340. 至多包含 K 个不同字符

样例1：
输入：S="e ceb a"并且k=3
输出：4
解释：T="e ceb "
样例2：
输入：S="W ORL D"并且k=4
输出：4
解释：T="W ORL " 或"OR LD"

```
public class Solution {
        public int length OfL ong est Sub str ing -
KDi sti nct (String s, int k) {

               if (s.len gth() == 0 || k == 0) {
                        return 0;
               }

               int left = 0, right = 0, cnt = 0;
               int charSet[] = new int[256];
               int ans = 0;

                while (right < s.leng th()) {
                       // 统计righ t指向的字符
                       // 当字符在窗口 内第一 次出现 -
时，字 符种类 数+1 ，该字 符出现次数+1
                       if (charS et[ s.c har At( -
right)] == 0) {
                                cnt++;
                       }
                        cha rSe t[s.ch arA t(r -
igh t)]++;
                        rig ht++;

                       // 向右移动le ft， 保持窗 口内
只 有k种 不同的字符
                        while (cnt > k) {
                                cha rSe t[s.ch -
arA t(l eft )]--;
                               // 当该字符在本 窗口
不 再出现 时，字 符种类数-1
```

340. 至多包含 K 个不同字符 (cont)

```
>               if (charSet[s.charAt(left)] == 0) {
                 cnt--;
               }
               left++;
             }

             // 更新答案，此时 cnt == k
             ans = Math.max(ans, right - left);
           }
           return ans;
         }
}
```

4. 寻找两个正序数组的中位数

示例 1：
输入：nums1 = [1,3], nums2 = [2]
输出：2.00000
解释：合并数组 = [1,2,3] ，中位数 2
示例 2：
输入：nums1 = [1,2], nums2 = [3,4]
输出：2.50000
解释：合并数组 = [1,2,3,4] ，中位数 (2 + 3) / 2 = 2.5

```
class Solution {
        public double findMe dia nSo rte dAr ray -
s(int[] nums1, int[] nums2) {
                int len1 = nums1.l ength;
                int len2 = nums2.l ength;
                // 减少操作次数
                if (len1 > len2) {
                        // 确保N1是短的部分
                         return findMe dia nSo -
rte dAr ray s(n ums2, nums1);
```

4. 寻找两个正序数组的中位数 (cont)

```
>       }
        // 开始从nums1中取元素
        int left = 0;
        int right = len1;
        while (left <= right) {
            // 从nums1和nums2中分别取m1和m2个元素组成k个元素
            int m1 = left + (right - left) / 2;
            int m2 = (len1 + len2) / 2 - m1;
            // k -> max(L1, L2), (k + 1) -> min(R1, R2)
            // nums1没有取元素?
            // 此时左边的元素一定从nums2中取，因此要使得L1 =
MIN_VALUE
            // 这样max(L1, L2)就会取到L2了
            // 这时所取得元素中num2的元素都要比nums1中的小
            // 因此在min(R1,R2)的时候，选中的也是R2
            double m1Left = (m1 == 0) ? Integer.MIN_VALUE :
nums1[m1 - 1];
            // nums1取了所有的元素?
            double m1Right = (m1 == len1) ? Integer.MAX_VALUE :
nums1[m1];
            // nums2没有取元素?
            double m2Left = (m2 == 0) ? Integer.MIN_VALUE :
nums2[m2 - 1];
            // nums2取了所有的元素?
            double m2Right = (m2 == len2) ? Integer.MAX_VALUE :
nums2[m2];
            // 左边取多了
            if (m1Left > m2Right) {
                right = m1 - 1;
            }else if (m2Left > m1Right) {
                // 左边取少了
                left = m1 + 1;
            } else {
```

4. 寻找两个正序数组的中位数 (cont)

```
>               // 此时是符合要求的排列情况
                if ((len1 + len2) % 2 == 0){
                    double medLeft = Math.max(m1Left, m2Left);
                    double medRight = Math.min(m1Right, m2Right);
                    return (medLeft + medRight) / 2.0;
                } else {
                    return Math.min(m1Right, m2Right);
                }
            }
        }
        return -1;
    }
}
```

662. 二叉树最大宽度

给你一棵二叉 树的根节点 `root` ，返回树的 最大宽度 。
树的 最大宽度 是所有层中最大的 宽度 。
每一层的 宽度 被定义为该层 最左和 最右的 非空节 点（即 ，两个 端点） 之间的 长度。 将这个 二叉树 视作与 满二叉 树结构 相同， 两端点 间会出 现一些 延伸到这一层的 `null` 节点，这些 `null` 节点也计入长度。
题目数据保证 答案将会在 `32` 位 带符号整数范围内。

示例 `1`：
输入：`root = [1,3,2 ,5, 3,n ull,9]`
输出：`4`
解释：最大宽 度出现在树的第 `3` 层，宽度为 `4 (5,3,n ull,9)` 。
示例 `2`：
输入：`root = [1,3,2 ,5, nul l,n ull ,9, 6,n ull,7]`
输出：`7`
解释：最大宽 度出现在树的第 `4` 层，宽度为 `7 (6,nul l,n - ull ,nu ll, nul l,n ull,7)` 。

662. 二叉树最大宽度 (cont)

> 示例 3：
输入：root = [1,3,2,5]
输出：2
解释：最大宽度出现在树的第 2 层，宽度为 2 (3,2) 。
此题求二叉树所有层的最大宽度，比较直观的方法是求出每一层的宽度，然后
求出最大值。求每一层的宽度时，因为两端点间的nul1节点也需要计入宽
度，因此可以对节点进行编号。一个编号为index的左子节点的编号记为
2xindex,右子节点的编号记为2×index+1,计算每层宽度时，用每层节点的最大编号减去最小编号再加1即为宽度。
遍历节点时，可以用广度优先搜索来遍历每一层的节点，并求出最大值。

```
class Solution {
    public int widthOfBinaryTree(TreeNode root) {
        int res = 1;
        List<Pair<TreeNode, Integer>> arr = new ArrayList<Pair<Tre-
eNode, Integer>>();
        arr.add(new Pair<TreeNode, Integer>(root, 1));
        while (!arr.isEmpty()) {
            List<Pair<TreeNode, Integer>> tmp = new ArrayList<Pair<-
TreeNode, Integer>>();
            for (Pair<TreeNode, Integer> pair : arr) {
                TreeNode node = pair.getKey();
                int index = pair.getValue();
                if (node.left != null) {
                    tmp.add(new Pair<TreeNode, Integer>(node.left, index *
2));
                }
                if (node.right != null) {
                    tmp.add(new Pair<TreeNode, Integer>(node.right, index
* 2 + 1));
                }
            }
```

