剑指offer25 合并两个排序的链表

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode m = new ListNode(0), current=m;
        while(l1 != null && l2 != null ){
            if(l1.val <= l2.val){
                current.next = l1;
                l1 = l1.next;
            }
            else{
                current.next = l2;
                l2 = l2.next;
            }
            current = current.next;
        }
        current.next = l1 != null ? l1 : l2;
        return m.next;
    }
}
```

剑指offer22 链表中倒数第k个节点

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
 //先遍历一遍，求出个数
class Solution {
    public ListNode getKthFromEnd(ListNode head, int k) {
        ListNode m = head, n = head;
        int length1 = 0, length2 = 0;
        while(m != null){
            length1++;
            m = m.next;
        }
        while(true){
```

```
            length2++;
            if(length2 == length1 - k + 1)
                return n;
            n = n.next;
        }
    }
}
```

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode getKthFromEnd(ListNode head, int k) {
        ListNode fast = head, low = head;
        int time_fast = 0, time_low = 0;
        while(fast != null){
            time_fast++;
            fast = fast.next;
            if(time_fast > k){ //先走k步
                low = low.next;
            }
        }
        return low;
    }
}
```

剑指offer 02.07 链表相交

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if(headA == null || headB == null)
            return null;
        ListNode m = headA, n = headB;
```

```
        int a = 0 , b = 0;
        while(m != n){
            if(m != null)
                m = m.next;
            else
                m = headB;
            if(n != null)
                n = n.next;
            else
                n = headA;
        }
        return m;
    }
}
```

```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        int lenA = 0, lenB = 0;
        ListNode m = headA, n = headB;
        //先计算长度
        while(m != null){
            lenA++;
            m = m.next;
        }
        while(n != null){
            lenB++;
            n = n.next;
        }
        //计算长度差
        int diffLen = lenA - lenB;
        m = headA;
        n = headB;
        //让短的先走
        if(diffLen >= 0){
            while(diffLen-- != 0)
                m = m.next;
        }
        else{
            while(diffLen++ != 0)
                n = n.next;
        }
        //相同时则找到公共结点
        while(m != null && n != null && m != n){
            m = m.next;
            n = n.next;
        }
        return m;
    }
}
```

141. 环形链表

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        Set<ListNode> set = new HashSet<>();
        while(head != null){
            if(set.contains(head)){
                return true;
            }
            else{
                set.add(head);
            }
            head = head.next;
        }
        return false;
    }
}
```

通过使用具有 不同速度 的快、慢两个指针遍历链表，空间复杂度可以被降低至 O(1)O(1)O(1)。慢指针每次移动一步，而快指针每次移动两步。

如果列表中不存在环，最终快指针将会最先到达尾部，此时我们可以返回 `false`。

现在考虑一个环形链表，把慢指针和快指针想象成两个在环形赛道上跑步的运动员（分别称之为慢跑者与快跑者）。而快跑者最终一定会追上慢跑者。这是为什么呢？考虑下面这种情况（记作情况 A）- 假如快跑者只落后慢跑者一步，在下一次迭代中，它们就会分别跑了一步或两步并相遇。

其他情况又会怎样呢？例如，我们没有考虑快跑者在慢跑者之后两步或三步的情况。但其实不难想到，因为在下一次或者下下次迭代后，又会变成上面提到的情况 A。

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
```

```
 */
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        if(head == null || head.next == null)
            return false;
        ListNode slow = head, fast = head.next;
        while(slow != fast){
            if(fast == null && fast.next == null)
                return false;
            slow = slow.next;
            fast = fast.next.next;
        }
        return true;
    }
}
```

剑指 Offer 24. 反转链表

> 定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode m1 = new ListNode(0), m2=m1;
        Stack<Integer> stack = new Stack();
        while(head != null){
            stack.push(head.val);
            head = head.next;
        }
        while(!stack.isEmpty()){
            ListNode n = new ListNode(stack.pop());
            m2.next = n;
            m2 = n;
```

```
        }
        return m1.next;
    }
}
```

双链表

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode newHead = null;
        while(head != null){
            ListNode temp = head.next;

            //将原来结点取下挂在新链表上
            head.next = newHead;
            newHead = head;

            head = temp;
        }
        return newHead;
    }
}
```

