1. 在一个二维数组中（每个一维数组的长度相同），每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。 请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

思路：第0行是每行最小的数，最后一列是第0行最大的数，可以从这里开始比较，如果相等就结束，如果该数大于要查找的，剔除所在列，如果该数小于要查找的，剔除所在行。

|  |
| --- |
| public boolean Find(int target, int [][] array) {  boolean flag = false;  if(array == null){  return flag;  }  int rows = array.length;  int cols = array[0].length;  if(rows > 0 && cols > 0){  int row = 0;  int col = cols - 1;  while(row < rows && col >= 0){  if(array[row][col] == target){  flag = true;  break;  }else if(array[row][col] > target){  col--;  }else{  row++;  }  }  }  return flag;  } |

1. 替换空格

实现一个函数，把字符串中的每个空格替换成%20，

\* 例如we are happy-》we%20are%20happy

\* 思路：如果从前往后找，有几个空格，代表有些字符需要重复往后移动n次

\* 时间复杂度：遍历字符串，O(n)，遇到空格并替换，移动n个字符O(n)，最后得到

\* O(n^2)

\* 因此：换种思路，从后往前，两个指针p1，p2；先计算多少个空格，每个空格替换后

\* 长度加2，因此设原来数组长度为n，m个空格，替换后长度为n + 2 \* m；

\* 确定长度后，p1,p2都指向表尾，p1往前移动，遇到字符就将其移到p2，p2往前一位

\* 直到遇到空格，此时p1往前一位，由于空格替换后需要3位，p2往前三位，直到所有

\* 空格替换完毕，此时p1==p2

|  |
| --- |
| **public** **static** String replaceSpace(String str) {  //获取空格数  **int** count = 0;  **for**(**int** i = 0; i < str.length(); i++){  **char** space = str.charAt(i);  **if**(space == ' '){  count++;  }  }    //计算长度并进行替换  **int** length = str.length();  **int** newLength = length + count \* 2;  **char**[] c = **new** **char**[newLength];  System.*arraycopy*(str.toCharArray(), 0, c, 0, length);  **int** indexOfOriginal = length - 1;  **int** indexOfNew = newLength - 1;  **while**(indexOfOriginal >= 0 && indexOfOriginal != indexOfNew){  **if**(c[indexOfOriginal] == ' '){  c[indexOfNew--] = '%';  c[indexOfNew--] = '2';  c[indexOfNew--] = '0';  }**else**{  c[indexOfNew--] = c[indexOfOriginal];  }  indexOfOriginal--;  }  **return** String.*valueOf*(c);  } |

1. 从尾到头打印链表

|  |
| --- |
| /\*  \* 输入一个链表，按链表值从尾到头的顺序返回一个ArrayList  \* 从尾到头输出，第一个进去的，最后一个出来，可以想到栈数据结构  \* 既然想到用栈，那么应该想到递归，因为递归就是栈实现的  \*  \*/  **import** java.util.ArrayList;  **import** java.util.Stack;  **public** **class** Test3 {  **class** ListNode {  **int** val;  ListNode next = **null**;  ListNode(**int** val) {  **this**.val = val;  }  }  **public** ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  ArrayList<Integer> list1 = **new** ArrayList<Integer>();  ArrayList<Integer> list2 = **new** ArrayList<Integer>();  **while**(listNode != **null**){  list1.add(listNode.val);  listNode = listNode.next;  }  **for**(**int** i = list1.size() - 1; i >= 0; i--){  **int** value = list1.get(i);  list2.add(value);  }  **return** list2;  }  //栈  **public** ArrayList<Integer> printListFromTailToHead1(ListNode listNode) {  Stack <Integer> stack = **new** Stack <>();  ArrayList <Integer> list = **new** ArrayList <>();  **while** (listNode != **null**) {  stack.push( listNode.val ); //push方法进栈  listNode = listNode.next;  }  **while** (!stack.empty()) { //empty()方法判断非空  list.add( stack.pop() ); //pop（）方法取出栈顶元素并加入新链表中  }  **return** list;  }  //递归  **public** ArrayList<Integer> printListFromTailToHead2(ListNode listNode) {  ArrayList<Integer> list = **new** ArrayList<Integer>();  **if**(listNode != **null**){  list = printListFromTailToHead2(listNode.next);  list.add(listNode.val);  }  **return** list;  }    } |

