链表专题

LeetCode 19. 删除链表的倒数第N个节点

19. 删除链表的倒数第N个节点

难度 中等 711

给定一个链表，删除链表的倒数第 *n* 个节点，并且返回链表的头结点。

示例：

```
给定一个链表：1->2->3->4->5, 和 n = 2.

当删除了倒数第二个节点后，链表变为 1->2->3->5.
```

说明：

给定的 *n* 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

```
/**双指针
    1.建立虚拟头结点
    2.first向后走n步
    3.first、second同时向后走，当first走到末尾时停止。
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode *dummy = new ListNode(-1);//创建虚拟头结点
        dummy->next = head;//虚拟头结点指向head节点
        ListNode *first = dummy, *second = dummy;//双指针
        for (int i = 0; i < n; i ++ ) first = first->next;
        while (first -> next)
        {
            first = first->next;
            second = second->next;
        }
        second->next = second->next->next;
        return dummy->next;//虚拟头结点的下一个节点，不能是head！！！
    }
};
```

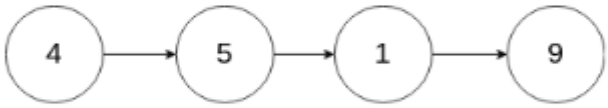
LeetCode 237. 删除链表中的节点

237. 删除链表中的节点

难度 简单 627

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

现有一个链表 -- head = [4,5,1,9]，它可以表示为:



示例 1:

```
输入: head = [4,5,1,9], node = 5
输出: [4,1,9]
解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

示例 2:

```
输入: head = [4,5,1,9], node = 1
输出: [4,5,9]
解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.
```

说明:

- 链表至少包含两个节点。
- 链表中所有节点的值都是唯一的。
- 给定的节点为非末尾节点并且一定是链表中的一个有效节点。
- 不要从你的函数中返回任何结果。

```cpp
/*直接指向下一个节点
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    void deleteNode(ListNode* node) {
        node->val = node->next->val;
        node->next = node->next->next;
        //还可以写成*(node) = *(node->next);
```

```
        //C++的特性，*(node)取node结构体的指针和地址！！！
    }
};
```

LeetCode 83.删除排序链表中的重复元素

83. 删除排序链表中的重复元素

难度 简单 266

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1:

```
输入：1->1->2
输出：1->2
```

示例 2:

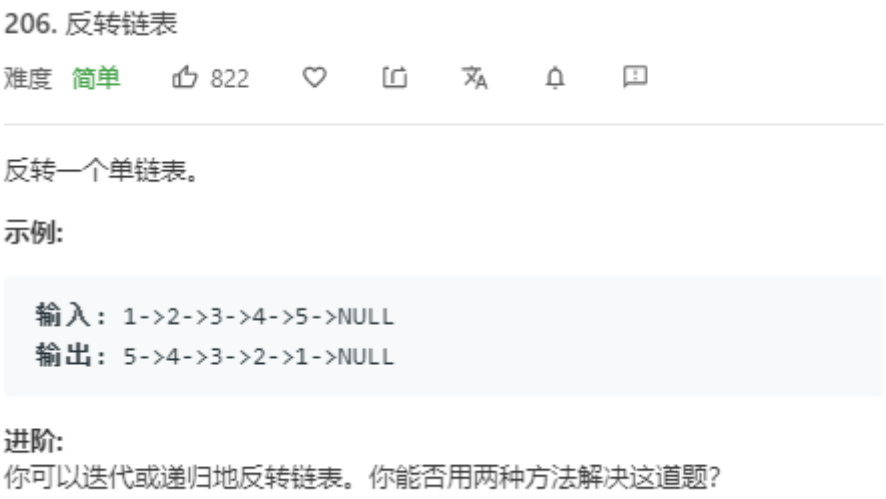
```
输入：1->1->2->3->3
输出：1->2->3
```