三指针

```
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head == null)
            return null;

        //定义三个指针
        ListNode ptr_pre = head;
        ListNode ptr_cur = head.next;
        if(ptr_cur == null)
            return head;
        ListNode ptr_next = ptr_cur.next;

        ptr_pre.next = null;
        while(ptr_next != null){
            ptr_cur.next = ptr_pre; //反转

            //向右移动
            ptr_pre = ptr_cur;
            ptr_cur = ptr_next;
            ptr_next = ptr_next.next;
        }

        //最后一个指向其前一个
        ptr_cur.next = ptr_pre;
```

```
            return ptr_cur;
        }
    }
```

递归(首递归)

```
public ListNode reverseList(ListNode head) {
    //终止条件
    if (head == null || head.next == null)
        return head;
    //保存当前节点的下一个结点
    ListNode next = head.next;
    //从当前节点的下一个结点开始递归调用
    ListNode reverse = reverseList(next);
    //reverse是反转之后的链表，因为函数reverseList
    // 表示的是对链表的反转，所以反转完之后next肯定
    // 是链表reverse的尾结点，然后我们再把当前节点
    //head挂到next节点的后面就完成了链表的反转。
    next.next = head;
    //这里head相当于变成了尾结点，尾结点都是为空的，
    //否则会构成环
    head.next = null;

    return reverse;
}
```

递归(尾递归)
~~有些看不懂 可放弃~~

```
class Solution {
    public ListNode reverseList(ListNode head) {
         return reverseListInt(head, null);
    }
    private ListNode reverseListInt(ListNode head, ListNode newHead) {
        if(head == null)
            return newHead;

        ListNode next = head.next;
        head.next = newHead;

        ListNode node = reverseListInt(next, head);
        return node;
    }
}
```

删除表的结点

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode deleteNode(ListNode head, int val) {
        if(head == null)
            return null;
        if(head.val ==  val)
            return head.next;
        ListNode a = head;
        while(a.next != null){
            if(a.next.next == null && a.next.val == val){
                a.next = null ;
                break;
            }
            if(a.next.val == val){
                a.next = a.next.next;
            }
            a = a.next;
        }
        return head;
    }
}
```

剑指 Offer 35. 复杂链表的复制

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

    public Node(int val) {
        this.val = val;
        this.next = null;
        this.random = null;
    }
}
*/
class Solution {
    public Node copyRandomList(Node head) {
        if(head == null)
            return null;
        Node n = head;
```

```
        HashMap<Node, Node> map =  new HashMap<>();
        while(n != null){
            map.put(n,new Node(n.val));
            n = n.next;
        }
        n = head;
        while(n != null){
            map.get(n).next = map.get(n.next); //给map中v值赋值
            map.get(n).random  = map.get(n.random);
            n = n.next;
        }
        return map.get(head);
    }
}
```

876. 链表的中间结点

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode middleNode(ListNode head) {
        ListNode a = head;
        int length1 = 0, length2 = 1;
        while(a != null){
            length1++;
            a = a.next;
        }
        a = head;
        while(length2 <= length1/2){
            a = a.next;
            length2++;
        }
        return a;
    }
}

//快慢指针法
class Solution {
    public ListNode middleNode(ListNode head) {
        ListNode slow = head, fast = head;
        while(fast != null && fast.next != null){ //位置不能变，fast不为null了才会判
断fast.next，否则会出现空指针异常
            slow = slow.next;
            fast = fast.next.next;
        }
```

```
        return slow;
    }
}

//数组
class Solution {
    public ListNode middleNode(ListNode head) {
        ListNode[] arr = new ListNode[100];
        int length = 0;
        while(head != null){
            arr[length++] = head;
            head = head.next;
        }
        return arr[length/2];