1. 输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

思路：前序的第一个节点是根节点，因此需要在中序中找到根节点的位置，根节点的左边是左子树节点，根节点的右边是右子树节点，找到了左右子树，就可以构建左右子树了。

|  |
| --- |
| **import** java.util.Arrays;  /\*  \* 输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。  思路：前序的第一个节点是根节点，因此需要在中序中找到根节点的位置，根节点的左边是左子树节点，根节点的右边是右子树节点，找到了左右子树，就可以构建左右子树了。  \*/  **public** **class** Test4 {  **class** TreeNode {  **int** val;  TreeNode left;  TreeNode right;  TreeNode(**int** x) {  val = x;  }  }  **public** TreeNode reConstructBinaryTree(**int** [] pre,**int** [] in) {  **if**(pre.length == 0 || in.length == 0){  **return** **null**;  }  //获取根节点，判断前序数组是否只有一个节点，是就返回根节点root  TreeNode root = **new** TreeNode(pre[0]);  **int** len = pre.length;  root.left = root.right = **null**;  **if**(len == 1){  **return** root;  }    //中序遍历找到根节点的位置  **int** rootIn = 0;  **while**(root.val != in[rootIn]){  rootIn++;  }    //得到左子树的长度  **if**(rootIn > 0){  //构建左子树  **int**[] leftPre = **new** **int**[rootIn + 1];  **int**[] leftIn = **new** **int**[rootIn + 1];  //该方法左闭右开  leftPre = Arrays.*copyOfRange*(pre, 1, rootIn + 1);  leftIn = Arrays.*copyOfRange*(in, 0, rootIn + 1);    root.left = reConstructBinaryTree(leftPre, leftIn);  }    //判断rootIn是不是pre的最后一个元素，如果不是，构建右子树  **if**(rootIn < len - 1){  //构建右子树  **int**[] rightPre = **new** **int**[len - rootIn - 1];  **int**[] rightIn = **new** **int**[len - rootIn - 1];  rightPre = Arrays.*copyOfRange*(pre, rootIn + 1, len);  rightIn = Arrays.*copyOfRange*(in, rootIn + 1, len);    root.right = reConstructBinaryTree(rightPre, rightIn);  }  **return** root;  }  **public** **static** **void** main(String[] args) {  **int**[] a = **new** **int**[]{1, 2, 4, 7, 3, 5, 6, 8};  **int**[] b = **new** **int**[]{4, 7, 2, 1, 5, 3, 8, 6};  TreeNode root = **new** Test4().reConstructBinaryTree(a, b);  **new** Test4().AfterOrder(root);  }  **public** **void** AfterOrder(TreeNode root){  **if**(root != **null**){  AfterOrder(root.left);  AfterOrder(root.right);  System.***out***.println(root.val);  }  }  } |

5.二叉树的下一个节点

找到二叉树中序遍历的下一个节点，节点中除了左右孩子的指针还有父节点的指针；

思路：如果该节点有右孩子，下一个节点就是右孩子的最左结点；如果该节点是其父节点的左孩子，则下一个就是父节点；如果没有右孩子，又是父节点的右孩子，那么就沿着父节点往上找，直到找到其某个父节点是其父节点的左孩子。