662. 二叉树最大宽度 (cont)

```
>           res = Math.max(res, arr.get(arr.size() - 1).getValue() -
arr.get(0).getValue() + 1);
            arr = tmp;
        }
        return res;
    }
}
```

236. 二叉树的最近公共祖先

给定一个二叉树， 找到该树中两个指定节点的最近公共祖先。
百度百科中最 近公共 祖先的 定义为 ：“对于有根树 T 的两个节点 p、q，最近 公共祖 先表示 为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能 大（一 个节点 也可以 是它自 己的祖先）。”

示例 1：
输入：root = [3,5,1 ,6, 2,0 ,8, nul l,n ull ,7,4], p = 5, q = 1
输出：3
解释：节点 5 和节点 1 的最近公共祖先是节点 3 。
示例 2：
输入：root = [3,5,1 ,6, 2,0 ,8, nul l,n ull ,7,4], p = 5, q = 4
输出：5
解释：节点 5 和节点 4 的最近公共祖先是节点 5 。因为根据定 义最近 公共祖 先节点 可以为 节点本身。
示例 3：
输入：root = [1,2], p = 1, q = 2
输出：1

```
class Solution {
        public TreeNode lowest Com mon Anc est -
or( Tre eNode root, TreeNode p, TreeNode q) {
                // base case
                if (root == null) return null;
                if (root == p || root == q) return
root;
```

236. 二叉树的最近公共祖先 (cont)

```
>       TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        // 情况 1
        if (left != null && right != null) {
            return root;
        }
        // 情况 2
        if (left == null && right == null) {
            return null;
        }
        // 情况 3
        return left == null ? right : left;
    }
}
```

25. K 个一组翻转链表

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]`

```
class Solution {
        public ListNode revers eKG rou p(L istNode head, int k) {
```

25. K 个一组翻转链表 (cont)

```
>       if (head == null) return null;
        // 区间 [a, b) 包含 k 个待反转元素
        ListNode a, b;
        a = b = head;
        for (int i = 0; i < k; i++) {
            // 不足 k 个，不需要反转，base case
            if (b == null) return head;
            b = b.next;
        }
        // 反转前 k 个元素
        ListNode newHead = reverse(a, b);
        // 递归反转后续链表并连接起来
        a.next = reverseKGroup(b, k);
        return newHead;
    }
    /* 反转区间 [a, b) 的元素，注意是左闭右开 */
    ListNode reverse(ListNode a, ListNode b) {
        ListNode pre, cur, nxt;
        pre = null;
        cur = a;
        nxt = a;
        // while 终止的条件改一下就行了
        while (cur != b) {
            nxt = cur.next;
            cur.next = pre;
            pre = cur;
            cur = nxt;
        }
```

25. K 个一组翻转链表 (cont)

```
>       // 返回反转后的头结点
        return pre;
    }
}
```

95. 不同的二叉搜索树 II

给你一个整数 `n` ，请你生成并返回所有由 `n` 个节点组成且节点值从 `1` 到 `n` 互不相同 二叉搜索树 。可以按 任意顺序 返回答案。

示例 `1`：
输入：`n = 3`
输出：`[[1 ,nu ll, 2,n ull ,3] ,[1 ,nu ll, 3,2 ],[ -2,1 ,3] ,[3 ,1, nul l,n ull ,2] ,[3 ,2, nul l,1]]`
示例 `2`：
输入：`n = 1`
输出：`[[1]]`

```
class Solution {
        public List<T ree Nod e> genera teT ree -s(int n) {
                if (n == 0) {
                        return new Linked Lis t<T -ree Nod e>();
                }
                 return genera teT rees(1, n);
        }
        public List<T ree Nod e> genera teT ree -s(int start, int end) {
                 Lis t<T ree Nod e> allTrees = new Linked Lis t<T ree Nod e>();
                if (start > end) {
                        all Tre es.a dd (null);
                        return allTrees;
                }
                // 枚举可行根节点
                for (int i = start; i <= end; i++) {
```

95. 不同的二叉搜索树 II (cont)

```
>           // 获得所有可行的左子树集合
            List<TreeNode> leftTrees = generateTrees(start, i - 1);
            // 获得所有可行的右子树集合
            List<TreeNode> rightTrees = generateTrees(i + 1, end);
            // 从左子树集合中选出一棵左子树，从右子树集合中选出一棵右子树，拼接到根节点上
            for (TreeNode left : leftTrees) {
                for (TreeNode right : rightTrees) {
                    TreeNode currTree = new TreeNode(i);
                    currTree.left = left;
                    currTree.right = right;
                    allTrees.add(currTree);
                }
            }
        }
        return allTrees;
    }
}
```

103. 二叉树的锯齿形层次

`103.` 二叉树的锯齿形层序遍历
给你二叉树的根节点 `root` ，返回其节点值的 锯齿形层序遍历 。（即先从左 往右， 再从右 往左进 行下一 层遍历 ，以此 类推， 层与层 之间交 替进行）。

示例 `1`：
输入：`root = [3,9,2 0,n ull ,nu ll, 15,7]`
输出：`[[3 ],[ 20, 9], [15,7]]`
示例 `2`：
输入：`root = [1]`
输出：`[[1]]`

103. 二叉树的锯齿形层次 (cont)