    }
}
```

142. 环形链表 II

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */

public class Solution {
    public ListNode detectCycle(ListNode head) {
        if(head == null || head.next == null)
            return null;
        HashMap<ListNode, Integer> map = new HashMap<>();
        int i = 0;
        while(head != null){
            if(map.containsKey(head)){
                return head;
            }
            map.put(head, i);
            head = head.next;
        }
        return null;
    }
}

public class Solution {
```

```
    public ListNode detectCycle(ListNode head) {
        if(head == null || head.next == null)
            return null;
        HashMap<ListNode, Integer> map = new HashMap<>();
        Set<ListNode> visitedNode = new HashSet<>();
        while(head != null){
            if(visitedNode.contains(head))
                return head;
            visitedNode.add(head);
            head = head.next;
        }
        return null;
    }
}

//folyd算法(https://leetcode-cn.com/problems/linked-list-cycle-ii/solution/xiang-xi-tu-jie-ken-ding-kan-de-ming-bai-by-xixili/)
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if(head == null || head.next == null)
            return null;
        ListNode slow = head, fast = head;
        while(fast != null && fast.next != null){
            slow = slow.next;
            fast = fast.next.next;
            if(slow == fast){
                fast = head;
                while(slow != fast){
                    slow = slow.next;
                    fast = fast.next;
                }
                return slow;
            }
        }
        return null;
    }

}
```

1290. 二进制链表转整数

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public int getDecimalValue(ListNode head) {
```

```
        int sum = 0;
        while(head != null){
            sum = sum * 2 + head.val;  // 题解中的是运用了反向运算操作，我们在获得二进制的时候是除于2取余数，要计算被除数则是要商乘于2加余数。
            head = head.next;
        }
        return sum;
    }
}


class Solution {
    public int getDecimalValue(ListNode head) {
        Stack<Integer> stack = new Stack();
        while(head != null){
            stack.push(head.val);
            head = head.next;
        }
        int size = stack.size(), sum = 0;
        for(int i = 0; i < size; i++){
            sum += (Math.pow(2, i)) * stack.pop();
        }
        return sum;
    }
}
```

86. 分隔链表

> 给定一个链表和一个特定值 x，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。

> 你应当保留两个分区中每个节点的初始相对位置。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode partition(ListNode head, int x) {
        List<Integer> list1 =  new LinkedList<>(), list2 =  new LinkedList<>();
        ListNode n = head;
        while(n != null){
            if(n.val < x){
                list1.add(n.val);
            }
            else{
```

```
                list2.add(n.val);
            }
            n = n.next;
        }
        Iterator<Integer> iterator1 = list1.iterator();
        Iterator<Integer> iterator2 = list2.iterator();
        n = head;
        while(iterator1.hasNext()){
            n.val = iterator1.next();
            n = n.next;
        }
        while(iterator2.hasNext()){
            n.val = iterator2.next();
            n = n.next;
        }
        return head;
    }
}
```

两个链表分别保存大于或小于x的值

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode partition(ListNode head, int x) {
        ListNode low = new ListNode(0), low0 = low, high = new ListNode(0), high0
= high;
        while(head != null){
            if(head.val < x){
                low.next = new ListNode(head.val);
                low = low.next;
            }
            else{
                high.next = new ListNode(head.val);
                high =high.next;
            }
            head = head.next;
        }
        low.next = high0.next;
        return low0.next;
    }
}
```

```
class Solution {
    public String countAndSay(int n) {
        if(n == 1){
            return "1";
        }
        String preString = countAndSay(n-1);
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < preString.length();){
            int count = 0;
            for(int j = i+1; j < preString.length(); j++){
                if(preString[i] == preString[j]){
                    count++;
                }
                else{
                    j--;
                    i = j;
                    break;
                }
            }
            result.append(count+preString[i]);
        }
        return result;
    }
}
```

剑指 Offer 06. 从尾到头打印链表

子烁

递归

```
class Solution {
    List<Integer> resList = new ArrayList<>();
    public int[] reversePrint(ListNode head) {
        getResList(head);
        int[] res = new int[resList.size()];
        int n = 0;
        for(int num : resList){
            res[n] = num;
            n++;
        }
        return res;
    }

    private List<Integer> getResList(ListNode node){
        if(node != null){
            getResList(node.next);
            resList.add(node.val);
        }
```

```
        return resList;
    }
}
```

栈

```
class Solution {
    public int[] reversePrint(ListNode head) {
        Stack<ListNode> stack=new Stack<ListNode>();
        ListNode Temp=head;
        while(Temp!=null){
            stack.push(Temp);
            Temp=Temp.next;
        }
        int size=stack.size();
        int[] result=new int[size];
        for(int i=0;i<size;i++){
            result[i]=stack.pop().val;
        }
        return result;
    }
}
```

循环插入首结点

