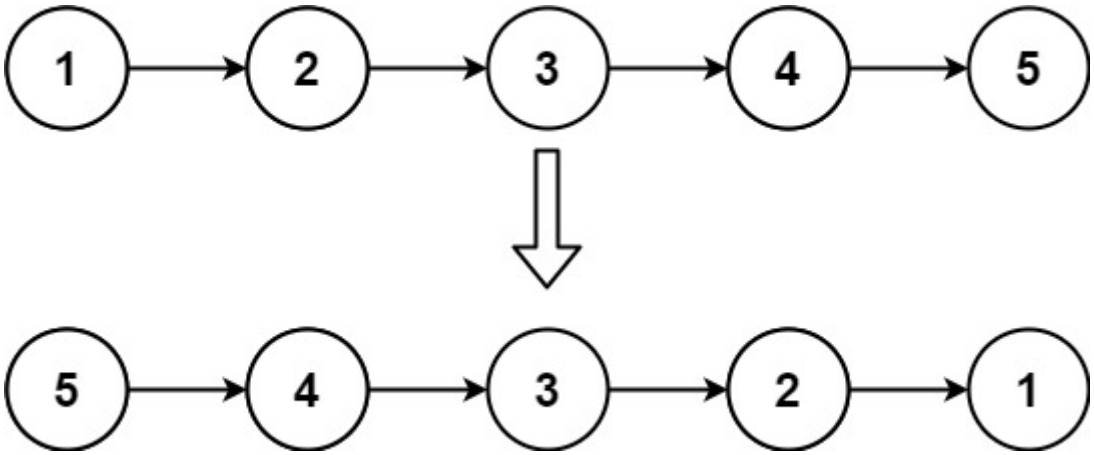
链表

一些题目

1.反转链表

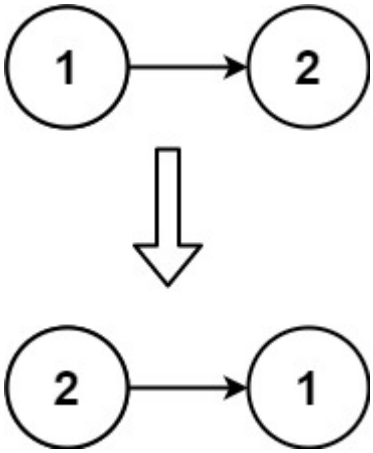
给你单链表的头节点 `head` ，请你反转链表，并返回反转后的链表。

示例 1：




```
输入：head = [1,2,3,4,5]
输出：[5,4,3,2,1]
```

示例 2：




```
输入：head = [1,2]
输出：[2,1]
```

示例 3：

```
输入：head = []
输出：[]
```

提示：

- 链表中节点的数目范围是 `[0, 5000]`
- `-5000 <= Node.val <= 5000`

进阶：链表可以选用迭代或递归方式完成反转。你能否用两种方法解决这道题？

```c
/*方法一：迭代*/
struct ListNode* reverseList(struct ListNode* head) {
    struct ListNode* prev = NULL;
    struct ListNode* curr = head;
    struct ListNode* next = NULL;

    while(curr){
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

递归的三步走：

1.确定终止条件

> 思考：
>
> 1. 最简单的情况是什么?
> 2. 什么时候可以直接返回结果?

2.设计子问题

> 思考：
>
> 1. 如何将问题规模变小?
> 2. 子问题与原问题的关系是什么

3.处理当前层

> 思考：
>
> 1. 当前层需要做什么操作?
> 2. 如何利用子问题的结果?

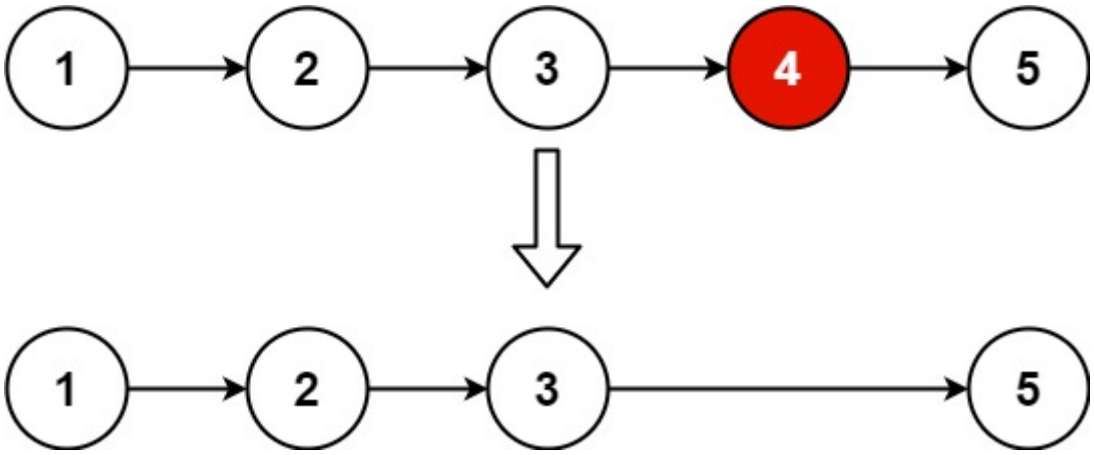
```c
/*方法二：递归*/
struct ListNode* reverseList(struct ListNode* head) {
   /*确定终止条件*/
    if(head == NULL || head->next == NULL){
        return head;

```