> 示例 3：
输入：root = []
输出：[]

```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        // 定义需要返回的变量
        List<List<Integer>> res = new ArrayList<>();
        // 创建双端队列
        Queue<TreeNode> queue = new ArrayDeque<>();
        // 创建层数索引
        int idx = 0;
        // 如果节点不为空
        if (root != null) {
            queue.add(root);
        }
        // 如果队列不为空
        while (!queue.isEmpty()) {
            // 计算这一层的根节点数
            int n = queue.size();
            idx++;
            // 新建List,用于存储每一层的节点数
            List<Integer> level = new ArrayList<>();
            for (int i = 0; i < n; i++) {
                TreeNode node = queue.poll();
                // 记录当前根节点
                level.add(node.val);
                // 将根节点删除，并依次加入所删除根节点的左节点与右节点。
                if (node.left != null) {
```

103. 二叉树的锯齿形层次 (cont)

>

```
                    queue.add(node.left);
                }
                if (node.right != null) {
                    queue.add(node.right);
                }
            }
            // 判断奇偶层
            if (idx%2 == 0){
                //反转level
                Collections.reverse(level);
            }
            res.add(level);
        }
        // 返回结果
        return res;
    }
}
```

700. 二叉搜索树中的搜索

给定二叉搜索树（BST）的根节点 root 和一个整数值 val。
你需要在 BST 中找到节点值等于 val 的节点。 返回以该节点 为根的子树。 如果节点不存 在，则返回 null 。
示例 1:
输入：root = [4,2,7 ,1,3], val = 2
输出：[2,1,3]
示例 2:
输入：root = [4,2,7 ,1,3], val = 5
输出：[]

```
class Solution {
```

700. 二叉搜索树中的搜索 (cont)

```
>    public TreeNode searchBST(TreeNode root, int target) {
       if (root == null) {
          return null;
       }
       // 去左子树搜索
       if (root.val > target) {
          return searchBST(root.left, target);
       }
       // 去右子树搜索
       if (root.val < target) {
          return searchBST(root.right, target);
       }
       return root;
    }
}
```

1038. 从二叉搜索树到更大和树

给定一个二叉搜索树 `root (BST)`，请将它的每个节点的值替换成树中大于或者等于该节点值的所有节点值之和。
提醒一下， 二叉搜索树 满足下列约束条件：
节点的左子树仅包含键 小于 节点键的节点。
节点的右子树仅包含键 大于 节点键的节点。
左右子树也必 须是二 叉搜索树。

输入：`[4, 1,6 ,0, 2,5 ,7, nul l,n ull ,nu ll, 3,n - ull ,nu ll, null,8]`
输出：`[30 ,36 ,21 ,36 ,35 ,26 ,15 ,nu ll, nul l,n - ull ,33 ,nu ll, nul l,n ull,8]`
示例 2：
输入：`root = [0,null,1]`
输出：`[1, null,1]`
`// 右根左遍历`

1038. 从二叉搜索树到更大和树 (cont)

```
> class Solution {
    int tmp = 0;
    public TreeNode bstToGst(TreeNode root) {
        traverse(root);
        return root;
    }
    public void traverse(TreeNode root) {
        if (root == null) return;
        traverse(root.right);
        tmp += root.val;
        root.val = tmp;
        traverse(root.left);
    }
}
```

20. 有效的括号

```
Stack<Character> stack = new Stack<>();
                for (char c : s.toCh arA rray()) {
                          switch (c) {
                                  case '(': stack.p -
us h(')'); break;
                                  case '[': stack.p -
us h(']'); break;
                                  case '{': stack.p -
us h('}'); break;
                                   def ault: if
(stack.is Empty() || c != stack.p op()) return
false;
                          }
                }
                 return stack.i sE mpty();
```

297. 二叉树的序列化与反序列化

输入：root = [1,2,3,null,null,4,5]
输出：[1, 2,3 ,nu ll, nul l,4,5]
示例 2：
输入：root = []
输出：[]
示例 3：
输入：root = [1]
输出：[1]
示例 4：
输入：root = [1,2]
输出：[1,2]

```
public class Codec {
        Lin ked Lis t<S tri ng> list = new Linked -
Lis t<>();
        public String serial ize (Tr eeNode root)
{
                return traver se( root);
       }
        public String traver se( Tre eNode root) {
               if (root == null) return " nul l";
                String left = traver se( roo t.l -
eft);
                String right = traver se( roo -
t.r ight);
                return root.val + " ," + left +
" ," + right;
       }
        public TreeNode deseri ali ze( String
data) {
                Str ing[] strs = data.s pli -
t(", ");
               for (String str : strs) {
                        lis t.a dd( str);
               }
                return traver se();
```

297. 二叉树的序列化与反序列化 (cont)

```
>   }
   public TreeNode traverse() {
      String str = list.removeFirst();
      if ("null".equals(str)) return null;
      TreeNode root = new TreeNode(Integer.valueOf(str));
      root.left = traverse();
      root.right = traverse();
      return root;
   }
}
```

114. 二叉树展开为链表

输入：root = [1,2,5,3,4,null,6]
输出：[1, nul l,2 ,nu ll, 3,n ull ,4, nul l,5 ,nu -
ll,6]
示例 2：
输入：root = []
输出：[]
示例 3：
输入：root = [0]
输出：[0]

```
class Solution {
       // 定义：将以 root 为根的树拉平为链表
        public void flatte n(T reeNode root) {
               // base case
               if (root == null) return;
               // 先递归拉平左右子树
                fla tte n(r oot.left);
                fla tte n(r oot.ri ght);
           *后序遍历位置*/
```

114. 二叉树展开为链表 (cont)

```
>       // 1、左右子树已经被拉平成一条链表
      TreeNode left = root.left;
      TreeNode right = root.right;
      // 2、将左子树作为右子树
      root.left = null;
      root.right = left;
      // 3、左子树的末端连接上当前的右子树
      TreeNode p = root;
      while (p.right != null) {
         p = p.right;
      }
      p.right = right;
   }
}
```

101. 对称二叉树

示例 1：
输入：root = [1,2,2 ,3, 4,4,3]
输出：true
示例 2：
输入：root = [1,2,2 ,nu ll, 3,n ull,3]
输出：false

```
class Solution {
        public boolean isSymm etr ic( Tre eNode
root) {
                return isSymm etr ic( roo t.left,
root.r ight);
       }
        public boolean isSymm etr ic( Tre eNode
left, TreeNode right) {
               if (left == null && right == null)
return true;
               if (left == null || right == null
|| left.val != right.val) return false;
```

