Ctrl+Shift+P（MacOS：cmd+shift+p）呼出命令面板，输入Markdown Preview Enhanced: Create Toc会生成一段类似，保存生成目录。

- 数组中的最长无重复子串的长度
- 判断链表是否为回文结构
- 岛屿的数量
- 在二叉树中找到两个节点的最近公共祖先
- 重复项数字的所有排列
- 最长回文子串的长度
- 最长公共子序列
- 最小编辑代价
- 矩阵的最小路径和
- 顺时针旋转数组
- 判断一棵树是否是搜索二叉树和完全二叉树
- 连续子数组的最大和（sum < 0置为0）
- 两数之和
- 删除有序链表中重复出现的元素
- 在转动过的有序数组中寻找目标值
- 数组中未出现的最小正整数
- 数组中最长连续子序列
- 判断二叉树是否对称
- 没有重复项数字的所有排列
- 集合的所有子集

翻转链表

```
// 非递归
public class Solution {
    public ListNode ReverseList(ListNode head) {
        ListNode pre = null;
        ListNode next = null;
        while(head != null){
            next = head.next;
            head.next = pre;
            pre = head;
            head = next;
        }
        return pre;
    }
}
```

```
// 递归
ListNode reverse(ListNode head) {
    if (head.next == null) return head;
    ListNode last = reverse(head.next);
    head.next.next = head;
    head.next = null;
    return last;
}
```

实现二叉树先序，中序和后序遍历

```
import java.util.*;

/*
 * public class TreeNode {
 *   int val = 0;
 *   TreeNode left = null;
 *   TreeNode right = null;
 * }
 */

public class Solution {
    /**
     *
     * @param root TreeNode类 the root of binary tree
     * @return int整型二维数组
     */
    public int[][] threeOrders (TreeNode root) {
        // write code here
        ArrayList<Integer> list1 = new ArrayList<>();
        ArrayList<Integer> list2 = new ArrayList<>();
        ArrayList<Integer> list3 = new ArrayList<>();
        front(root,list1,list2,list3);
        int[][] ints = new int[3][list1.size()];
        for (int i = 0; i < list1.size(); i++) {
            ints[0][i] = list1.get(i);
            ints[1][i] = list2.get(i);
            ints[2][i] = list3.get(i);
        }
        return ints;
    }

    public void front(TreeNode root,ArrayList<Integer> list1,
            ArrayList<Integer> list2,ArrayList<Integer> list3){
        if(root == null){
            return;
        }

        list1.add(root.val);
        front(root.left,list1,list2,list3);
        list2.add(root.val);
        front(root.right,list1,list2,list3);
        list3.add(root.val);
    }
}
```

- 非递归遍历
- 前序遍历

用栈来保存信息，但是遍历的时候，是：**先输出根节点信息，然后压入右节点信息，然后再压入左节点信息。**

```
public void pre(Node head){
    if(head == null){
        return;
    }
    Stack<Integer> stack = new Stack<>();
    stack.push(head);
    while(!stack.isEmpty()){
        head = stack.poll();
        System.out.println(head.value + " ");
        if(head.right != null){
            stack.push(head.right);
        }
        if(head.left != null){
            stack.push(head.left);
        }
    }
    System.out.println();
}
```

- 中序遍历

中序遍历的顺序是**左中右**，先一直左节点遍历，并压入栈中，当做节点为空时，输出当前节点，往右节点遍历。

```
public void inorder(Node head){
    if(head == null){
        return;
    }
    Stack<Integer> stack = new Stack<>();
    stack.push(head);
    while(!stack.isEmpty() || head != null){
        if(head != null){
            stack.push(head);
            head = head.left
        } else {
            head = stack.poll();
            System.out.println(head.value + " ");
            head = head.right;
        }
    }
    System.out.println();
}
```

- 后序遍历

用两个栈来实现，压入栈1的时候为**先左后右**，栈1弹出来就是**中右左**，栈2收集起来就是**左右中**。

设计LRU缓存结构

```java
import java.util.*;


public class Solution {
    /**
     * lru design
     * @param operators int整型二维数组 the ops
     * @param k int整型 the k
     * @return int整型一维数组
     */
    public int[] LRU (int[][] operators, int k) {
        // write code here
        ArrayList<Integer> list = new ArrayList<Integer>();
        LRUCache cache = new LRUCache(k);
        for(int[] op : operators){
            if(op[0]==1){
                cache.put(op[1],op[2]);
            }else{
                int val = cache.get(op[1]);
                list.add(val);
            }
        }
        int[] ans = new int[list.size()];
        for(int i=0;i<list.size();i++){
            ans[i] = list.get(i);
        }
        return ans;
    }
}

class Node {
    public int key;
    public int value;
    Node pre,next;

    public Node(int key , int value){
        this.key = key;
        this.value = value;
    }
}

class LRUCache{
    public HashMap<Integer,Node> map;
    public LinkedList<Node> list;
    public int capacity;

    public LRUCache(int capacity){
        this.capacity = capacity;
        map = new HashMap<>();
        list = new LinkedList<>();
    }
```

```
    public int get(int key){
        if(!map.containsKey(key)){
            return -1;
        }
        Node temp = map.get(key);
        put(key,temp.value);
        return temp.value;
    }

    public void put(int key, int value){
        Node node = new Node(key,value);
        if(map.containsKey(key)){
            Node temp = map.get(key);
            list.remove(temp);
            list.addFirst(node);
            map.put(key,node);
        } else {
            if(map.size() == capacity){
                Node last = list.removeLast();
                map.remove(last.key);
            }
            list.addFirst(node);
            map.put(key,node);
        }
    }

}
```

两个链表的第一个公共结点

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/
public class Solution {
    public ListNode FindFirstCommonNode(ListNode pHead1,
                    ListNode pHead2) {
        if(pHead1 == null || pHead2 == null){
            return null;
        }

        ListNode p1 = pHead1;
        ListNode p2 = pHead2;

        while(p1 != p2){
            p1 = p1.next;
            p2 = p2.next;
            if(p1 != p2){
                if(p1 == null) p1 = pHead2;
                if(p2 == null) p2 = pHead1;
            }
        }

        return p1;