```
class Solution {
    public int[] reversePrint(ListNode head) {
        List<Integer> res = new ArrayList<>();
        ListNode node = head;
        while(node != null){
            res.add(0, node.val);
            node = node.next;
        }
        int[] result = new int[res.size()];
        int n = 0;
        for(int num : res){
            result[n] = num;
            n++;
        }
        return result;
    }
}
```

面试题 17.04. 消失的数字

> 数组nums包含从0到n的所有整数，但其中缺了一个。请编写代码找出那个缺失的整数。你有办法在O(n)时间内完成吗?

```
class Solution {
    public int missingNumber(int[] nums) {
        Arrays.sort(nums);
        for(int i = 0; i < nums.length; i++){
            if(i != nums[i])
                return i;
        }
        return nums.length;
    }
}
```

数组的长度应该就是其最大值

```
class Solution {
    public int missingNumber(int[] nums) {
        int sum = nums.length;
        for(int i = 0; i < nums.length; i++){
            sum -= nums[i];
            sum += i;
        }
        return sum;
    }
}
```

异或

> i, nums[i]（有序数组） 相与应该全为0（如果不缺的话）

```
class Solution {
    public int missingNumber(int[] nums) {

        int res = 0;

        for(int i = 1; i <= nums.length; i++){ //一定是等于，因为少了一位
            res ^= i;

        }

        for(int num : nums){
            res ^= num;
        }

        return res;
    }
}
```

面试题 04.02. 最小高度树

> 给定一个有序整数数组，元素各不相同且按升序排列，编写一个算法，创建一棵高度最小的二叉搜索树。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return helper(nums, 0, nums.length - 1);
    }

    private TreeNode helper(int[] nums, int low, int high){
        if (low > high) {   // low > high表示子数组为空
            return null;
        }
        // 以mid作为根节点
        int mid = (high - low) / 2 + low;
        TreeNode node = new TreeNode(nums[mid]);
        // 左子数组[low, mid -1]构建左子树
        node.left = helper(nums, low, mid - 1);
        // 右子数组[mid + 1, high]构建右子树
        node.right = helper(nums,mid + 1, high);
        return node;
}
```

237. 删除链表中的节点

> 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。传入函数的唯一参数为 要被删除的节点 。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public void deleteNode(ListNode node) {
        node.val = node.next.val;
        node.next = node.next.next;
    }
}
```

删除链表中的重复结点

子烁

> 在一个排序的链表中，存在这重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。例如，链表1->2->3->3->4->4->5处理后为1->2->5.

```
public class Offer_54{
    publick ListNode deletDuplication(ListNode pHead){
        if(pHead == null || pHead.next == null)
            return Phead;
        ListNode temp = new List(Integer.MIN_VALUE);    //为了防止第一个元素就是重复元素的情况
        temp.next = pHead;
        ListNode pre =  temp;
        ListNode cur =  temp;
        while(curr != null){
            //如果当前元素就是重复元素，就一直遍历，直至下一个元素和当前元素不一样
            while(cur.next != null && cur.val == cur.next.val)
                cur = cur.next;
            cur = cur.next;
            //如果下一个重复元素，就continue跳出当前循环，从头又开始判断
            if(cur != null && cur.next != null && cur.val == cur.next.val)
                continue;
            pre.next = cur;
            pre = pre.next;
        }
        return temp.next;
    }
}
```

【滑动窗口】【动态规划】剑指 Offer 42. 连续子数组的最大和

滑动窗口

```
class Solution {
    public int maxSubArray(int[] nums) {
        int left = 0;
        int res = Integer.MIN_VALUE;
        int sum = 0;    //记录滑动窗口中所有数的和
        for(int right = 0; right < nums.length; right++){
            sum += nums[right];
            result = Math.max(res, sum);
            //如果当前和为负数，则右移，注意left = right时left也要右移动(sum就是比当前负数大一点的负数，然后一减去sum就为0或正数，然后for循环后，right++,right又等于left了，不矛盾)
            while(left <= right && sum <= 0){
                sum -= nums[left];
```

```
                    left++;
                }
            }
        }
    }
```