```
    }

    /*设计子问题*/
    struct ListNode* newHead = reverseList(head->next);

    /*处理当前层*/
    head->next->next = head;
    head->next = NULL;
    return newHead;


}
```

2.删除链表的倒数第N个节点

给你一个链表，删除链表的倒数第 `n` 个结点，并且返回链表的头结点。

示例 1：




```
输入：head = [1,2,3,4,5], n = 2
输出：[1,2,3,5]
```

示例 2：

```
输入：head = [1], n = 1
输出：[]
```

示例 3：

```
输入：head = [1,2], n = 1
输出：[1]
```

提示：

- 链表中结点的数目为 `sz`
- `1 <= sz <= 30`

- `0 <= Node.val <= 100`
- `1 <= n <= sz`

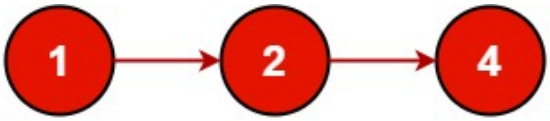
进阶：你能尝试使用一趟扫描实现吗？

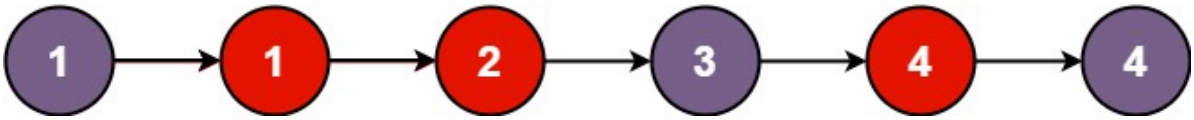
```c
struct ListNode* removeNthFromEnd(struct ListNode* head, int n) {
    // 创建dummy节点
    struct ListNode* dummy = (struct ListNode*)malloc(sizeof(struct ListNode));
    dummy->next = head;

    // 定义双指针
    struct ListNode* left = dummy;
    struct ListNode* right = dummy;

    // 右指针先走n步 迭代n次可以用n--
    while (n--) {
        right = right->next;
    }

    // 双指针同时移动
    while (right->next) {
        left = left->next;
        right = right->next;
    }

    // 删除目标节点
    struct ListNode* nxt = left->next;
    left->next = left->next->next;
    free(nxt);

    // 获取新的头节点
    struct ListNode* newHead = dummy->next;

    // 释放dummy节点
    free(dummy);

    return newHead;
}
```

3.合并两个有序链表/合并K个有序链表

将两个升序链表合并为一个新的 **升序** 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例 1：





```
输入：l1 = [1,2,4], l2 = [1,3,4]
输出：[1,1,2,3,4,4]
```

示例 2：

```
输入：l1 = [], l2 = []
输出：[]
```

示例 3：

```
输入：l1 = [], l2 = [0]
输出：[0]
```

提示：

- 两个链表的节点数目范围是 `[0, 50]`
- `-100 <= Node.val <= 100`
- `l1` 和 `l2` 均按 **非递减顺序** 排列

```c
struct ListNode* mergeTwoLists(struct ListNode* l1, struct ListNode* l2) {
        //创建头节点（新链表）
        struct ListNode* dummy = malloc(sizeof(struct ListNode));

        //在新链表中设置指针
        struct ListNode* curr = dummy;