|  |
| --- |
| /\*  \* 找到二叉树中序遍历的下一个节点，节点中除了左右孩子的指针还有父节点的指针  \* 思路：如果该节点有右孩子，下一个节点就是右孩子的最左结点；如果该节点是其父节点的左孩子，  \* 则下一个就是父节点；如果没有右孩子，又是父节点的右孩子，那么就沿着父节点往上找，  \* 直到找到其某个父节点是其父节点的左孩子。  \* 编码：  \* 如果有右孩子：  \* 否则，如果有父节点  \* 是父节点的右孩子：  \* 是父节点的左孩子：  \*/  **public** **class** Test8 {  **class** TreeLinkNode {  **int** val;  TreeLinkNode left = **null**;  TreeLinkNode right = **null**;  TreeLinkNode next = **null**;  TreeLinkNode(**int** val) {  **this**.val = val;  }  }  **public** TreeLinkNode getNext(TreeLinkNode pNode)  {  **if**(pNode == **null**){  **return** **null**;  }  TreeLinkNode pNext = **null**;  **if**(pNode.right != **null**){  TreeLinkNode pRight = pNode.right;  **while**(pRight.left != **null**){  pRight = pRight.left;  }  pNext = pRight;  }  **else** **if**(pNode.next != **null**){  //遇到这种需要找父节点的，都要定义两个，当前节点和父节点；  TreeLinkNode current = pNode;  TreeLinkNode parent = pNode.next;  **while**(parent != **null** && current == parent.right){  current = parent;  parent = parent.next;  }  //此处有点巧妙；如果是parent==null，说明没找到，parent为null  //如果current==parent.left，说明parent就是要找的，总之都是parent；  pNext = parent;  }  **return** pNext;  }  } |

6.两个栈实现一个队列

利用两个栈实现一个队列。完成队列的Push和Pop操作。 队列中的元素为int类型

思路：stack1用于插入，stack2用于弹出；stack2的顺序一直就是插入的顺序；

|  |
| --- |
| **import** java.util.Stack;  /\*  \* 利用两个栈实现一个队列。完成队列的Push和Pop操作。 队列中的元素为int类型  \* stack1用于插入，stack2用于弹出  \* stack2的顺序一直就是插入的顺序；  \*/  **public** **class** Test9 {  Stack<Integer> stack1 = **new** Stack<Integer>();  Stack<Integer> stack2 = **new** Stack<Integer>();  **public** **void** push(**int** node) {  stack1.push(node);  }  **public** **int** pop() {  **if**(stack2.isEmpty()){  **while**(!stack1.isEmpty()){  stack2.push(stack1.pop());  }  }    **return** stack2.pop();  }    **public** **static** **void** main(String[] args) {  Test9 test = **new** Test9();  **for**(**int** i = 0; i < 5; i++){  test.push(i);  }  **for**(**int** i = 0; i < 5; i++){  System.***out***.println(test.pop());  }  }  } |

7.斐波那契数列

|  |
| --- |
| /\*  \* 斐波那契数列  \*/  **public** **class** Test10 {  **int**[] f;  **public** Test10(**int** n){  f = **new** **int**[n];  }  //效率低下的递归  **public** **int** fib(**int** n){  **if**(n < 2){  **return** n;  }  **return** fib(n - 1) + fib(n - 2);  }  //效率较好，但是空间复杂读差的递归  **public** **int** fib2(**int** n){  **if**(n < 2){  **return** n;  }    **if**(f[n - 1] > 0){  **return** f[n - 1];  }    f[n - 1] = fib2(n -1) + fib2(n -2);  **return** f[n - 1];  }    //效率好，空间复杂也低的递归  **public** **int** fib3(**int** n){  **if**(n < 2){  **return** n;  }  **for**(**int** i = 3; i < n; i++){  f[n - 1] = f[n - 2] + f[n - 3];  }  **return** f[n - 1];  }    **public** **static** **void** main(String[] args) {  Test10 t = **new** Test10(40);  System.***out***.println(System.*currentTimeMillis*());  System.***out***.println(t.fib2(40));  System.***out***.println(System.*currentTimeMillis*());  System.***out***.println(t.fib(40));  System.***out***.println(System.*currentTimeMillis*());  System.***out***.println(t.fib3(40));  System.***out***.println(System.*currentTimeMillis*());  }  } |