101. 对称二叉树 (cont)

```
>       return isSymmetric(left.left, right.right) && isSymmetric(left.r-
ight, right.left);
   }
}
```

144. 二叉树的前序遍历

```
class Solution {
        Lis t<I nte ger> res = new Linked Lis t<>
();
        public List<I nte ger> preord erT rav -
ers al( Tre eNode root) {
                tra ver se( root);
                return res;
       }
        public void traver se( Tre eNode root) {
               if (root == null) {
                        return;
               }
                res.ad d(r oot.val);
                tra ver se( roo t.l eft);
                tra ver se( roo t.r ight);
       }
}
```

111. 二叉树的最小深度

```
class Solution {
        public int minDep th( Tre eNode root) {
               if (root == null) return 0;
                Lin ked Lis t<T ree Nod e> list =
new Linked Lis t<>();
                lis t.a dd( root);
               int depth = 0;
                while (!list.is Emp ty()) {
```

111. 二叉树的最小深度 (cont)

```
>           int size = list.size();
          depth++;
          for (int i = 0; i < size; i++) {
             TreeNode cur = list.removeFirst(); // eaily wrong
             if (cur.left == null && cur.right == null) {
                return depth;
             }
             if (cur.left != null) list.add(cur.left);
             if (cur.right != null) list.add(cur.right);
          }

       }
       return depth;
    }
}
```

105. 从前序与中序遍历序列构造二叉树

```
class Solution {
        Has hMa p<I nteger, Intege r> inOrde rVa -
lMapIdx = new HashMa p<>();
        public TreeNode buildT ree (int[]
preOrder, int[] inOrder) {
               for (int i = 0; i < preOrd er.l -
engths; i++) inOrde rVa lMa pId x.p ut( inO rde -
r[i], i);
                return buildT ree (pr eOrder,
inOrder, 0, preOrd er.l ength - 1, 0, inOrde r.l -
engths - 1);
       }
       // pre: 3 9 20 15 7
       // T L
       // in : 9 3 15 20 7
       // L T
        public TreeNode buildT ree (int[]
preOrder, int[] inOrder, int preLeft, int preRight,
int inLeft, int inRight) {
               if (preLeft > preRight) return null;
```

105. 从前序与中序遍历序列构造二叉树 (cont)

```
>       int rootVal = preOrder[preLeft];
       TreeNode root = new TreeNode(rootVal);
       int rootIdx = inOrderValMapIdx.get(rootVal);
       int inLeftSize = rootIdx - inLeft; // 1
       root.left = buildTree(preOrder, inOrder, preLeft + 1, preLeft +
inLeftSize, inLeft, inLeft + inLeftSize - 1);
       root.right = buildTree(preOrder, inOrder, preLeft + inLeftSize +
1, preRight, rootIdx + 1, inRight);
       return root;
   }
}
```

124. 二叉树中的最大路径和

```
class Solution {
       int res = Intege r.M IN_ VALUE;
        public int maxPat hSu m(T reeNode root) {
                get One Sid eMa x(r oot);
                return res;
       }
        public int getOne Sid eMa x(T reeNode
root) {
               if (root == null) {
                        return 0;
               }
               int left = Math.m ax(0, getOne -
Sid eMa x(r oot.le ft));
               int right = Math.m ax(0, getOne -
Sid eMa x(r oot.ri ght));
               res = Math.m ax(left + right +
root.val, res);
                return Math.m ax( left, right) +
root.val;
       }
}
```

124. 二叉树中的最大路径和 (cont)

> 543. 二叉树的直径

```
class Solution {
    int res = 0;
    public int diameterOfBinaryTree(TreeNode root) {
        traverse(root);
        return res;
    }
    public int traverse(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int left = traverse(root.left);
        int right = traverse(root.right);
        res = Math.max(left + right, res);
        return Math.max(left, right) + 1;
    }
}
```

2. 两数相加 - 力扣（Leetcode）

链表格式

```
class Solution {
        public ListNode addTwo Num ber s(L istNode
l1, ListNode l2) {
                Lis tNode first = l1, second = l2;
               int add = 0;
                Lis tNode fakeHead = new ListNo -
de(-1);
                Lis tNode res = fakeHead;
                while (first != null || second !=
null) {
                        int sum = (first == null ? 0
: first.val) + (second == null ? 0 : second.val) +
add;
                        add = sum / 10;
```

2. 两数相加 - 力扣（Leetcode）(cont)

```
>           res.next = new ListNode(sum % 10);
            res = res.next;
            if (first != null) first = first.next;
            if (second != null) second = second.next;
        }
        if (add != 0) {
            res.next = new ListNode(add % 10);
        }
        return fakeHead.next;
    }
}
```

138. 复制带随机指针的链表

示例 1：

输入：`head = [[7,nu ll] ,[1 3,0 ],[ 11, 4], -[10 ,2] ,[1,0]]`

输出：`[[7 ,nu ll] ,[1 3,0 ],[ 11, 4], [10 ,2] ,[1,0]]`

示例 2：

输入：`head = [[1,1] ,[2,1]]`

输出：`[[1 ,1] ,[2,1]]`

示例 3：

输入：`head = [[3,nu ll] ,[3 ,0] ,[3 ,null]]`

输出：`[[3 ,nu ll] ,[3 ,0] ,[3 ,null]]`

```
class Solution {
        Map <Node, Node> map = new HashMa p<Node,
Node>();
        public Node copyRa ndo mLi st(Node
oldNode) {
                if (oldNode == null) {
                        return null;
                }
                if (map.c ont ain sKe y(o ldN ode))
{
```

138. 复制带随机指针的链表 (cont)

```
>           return map.get(oldNode);
      } else {
         Node newNode = new Node(oldNode.val);

         newNode.next = copyRandomList(oldNode.next);
         newNode.random = copyRandomList(oldNode.random);
         map.put(oldNode, newNode);
         return newNode;
      }
   }

}
```