    }
}
```

求平方根

```java
import java.util.*;


public class Solution {
    /**
     *
     * @param x int整型
     * @return int整型
     */
    public int sqrt (int x) {
        // write code here
        if(x < 2){
            return x;
        }
        int left = 1;
        int right = x / 2;
        while(left <= right){
            int mid = left + (right - left) / 2;
            if(x / mid == mid){
                return mid;
            } else if(x / mid < mid){
                right = mid - 1;
            } else if(x / mid > mid){
                left = mid + 1;
            }
        }

        return right;
    }
}
```

寻找第K大

```java
import java.util.*;

public class Finder {
    public int findKth(int[] a, int n, int K) {
        // write code here
        return find(a, 0, n-1, K);
    }

    public int find(int[] a, int low, int high, int K){
        int pivot = partition(a, low, high);

        if(pivot + 1 < K){
            return find(a, pivot + 1, high, K);
        } else if(pivot + 1 > K){
            return find(a, low, pivot - 1, K);
        } else {
            return a[pivot];
        }
    }

    int partition(int arr[], int startIndex, int endIndex){
        int small = startIndex - 1;
        for (int i = startIndex; i < endIndex; ++i) {
            if(arr[i] > arr[endIndex]) {
                swap(arr,++small, i);
            }
        }
        swap(arr,++small,endIndex);
        return small;
    }

    public void swap(int[] arr, int i, int j){
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

判断链表中是否有环

```java
/**
 * 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){
            return false;
        }
        ListNode p1 = head, p2 = head;
        while(p1.next != null && p1.next.next != null){
            p1 = p1.next.next;
            p2 = p2.next;
            if(p1 == p2){
                return true;
            }
        }
        return false;
    }
}
```

合并有序链表

```
import java.util.*;

/*
 * public class ListNode {
 *   int val;
 *   ListNode next = null;
 * }
 */

public class Solution {
    /**
     *
     * @param l1 ListNode类
     * @param l2 ListNode类
     * @return ListNode类
     */
    public ListNode mergeTwoLists (ListNode l1, ListNode l2) {
        ListNode node = new ListNode(0);
        ListNode res = node;
        while(l1 != null && l2 != null){
            if(l1.val > l2.val){
                node.next = l2;
                l2 = l2.next;
            } else {
                node.next = l1;
                l1 = l1.next;
            }
            node = node.next;
        }

        if(l1 != null){
            node.next = l1;
        }

        if(l2 != null){
            node.next = l2;
        }

        return res.next;
    }
}
```

合并k个已排序的链表

```java
import java.util.*;
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode mergeKLists(ArrayList<ListNode> lists) {
        if(lists == null || lists.size() == 0){
            return null;
        }

        return mergeList(lists,0,lists.size()-1);
    }

    public ListNode mergeList(ArrayList<ListNode> lists, int low, int high){
        if(low >= high){
            return lists.get(low);
        }

        int mid = low + (high - low)/2;
        ListNode left = mergeList(lists,low,mid);
        ListNode right = mergeList(lists,mid+1,high);
        return merge(left,right);
    }

    public ListNode merge(ListNode left, ListNode right){
        ListNode h = new ListNode(-1);
        ListNode tmp = h;
        while(left != null && right != null){
            if(left.val < right.val){
                tmp.next = left;
                left = left.next;
            } else {
                tmp.next = right;
                right = right.next;
            }
            tmp = tmp.next;
        }

        if(left != null){
            tmp.next = left;
        }

        if(right != null){
```

```
            tmp.next = right;
        }

        return h.next;
    }
}
```

数组中相加和为0的三元组

```
import java.util.*;

public class Solution {
    public ArrayList<ArrayList<Integer>> threeSum(int[] num) {
        ArrayList<ArrayList<Integer>> list = new ArrayList<>();
        Arrays.sort(num);
        int left,right,sum;
        for(int i = 0; i < num.length - 2; i++){
            if(i > 0 && num[i] == num[i-1]) continue;
            left = i + 1;
            right = num.length - 1;
            while(left < right){
                sum = num[i] + num[left] + num[right];
                if(sum == 0){
                    ArrayList<Integer> temp = new ArrayList<>();
                    temp.add(num[i]);
                    temp.add(num[left]);
                    temp.add(num[right]);
                    list.add(temp);
                    right--;
                    left++;
                    while(left < right && num[left] == num[left-1]){
                        left++;
                    }
                    while(left < right && num[right] == num[right+1]){
                        right--;
                    }
                } else if(sum < 0){
                    left++;
                } else {
                    right--;
                }
            }
        }
        return list;
    }
}
```

删除链表的倒数第n个节点

```
import java.util.*;

/*
 * public class ListNode {
 *   int val;
 *   ListNode next = null;
 * }
 */

public class Solution {
    /**
     *
     * @param head ListNode类
     * @param n int整型
     * @return ListNode类
     */
    public ListNode removeNthFromEnd (ListNode head, int n) {
        // write code here
        ListNode dummyNode = new ListNode(0);
        dummyNode.next = head;
        ListNode fast = dummyNode;
        ListNode slow = dummyNode;
        for(int i = 0; i <= n; i++){
            fast = fast.next;
        }

        while(fast != null){
            fast = fast.next;
            slow = slow.next;
        }

        slow.next = slow.next.next;

        return dummyNode.next;
    }
}
```

二分查找

```java
import java.util.*;


public class Solution {
    /**
     * 二分查找
     * @param n int整型 数组长度
     * @param v int整型 查找值
     * @param a int整型一维数组 有序数组
     * @return int整型
     */
    public int upper_bound_ (int n, int v, int[] a) {
        // write code here
        int left = 0, right = n;
        while(left < right){
            int mid = left + (right - left) / 2;
            if(a[mid] == v){
                right = mid;
            } else if(a[mid] > v){
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left+1;
    }
}
```

两个链表生成相加链表

```java
import java.util.*;

/*
 * public class ListNode {
 *   int val;
 *   ListNode next = null;
 * }
 */

public class Solution {
    /**
     *
     * @param head1 ListNode类
     * @param head2 ListNode类
     * @return ListNode类
     */
    public ListNode addInList (ListNode head1, ListNode head2) {
        // write code here
        if(head1==null) return head2;
        if(head2==null) return head1;
        ListNode l1=reverse(head1);
        ListNode l2=reverse(head2);
        ListNode result=new ListNode(0);
        int c=0;
        while(l1!=null||l2!=null||c!=0)
        {
            int v1=l1!=null?l1.val:0;
            int v2=l2!=null?l2.val:0;
            int val=v1+v2+c;
            c=val/10;
            ListNode cur=new ListNode(val%10);
            cur.next=result.next;
            result.next=cur;
            if(l1!=null)
                l1=l1.next;
            if(l2!=null)
                l2=l2.next;
        }
        return result.next;
    }

    public ListNode reverse(ListNode node)
    {
        if(node==null) return node;
        ListNode pre=null,next=null;
        while(node!=null)
        {
            next=node.next;
            node.next=pre;
            pre=node;
            node=next;
```

```
        }
        return pre;
    }
}
```