```
/*相同的都排在一起，只取第一个
1.下一个点和当前点相同，则删掉下一点
2.下一个点和当前点不相同，则移动指针
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head) return NULL;
        ListNode *first, *second;
        first = second = head;
        while (first)
        {
            if (first->val != second->val)
            {
                second->next = first;
                second = first;
            }
            first = first->next;
        }
        second->next = NULL;
        return head;
    }
};
```

简洁版：

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        auto cur = head;
        while(cur){
            if(cur->next && cur->next->val == cur->val)
                cur->next = cur->next->next;
            else cur = cur->next;
        }
        return head;
    }
};
```

LeetCode 206. 反转链表



206. 反转链表

难度 简单 822

反转一个单链表。

示例:

输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL

进阶:
你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

```
/*指针变方向，递归
双指针，
c = b->next
b ->next = a;
a = b,b = c;
b为空时a为头结点
链表形式：a--b--c....
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (!head) return NULL;
        if (!head->next) return head;
        ListNode *p = reverseList(head->next);
        head->next->next = head;
        head->next = NULL;
        return p;
    }
```

```
};
********************************************************************************
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(!head) return NULL;

        auto a = head,b = head->next;
        while(b)
        {
            auto c = b->next;
            b->next = a;
            a = b,b = c;
        }
        head->next = NULL;
        return a;
    }
};
```

LeetCode 92. 反转链表 II

92. 反转链表 II

难度 中等 312

反转从位置 *m* 到 *n* 的链表。请使用一趟扫描完成反转。

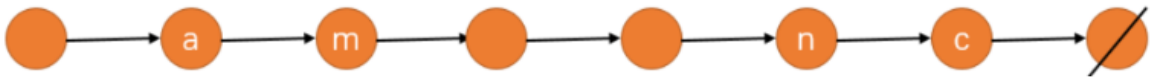
说明:
1 ≤ *m* ≤ *n* ≤ 链表长度。

示例:

```
输入: 1->2->3->4->5->NULL, m = 2, n = 4
输出: 1->4->3->2->5->NULL
```

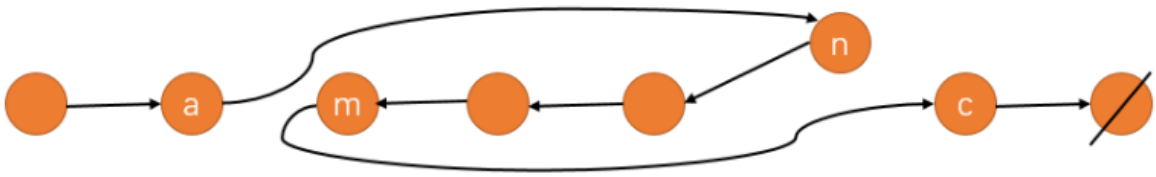
(模拟) $O(n)$

假设初始链表如下所示：



第一步，我们先将 m 到 n 之间的指针翻转（不包含第 m 个节点），如下所示：

第二步我们将 m 的指针指向 c，将 a 的指针指向 n，如下所示：



此时我们就完成了 m 到 n 之间的翻转！

时间复杂度分析：整个链表只遍历了一遍，所以时间复杂度是 $O(n)$。

```
/*先求三个点的位置
/**
```

```
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int m, int n) {
        if (m == n) return head;//不用操作
        ListNode *dummy = new ListNode(-1);
        dummy->next = head;
        ListNode *b = dummy;
        for (int i = 0; i < m - 1; i ++ ) b = b->next;
        ListNode *a = b;
        b = b->next;
        ListNode *c = b->next;
        for (int i = 0; i < n - m; i ++ )
        {
            ListNode *t = c->next;
            c->next = b;
            b = c, c = t;
        }
        ListNode *mp = a->next;
        ListNode *np = b;
        a->next = np, mp->next = c;
        return dummy->next;
    }
};
```

LeetCode 61. 旋转链表

61. 旋转链表

难度 中等 208

给定一个链表，旋转链表，将链表每个节点向右移动 k 个位置，其中 k 是非负数。

示例 1:

```
输入: 1->2->3->4->5->NULL, k = 2
输出: 4->5->1->2->3->NULL
解释:
向右旋转 1 步: 5->1->2->3->4->NULL
向右旋转 2 步: 4->5->1->2->3->NULL
```