        //循环终止条件 L1或L2中有一个为空
        while(l1 && l2){
            if(l1->val <= l2->val){
                curr->next = l1; //把值赋给新链表
                l1 = l1->next;  //L2继续移动
            }else{
                curr->next = l2; //把值赋给新链表
                l2 = l2->next; //L2继续移动
            }
```

```
            curr = curr->next; //新链表指针移动
        }

        curr->next = l1? l1 : l2; //将剩下的链表全数接回 （L1不为空则接L1  反之亦然）
        struct ListNode* res = dummy->next;
        free(dummy);
        return res;

}
```

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

```
输入：lists = [[1,4,5],[1,3,4],[2,6]]
输出：[1,1,2,3,4,4,5,6]
解释：链表数组如下：
[
  1->4->5,
  1->3->4,
  2->6
]
将它们合并到一个有序链表中得到。
1->1->2->3->4->4->5->6
```

示例 2：

```
输入：lists = []
输出：[]
```

示例 3：

```
输入：lists = [[]]
输出：[]
```

提示：

- `k == lists.length`
- `0 <= k <= 10^4`
- `0 <= lists[i].length <= 500`
- `-10^4 <= lists[i][j] <= 10^4`
- `lists[i]` 按 **升序** 排列
- `lists[i].length` 的总和不超过 `10^4`

这道题可以在基于合并两个有序链表的基础上 利用**分治**策略来完成本道题

分治法的步骤可以按照类似的结构梳理，结合分治的核心特点进行细化：

####

1. **划分子问题**

   > **思考**：
   >
   > - 问题能否分解为多个规模更小但结构相似的子问题？
   > - 如何划分子问题以确保问题规模减少且不遗漏重要信息？

   要点：

   - 将问题分解为**互相独立的子问题。**
   - 确保子问题的规模足够小，最终可直达终止条件。
   - 子问题的划分方式可以是对半、按块、递减等，具体取决于问题的特点。
2. **解决子问题**

   > **思考**：
   >
   > - 子问题是否可以直接用递归来解决？
   > - 子问题的终止条件是什么？

   要点：

   - 如果子问题足够小（达到终止条件），直接求解。
   - 否则递归调用分治步骤，对子问题进一步划分和求解。
   - 确保每个子问题在规模足够小后，都有明确的解法。
3. **合并子问题结果**

   > **思考**：
   >
   > - 子问题的结果如何组合成原问题的解？
   > - 合并的逻辑是否高效？

   要点：

   - 子问题的结果可能需要排序、汇总、连接或其他操作来生成最终结果。
   - 合并步骤通常是算法优化的重点部分，可以关注其时间复杂度和数据结构选择。
4. **处理边界情况**

   > **思考**：
   >
   > - 最小问题规模下的直接解法是否准确？
   > - 是否需要特殊处理空输入、边界输入或异常情况？

   要点：

   - 明确分治法适用的前提条件和限制（如输入规模、数据特点）。
   - 增加边界测试用例，确保所有路径都可覆盖。

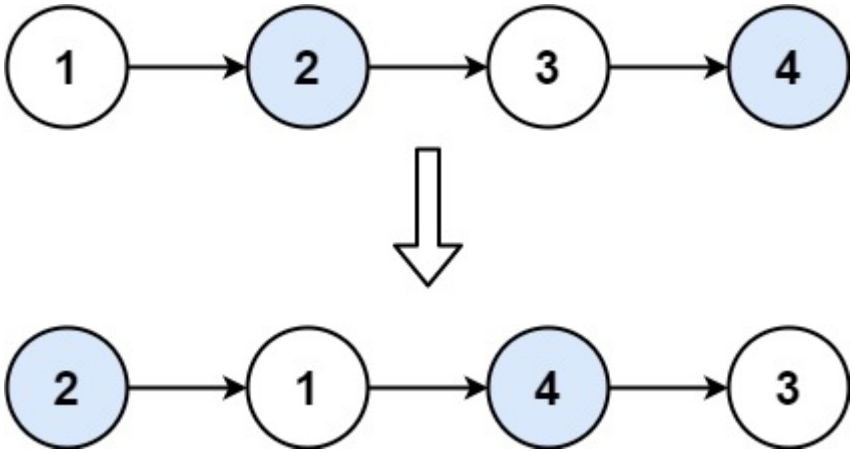
```c
struct ListNode* mergeTwoLists(struct ListNode* l1, struct ListNode* l2) {
        struct ListNode* dummy = malloc(sizeof(struct ListNode));
        struct ListNode* curr = dummy;
        while(l1 != NULL && l2 != NULL){
            if(l1->val <= l2->val){
                curr->next = l1;
                l1 = l1->next;
```