146. LRU缓存机制

```
public class LRUCache {
        Map <In teger, Intege r> map = new Linked -
Has hMa p<I nteger, Intege r>();
       int cap;

        public LRUCac he(int capacity) {
               cap = capacity;
       }

        public int get(int key) {
               if (!map.c on tai nsK ey( key))
                        return -1;

               int val = map.re mov e(key);
                map.pu t(key, val);
                return val;
       }

        public void set(int key, int value) {
               if (map.c ont ain sKe y(key)) {
                        map.re mov e(key);
                        map.pu t(key, value);
```

146. LRU缓存机制 (cont)

```
>           return;
      }

      map.put(key, value);

      if (map.size() > cap)
         map.remove(map.entrySet().iterator().next().getKey());
   }
}
```

93. 复原IP地址

示例 1：
输入：s = " 255 255 111 35"
输出：["25 5.2 55.1 1.1 35 " ,"25 5.2 55.1 - 11.35 "]
示例 2：
输入：s = " 000 0"
输出：["0.0.0.0"]
示例 3：
输入：s = " 101 023 "
输出：["1.0.1 0.2 3","1.0.1 02.3", " - 10.1.0.23 " ,"10.10.2.3 ","1 01.0.2.3"]

```
class Solution {
       // Temporary list to store parts of the IP
address
        Lis t<S tri ng> temp = new ArrayL ist <>();
       // Result list to store all valid IP
addresses
        Lis t<S tri ng> res = new ArrayL ist <>();

       // Public method to start the process of
restoring IP addresses
        public List<S tri ng> restor eIp Add res -
ses (String s) {
               // Start backtr acking from the
first character
                bac ktr ack(s, 0);
               // Return the result list
                return res;
```

93. 复原IP地址 (cont)

```
>     }
    // Private method to perform backtracking
    private void backtrack(String s, int curIdx) {
        // If we have 4 parts in the IP address and we haven't reached
the end of the string, return
        if (temp.size() == 4 && curIdx != s.length()) return;
        // If we have 4 parts in the IP address and we have reached the
end of the string, add to result
        if (temp.size() == 4 && curIdx == s.length()){
            // Join the parts of the IP address with a dot and add to the
result list
            res.add(String.join(".", temp)); //这个join函数用时2ms，手动-
StringBuilder的话就是1ms
            return;
        }
        // Loop through the next 3 characters of the string
        for(int i = curIdx; i < s.length() && i < curIdx + 3; i++){
            // Get the substring from the current index to the current index
+ 1
            String sub = s.substring(curIdx, i + 1);
            // If the integer value of the substring is less than or equal to
255
            if (Integer.parseInt(sub) <= 255){
                // If the substring has more than one character and the first
character is '0', return
                if(sub.length() > 1 && s.charAt(curIdx) == '0'){
                    return;
                }
                // Add the substring to the temporary list
                temp.add(sub);
                // Continue backtracking from the next character
                backtrack(s, i + 1);
                // Remove the last element from the temporary list
                temp.remove(temp.size() - 1);
            } else {
```

93. 复原IP地址 (cont)

```
>                 // If the integer value of the substring is greater than 255,
return
                return;
            }
        }
    }
}
```

513. 找树左下角的值

示例 1:

输入: root = [2,1,3]

输出: 1

示例 2:

输入: [1,2,3 ,4, nul l,5 ,6, nul l,n ull,7]

输出: 7

```
class Solution {
        public int findBo tto mLe ftV alu e(T -
reeNode root) {
                Que ue< Tre eNo de> q = new
ArrayD equ e<>();
                q.o ffe r(r oot);
               int ans = 0;
                while (!q.is Emp ty()) {
                        ans = q.peek ().val;
                        for (int i = q.size(); i >
0; --i) {
                                Tre eNode node =
q.poll();
                                if (node.left !=
null) {
                                        q.o ffe -
r(n ode.left);
                                }
                                if (node.r ight !=
null) {
                                        q.o ffe -
r(n ode.ri ght);
                                }
```

513. 找树左下角的值 (cont)

```
>         }
      }
      return ans;
    }
}
```

129. 求根节点到叶节点数字之和

给你一个二叉树的根节点 `root` ，树中每个节点都存放有一个 `0` 到 `9` 之间的数字。
每条从根节点 到叶节 点的路 径都代 表一个数字：
例如，从根节 点到叶 节点的路径 `1 -> 2 -> 3` 表示数字 `123` 。
计算从根节点 到叶节点生成的 所有数字之和 。
叶节点 是指没有子节 点的节点。

示例 `1`：
输入：`root = [1,2,3]`
输出：`25`
解释：
从根到叶子节点路径 `1->2` 代表数字 `12`
从根到叶子节点路径 `1->3` 代表数字 `13`
因此，数字总和 `= 12 + 13 = 25`
示例 `2`：
输入：`root = [4,9,0 ,5,1]`
输出：`1026`
解释：
从根到叶子节点路径 `4-> 9->5` 代表数字 `495`
从根到叶子节点路径 `4-> 9->1` 代表数字 `491`
从根到叶子节点路径 `4->0` 代表数字 `40`
因此，数字总和 `= 495 + 491 + 40 = 1026`

```
class Solution {
```

129. 求根节点到叶节点数字之和 (cont)

```
>    public int sumNumbers(TreeNode root) {
       return cal(root, 0);
    }
    public int cal(TreeNode root, int cur) {
       if (root == null) return 0;
       cur = cur * 10 + root.val;
       if (root.left == null && root.right == null) return cur;
       return cal(root.left, cur) + cal(root.right, cur);
    }
}
```

958. 二叉树的完全性检验

给你一棵二叉树的根节点 `root` ，请你判断这棵树是否是一棵 完全二叉树 。
在一棵 完全二叉树 中，除了最后 一层外 ，所有 层都被 完全填 满，并 且最后 一层中 的所有 节点都 尽可能 靠左。 最后一层（第 `h` 层）中可以包含 `1` 到 `2h` 个节点。