示例 2:

```
输入: 0->1->2->NULL, k = 4
输出: 2->0->1->NULL
解释:
向右旋转 1 步: 2->0->1->NULL
向右旋转 2 步: 1->2->0->NULL
向右旋转 3 步: 0->1->2->NULL
向右旋转 4 步: 2->0->1->NULL
```

```
/*把最后k个移动到开头
1.k %= n 保证k在0 ~ n之间！！！
2.双指针，first指针从头往后走K步
3.first、second同时往后走，当first走到结尾时停止
4.移动指针
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head) return head;

        int n = 0;
        ListNode *p = head;
        while (p)
        {
            n ++ ;
            p = p->next;
        }

        k %= n;
        if (!k) return head;
```

```
        ListNode *first = head;
        while (k -- && first) first = first->next;
        ListNode *second = head;
        while (first->next)
        {
            first = first->next;
            second = second->next;
        }
        //移动指针
        first->next = head;
        head = second->next;
        second->next = NULL;
        return head;
    }
};

**********简洁版**********
 /**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if(!head) return NULL;

        int n = 0;
        for(auto p = head;p;p = p->next) n ++;

        k %= n;
        auto first = head,second = head;
        while(k --) first = first->next;
        while(first->next)
        {
            first = first->next;
            second = second->next;
        }
        first->next = head;
        head = second->next;
        second->next = NULL;

        return head;
    }
};
```

LeetCode 24.两两交换链表中的节点

24. 两两交换链表中的节点

难度 中等 427

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例:

```
给定 1->2->3->4, 你应该返回 2->1->4->3.
```

通过次数 78,145 | 提交次数 120,805

```
/*相邻两点之间交换
1.建立虚拟头结点
2.交换指针，移动指针
p->next = b;
a->next = b->next;
b->next = a;
p = a;
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        auto dummy = new ListNode(-1);
        dummy->next = head;

        for(auto p = dummy;p && p->next && p->next->next;p = p->next)
        {
            auto a = p->next,b = a->next;
            p->next = b;
            a->next = b->next;
            b->next = a;
            p = a;
        }
        return dummy->next;
    }
};
```

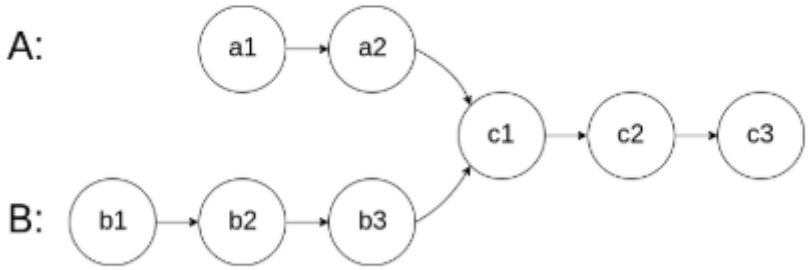
LeetCode 160. 相交链表 !!!

160. 相交链表

难度 简单 554

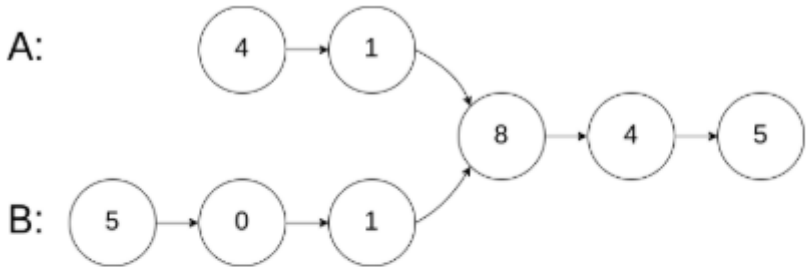
编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



在节点 c1 开始相交。

示例 1：



输入： intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出： Reference of the node with value = 8
输入解释： 相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

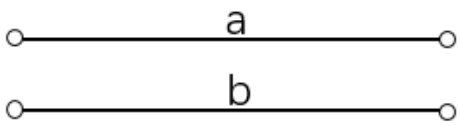
这题的思路很巧妙，我们先给出做法，再介绍原理。

算法步骤：

1. 用两个指针分别从两个链表头部开始扫描，每次分别走一步；
2. 如果指针走到 `null`，则从另一个链表头部开始走；
3. 当两个指针相同时，
   - 如果指针不是 `null`，则指针位置就是相遇点；
   - 如果指针是 `null`，则两个链表不相交；