8.快排的实现

|  |
| --- |
| **import** java.util.Arrays;  **public** **class** QuickSort {  **public** **static** **int** partition(**int**[]a, **int** start, **int** end){  //求出start到end的之间的随机数，包含end和start  **int** index = (**int**) (Math.*random*() \* start) % (end - start + 1) + start;  *swap*(a, index, end);  **int** small = start - 1;//指向最后一个比标准值小的位置；  **for**(index = start; index < end; index++){  **if**(a[index] < a[end]){  ++small;  **if**(small != index){  *swap*(a, index, small);  }  }  }  //此时由于small指向最后一个比标准值小的位置，因此+1，  //为了指向第一个比标准值大的值，然后与end交换，这样就能保证  //小的在一边，大的在一边  ++small;  *swap*(a, small, end);  **return** small;  }  **public** **static** **void** quickSort(**int**[] a, **int** start, **int** end){  **if**(start == end){  **return**;  }  **int** index = *partition*(a, start, end);  **if**(start < index){  *quickSort*(a, start, index - 1);  }  **if**(index < end){  *quickSort*(a, index + 1, end);  }  }    **private** **static** **void** swap(**int**[] a, **int** index, **int** end) {  // **TODO** Auto-generated method stub  **int** temp = a[index];  a[index] = a[end];  a[end] = temp;  }  **public** **static** **void** main(String[] args){  **int**[] a = **new** **int**[]{1,4,2,6,5};  *quickSort*(a, 0, a.length - 1);  System.***out***.println(Arrays.*toString*(a));  }  } |

9.求公司员工按照年龄顺序排序的序列

岁数是0-99，定义100个容量的数组，下标表示岁数，元素值表示该岁数有多少人。

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Arrays;  **import** java.util.List;  /\*  \* 求公司员工按照年龄顺序排序的序列  岁数是0-99，定义100个容量的数组，下标表示岁数，元素值表示该岁数有多少人。  \*/  **public** **class** SortAge {  **public** **static** **void** sortAge(List<Integer> list){  **if**(list == **null** || list.size() <= 0){  **return**;  }    **int** maxAge = 99;  List<Integer> ageList = **new** ArrayList<Integer>(maxAge + 1);  **for**(**int** i = 0 ; i < maxAge + 1; i++){  ageList.add(0);  }  **for**(**int** i = 0; i < list.size(); i++){  **int** age = list.get(i);  **int** value = ageList.get(age);  ageList.set(age, ++value);  }    **int** index = 0;  **for**(**int** i = 0; i <= maxAge ; i++){  **for**(**int** j = 0; j < ageList.get(i); j++){  list.set(index, i);  index++;  }  }  }  **public** **static** **void** main(String[] args){  List<Integer> list = **new** ArrayList<Integer>();  list.add(5);  list.add(5);  list.add(6);  list.add(5);  list.add(3);  list.add(5);  list.add(6);  list.add(7);  list.add(5);  list.add(8);  list.add(9);  *sortAge*(list);  System.***out***.println(Arrays.*toString*(list.toArray()));  }  } |

10.旋转数组中的最小数字

设定两个指针，第一个指向第一个位置，第二个指向最后一个位置，如果第一个指针小于第二个指针，那么直接已经有序，返回第一个指针的元素；否则，找到mid，如果mid大于或等于第一个指针的元素，说明最小值在mid和第二个指针之间，如果mid小于或等于第二个指针的元素，说明最小值在mid和第一个指针之间。如果前中后三者元素相同，那么只能顺序查找。

|  |
| --- |
| **import** java.util.Arrays;  /\*  \* 旋转数组中的最小数字  设定两个指针，第一个指向第一个位置，第二个指向最后一个位置，如果第一个指针小于第二个指针，  那么直接已经有序，返回第一个指针的元素；否则，找到mid，如果mid大于或等于第一个指针的元素，  说明最小值在mid和第二个指针之间，如果mid小于或等于第二个指针的元素，说明最小值在mid和第一个指针之间。  直到两个指针相邻，则第二个指针值的就是最小值，如果前中后三者元素相同，那么只能顺序查找。  \*/  **public** **class** Test11 {  **public** **static** **int** getMin(**int**[] nums){  **int** len = nums.length;  **if**( len<= 0){  **return** -1;  }  **int** low = 0;  **int** high = len - 1;  **int** mid = low;  **while**(nums[low] >= nums[high]){  **if**(high - low == 1){  mid = high;  **break**;  }  mid = (low + high) / 2;    **if**(nums[low] == nums[high] && nums[high] == nums[mid]){  **int** min = nums[low];  **for**(**int** i = low + 1; i <= high; i++){  **if**(min > nums[i]){  min = nums[i];  }  }  **return** min;  }    **if**(nums[mid] >= nums[low]){  low = mid;  }  **if**(nums[mid] <= nums[high]){  high = mid;  }  }  **return** nums[mid];  }  **public** **static** **void** main(String[] args){  **int**[] a = **new** **int**[]{1,1,0,1,1};  **int**[] b = **new** **int**[]{1,2,3,4,5};  **int**[] c = **new** **int**[]{3,4,5,1,2};  System.***out***.println(*getMin*(c));  }  } |

