# 题目

给你一个链表，每k个节点一组进行翻转，请你返回翻转后的链表。

k是一个正整数，它的值小于或等于链表的长度。

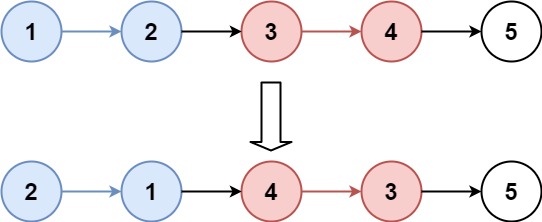
如果节点总数不是k的整数倍，那么请将最后剩余的节点保持原有顺序。

进阶：

你可以设计一个只使用常数额外空间的算法来解决此问题吗？

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。

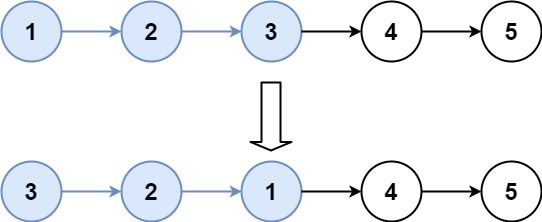
示例 1：



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]

示例 3：

输入：head = [1,2,3,4,5], k = 1

输出：[1,2,3,4,5]

示例 4：

输入：head = [1], k = 1

输出：[1]

提示：

列表中节点的数量在范围 sz 内

1 <= sz <= 5000

0 <= Node.val <= 1000

1 <= k <= sz

# 分析

## 方法一：遍历

思路：

1、定义一个辅助节点 dummy，并将其指向头节点，用于处理头节点可能会被翻转的情况。

2、使用两个指针 pre 和 cur 分别指向当前翻转段的前一个节点和当前节点。

3、进入循环，每次循环中，先遍历找到待翻转的一组节点的结束节点 cur，并进行边界判断，若不足 k 个节点则直接返回。

4、然后将当前翻转段的头节点 start 记录下来，翻转这一段链表，并将翻转后的头节点与 pre 连接，将翻转后的尾节点与下一段链表连接。

5、更新 pre 和 cur 指针，使其分别指向下一段待翻转的前一个节点和当前节点。

6、重复步骤 3-5，直到遍历完整个链表。

在翻转链表的过程中，需要使用 reverseList 函数来实现链表的翻转操作。

代码：

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

ListNode\* prev = nullptr;

ListNode\* curr = head;

while (curr != nullptr) {

ListNode\* nextTemp = curr->next;

curr->next = prev;

prev = curr;

curr = nextTemp;

}

return prev;

}

ListNode\* reverseKGroup(ListNode\* head, int k) {

if (head == nullptr || head->next == nullptr || k == 1) {

return head;

}

ListNode\* dummy = new ListNode(0);

dummy->next = head;

ListNode\* pre = dummy;

ListNode\* cur = head;

while (cur != nullptr) {

ListNode\* start = cur;

int count = 1;

while (count < k && cur->next != nullptr) { // 每隔k节点反转

cur = cur->next;

count++;

}

if (count == k) {

ListNode\* next = cur->next; // 记录下当前翻转段的下一个节点，以便后续连接下一段链表

cur->next = nullptr; // 将当前节点的 next 指针置空，这一步是为了在翻转链表时，将当前节点的指针反向连接到前一个节点

pre->next = reverseList(start); // 将前一个翻转段的尾节点（pre 指向的节点）的 next 指针指向当前翻转段翻转后的头节点。reverseList(start) 的作用是翻转以 start 开头的链表段

start->next = next; // 将当前翻转段翻转后的尾节点的 next 指针指向下一个翻转段的开始节点，即连接下一段链表

pre = start; // 更新 pre 指针，使其指向当前翻转段翻转后的尾节点，为下一次翻转做准备

cur = next; //更新 cur 指针，使其指向下一个翻转段的开始节点，为下一次翻转做准备

} else {

pre->next = start;

break;

}

}

return dummy->next;

}

};

或：

思路：

你可以按照如下步骤实现每 k 个节点一组进行翻转的功能：

1、定义一个函数 reverseList，用于反转链表。

2、在 reverseKGroup 函数中，遍历链表，每次处理 k 个节点：

- 找到当前 k 个节点的起始节点和结束节点，将其分割出来。

- 将这 k 个节点反转，得到新的头节点和尾节点。

- 将反转后的链表连接回原链表中。

3、返回头节点。

以下是具体的实现：

class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

ListNode\* prev = nullptr;

ListNode\* curr = head;

while (curr != nullptr) {

ListNode\* nextTemp = curr->next;

curr->next = prev;

prev = curr;

curr = nextTemp;

}

return prev;

}

ListNode\* reverseKGroup(ListNode\* head, int k) {

if (head == nullptr || head->next == nullptr || k == 1) {

return head;

}

ListNode\* dummy = new ListNode(0);

dummy->next = head;

ListNode\* pre = dummy;

ListNode\* end = dummy;

while (end->next != nullptr) {

for (int i = 0; i < k && end != nullptr; ++i) {

end = end->next;

}

if (end == nullptr) {

break;

}

ListNode\* start = pre->next;

ListNode\* next = end->next;

end->next = nullptr;

pre->next = reverseList(start);

start->next = next;

pre = start;

end = pre;

}

return dummy->next;

}

};