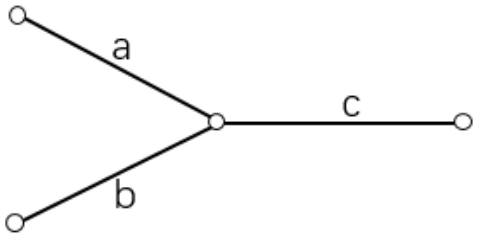
此题我们画图讲解，一目了然：

1. 两个链表不相交：



a, b 分别代表两个链表的长度，则两个指针分别走 $a + b$ 步后都变成 `null`。

2. 两个链表相交：



则两个指针分别走 $a + b + c$ 步后在两链表交汇处相遇。

时间复杂度分析：每个指针走的长度不大于两个链表的总长度，所以时间复杂度是 $O(n)$。

```
/*p,q都走完a+b+c之后一定相遇！！！妙啊~
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        ListNode *p = headA, *q = headB;
        while (p != q)
        {
            if (p) p = p->next;
            else p = headB;
            if (q) q = q->next;
            else q = headA;
        }
        return p;
    }
};
```

LeetCode 142. 环形链表 II

142. 环形链表 II

难度 中等 383

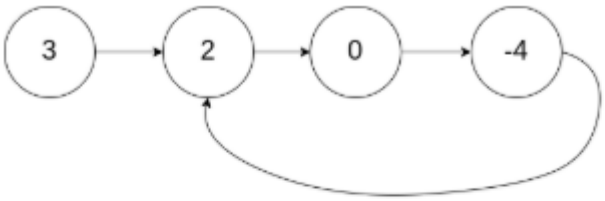
给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 `null`。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。

说明： 不允许修改给定的链表。

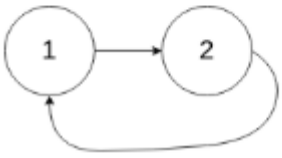
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：tail connects to node index 1
解释：链表中有一个环，其尾部连接到第二个节点。
```

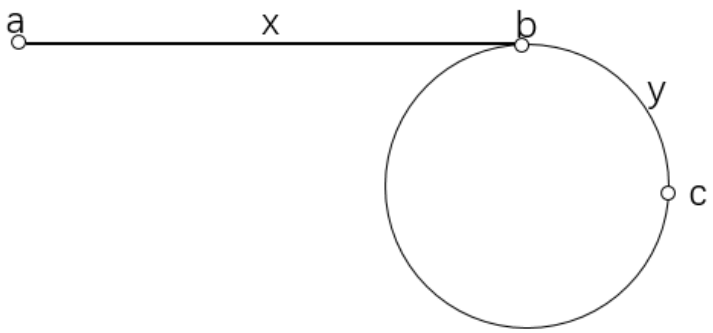


示例 2：

