h2->next = new ListNode(2);
h2->next->next = new ListNode(2);
h2->next->next->next = new ListNode(3);
h2->next->next->next->next = new ListNode(4);
h2->next->next->next->next->next = new ListNode(4);
h2->next->next->next->next->next->next = new ListNode(5);
cout << "链表去重前: "; printList(h2);
removeDupList(h2);
cout << "链表去重后: "; printList(h2);
freeList(h2);

// 问题3
ListNode* a = new ListNode(1);
a->next = new ListNode(3);
a->next->next = new ListNode(5);
ListNode* b = new ListNode(2);
b->next = new ListNode(4);
b->next->next = new ListNode(6);
cout << "\nA: "; printList(a);
cout << "B: "; printList(b);
ListNode* merged = mergeLists(a, b);
cout << "合并后: "; printList(merged);
freeList(merged);

// 问题4
vector<int> v4 = {1,2,3,4,5};
cout << "\n原顺序表: "; printVector(v4);
rotateLeft(v4, 2);
cout << "左移2位: "; printVector(v4);

ListNode* h4 = new ListNode(1);
h4->next = new ListNode(2);
h4->next->next = new ListNode(3);
h4->next->next->next = new ListNode(4);
h4->next->next->next->next = new ListNode(5);
cout << "链表原: "; printList(h4);
h4 = rotateLeftList(h4, 2);
cout << "链表左移2位: "; printList(h4);
freeList(h4);

// 问题5
cout << "\n约瑟夫环 (n=5, m=3)" << endl;
int res = josephus(5, 3);
cout << "最后剩下: " << res << endl;
```

```
    return 0;  
}
```