```
            }else{
                curr->next = l2;
                l2 = l2->next;
            }
            curr = curr->next;
        }

        curr->next = l1? l1 : l2;
        struct ListNode* res = dummy->next;
        free(dummy);
        return res;

}

struct ListNode* merge(struct ListNode** lists, int left, int right) {
    // 1. 终止条件
    if(left == right) {
        return lists[left];  // 只剩一个链表时直接返回
    }
    if(left > right) {
        return NULL;  // 无效区间
    }

    // 2. 分治处理
    // 计算中间位置，避免溢出
    int mid = left + (right - left) / 2;

    // 递归合并左半部分
    struct ListNode* l1 = merge(lists, left, mid);
    // 递归合并右半部分
    struct ListNode* l2 = merge(lists, mid + 1, right);

    // 3. 合并两个有序链表
    return mergeTwoLists(l1, l2);
}


struct ListNode* mergeKLists(struct ListNode** lists, int listsSize) {
    // 处理空数组情况
    if(!listsSize) return NULL;

    // 调用分治合并函数
    return merge(lists, 0, listsSize - 1);
}
```

4.两两交换链表中的节点

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

示例 1：




```
输入：head = [1,2,3,4]
输出：[2,1,4,3]
```

示例 2：

```
输入：head = []
输出：[]
```

示例 3：

```
输入：head = [1]
输出：[1]
```

提示：

- 链表中节点的数目在范围 `[0, 100]` 内
- `0 <= Node.val <= 100`

```c
//解法一：迭代解法
struct ListNode* swapPairs(struct ListNode* head) {
    //创建头节点（新链表）
    struct ListNode* dummy = malloc(sizeof(struct ListNode));
    dummy->next = head;

    //在新链表中设置指针
    struct ListNode* curr = dummy;
    while (curr->next != NULL && curr->next->next != NULL){
        struct ListNode* first = curr->next;
        struct ListNode* second = curr->next->next;

        //每两两进行交换
        first->next = second->next;
        second->next = first;
        curr->next = second;

        //移至下一对
        curr = first;
    }

```

```
    struct ListNode* res = dummy->next;
    free(dummy);
    return res;

}
```

```
//解法二：递归解法
struct ListNode* swapPairs(struct ListNode* head) {
    //终止条件
    if(head == NULL || head->next == NULL){
        return head;
    }

    struct ListNode* newhead = head->next;
    struct ListNode* next = newhead->next;

    //处理当前层
    newhead->next = head;

    //分解子问题（子问题和当前问题连接）
    head->next = swapPairs(next);
    return newhead;