示例 `1`：
输入：`root = [1,2,3 ,4,5,6]`
输出：`true`
解释：最后一 层前的 每一层 都是满 的（即 ，节点值为 `{1}` 和 `{2,3}` 的两层），且 最后一 层中的 所有节 点（`{ -4,5 ,6}` ）尽可能靠左。
示例 `2`：
输入：`root = [1,2,3 ,4, 5,n ull,7]`
输出：`false`
解释：值为 `7` 的节点不满足 条件「 节点尽 可能靠左」。

```
// Java层序 遍历， 判断s ize 和in dex 是否相等即可
class Solution {
        public boolean isComp let eTr ee( Tre -
eNode root) {
                if (root == null) return false;
```

958. 二叉树的完全性检验 (cont)

```
>       LinkedList<TreeNode> queue = new LinkedList<>();
      LinkedList<Integer> indexQueue = new LinkedList<>();
      queue.offer(root);
      indexQueue.offer(0);
      int maxIndex = 0;
      int size = -1;
      while (!queue.isEmpty()) {
         TreeNode node = queue.poll();
         int index = indexQueue.poll();
         maxIndex = Math.max(index, maxIndex);
         size++;
         if (node.left != null) {
            queue.offer(node.left);
            indexQueue.offer(index * 2 + 1);
         }
         if (node.right != null) {
            queue.offer(node.right);
            indexQueue.offer(index * 2 + 2);
         }
      }
      return size == maxIndex;
   }
}
```

199. 二叉树的右视图

输入：[1,2,3,null,5,null,4]
输出：[1,3,4]
示例 2：
输入：[1,null,3]
输出：[1,3]
示例 3：
输入：[]
输出：[]

```
// BFS
class Solution {
        public List<I nte ger> rightS ide Vie -
w(T reeNode root) {
                Lis t<I nte ger> res = new ArrayL -
ist <>();
               if (root == null) {
                        return res;
               }
                Que ue< Tre eNo de> queue = new
Linked Lis t<>();
                que ue.o ff er( root);
                while (!queu e.i sEm pty()) {
                       int size = queue.s ize();
                       for (int i = 0; i < size;
i++) {
                                Tre eNode node =
queue.p oll();
                               if (node.left !=
null) {
                                        que ue.o -
ff er( nod e.l eft);
                               }
                               if (node.r ight !=
null) {
                                        que ue.o -
ff er( nod e.r ight);
                               }
                               if (i == size - 1)
{ //将当前层 的最后 一个节 点放入结果列表
```

199. 二叉树的右视图 (cont)

```
>                 res.add(node.val);
            }
          }
        }
        return res;
    }
}
// DFS
我们按照 「根结点 -> 右子树 -> 左子树」 的顺序访问 ， 就可以保证每层都是最先访问最右边的节点的。
（ 与先序遍历 「根结点 -> 左子树 -> 右子树」 正好相反 ，先序遍历每层最先访问的是最左边的节点 ）
class Solution {
    List<Integer> res = new ArrayList<>();
    public List<Integer> rightSideView(TreeNode root) {
        dfs(root, 0); // 从根节点开始访问 ，根节点深度是0
        return res;
    }
    private void dfs(TreeNode root, int depth) {
        if (root == null) {
            return;
        }
        // 先访问 当前节点 ，再递归地访问 右子树 和 左子树。
        if (depth == res.size()) { // 如果当前节点所在深度还没有出现在res里 ，说明在该深度下当前节点是第一个被访问的节点 ，因此将当前节点加入res中。
            res.add(root.val);
        }
        depth++;
```

199. 二叉树的右视图 (cont)

```
>         dfs(root.right, depth);
        dfs(root.left, depth);
    }
}
```

206. 反转链表

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：[]

```
class Solution {
        public ListNode revers eLi st( Lis tNode head) {
                Lis tNode prev = null, next;
                Lis tNode curr = head;
                while (curr != null) {
                        next = curr.next;
                         cur r.next = prev;
                        prev = curr;
                        curr = next;
                }
                return prev;
        }
}
```

222. 完全二叉树的节点个数

给你一棵 完全二叉树 的根节点 root ，求出该树的节点个数。
完全二叉树 的定义如下： 在完全 二叉树 中，除 了最底 层节点 可能没 填满外 ，其余 每层节 点数都 达到最 大值， 并且 最 下面一 层的节 点都集 中在该 层最左 边的若 干位置 。若最底层为第 h 层，则该层包含 1~ 2h 个节点。

示例 1：
输入：root = [1,2,3 ,4,5,6]
输出：6
示例 2：
输入：root = []
输出：0
示例 3：
输入：root = [1]
输出：1

```
class Solution {
        public int countN ode s(T reeNode root) {
               if (root == null) return 0;
                Tre eNode left = root, right =
root;
               int left_cnt = 1, right_cnt = 1;
                while (left.left != null) {
                       left = left.left;
                        lef t_c nt++;
               }
                while (right.right != null) {
                        right = right.r ight;
                        rig ht_ cnt++;
               }
               if (left_cnt == right_cnt) {
                        return (int)M ath.pow(2,
left_cnt) - 1;
>       }
      return 1 + countNodes(root.left) + countNodes(root.right);
  }
}
```

235. 二叉搜索树的最近公共祖先

给定一个二叉搜索树， 找到该树中两个指定节点的最近公共祖先。
百度百科中最 近公共 祖先的 定义为 ：“对于有根树 T 的两个结点 p、q，最近 公共祖 先表示 为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能 大（一 个节点 也可以 是它自 己的祖先）。”
例如，给定如 下二叉搜索树：root = [6,2,8 ,0, 4,7 ,9, -nul l,n ull ,3,5]
示例 1:
输入：root = [6,2,8 ,0, 4,7 ,9, nul l,n ull ,3,5], p = 2, q = 8
输出：6
解释：节点 2 和节点 8 的最近公共祖先是 6。
示例 2:
输入：root = [6,2,8 ,0, 4,7 ,9, nul l,n ull ,3,5], p = 2, q = 4
输出：2
解释：节点 2 和节点 4 的最近公共祖先是 2，因为根据定义 最近公 共祖先 节点可 以为节点本身。

```
class Solution {
        public TreeNode lowest Com mon Anc est -
or( Tre eNode root, TreeNode p, TreeNode q) {
               if (p.val > q.val) {
                        return traver se( root, q,
p);
               }
                return traver se( root, p, q);
       }
        public TreeNode traver se( Tre eNode root,
TreeNode p, TreeNode q) {
               if (root == null) return null;
```