二叉树的之字形层序遍历

```
import java.util.*;

/*
 * public class TreeNode {
 *   int val = 0;
 *   TreeNode left = null;
 *   TreeNode right = null;
 * }
 */

public class Solution {
    /**
     *
     * @param root TreeNode类
     * @return int整型ArrayList<ArrayList<>>
     */
    public ArrayList<ArrayList<Integer>> zigzagLevelOrder (TreeNode root) {
        // write code here
        Queue<TreeNode> queue = new LinkedList<>();
        ArrayList<ArrayList<Integer>> list = new ArrayList<>();
        if(root != null) queue.add(root);
        while(!queue.isEmpty()){
            ArrayList<Integer> temp = new ArrayList<>();
            for(int i = queue.size(); i > 0; i--){
                TreeNode node = queue.poll();
                temp.add(node.val);
                if(node.left != null){
                    queue.add(node.left);
                }
                if(node.right != null){
                    queue.add(node.right);
                }
            }
            if(list.size() % 2 == 1){
                Collections.reverse(temp);
            }
            list.add(temp);
        }
        return list;
    }
}
```

链表内指定区间反转

```
public class Solution {
    /**
     *
     * @param head ListNode类
     * @param m int整型
     * @param n int整型
     * @return ListNode类
     */
    public ListNode reverseBetween (ListNode head, int m, int n) {
        // write code here
        if(head == null || n == m){
            return head;
        }

        ListNode dummy = new ListNode(0);
        dummy.next = head;

        ListNode cur = dummy;
        for(int i = 1; i < m; i++){
            cur = cur.next;
        }

        // t1代表头，t2代表尾
        ListNode t1 = cur;
        ListNode t2 = cur.next;
        cur = t2;

        ListNode pre = null;
        // 翻转链表
        for(int i = 0; i <= n - m; i++){
            ListNode temp = cur.next;
            cur.next = pre;
            pre = cur;
            cur = temp;
        }

        t1.next = pre;
        t2.next = cur;

        return dummy.next;
    }
}
```

二叉树的镜像

```java
public class Solution {
    public void Mirror(TreeNode root) {
        if(root == null){
            return;
        }
        if(root.left == null && root.right == null){
            return;
        }
        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);
        while(!stack.isEmpty()){
            TreeNode node = stack.pop();

            if(node.left != null || node.right != null){
                TreeNode temp = node.left;
                node.left = node.right;
                node.right = temp;
            }

            if(node.left != null){
                stack.push(node.left);
            }

            if(node.right != null){
                stack.push(node.right);
            }
        }
    }
}

/*
public class Solution {
    public void Mirror(TreeNode root) {
        if(root == null){
            return;
        }
        if(root.left == null && root.right == null){
            return;
        }

        TreeNode temp = root.left;
        root.left = root.right;
        root.right = temp;

        if(root.left != null){
            Mirror(root.left);
        }

        if(root.right != null){
            Mirror(root.right);
        }
```

```
    }
}
*/
```

数组中只出现一次的数字

```
public class Solution {

    public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) {
        int num = 0;
        for(int i = 0; i < array.length; i++){
            num^=array[i];
        }

        int count = 0;
        // 标志位，记录num中的第一个1出现的位置
        for(;count < array.length; count++){
            if((num&(1<<count)) != 0){
                break;
            }
        }
        num1[0] = 0;
        num2[0] = 0;
        for(int i = 0; i < array.length; i++){
            // 标志位为0的为一组，异或后必得到一个数字（这里注意==的优先级高于&，需在前面加（））
            if((array[i] & (1 << count)) == 0){
                num1[0] ^= array[i];
            } else {
                // 标志位为1的为一组
                num2[0] ^= array[i];
            }
        }
    }

    /*
    public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) {
        HashSet<Integer> set = new HashSet<>();
        for(int i = 0; i < array.length; i++){
            if(!set.add(array[i])){
                set.remove(array[i]);
            }
        }

        Object[] temp = set.toArray();
        num1[0] = (int)temp[0];
        num2[0] = (int)temp[1];
    }*/
}
```

最长的括号子串

```java
public class Solution {
    /**
     *
     * @param s string字符串
     * @return int整型
     */
    public int longestValidParentheses (String s) {
        // write code here
        if(s == null || s.length() <= 0){
            return 0;
        }

        Stack<Integer> stack = new Stack<>();
        int last = -1;
        int maxLen = 0;
        for(int i = 0; i < s.length(); i++){
            if(s.charAt(i) == '('){
                stack.push(i);
            } else {
                if(stack.isEmpty()){
                    last = i;
                } else {
                    stack.pop();
                    if(stack.isEmpty()){
                        maxLen = Math.max(maxLen, i - last);
                    } else {
                        maxLen = Math.max(maxLen, i - stack.peek());
                    }
                }
            }
        }

        return maxLen;
    }
}

// 动态规划
public int longestValidParentheses2(String s) {
    if (s == null || s.length() == 0)
        return 0;
    int[] dp = new int[s.length()];
    int ans = 0;
    for (int i = 1; i < s.length(); i++) {
        // 如果是'('直接跳过，默认为0
        if (s.charAt(i) == ')') {
            if (s.charAt(i - 1) == '(')
                dp[i] = (i >= 2 ? dp[i - 2] : 0) + 2;
            // 说明s.charAt(i - 1)==')'
            else if (i - dp[i - 1] > 0 &&
                s.charAt(i - dp[i - 1] - 1) == '(') {
                dp[i] = (i - dp[i - 1] > 1 ?
```

```
                    dp[i - dp[i - 1] - 2] : 0) + dp[i - 1] + 2;
                // 因为加了一个左括号和一个右括号，所以是加2
            }
        }
        ans = Math.max(ans, dp[i]);
    }
    return ans;
}
```