```
输入：head = [1,2], pos = 0
输出：tail connects to node index 0
解释：链表中有一个环，其尾部连接到第一个节点。
```



本题的做法比较巧妙。
用两个指针 $first, second$ 分别从起点开始走，$first$ 每次走一步，$second$ 每次走两步。
如果过程中 $second$ 走到 `null`，则说明不存在环。否则当 $first$ 和 $second$ 相遇后，让 $first$ 返回起点，$second$ 待在原地不动，然后两个指针每次分别走一步，当相遇时，相遇点就是环的入口。



证明：如上图所示，a 是起点，b 是环的入口，c 是两个指针的第一次相遇点，ab 之间的距离是 x，bc 之间的距离是 y。
则当 $first$ 走到 b 时，由于 $second$ 比 $first$ 多走一倍的路，所以 $second$ 已经从 b 开始在环上走了 x 步，可能多余1圈，距离 b 还差 y 步（这是因为第一次相遇点在 b 之后 y 步，我们让 $first$ 退回 b 点，则 $second$ 会退 $2y$ 步，也就是距离 b 点还差 y 步）；所以 $second$ 从 b 点走 $x+y$ 步即可回到 b 点，所以 $second$ 从 c 点开始走，走 x 步即可恰好走到 b 点，同时让 $first$ 从头开始走，走 x 步也恰好可以走到 b 点。所以第二次相遇点就是 b 点。

另外感谢@watay147提供的另一种思路，可以用公式来说明：a, b, c, x, y 的含义同上，我们用 z 表示从 c 点顺时针走到 b 的距离。则第一次相遇时 $second$ 所走的距离是 $x+(y+z)*n+y$，n 表示圈数，同时 $second$ 走过的距离是 $first$ 的两倍，也就是 $2(x+y)$，所以我们有
$x+(y+z)*n+y=2(x+y)$，所以 $x=(n-1)\times(y+z)+z$。那么我们让 $second$ 从 c 点开始走，走 x 步，会恰好走到 b 点；让 $first$ 从 a 点开始走，走 x 步，也会走到 b 点。

时间复杂度分析：$first$ 总共走了 $2x+y$ 步，$second$ 总共走了 $2x+2y+x$ 步，所以两个指针总共走了 $5x+3y$ 步。由于当第一次 $first$ 走到 b 点时，$second$ 最多追一圈即可追上 $first$，所以 y 小于环的长度，所以 $x+y$ 小于等于链表总长度。所以总时间复杂度是 $O(n)$。

```
/*
1.快慢指针，快的每次走一步，慢的每次走两步。
2.在环中每走y步两指针距离就少一
3.相遇后把慢的放到开头
4.快慢每次都走一步，当再次相遇时就是环的入口
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if (!head || !head->next) return 0;
        ListNode *first = head, *second = head;

        while (first && second)
        {
            first = first->next;
            second = second->next;
```

```
            if (second) second = second->next;
            else return 0;

            if (first == second)
            {
                first = head;
                while (first != second)
                {
                    first = first->next;
                    second = second->next;
                }
                return first;
            }
        }

        return 0;
    }
};

**********************************************************************
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        auto fast = head,slow = head;
        while(fast)
        {
            fast = fast->next;
            slow = slow->next;
            if(fast) fast = fast->next;
            else break;

            if(fast == slow)
            {
                fast = head;
                while(slow != fast)
                {
                    fast = fast->next;
                    slow = slow->next;
                }
                return slow;
            }
        }
        return NULL;
    }
};
```

LeetCode 148. 排序链表

148. 排序链表

难度 中等 433

在 O(*n* log *n*) 时间复杂度和常数级空间复杂度下，对链表进行排序。

示例 1:

```
输入：4->2->1->3
输出：1->2->3->4
```

示例 2:

```
输入：-1->5->3->4->0
输出：-1->0->3->4->5
```

通过次数 44,584 | 提交次数 69,587

自顶向下递归形式的归并排序，由于递归需要使用系统栈，递归的最大深度是 $logn$，所以需要额外 $O(logn)$ 的空间。
所以我们需要使用自底向上非递归形式的归并排序算法。
基本思路是这样的，总共迭代 $logn$ 次：

1. 第一次，将整个区间分成连续的若干段，每段长度是2：$[a_0,a_1],[a_2,a_3],...[a_{n-1},a_{n-1}]$，然后将每一段内排好序，小数在前，大数在后；
2. 第二次，将整个区间分成连续的若干段，每段长度是4：$[a_0,...,a_3],[a_4,...,a_7],...[a_{n-4},...,a_{n-1}]$，然后将每一段内排好序，这次排序可以利用之前的结果，相当于将左右两个有序的半区间合并，可以通过一次线性扫描来完成；
3. 依此类推，直到每段小区间的长度大于等于 n 为止；

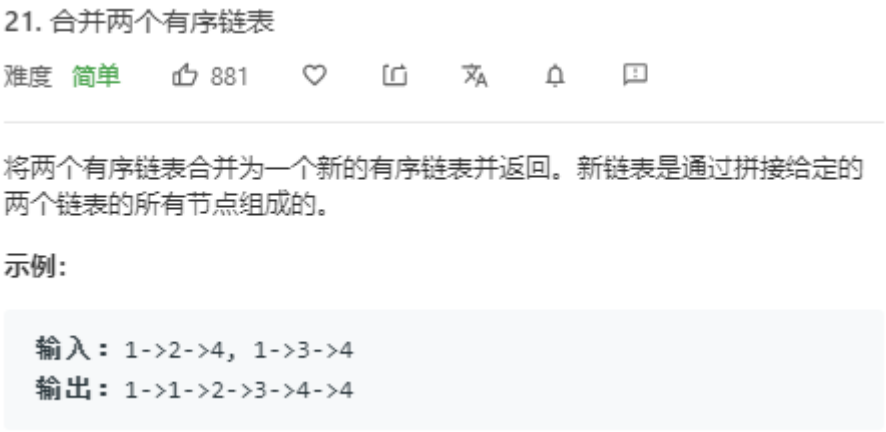
另外，当 n 不是2的整次幂时，每次迭代只有最后一个区间会比较特殊，长度会小一些，遍历到指针为空时需要提前结束。

时间复杂度分析：整个链表总共遍历 $logn$ 次，每次遍历的复杂度是 $O(n)$，所以总时间复杂度是 $O(nlogn)$。
空间复杂度分析：整个算法没有递归，迭代时只会使用常数个额外变量，所以额外空间复杂度是 O(1)$.