**11.矩阵中的路径**

矩阵中选一个格子作为起点，观察该格子是否是需要的字符，如果是在往左右下找下一个字符。

当矩阵定位了前n个字符之后，在与第n个字符对应的格子的周围都没有找到第n+1个字符，那么只好在原路径上回到n-1个字符，重新定位。由于路径不能重复，还需定义和矩阵一样的大小的布尔矩阵

|  |
| --- |
| **boolean** hasPath(**char**[][] matrix, **int** rows, **int** cols, **char**[] str){  **if**(matrix == **null** || rows < 1 || cols < 1 || str == **null**){  **return** **false**;  }  //用于表示是否被访问过  **boolean**[][] visited = **new** **boolean**[rows][cols];  **for**(**int** i = 0; i < rows; i++){  **for**(**int** j = 0; j < cols; j++){  visited[i][j] = **false**;  }  }    **int** pathLength = 0;//表示需要找的str串中的下标；  **for**(**int** row = 0; row < rows; row++){  **for**(**int** col = 0; col < cols; col++){  **if**(hasPathCore(matrix, rows, cols, row, col, str, pathLength, visited)){  **return** **true**;  }  }  }  **return** **false**;  }  **private** **boolean** hasPathCore(**char**[][] matrix, **int** rows, **int** cols, **int** row, **int** col, **char**[] str, **int** pathLength,  **boolean**[][] visited) {  **if**(str[pathLength] == '#'){  **return** **true**;  }    **boolean** hasPath = **true**;    **if**(row >= 0 && col >=0 && row < rows && col < cols && matrix[row][col] == str[pathLength] && !visited[row][col]){  ++pathLength;  visited[row][col] = **true**;  hasPath = hasPathCore(matrix, rows, cols, row - 1, col, str, pathLength, visited)  || hasPathCore(matrix, rows, cols, row, col - 1, str, pathLength, visited)  || hasPathCore(matrix, rows, cols, row + 1, col, str, pathLength, visited)  || hasPathCore(matrix, rows, cols, row, col + 1, str, pathLength, visited);  **if**(!hasPath){  --pathLength;  visited[row][col] = **true**;    }  }    **return** hasPath;  } |

**12.数独**

**13.机器人的运动范围**

题目：地上有一个m行n列的方格，一个机器人从坐标(0,0)的格子开始移动，它每次可以向左、右、上、下移动一格，但不能进入行坐标和列坐标的数位之和大于k的各自，例如k为18时，机器人能够进入的方格数(35,37)，因为3+5+3+7 = 18，但它不能进入方格（35,38），因为3+5+3+8=19，请问机器人能够到达多少个格子。