把二叉树打印成多行

```java
import java.util.*;


/*
public class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;

    public TreeNode(int val) {
        this.val = val;

    }

}
*/
public class Solution {
    ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {
        if(pRoot == null){
            return new ArrayList<ArrayList<Integer>>();
        }
        ArrayList<ArrayList<Integer>> list = new ArrayList<>();

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(pRoot);
        while(!queue.isEmpty()){
            ArrayList<Integer> temp = new ArrayList<>();
            for(int i = queue.size(); i > 0; i--){
                TreeNode node = queue.poll();
                temp.add(node.val);
                if(node.left != null){
                    queue.add(node.left);
                }
                if(node.right != null){
                    queue.add(node.right);
                }
            }
            list.add(temp);
        }

        return list;
    }

}
```

合并两个有序的数组

```
public class Solution {
    public void merge(int A[], int m, int B[], int n) {
        int i = m-1, j = n-1, k = m+n-1;
        while(i >= 0 && j >= 0){
            if(A[i] > B[j]){
                A[k--] = A[i--];
            } else {
                A[k--] = B[j--];
            }
        }

        while(j >= 0){
            A[k--] = B[j--];
        }
    }
}
```

二叉树的最大路径和

```
public class Solution {
    int max = Integer.MIN_VALUE;
    /**
     *
     * @param root TreeNode类
     * @return int整型
     */
    public int maxPathSum (TreeNode root) {
        // write code here
        maxSum(root);
        return max;
    }

    public int maxSum(TreeNode root){
        if(root == null){
            return 0;
        }

        //三种情况：1.包含一个子树和顶点，2.仅包含顶点，3.包含左子树和右子树以及顶点。
        int left = Math.max(maxSum(root.left),0);
        int right = Math.max(maxSum(root.right),0);

        max = Math.max(max,left+right+root.val);

        //对于每一个子树，返回包含该子树顶点的深度方向的路径和的最大值。
        return root.val + Math.max(left,right);
    }
}
```

买卖股票的最佳时机

base case：
dp[-1][k][0] = dp[i][0][0] = 0
dp[-1][k][1] = dp[i][0][1] = -infinity

状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

```
public class Solution {
    /**
     *
     * @param prices int整型一维数组
     * @return int整型
     */
    public int maxProfit (int[] prices) {
        if(prices.length == 0) return 0;
        // write code here
        int n = prices.length;
        int[][] dp = new int[n][2];
        for(int i = 0; i < n; i++){
            if(i - 1 == -1){
                dp[i][0] = 0;
                dp[i][1] = -prices[i];
                continue;
            }
            dp[i][0] = Math.max(dp[i-1][0],
                                dp[i-1][1] + prices[i]);
            dp[i][1] = Math.max(dp[i-1][1],-prices[i]);
        }

        return dp[n-1][0];
    }
}
```

二叉树中是否存在节点和为指定值的路径

```java
public class Solution {
    /**
     *
     * @param root TreeNode类
     * @param sum int整型
     * @return bool布尔型
     */
    public boolean hasPathSum (TreeNode root, int sum) {
        // write code here
        if(root == null){
            return false;
        }

        if(root.left == null && root.right == null){
            return sum - root.val == 0;
        }

        return hasPathSum(root.left,sum - root.val) ||
                hasPathSum(root.right,sum - root.val);
    }
}
```

设计getMin功能的栈

```
public class Solution {
    /**
     * return a array which include all ans for op3
     * @param op int整型二维数组 operator
     * @return int整型一维数组
     */
    public int[] getMinStack (int[][] op) {
        // write code here
        LinkedList<Integer> stack = new LinkedList<>();
        LinkedList<Integer> minStack = new LinkedList<>();
        ArrayList<Integer> ans = new ArrayList<>();

        for(int i = 0; i < op.length; i++){
            int type = op[i][0];
            if(type == 1){
                if(minStack.size() == 0){
                    minStack.push(op[i][1]);
                } else if(op[i][1] <= minStack.peek()){
                    minStack.push(op[i][1]);
                }
                stack.push(op[i][1]);
            } else if(type == 2) {
                 if(stack.peek().equals(minStack.peek())){
                     minStack.pop();
                 }
                 stack.pop();
            } else {
                ans.add(minStack.peek());
            }
        }

        int[] res = new int[ans.size()];
        for(int i = 0; i < ans.size(); i++){
            res[i] = ans.get(i);
        }
        return res;
    }
}
```

LFU缓存结构设计

```
// 解法一

public class Solution {
    /**
     * lfu design
     * @param operators int整型二维数组 ops
     * @param k int整型 the k
     * @return int整型一维数组
     */
    public int[] LFU (int[][] operators, int k) {
        // write code here
        if(operators == null) return new int[]{-1};
        HashMap<Integer,Integer> map = new HashMap<>();// key -> value
        HashMap<Integer,Integer> count = new HashMap<>(); // key -> count
        List<Integer> list = new ArrayList<>();
        for(int[] ops : operators){
            int type = ops[0];
            int key = ops[1];
            if(type == 1){
                // set操作
                if(map.containsKey(key)){
                    map.put(key,ops[2]);
                    count.put(key,count.get(key)+1);
                } else {
                    if(map.size() == k){
                        int minKey = getMinKey(count);
                        map.remove(minKey);
                        count.remove(minKey);
                    }
                    map.put(key,ops[2]);
                    if(count.containsKey(key)){
                        count.put(key,count.get(key)+1);
                    } else {
                        count.put(key,1);
                    }
                }
            } else if(type == 2) {
                if(map.containsKey(key)){
                    int value = map.get(key);
                    count.put(key,count.get(key)+1);
                    list.add(value);
                } else {
                    list.add(-1);
                }
            }
        }

        int[] ans = new int[list.size()];
        for(int i = 0; i < list.size(); i++){
            ans[i] = list.get(i);
        }
```

```
        return ans;
    }

    public int getMinKey(HashMap<Integer,Integer> map){
        int minCount = Integer.MAX_VALUE;
        int key = 0;
        for(Entry<Integer,Integer> entry : map.entrySet()){
            if(entry.getValue() < minCount){
                minCount = entry.getValue();
                key = entry.getKey();
            }
        }
        return key;
    }
}

// 解法二

import java.util.*;


public
class Solution {

    public class LFUCache {

        private class Node {
            int k;
            int v;
            int count; //调用次数

            public Node(int k, int v) {
                this.k = k;
                this.v = v;
                count = 1;
            }
        }

        private int size;
        private int maxSize;
        private Map<Integer, Node> key2node;
        private Map<Integer, LinkedList<Node>> count2list; //<调用次数，对于调用次数的链表>

        public LFUCache(int maxSize) {
            this.maxSize = maxSize;
            size = 0;
            key2node = new HashMap<>();
            count2list = new HashMap<>();
        }

        public void set(int k, int v) {
```

```
            if (key2node.containsKey(k)) { //存在
                key2node.get(k).v = v;
                get(k); //get 一次用于改变调用次数
                return;
            }
            //不存在：建一个新节点
            Node node = new Node(k, v);
            //如果调用次数map中不存在，就添加一个新链表
            if (!count2list.containsKey(node.count))
                count2list.put(node.count, new LinkedList<>());
            LinkedList<Node> list = count2list.get(node.count);
            list.addFirst(node);//插入到该链表的头部，表示该调用次数中，是最近调用的，链表尾才是最
            key2node.put(k, node);//同时加入核心map
            size++;
            if (size > maxSize) {//超过容量了，删除一个
                key2node.remove(list.getLast().k);
                list.removeLast();
                size--;
            }
        }

        public int get(int k) {
            if (!key2node.containsKey(k)) return -1;
            Node node = key2node.get(k);//获取之后
            //还需要更新调用次数：
            LinkedList<Node> oldList = count2list.get(node.count);
            oldList.remove(node);//原来的链表中删除
            if (oldList.isEmpty()) count2list.remove(node.count);
            node.count++;
            //建立并加入新链表
            if (!count2list.containsKey(node.count))
                count2list.put(node.count, new LinkedList<>());
            LinkedList<Node> list = count2list.get(node.count);
            list.addFirst(node);
            return node.v;
        }
    }

    public int[] LFU(int[][] operators, int k) {
        LFUCache cache = new LFUCache(k);
        List<Integer> list = new ArrayList<>();
        for (int[] e : operators) {
            if (e[0] == 1) cache.set(e[1], e[2]);
            else list.add(cache.get(e[1]));
        }
        int[] res = new int[list.size()];
        for (int i = 0; i < res.length; i++) res[i] = list.get(i);
        return res;
    }
}
```