```
/*不能用递归，用归并排序
空间复杂度O(1)
/**
 * Definition for singly-linked list.
 * struct ListNode {
```

```
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        int n = 0;
        for (ListNode *p = head; p; p = p->next) n ++ ;

        ListNode *dummy = new ListNode(-1);
        dummy->next = head;
        for (int i = 1; i < n; i *= 2)
        {
            ListNode *begin = dummy;
            for (int j = 0; j + i < n; j += i * 2)
            {
                ListNode *first = begin->next, *second = first;
                for (int k = 0; k < i; k ++ )
                    second = second->next;
                int f = 0, s = 0;
                while (f < i && s < i && second)
                    if (first->val < second->val)
                    {
                        begin = begin->next = first;
                        first = first->next;
                        f ++ ;
                    }
                    else
                    {
                        begin = begin->next = second;
                        second = second->next;
                        s ++ ;
                    }

                while (f < i)
                {
                    begin = begin->next = first;
                    first = first->next;
                    f ++ ;
                }
                while (s < i && second)
                {
                    begin = begin->next = second;
                    second = second->next;
                    s ++ ;
                }

                begin->next = second;
            }
        }

        return dummy->next;
    }
};
```

LeetCode 21. 合并两个有序链表



21. 合并两个有序链表

难度 简单 881

将两个有序链表合并为一个新的有序链表并返回。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：

```
输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

```
/*建立虚拟节点，合并节点。
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        auto dummy = new ListNode(-1);
        auto p = dummy;
        while(l1 && l2)
        {
            if (l1->val < l2->val)
            {
                p->next = l1;
                p = l1;
                l1 = l1->next;
            }
            else
            {
                p->next = l2;
                p = l2;
                l2 = l2->next;
            }
        }
        if (!l1) l1 = l2;
        while(l1)
        {
            p->next = l1;
            p = l1;
            l1 = l1->next;
        }
        return dummy->next;
    }
};
```

LeetCode 141. 环形链表

141. 环形链表

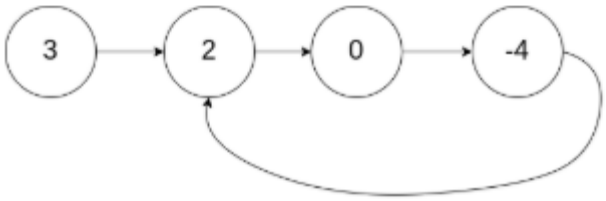
难度 简单 511

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。

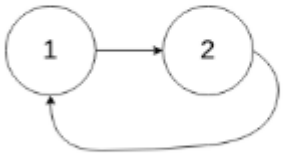
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。
```



示例 3：