235. 二叉搜索树的最近公共祖先 (cont)

```
>       if (root == p || root == q) return root;
      if (root.val < p.val) {
         return traverse(root.right, p, q);
      } else if (root.val > q.val) {
         return traverse(root.left, p, q);
      }
      return root;
   }
}
```

341. 扁平化嵌套列表迭代器

示例 1：

输入：nes tedList = [[1,1] ,2, [1,1]]

输出：[1, 1,2 ,1,1]

解释：通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的 顺序应该是：[1,1,2 ,1,1]。

示例 2：

输入：nes tedList = [1,[4, [6]]]

输出：[1,4,6]

解释：通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的 顺序应该是：[1,4,6]。

```
public class Nested Ite rator implements Iterat -
or< Int ege r> {
        Lin ked Lis t<N est edI nte ger> list;
        public Nested Ite rat or( Lis t<N est edI -
nte ger> nested List) {
               list = new Linked Lis t<> (ne ste -
dList);
       }
        @Ov erride
        public Integer next() {
                return list.r emo veF irs t().ge -
tIn teg er();
```

341. 扁平化嵌套列表迭代器 (cont)

```
>   }
   @Override
   public boolean hasNext() {
      while (!list.isEmpty() && !list.getFirst().isInteger()) {
         List<NestedInteger> cur_list = list.removeFirst().getList();
         for (int i = cur_list.size() - 1; i >= 0; i--) {
            list.addFirst(cur_list.get(i));
         }
      }
      return !list.isEmpty();
   }
}
```

96. 不同的二叉搜索树

给你一个整数 n ，求恰由 n 个节点组成且节点值从 1 到 n 互不相同的 二叉搜索树 有多少种？返回满足题意的二叉搜索树的种数。

示例 1：

输入：n = 3

输出：5

示例 2：

输入：n = 1

输出：1

```
class Solution {
        public int numTre es(int n) {
                return sum(1,n);
       }

        private int sum(int left,int right){
                if( rig ht< =left) return 1;
               int sum = 0;
```

96. 不同的二叉搜索树 (cont)

```
>       for(int i=left;i<=right;i++) sum += sum(left,i-1) * sum(i+1,right);
        return sum;
    }
}
```

98. 验证二叉搜索树

给你一个二叉树的根节点 `root` ，判断其是否是一个有效的二叉搜索树。
有效 二叉搜索树定义如下：
节点的左子树只包含 小于 当前节点的数。
节点的右子树只包含 大于 当前节点的数。
所有左子树和 右子树 自身必 须也是 二叉搜索树。

示例 1：
输入：`root = [2,1,3]`
输出：`true`
示例 2：
输入：`root = [5,1,4 ,nu ll, nul l,3,6]`
输出：`false`
解释：根节点的值是 `5` ，但是右子节点的值是 `4` 。

```
class Solution {
        Tre eNode pre = null;
        public boolean isVali dBS T(T reeNode
root) {
                if (root == null) {
                        return true;
                }
                if (! isVali dBS T(r oot.left)) {
                        return false;
                }
                if (pre != null && root.val <=
pre.val) {
                        return false;
```

98. 验证二叉搜索树 (cont)

```
>       }
        pre = root;
        return isValidBST(root.right);
    }
}
```

652. 寻找重复的子树

输入：`root = [1,2,3,4,null,2,4,null,null,4]`
输出：`[[2 ,4] ,[4]]`
示例 2：
输入：`root = [2,1,1]`
输出：`[[1]]`
示例 3：
输入：`root = [2,2,2 ,3, nul l,3 ,null]`
输出：`[[2 ,3] ,[3]]`

```
class Solution {
        Has hMa p<S tring, Intege r> map = new
HashMa p<>();
        Lis t<T ree Nod e> res = new Linked Lis t<>
();
        public List<T ree Nod e> findDu pli cat -
eSu btr ees (Tr eeNode root) {
                tra ver se( root);
                return res;
        }
        public String traver se( Tre eNode root) {
                if (root == null) return " nul l";
                String left = traver se( roo t.l -
eft);
                String right = traver se( roo -
t.r ight);
                String str = root.val + " ," +
left + " ," + right;
                if (map.c ont ain sKe y(str) &&
map.ge t(str) == 1) {
                        res.ad d(r oot);
```

652. 寻找重复的子树 (cont)

```
>       }
        map.put(str, map.getOrDefault(str, 0) + 1);
        return str;
    }
}
```

654. 最大二叉树

给定一个不重复的整数数组 `nums` 。 最大二叉树 可以用下面的算法从 `nums` 递归地构建：

创建一个根节 点，其值为 `nums` 中的最大值。

递归地在最大值 左边 的 子数组前缀上 构建左子树。

递归地在最大值 右边 的 子数组后缀上 构建右子树。

返回 `nums` 构建的 最大二叉树 。

输入：`nums = [3,2,1 ,6,0,5]`

输出：`[6, 3,5 ,nu ll, 2,0 ,nu ll, null,1]`

解释：递归调 用如下所示：

- `[3,2,1 ,6,0,5]` 中的最大值是 `6` ，左边部分是 `[3,2,1]` ，右边部分是 `[0,5]` 。
    - `[3,2,1]` 中的最大值是 `3` ，左边部分是 `[]` ，右边部分是 `[2,1]` 。
        - 空数组，无子节点。
        - `[2,1]` 中的最大值是 `2` ，左边部分是 `[]` ，右边部分是 `[1]` 。
            - 空数组，无子节点。
            - 只有一个元素 ，所以 子节点 是一个值为 `1` 的节点。
    - `[0,5]` 中的最大值是 `5` ，左边部分是 `[0]` ，右边部分是 `[]` 。
        - 只有一个元素 ，所以 子节点 是一个值为 `0` 的节点。
        - 空数组，无子节点。

```
class Solution {
        public TreeNode constr uct Max imu mBi -
nar yTr ee( int[] nums) {
                return maxTre e(nums, 0, nums.l -
ength - 1);
        }
```