N皇后问题

```java
public class Solution {
    /**
     *
     * @param n int整型 the n
     * @return int整型
     */
     public int Nqueen (int n) {
        // write code here
        List<Integer> res=new ArrayList<>();
        char[][] chess=new char[n][n];
        for(int i=0;i<n;i++){ //棋盘
            for(int j=0;j<n;j++){
                chess[i][j]='.';
            }
        }

        backtrack(0,chess,res);
        return res.size();
    }

    // 回溯
    public void backtrack(int row,char[][] chess,List<Integer> res){
        if(row==chess.length){
            res.add(1);
            return;
        }
        for(int col=0;col<chess.length;col++){
            if(!isValid(row,col,chess)) continue;

            chess[row][col]='Q';
            backtrack(row+1,chess,res);
            chess[row][col]='.';
        }
    }

    public boolean isValid(int row,int col,char[][] chess){
        for(int i=0;i<chess.length;i++){
            if(chess[i][col]=='Q'){
                return false;
            }
        }
        for(int i=row-1,j=col+1;i>=0&&j<chess.length;i--,j++){
            if(chess[i][j]=='Q'){
                return false;
            }
        }
        for(int i=row-1,j=col-1;j>=0&&i>=0;i--,j--){
            if(chess[i][j]=='Q'){
                return false;
            }
        }
```

```
        return true;
    }
}
```

带权值的最小路径和

```
public class Solution {
    /**
     *
     * @param grid int整型二维数组
     * @return int整型
     */
    public int minPathSum (int[][] grid) {
        // write code here
        int row = grid.length;
        int col = grid[0].length;
        int[][] dp = new int[row][col];
        dp[0][0] = grid[0][0];
        for(int i = 1; i < row; i++){
            dp[i][0] = dp[i-1][0] + grid[i][0];
        }
        for(int j = 1; j < col; j++){
            dp[0][j] = dp[0][j-1] + grid[0][j];
        }

        for(int i = 1; i < row; i++){
            for(int j = 1; j < col; j++){
                dp[i][j] = Math.min(dp[i][j-1],dp[i-1][j]) + grid[i][j];
            }
        }

        return dp[row-1][col-1];
    }
}
```

反转数字

```
public class Solution {
    /**
     *
     * @param x int整型
     * @return int整型
     */
    public int reverse (int x) {
        // write code here
        int res = 0;
        while(x != 0){
            // 获取最后一位
            int tail = x % 10;
            int newRes = res * 10 + tail;
            // 如果不等于，说明溢出
            if((newRes - tail) / 10 != res){
                return 0;
            }
            res = newRes;
            x /= 10;
        }

        return res;
    }
}
```

二叉搜索树的第k个结点

```
public class Solution {
    int index = 0;
    TreeNode target = null;
    TreeNode KthNode(TreeNode pRoot, int k){
        getKthNode(pRoot,k);
        return target;
    }

    public void getKthNode(TreeNode pRoot, int k){
        if(pRoot == null){
            return;
        }
        getKthNode(pRoot.left,k);
        index++;
        if(index == k){
            target = pRoot;
            return;
        }
        getKthNode(pRoot.right,k);
    }


}
```

子数组最大乘积

```java
public class Solution {
    public double maxProduct(double[] arr) {
        if(arr.length == 0 || arr == null){
            return 0.0;
        }
        double[] max = new double[arr.length];
        double[] min = new double[arr.length];
        max[0] = min[0] = arr[0];
        for(int i = 1; i < arr.length; i++){
            max[i] = Math.max(Math.max(max[i-1]*arr[i],
                            min[i-1]*arr[i]),arr[i]);
            min[i] = Math.min(Math.min(max[i-1]*arr[i],
                            min[i-1]*arr[i]),arr[i]);
        }

        double ans = max[0];
        for(int i = 0; i < max.length; i++){
            if(max[i] > ans){
                ans = max[i];
            }
        }
        return ans;
    }
}
```

最长递增子序列

```
public int[] LIS (int[] arr) {
    // write code here
    if(arr == null || arr.length <= 0){
        return null;
    }

    int len = arr.length;
    int[] count = new int[len];              // 存长度
    int[] end = new int[len];                // 存最长递增子序列

    //init
    int index = 0;                           // end 数组下标
    end[index] = arr[0];
    count[0] = 1;

    for(int i = 0; i < len; i++){
        if(end[index] < arr[i]){
            end[++index] = arr[i];
            count[i] = index;
        }
        else{
            int left = 0, right = index;
            while(left <= right){
                int mid = (left + right) >> 1;
                if(end[mid] >= arr[i]){
                    right = mid - 1;
                }
                else{
                    left = mid + 1;
                }
            }
            end[left] = arr[i];
            count[i] = left;
        }
    }

    //因为返回的数组要求是字典序，所以从后向前遍历
    int[] res = new int[index + 1];
    for(int i = len - 1; i >= 0; i--){
        if(count[i] == index){
            res[index--] = arr[i];
        }
    }
    return res;
}
```