```
输入：head = [1], pos = -1
输出：false
解释：链表中没有环。
```

(链表，指针扫描) $O(n)$

用两个指针从头开始扫描，第一个指针每次走一步，第二个指针每次走两步。如果走到 `null`，说明不存在环；否则如果两个指针相遇，则说明存在环。

为什么呢?
假设链表存在环，则当第一个指针走到环入口时，第二个指针已经走到环上的某个位置，距离环入口还差 x 步。
由于第二个指针每次比第一个指针多走一步，所以第一个指针再走 x步，两个指针就相遇了。

时间复杂度分析：第一个指针在环上走不到一圈，所以第一个指针走的总步数小于链表总长度。而第二个指针走的路程是第一个指针的两倍，所以总时间复杂度是 $O(n)$。

```cpp
/*快慢指针，只要有环就一定能追上！！！
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(!head || !head->next) return NULL;
        auto first = head,second = head->next;
        while(first && second)
        {
            if(first == second) return true;
            first = first->next;
            second = second->next;
            if(second) second = second->next;
        }
        return false;
    }
};
```

LeetCode 147. 对链表进行插入排序

插入排序的动画演示如上。从第一个元素开始，该链表可以被认为已经部分排序（用黑色表示）。
每次迭代时，从输入数据中移除一个元素（用红色表示），并原地将其插入到已排好序的链表中。

插入排序算法：

1. 插入排序是迭代的，每次只移动一个元素，直到所有元素可以形成一个有序的输出列表。
2. 每次迭代中，插入排序只从输入数据中移除一个待排序的元素，找到它在序列中适当的位置，并将其插入。
3. 重复直到所有输入数据插入完为止。

示例 1：

```
输入：4->2->1->3
输出：1->2->3->4
```

示例 2：

```
输入：-1->5->3->4->0
输出：-1->0->3->4->5
```

```
/*建立虚拟头结点，指向原链表头部。
然后扫描原链表，对于每个节点v，从前往后扫描结果链表，找到第一个比v大的节点u，将v插入到u之前。
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        auto dummy = new ListNode(-1);
        while(head)
        {
            auto p = head->next,q = dummy;
            while(q->next && q->next->val <= head->val) q = q->next;

            head->next = q->next;
            q->next = head;

            head = p;
        }
        return dummy->next;
    }
};
```

LeetCode 146. LRU缓存机制 !!!

146. LRU缓存机制

难度 中等 433

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 `get` 和 写入数据 `put` 。

获取数据 `get(key)` - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。
写入数据 `put(key, value)` - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 **O(1)** 时间复杂度内完成这两种操作?

示例:

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得密钥 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得密钥 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

通过次数 36,674 | 提交次数 79,508

使用两个双链表和一个哈希表：

- 第一个双链表存储未被使用的位置；
- 第二个双链表存储已被使用的位置，且按最近使用时间从左到右排好序；
- 哈希表存储key对应的链表中的节点地址；

初始化：

- 第一个双链表插入 n 个节点，n 是缓存大小；
- 第二个双链表和哈希表都为空；

`get(key)`：
首先用哈希表判断key是否存在：

- 如果key存在，则返回对应的value，同时将key对应的节点放到第二个双链表的最左侧；
- 如果key不存在，则返回-1；

`set(key, value)`：
首先用哈希表判断key是否存在：

- 如果key存在，则修改对应的value，同时将key对应的节点放到第二个双链表的最左侧；
- 如果key不存在：
  - 如果缓存已满，则删除第二个双链表最右侧的节点（上次使用时间最老的节点），同时更新三个数据结构；
  - 否则，插入(key, value)：从第一个双链表中随便找一个节点，修改节点权值，然后将节点从第一个双链表删除，插入第二个双链表最左侧，同时更新哈希表；

时间复杂度分析：双链表和哈希表的增删改查操作的时间复杂度都是 $O(1)$，所以 `get` 和 `set` 操作的时间复杂度也都是 $O(1)$。

```
class LRUCache {
public:
    struct Node
    {
        int val, key;
        Node *left, *right;
        Node() : key(0), val(0), left(NULL), right(NULL) {}
    };
    Node *hu, *tu; // hu: head_used, tu: tail_used; head在左侧，tail在右侧
    Node *hr, *tr; // hr: head_remains, tr: tail_remains; head在左侧，tail在右侧
    int n;
    unordered_map<int, Node*> hash;

    void delete_node(Node *p)
    {
        p->left->right = p->right, p->right->left = p->left;
    }

    void insert_node(Node *h, Node *p)
    {
        p->right = h->right, h->right = p;
        p->left = h, p->right->left = p;
    }