654. 最大二叉树 (cont)

```
>   public TreeNode maxTree(int[] nums, int l, int r){
        if(l > r){
            return null;
        }
        //bond为当前数组中最大值的索引
        int idx = findMax(nums, l, r);
        TreeNode root = new TreeNode(nums[bond]);
        root.left = maxTree(nums, l, idx - 1);
        root.right = maxTree(nums, idx + 1, r);
        return root;
    }
    //找最大值的索引
    public int findMax(int[] nums, int l, int r){
        int max = Integer.MIN_VALUE, maxIndex = l;
        for(int i = l; i <= r; i++){
            if(max < nums[i]){
                max = nums[i];
                maxIndex = i;
            }
        }
        return maxIndex;
    }
}
```

116. 填充每个节点的下一个右指针

```
class Solution {
        public Node connec t(Node root) {
               if (root == null) return null;
                tra ver se( roo t.left, root.r -
ight);
                return root;
       }
        public void traver se(Node left, Node
right) {
               if (left == null || right == null)
return;
                lef t.next = right;
                tra ver se( lef t.left, left.r -
ight);
                tra ver se( rig ht.l eft, right.r -
ight);
                tra ver se( lef t.r ight, right.l -
eft);
       }
}
```

226. 翻转二叉树

```
class Solution {
        public TreeNode invert Tre e(T reeNode
root) {
               if (root == null) return null;
                Tre eNode left = invert Tre e(r -
oot.left);
                Tre eNode right = invert Tre e(r -
oot.ri ght);
                roo t.right = left;
                roo t.left = right;
                return root;
       }
}
```

515. 在每个树行中找最大值

示例1：

输入：root = [1,3,2 ,5, 3,n ull,9]

输出：[1,3,9]

示例2：

输入：root = [1,2,3]

输出：[1,3]

```
// dfs(root, 结果 , 层级)
// 如果root 是null返回
//
class Solution {
       // 二叉树层序遍 历寻找最大值
        public List<I nte ger> larges tVa lue -
s(T reeNode root) {
                Lis t<I nte ger> res = new Linked -
Lis t<>();
               if (null == root) {
                        return res;
               }
                Lin ked Lis t<T ree Nod e> queue =
new Linked Lis t<>();
                que ue.a dd (root);
                while (!queu e.i sEm pty()) {
                       int size = queue.s ize();
                       int max = queue.g et Fir -
st( ).val;
                       for (int i = 0; i < size;
i++) {
                            Tre eNode node =
queue.r em ove Fir st();
                           if (node.val > max)
{
                                   max =
node.val;
                           }
                           if (node.left !=
null) {
                                    que ue.a -
dd Las t(n ode.left);
```

515. 在每个树行中找最大值 (cont)

```
>           }
            if (node.right != null) {
                queue.addLast(node.right);
            }
        }
        res.add(max);
    }
    return res;
  }
}
```

230. 二叉搜索树中第K小的元素

```
class Solution {
       int k = 0;
       int res = 0;
        public int kthSma lle st( Tre eNode root,
int k) {
                this.k = k;
                tra ver se( root);
                return res;
       }
        public void traver se( Tre eNode root) {
               if (root == null) {
                        return;
               }
                tra ver se( roo t.l eft);
               k--;
               if (k == 0) {
                       res = root.val;
                        return;
```

230. 二叉搜索树中第K小的元素 (cont)

```
>     }
      traverse(root.right);
   }
}
```

78. 子集

示例 1：

输入：nums = [1,2,3]

输出：[[] ,[1],[2], [1, 2], [3] ,[1 ,3] ,
[2 ,3] ,[1 ,2,3]]

示例 2：

输入：nums = [0]

输出：[[] ,[0]]

```
class Solution {
        //定 义二维 数组r es用 于存储结果
        Lis t<L ist <In teg er>> res = new Linked -
Lis t<>();
        public List<L ist <In teg er>> subset -
s(int[] nums) {
                //定 义路径数组
                Lis t<I nte ger> track = new
Linked Lis t<>();
                bac ktr ack (nums, 0, track);
                return res;
       }
        public void backtr ack (int[] nums, int
start, List<I nte ger> track) {
                //添 加路径 数组到 结果数组中
                res.ad d(new Linked Lis t<> (tr -
ack));
                //f or循 环遍历 数组nums
               for (int i = start; i < nums.l -
ength; i++) {
                        //做 选择， 将选择 添加到 路径
数组中
                        tra ck.a dd (nu ms[i]);
                        //回 溯，继 续向后遍历
```

78. 子集 (cont)

```
>            backtrack(nums, i + 1, track);
         //撤销选择，将选择从路径中删除
         track.remove(track.size() - 1);
      }
   }
}
```

125. 验证回文串

示例 1：

输入：s = "A man, a plan, a canal: Panama "

输出：true

解释：" ama nap lan aca nal pan ama " 是回文串。

示例 2：

输入：s = "race a car"

输出：false

解释：" rac eac ar" 不是回文串。

示例 3：

输入：s = " "

输出：true

解释：在移除 非字母 数字字 符之后，s 是一个空字符串 " " 。

由于空字符串 正着反 着读都 一样， 所以是回文串。

```
class Solution {
        public boolean isPali ndr ome (String s) {
               int n = s.leng th();
               int left = 0, right = n - 1;
                while (left < right) {
                        while (left < right &&
!Chara cte r.i sLe tte rOr Dig it( s.c har At( -
left))) {
                                left++;
                       }
```

125. 验证回文串 (cont)

```
>            while (left < right && !Character.isLetterOrDigit(s.charAt(rig-
ht))) {
                right--;
            }
            if (left < right) {
                if (Character.toLowerCase(s.charAt(left)) != Character.to-
LowerCase(s.charAt(right))) {
                    return false;
                }
                left++;
                right--;
            }
        }
        return true;
    }
}
```