}
```

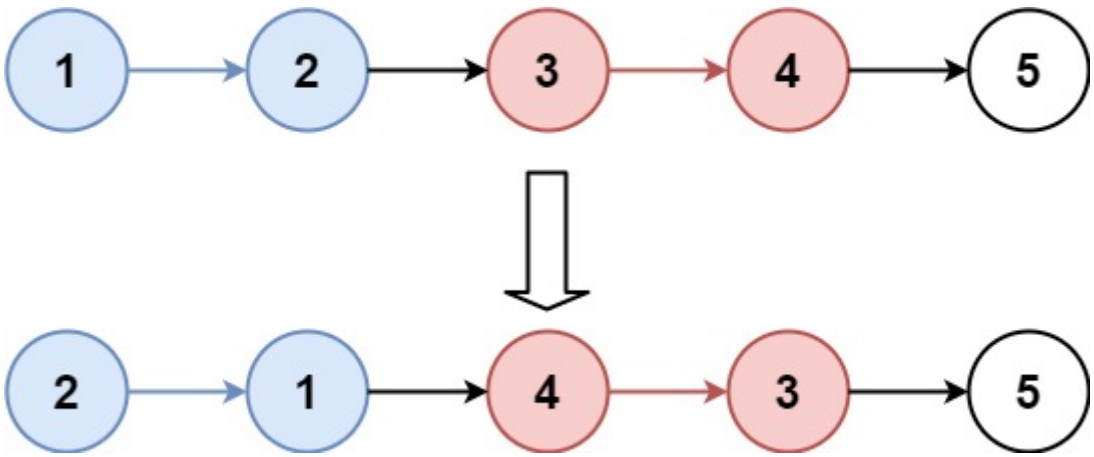
5.K个一组反转链表

给你链表的头节点 `head` ，每 `k` 个节点一组进行翻转，请你返回修改后的链表。

`k` 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 `k` 的整数倍，那么请将最后剩余的节点保持原有顺序。

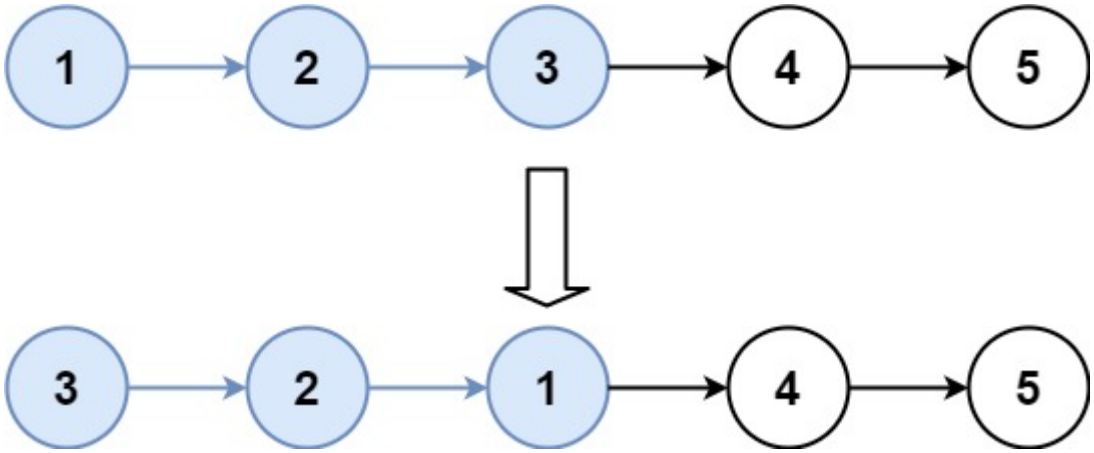
你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

示例 1：



```
输入：head = [1,2,3,4,5], k = 2
输出：[2,1,4,3,5]
```

示例 2：




```
输入：head = [1,2,3,4,5], k = 3
输出：[3,2,1,4,5]
```

提示：

- 链表中的节点数目为 `n`
- `1 <= k <= n <= 5000`
- `0 <= Node.val <= 1000`

进阶：你可以设计一个只用 `O(1)` 额外内存空间的算法解决此问题吗？

`O(1)` 额外内存空间算法：

```c
struct ListNode* reverse(struct ListNode* head) {
    struct ListNode* curr = head;
    struct ListNode* pre = NULL;
    while(curr != NULL) {
        struct ListNode* next = curr->next;
        curr->next = pre;
        pre = curr;
        curr = next;
    }
    return pre;
}

struct ListNode* reverseKGroup(struct ListNode* head, int k) {
    // 终止条件
    if(head == NULL || k == 1) {
        return head;
    }

    // 创建虚拟头节点
    struct ListNode* dummy = malloc(sizeof(struct ListNode));
    dummy->next = head;
```

```
    // 每组的前缀节点
    struct ListNode* preGroup = dummy;

    while(1) {
        // 找到每组的起始节点
        struct ListNode* start = preGroup->next;
        // 找到每组的结束节点
        struct ListNode* end = start;

        // 向后移动k-1步找到end
        for(int i = 0; i < k - 1 && end != NULL; i++) {
            end = end->next;
        }

        // 如果不足k个节点，退出
        if(end == NULL) break;

        // 保存下一组的起始节点
        struct ListNode* nextStart = end->next;
        // 断开与下一组的连接
        end->next = NULL;

        // 反转当前组
        struct ListNode* currGroup = reverse(start);

        // 连接反转后的组
        preGroup->next = currGroup;
        // 连接下一组（此时start变成了尾节点）
        start->next = nextStart;
        // 更新前缀节点
        preGroup = start;
    }

    struct ListNode* result = dummy->next;
    free(dummy);
    return result;
}
```

主要算法步骤：

```
while(1) {
    // 1. 找到当前组的end节点
    // 2. 断开与下一组的连接
    // 3. 反转当前组
    // 4. 重新连接
    // 5. 更新前缀节点
}
```

关键变量：

```
struct ListNode* preGroup  // 每组的前缀节点
struct ListNode* start     // 当前组的起始节点
struct ListNode* end       // 当前组的结束节点
struct ListNode* nextStart // 下一组的起始节点
```