在两个长度相等的排序数组中找到上中位数

```
public int findMedianinTwoSortedAray (int[] arr1, int[] arr2) {
    // write code here
    int n = arr1.length;
    if(n==0){
        return 0;
    }
    //arr1左右两端
    int l1=0,r1=n-1;
    //arr2左右两端
    int l2=0,r2=n-1;
    int mid1,mid2;
    //终止条件为l1=r1，即两个数组都只有一个元素，此时的上中位数为两数的最小值
    while(l1< r1){
        //arr1中位数
        mid1 = l1+((r1-l1)>>1);
        //arr2中位数
        mid2 = l2+((r2-l2)>>1);
        int k = r1-l1+1;
        if(arr1[mid1] == arr2[mid2]){ //若两数组中位数相等，整体中位数也是这个
            return arr1[mid1];
        }
        else if(arr1[mid1] > arr2[mid2]){
            if(k%2 == 0){//区间元素个数为偶数
                r1 = mid1; //整体中位数在arr1左区间,包括mid1
                l2 = mid2+1; //整体中位数在arr2右区间，不包括mid2
            }
            else if(k%2 == 1){ //区间元素个数为奇数
                r1 = mid1; //整体中位数在arr1左区间,包括mid1
                l2 = mid2; //整体中位数在arr2右区间，包括mid2
            }
        }
        else if (arr1[mid1] < arr2[mid2]){
            if(k%2 == 0){//区间元素个数为偶数
                r2 = mid2; //整体中位数在arr2左区间,包括mid2
                l1 = mid1+1; //整体中位数在arr1右区间，不包括mid1
            }
            else if(k%2 == 1){ //区间元素个数为奇数
                r2 = mid2; //整体中位数在arr2左区间,包括mid2
                l1 = mid1; //整体中位数在arr1右区间，包括mid1
            }
        }
    }
    //当区间内只有一个元素时，两个区间中最小值即为整体中位数
    return Math.min(arr1[l1],arr2[l2]);
}
```

判断t1树中是否有与t2树拓扑结构完全相同的子树

```
/**
     *
     * @param root1 TreeNode类
     * @param root2 TreeNode类
     * @return bool布尔型
     */
public boolean isContains (TreeNode root1, TreeNode root2) {
    // write code here
    if(root1 == null || root2 == null){
        return false;
    }
    return recur(root1, root2) || isContains(root1.left,root2)
                               || isContains(root1.right,root2);
}

public boolean recur(TreeNode root1, TreeNode root2){
    if(root2 == null){
        return true;
    }
    if(root1 == null || root1.val != root2.val){
        return false;
    }
    return recur(root1.left,root2.left) &&
                   recur(root1.right,root2.right);
}
```

反转字符串

```java
/**
* 反转字符串
* @param str string字符串
* @return string字符串
*/
public String solve (String str) {
    // write code here
    if(str == null){
        return null;
    }
    char[] c = new char[str.length()];
    int left = 0, right = str.length() - 1;

    while(left <= right){
        c[left] = str.charAt(right);
        c[right] = str.charAt(left);
        left++;
        right--;
    }
    return new String(c);
}
```

最大正方形

```java
/**
* 最大正方形
* @param matrix char字符型二维数组
* @return int整型
*/
public int solve (char[][] matrix) {
    // write code here
    int m = matrix.length;
    int n = matrix[0].length;
    int dp[][] = new int [m][n];
        for(int i = 0; i < m; i++){
            dp[i][0] = matrix[i][0] - '0';
        }
    for(int j = 0; j < n; j++){
        dp[0][j] = matrix[0][j] - '0';
    }
    int max = 0;
    for(int i = 1; i < m; i++){
        for(int j = 1; j < n; j++){
            if(matrix[i][j] == '1'){
                dp[i][j] = Math.min(Math.min(dp[i-1][j-1],
                                dp[i][j-1]),dp[i-1][j]) + 1;
                max = Math.max(max,dp[i][j]);
            }
        }
    }
    return max*max;
}
```

链表中的节点每K个一组翻转

```
//明显递归解决，翻转第一组之后，以第二组的开头为头节点，继续翻转，转翻到最后，返回。
public ListNode reverseKGroup(ListNode head, int k) {
    if(head==null||head.next==null)
        return head;
    ListNode h=new ListNode(0);
    h.next=head;
    ListNode next=null,tmp=head,cur=head;
    for(int i=1;i<k;i++){
        cur=cur.next;
        if(cur==null)
            return head;
    }
    next=cur.next;
    while(head.next!=next){
        tmp=head.next;
        head.next=tmp.next;
        tmp.next=h.next;
        h.next=tmp;
    }
    head.next=reverseKGroup(next,k);
    return h.next;
}
```

数组中的最长无重复子串的长度

```
public int maxLength (int[] arr) {
    int left = 0, right = 0;
    Set<Integer> set = new HashSet<>();
    int res = 1;
    while(right < arr.length){
        if(!set.contains(arr[right])){
            set.add(arr[right]);
            right++;
        }else{
            set.remove(arr[left]);
            left++;
        }
        res = Math.max(res, set.size());
    }
    return res;
}
```

判断链表是否为回文结构

```java
import java.util.*;

public class Solution {
    /**
     *
     * @param head ListNode类 the head
     * @return bool布尔型
     */
    public boolean isPail (ListNode head) {
        ListNode slow = head;
        ListNode fast = head;
        while(fast != null && fast.next != null){
            fast = fast.next.next;
            slow = slow.next;
        }

        Stack<Integer> stack = new Stack<>();
        while(slow != null){
            stack.add(slow.val);
            slow = slow.next;
        }

        while(!stack.isEmpty()){
            if(stack.pop() != head.val){
                return false;
            }

            head = head.next;
        }

        return true;
    }
}
```