    LRUCache(int capacity) {
        n = capacity;
        hu = new Node(), tu = new Node();
        hr = new Node(), tr = new Node();
        hu->right = tu, tu->left = hu;
        hr->right = tr, tr->left = hr;
```

```
        for (int i = 0; i < n; i ++ )
        {
            Node *p = new Node();
            insert_node(hr, p);
        }
    }

    int get(int key) {
        if (hash[key])
        {
            Node *p = hash[key];
            delete_node(p);
            insert_node(hu, p);
            return p->val;
        }
        return -1;
    }

    void put(int key, int value) {
        if (hash[key])
        {
            Node *p = hash[key];
            delete_node(p);
            insert_node(hu, p);
            p->val = value;
            return;
        }

        if (!n)
        {
            n ++ ;
            Node *p = tu->left;
            hash[p->key] = 0;
            delete_node(p);
            insert_node(hr, p);
        }

        n -- ;
        Node *p = hr->right;
        p->key = key, p->val = value, hash[key] = p;
        delete_node(p);
        insert_node(hu, p);
    }
};

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
 */
```

LeetCode 138.复制带随机指针的链表

138. 复制带随机指针的链表

难度 中等 207

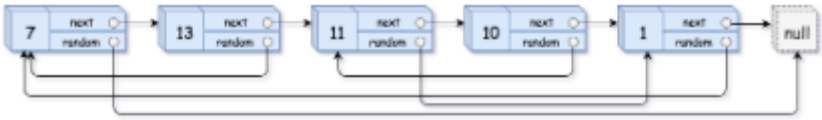
给定一个链表，每个节点包含一个额外增加的随机指针，该指针可以指向链表中的任何节点或空节点。

要求返回这个链表的 **深拷贝**。

我们用一个由 `n` 个节点组成的链表来表示输入/输出中的链表。每个节点用一个 `[val, random_index]` 表示：

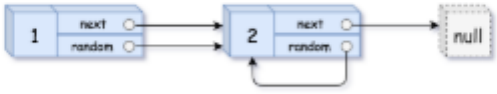
- `val`：一个表示 `Node.val` 的整数。
- `random_index`：随机指针指向的节点索引（范围从 `0` 到 `n-1`）；如果不指向任何节点，则为 `null` 。

示例 1：




```
输入：head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出：[[7,null],[13,0],[11,4],[10,2],[1,0]]
```

示例 2：




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

(哈希表) $O(n)$

用哈希表维护新旧链表节点之间的对应关系。

从前往后扫描旧链表，对于每个节点的两条边（`*next` 以及 `*random`），如果新链表中对应的点还未创建，则创建节点，并将新节点与旧链表中的节点关联起来，然后根据节点之间的映射关系，在新链表中添加这两条边（`*next` 以及 `*random`）。

时间复杂度分析：整个链表仅被扫描一遍，所以时间复杂度是 $O(n)$。

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* next;
    Node* random;
```

```
    Node() {}

    Node(int _val, Node* _next, Node* _random) {
        val = _val;
        next = _next;
        random = _random;
    }
};
*/
class Solution {
public:
    Node *copyRandomList(Node *head) {
        if (!head) return 0;
        unordered_map<Node*, Node*> hash;
        Node *root = new Node(head->val, NULL, NULL);
        hash[head] = root;
        while (head->next)
        {
            if (hash.count(head->next) == 0)
                hash[head->next] = new Node(head->next->val, NULL, NULL);
            hash[head]->next = hash[head->next];

            if (head->random && hash.count(head->random) == 0)
                hash[head->random] = new Node(head->random->val, NULL, NULL);
            hash[head]->random = hash[head->random];

            head = head->next;
        }

        if (head->random && hash.count(head->random) == 0)
            hash[head->random] = new Node(head->random->val, NULL, NULL);
        hash[head]->random = hash[head->random];

        return root;
    }
};
```