这段代码首先定义了一个 `reverseList` 函数用于反转链表。在 `reverseKGroup` 函数中，使用 `pre` 和 `end` 指针来确定每个 k 个节点的范围，然后将这部分链表反转，并将其接回原链表中。

或：

思路：

参考链表原地翻转的思路进行。

/ \*

\*prev

\*tail head

\*dummy 1 2 3 4 5

\*----------------------------

\*prev head tail

\*dummy 1 2 3 4 5

\* cur

\*----------------------------

\* 每次让prev.next的元素插入到当前tail之后,这样tail不断前移,被挪动的元素头插入tail之后的链表

\*prev tail head

\*dummy 2 3 1 4 5

\* cur

\*----------------------------

\*prev tail head

\*dummy 3 2 1 4 5

\* cur

\*----------------------------

\* prev

\* tail

\* head

\*dummy 3 2 1 4 5

\*----------------------------

\* prev head tail

\*dummy 3 2 1 4 5 null

\*----------------------------

\*/

代码：

class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

ListNode\* dummy=new ListNode(0);

dummy->next=head;

ListNode\* pre=dummy; // 三个指针实现翻转（链表原地翻转）

ListNode\* cur=head;

ListNode\* tail=dummy;

while(true){

int count=0;

tail=pre;

while(tail!=NULL&&count<k){

tail=tail->next; //退出循环后tail指向待反转链表的末尾节点

count++;

}

if(tail==NULL)

break;

while(pre->next!=tail){ //pre->next==tail时退出循环

cur=pre->next;//(1)

pre->next=cur->next; //(1) 步骤(1):将cur从链表中切出来

cur->next=tail->next;//(2)

tail->next=cur; //(2) 步骤(2):将cur添加到tail后

}

pre=head;

tail=head;

head=pre->next;//head指向新的待翻转的链表头

}

return dummy->next;

}

};

或：

思路：

可以按照两个链表/k链表翻转的思路进行。

代码：

class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

vector<ListNode\*> reversedLists;

ListNode\* cur = head;

while (cur != nullptr) {

vector<ListNode\*> group;

for (int i = 0; i < k && cur != nullptr; ++i) {

group.push\_back(cur);

cur = cur->next;

}

if (group.size() == k) {

ListNode\* reversed = mergeKLists(group);

reversedLists.push\_back(reversed);

} else {

reversedLists.push\_back(group[0]); // 不足 k 个节点，直接加入结果中

}

}

return mergeKLists(reversedLists);

}

ListNode\* mergeKLists(vector<ListNode\*>& lists) {

ListNode \*ret = nullptr;

for(int i = 0; i < lists.size(); i++) {

ret = merge2Lists(ret, lists.at(i));

}

return ret;

}

ListNode\* merge2Lists(ListNode\* p1, ListNode\* p2) {

ListNode\* dummyNode = new ListNode(-1);

ListNode\* tmpNode = dummyNode;

while (p1 && p2) {

if (p1->val < p2->val) {

tmpNode->next = p1;

p1 = p1->next;

} else {

tmpNode->next = p2;

p2 = p2->next;

}

tmpNode = tmpNode->next;

}

if (p1)

tmpNode->next = p1;

if (p2)

tmpNode->next = p2;

return dummyNode->next;

}

};

上述代码会执行超时。

## 方法二：递归

**思路：**

**代码：**

class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

if (head == NULL) return NULL;

ListNode \*a = head;

ListNode \*b = head;

for (int i = 0; i < k; i++) {

if (b == NULL) return head;

b = b->next;

}

ListNode \*newNode = reverseOperator(a,b);

a->next = reverseKGroup(b,k);

return newNode;

}

ListNode\* reverseOperator(ListNode\* n,ListNode \*b) {

ListNode \*pre, \*cur, \*nxt;

pre = NULL; cur = n; nxt = n;

while (cur != b) {

nxt = cur->next;

cur->next = pre;

pre = cur;

cur = nxt;

}

return pre;

}

};

注释：

class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

if(head==nullptr) return nullptr;//递归的结束条件

ListNode\*a=head;

ListNode\*b=head;

for(int i=0;i<k;i++){

//下面是递归时一条链接的情况

if(b==nullptr)return head;//说明不是k的整数倍，直接返回此时的head

b=b->next;//把b移动到k的前一个位置

}

//把a到b之间的节点反转，不算b

ListNode\*newNode = reverseOperator(a,b);

a->next= reverseKGroup(b,k);//递归调用本函数，这个太难想到了

return newNode;

}

ListNode\* reverseOperator(ListNode\* n, ListNode\* b ){

ListNode\* pre=nullptr;

ListNode\*cur=n;

ListNode\*nxt=n;

//循环条件就限定了只反转b之前的节点

while(cur!=b){

nxt=cur->next;

cur->next=pre;

pre=cur;

cur=nxt;

}

return pre;

}

};

另一种写法：

class Solution {

public:

ListNode\* reverseKGroup(ListNode\* head, int k) {

ListNode\* pre=head;

int count=0;

while(pre!=NULL&&count<k){

pre=pre->next;

count++;

} //退出循环后pre指向第k+1个节点

if(count==k) {

pre=reverseKGroup(pre,k);

while(count>0){

/\* head指向本次循环反转指向的节点

temp指向下次循环反转指向的节点

\*/

ListNode\* temp=head->next;

head->next=pre;

pre=head;

head=temp;

count--;

}

head=pre;

}

return head;

}

};