岛屿的数量

```
class Solution {
    public int numIslands(char[][] grid) {
        int count = 0;
        for(int i = 0; i < grid.length; i++) {
            for(int j = 0; j < grid[0].length; j++) {
                if(grid[i][j] == '1'){
                    bfs(grid, i, j);
                    count++;
                }
            }
        }
        return count;
    }

    public void bfs(char[][] grid, int i, int j){
        Queue<int[]> list = new LinkedList<>();
        list.add(new int[] { i, j });
        while(!list.isEmpty()){
            int[] cur = list.remove();
            i = cur[0]; j = cur[1];
            if(inArea(i,j,grid) && grid[i][j] == '1') {
                grid[i][j] = '0';
                list.add(new int[] { i + 1, j });
                list.add(new int[] { i - 1, j });
                list.add(new int[] { i, j + 1 });
                list.add(new int[] { i, j - 1 });
            }
        }
    }

    public boolean inArea(int i, int j, char[][] grid){
        return i >=0 && j >= 0 && i < grid.length
                        && j < grid[0].length;
    }
}
```

在二叉树中找到两个节点的最近公共祖先

```
public class Solution {
    /**
     *
     * @param root TreeNode类
     * @param o1 int整型
     * @param o2 int整型
     * @return int整型
     */
    public int lowestCommonAncestor (TreeNode root, int o1, int o2) {
        if (root == null) {
            return 0;
        }
        if (root.val == o1 || root.val == o2) {
            return root.val;
        }
        int left = lowestCommonAncestor(root.left, o1, o2);
        int right = lowestCommonAncestor(root.right, o1, o2);
        if (left != 0 && right != 0) {
            return root.val;
        }
        if (left == 0) {
            return right;
        }
        if (right == 0) {
            return left;
        }
        return 0;
    }
}
```

重复项数字的所有排列

```java
public class Solution {
    private ArrayList<ArrayList<Integer>> res;
    private boolean[] visited;

    public ArrayList<ArrayList<Integer>> permute(int[] nums) {

        res = new ArrayList<>();
        visited = new boolean[nums.length];
        List<Integer> list = new ArrayList<>();
        backtrace(nums, list);

        return res;
    }

    private void backtrace(int[] nums, List<Integer> list) {

        if (list.size() == nums.length) {
            res.add(new ArrayList<>(list));
        }

        for (int i = 0; i < nums.length; i++) {

            if (visited[i]) continue;

            visited[i] = true;
            list.add(nums[i]);

            backtrace(nums, list);

            visited[i] = false;
            list.remove(list.size() - 1);

        }
    }
}
```

最长回文子串的长度

```java
public class Palindrome {
    public int getLongestPalindrome(String A, int n) {
        // write code here
        int max=0;
        for (int i=1;i<n;i++){
            int left = i-1;
            int right = i;
            while (left>=0&&right<n
                &&A.charAt(left)==A.charAt(right)){
                max=Math.max(max,right-left+1);
                left--;
                right++;
            }
            left = i-1;
            right=i+1;
            while (left>=0&&right<n
                &&A.charAt(left)==A.charAt(right)){
                max=Math.max(max,right-left+1);
                left--;
                right++;
            }
        }
        return max;
    }
}
```

最长公共子序列

```java
public class Solution {
    /**
     * longest common subsequence
     * @param s1 string字符串 the string
     * @param s2 string字符串 the string
     * @return string字符串
     */
    public String LCS (String s1, String s2) {
        // write code here
        int m = s1.length();
        int n = s2.length();
        int[][] dp = new int[m+1][n+1];

        StringBuilder res = new StringBuilder();
        for(int i = 1; i <= m;i++){
            for(int j = 1; j <= n;j++){
                if(s1.charAt(i-1) == s2.charAt(j-1)){
                    dp[i][j] = dp[i-1][j-1] + 1;
                    res.append(s1.charAt(i-1));
                }else{
                    dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
                }
            }
        }
        return res.toString();
    }
}
```

最小编辑代价

```java
import java.util.*;


public class Solution {
    /**
     * min edit cost
     * @param str1 string字符串 the string
     * @param str2 string字符串 the string
     * @param ic int整型 insert cost
     * @param dc int整型 delete cost
     * @param rc int整型 replace cost
     * @return int整型
     */
    public int minEditCost (String str1, String str2,
                            int ic, int dc, int rc) {
        // write code here
        int len1=str1.length();
        int len2=str2.length();
        char[] char1=str1.toCharArray();
        char[] char2=str2.toCharArray();
        int[][] dp=new int[len1+1][len2+1];
        for(int i=1;i<=len1;i++){
            dp[i][0]=dp[i-1][0]+dc;
        }
        for(int i=1;i<=len2;i++){
            dp[0][i]=dp[0][i-1]+ic;
        }
        for(int i=0;i<len1;i++){
            for(int j=0;j<len2;j++){
                if(char1[i]==char2[j])dp[i+1][j+1]=dp[i][j];
                else {
                    dp[i+1][j+1]=Math.min(Math.min(dp[i][j]+rc,
                        dp[i][j+1]+dc),dp[i+1][j]+ic);
                }
            }
        }
        return dp[len1][len2];
    }
}
```

矩阵的最小路径和

```java
import java.util.*;


public class Solution {
    /**
     *
     * @param matrix int整型二维数组 the matrix
     * @return int整型
     */
    public int minPathSum (int[][] matrix) {
        // write code here
        int m = matrix.length, n = matrix[0].length;
        if (m == 0 || n == 0) return 0;

        for (int i = 1; i < m; i++)
            matrix[i][0] += matrix[i-1][0];
        for (int i = 1; i < n; i++)
            matrix[0][i] += matrix[0][i-1];

        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                matrix[i][j] +=
                        Math.min(matrix[i-1][j], matrix[i][j-1]);
            }
        }
        return matrix[m-1][n-1];
    }
}
```

顺时针旋转数组

```java
// 找规律：mat[i][j]被旋转到了mat[j][n-i-1]的位置
public class Rotate {
    public int[][] rotateMatrix(int[][] mat, int n) {
        // write code here
        int[][] temp=new int[n][n];

        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
            {
                temp[j][n-1-i]=mat[i][j];
            }
        }
        return temp;
    }
}
```

判断一棵树是否是搜索二叉树和完全二叉树

```java
import java.util.*;

/*
 * public class TreeNode {
 *   int val = 0;
 *   TreeNode left = null;
 *   TreeNode right = null;
 * }
 */

public class Solution {
    /**
     *
     * @param root TreeNode类 the root
     * @return bool布尔型一维数组
     */
    public boolean[] judgeIt (TreeNode root) {
        boolean[] result = new boolean[2];
        result[0]=isBST(root,Integer.MIN_VALUE,Integer.MAX_VALUE);
        result[1]=isComplete(root);
        return result;
    }

    private boolean isBST(TreeNode root,int min,int max){
        if(root == null){
            return true;
        }
        if(root.val <= min || root.val >=max){
            return false;
        }
        return  isBST(root.left,min,root.val) && isBST(root.right,root.val,max);
    }


    private boolean isComplete(TreeNode root){
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        while(!queue.isEmpty()){
            TreeNode tmp = queue.poll();
            if(tmp == null){
                break;
            }
            queue.offer(tmp.left);
            queue.offer(tmp.right);
        }
        while(!queue.isEmpty()){
            if(queue.poll() != null){
                return false;
            }
        }
```

```
            return true;
        }
    }
```

连续子数组的最大和（sum < 0置为0）

```
// dp
public int FindGreatestSumOfSubArray(int[] array) {
    if(array.length == 0){
        return 0;
    }

    int max = Integer.MIN_VALUE;
    int[] dp = new int[array.length];
    for(int i = 0; i < array.length; i++){
        dp[i] = array[i];
    }
    for(int i = 1; i < array.length; i++){
        dp[i] = Math.max(dp[i-1] + array[i],dp[i]);
    }

    for(int i = 0; i < dp.length; i++){
        if(dp[i] > max){
            max = dp[i];
        }
    }
    return max;
}

// sum < 0置为0
public int FindGreatestSumOfSubArray(int[] array) {
    if(array.length == 0){
        return 0;
    }

    int max = Integer.MIN_VALUE;
    int cur = 0;
    for(int i = 0; i < array.length; i++){
        cur += array[i];
        max = Math.max(max,cur);
        cur = cur < 0 ? 0 : cur;
    }
    return max;
}
```

两数之和

```java
import java.util.HashMap;
public class Solution {
    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer,Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            if (map.containsKey(nums[i])){
                return new int[]{map.get(nums[i])+1,i+1};
            }
            map.put(target - nums[i],i);
        }
        return null;
    }
}
```

删除有序链表中重复出现的元素

```java
public ListNode deleteDuplicates (ListNode head) {
    ListNode dummy=new ListNode(0);
    dummy.next=head;
    ListNode pre=dummy;
    ListNode p=head;
    while(p!=null&&p.next!=null){
        if(p.val==p.next.val){
            while(p.next!=null&&p.val==p.next.val){
                p=p.next;
            }
            pre.next=p.next;
            p=p.next;
        }
        else{
            pre=p;
            p=p.next;
        }
    }
    return dummy.next;
}
```

在转动过的有序数组中寻找目标值

```java
public int search (int[] a, int target) {
    // write code here
    if(a==null||a.length<=0){
        return -1;
    }
    int low = 0;
    int high = a.length-1;
    while(low<=high){
        int mid = low+(high-low)/2;
        if(a[mid]==target){
            return mid;
        }else if(a[mid]<a[high]){
            if(a[mid]<target&&target<=a[high]){
                low = mid+1;
            }else{
                high = mid-1;
            }
        }else{
            if(a[low]<=target&&target<a[mid]){
                high = mid-1;
            }else{
                low = mid+1;
            }
        }
    }
    return -1;
}
```

数组中未出现的最小正整数

```
public int minNumberdisappered (int[] arr) {
    int n=arr.length;
    for(int i=0;i<n;i++){
        while(arr[i]>=1&&arr[i]<=n&&arr[arr[i]-1]!=arr[i]){
            swap(arr,arr[i]-1,i);
        }
    }
    for(int i=0;i<n;i++){
        if(arr[i]!=i+1){
            return i+1;
        }
    }
    return n+1;
}
private void swap(int[] arr,int i,int j){
    int temp=arr[i];
    arr[i]=arr[j];
    arr[j]=temp;
}
```

数组中最长连续子序列

```
public int MLS (int[] arr) {
    if(arr==null || arr.length==0)
        return 0;
    if(arr.length==1)
        return 1;
    Arrays.sort(arr);
    int max = -1;
    int count = 1;
    for(int i = 0;i<arr.length-1;i++){
        int c = arr[i+1]-arr[i];
        if(c==0)
            continue;
        if(c==1)
            count++;
        else{
            max = max<count?count:max;
            count = 1;
        }
    }
    max=max<count?count:max;
    return max;
}
```

判断二叉树是否对称

```java
public class Solution {
    /**
     *
     * @param root TreeNode类
     * @return bool布尔型
     */
    public boolean isSymmetric (TreeNode root) {
        // write code here
        return isSymmetricNode(root,root);
    }

    public boolean isSymmetricNode(TreeNode node1, TreeNode node2){
        if(node1 == null && node2 == null){
            return true;
        }
        if(node1 == null || node2 == null){
            return false;
        }
        if(node1.val != node2.val){
            return false;
        }
        return isSymmetricNode(node1.left,node2.right)
            && isSymmetricNode(node1.right,node2.left);
    }
}
```

没有重复项数字的所有排列

```
public class Demo1 {
        ArrayList<ArrayList<Integer>> res;

        public ArrayList<ArrayList<Integer>> permute(int[] nums) {
                res = new ArrayList<ArrayList<Integer>>();
                if (nums == null || nums.length < 1)
                        return res;
                //对数组元素进行从小到大排序
                Arrays.sort(nums);
                ArrayList<Integer> list = new ArrayList<Integer>();

                solve(list, nums);

                return res;
        }

        private void solve(ArrayList<Integer> list, int[] nums) {
                if (list.size() == nums.length) {
                        res.add(new ArrayList<Integer>(list));
                        return;
                }
                for (int i = 0; i < nums.length; i++) {
                        if (!list.contains(nums[i])) {
                                list.add(nums[i]);
                                solve(list, nums);
                                list.remove(list.size() - 1);
                        }
                }
        }
}
```

集合的所有子集

```java
import java.util.*;
public class Solution {
    public ArrayList<ArrayList<Integer>> subsets(int[] S) {
        ArrayList<ArrayList<Integer>> res=new ArrayList<>();
        Arrays.sort(S);
        LinkedList<Integer> list=new LinkedList<>();
        dfs(res,list,0,S);
        return res;
    }
    public void dfs(ArrayList<ArrayList<Integer>> res,
                    LinkedList<Integer> list, int k, int[] S){
        res.add(new ArrayList<>(list));
        for(int i=k;i<S.length;i++){
            list.add(S[i]);
            dfs(res,list,i+1,S);
            list.removeLast();
        }
    }
}
```