|  |
| --- |
| **import** java.util.Scanner;  /\*  \* 题目：地上有一个m行n列的方格，一个机器人从坐标(0,0)的格子开始移动，  \* 它每次可以向左、右、上、下移动一格，但不能进入行坐标和列坐标的数位之和大于k的各自，  \* 例如k为18时，机器人能够进入的方格数(35,37)，因为3+5+3+7 = 18，但它不能进入方格（35,38），  \* 因为3+5+3+8=19，请问机器人能够到达多少个格子。  \*  \* 思路：从0,0开始移动，当他进入i，j时，通过检查坐标数位和与k的关系来判断机器人是否能进入该格子；  \* 如果可以，在判断是否能够进入四个相邻的格子  \*/  **public** **class** RobotRange {  **public** **static** **int** movingCount(**int** threshold, **int** rows, **int** cols){  **if**(threshold < 0 || rows <= 0 || cols <= 0){  **return** 0;  }    **boolean**[][] visited = **new** **boolean**[rows][cols];    **int** count = *movingCountCore*(threshold, rows, cols, 0, 0, visited);    **return** count;    }  **private** **static** **int** movingCountCore(**int** threshold, **int** rows, **int** cols, **int** row, **int** col, **boolean**[][] visited) {  **int** count = 0;  **if**(*check*(threshold, rows, cols, row, col, visited)){  visited[row][col] = **true**;  //自身算一次，其他就是往左，往右，上，往下能走几格子  count = 1 + *movingCountCore*(threshold, rows, cols, row - 1, col, visited)+  *movingCountCore*(threshold, rows, cols, row, col - 1, visited) +  *movingCountCore*(threshold, rows, cols, row + 1, col, visited) +  *movingCountCore*(threshold, rows, cols, row, col + 1, visited);  }  **return** count;  }  **private** **static** **boolean** check(**int** threshold, **int** rows, **int** cols, **int** row, **int** col, **boolean**[][] visited) {  // **TODO** Auto-generated method stub  **if**(row >= 0 && col >= 0 && row < rows && col < cols  && *getDigitSum*(row) + *getDigitSum*(col) <= threshold  && !visited[row][col]){  **return** **true**;  }  **return** **false**;  }  **private** **static** **int** getDigitSum(**int** num) {  **int** sum = 0;  **while**(num > 0){  sum += num % 10;  num /= 10;  }  **return** sum;  }    **public** **static** **void** main(String[] args) {  Scanner sc = **new** Scanner(System.***in***);  **int** m = sc.nextInt();  **int** n = sc.nextInt();  **int** k = sc.nextInt();  **int**[][] matrix = **new** **int**[m][n];  **for**(**int** i = 0; i < m; i++){  **for**(**int** j = 0; j < n; j++){  matrix[i][j] = 0;  }  }  System.***out***.println(*movingCount*(k, m, n));  }  } |

14.剪绳子的问题

长度为n的生字，剪成m段，每段绳子长度记为k[0],k[1]…k[m]，求k[0] \* k[1] \* …\*k[m]最大。

|  |
| --- |
| **public** **class** Cutting {  // static int[] maxProduct;  **public** **static** **void** main(String[] args) {  // maxProduct = new int[5];  // for(int i = 0; i < maxProduct.length; i++){  // maxProduct[i] = -1;  // }  System.***out***.println(*max2*(5));  }  // private static int max(int len) {  // // **TODO** Auto-generated method stub  // if(len <= 3){  // return len;  // }else{  // int max = 0;  // for(int i = 1; i <= len / 2; i++){  // int res = max(i) \* max(len - i);  // if(max < res){  // max = res;  // }  // }  // return max;  // }  // }  //  // private static int max1(int len) {  // // **TODO** Auto-generated method stub  // if(len <= 3){  // return len;  // }  //  // if(maxProduct[len - 1] >= 0){  // return maxProduct[len - 1];  // }  // int max = 0;  // for(int i = 1; i <= len / 2; i++){  // int res = max(i) \* max(len - i);  // if(max < res){  // max = res;  // }  // }  // maxProduct[len - 1] = max;  // return maxProduct[len - 1];  //  // }    **private** **static** **int** max2(**int** len) {  // **TODO** Auto-generated method stub  **if**(len < 2){  **return** 0;  }  **if**(len == 2){  **return** 1;  }    **int**[] maxProduct = **new** **int**[len];  maxProduct[0] = 0;  maxProduct[1] = 1;  maxProduct[2] = 2;  maxProduct[3] = 3;    **for**(**int** i = 4; i <= len; i++){  **int** max = 0;  **for**(**int** j = 1; j <= i / 2; j++){  **int** res = maxProduct[j] \* maxProduct[i - j];  **if**(max < res){  max = res;  }  }  maxProduct[len - 1] = max;  }    **return** maxProduct[len - 1];    }  }  //贪心法:当n大于5是，尽可能多的剪成3，如果len - 3\*num3 = 1，说明最后剩下4；如果最后剩下的长度为4，就剪成2\*2；  **private** **static** **int** max3(**int** len){  **if**(len < 2){  **return** 0;  }  **if**(len == 2){  **return** 1;  }  **if**(len == 3){  **return** 2;  }  //表示能被剪成3的最大段数  **int** num3 = len / 3;    //当绳子最后剩下长度为4的时候，不能再去剪长度为3的了，应该剪2,2\*2>3\*1  **if**(len - num3 \* 3 == 1){  num3 --;  }    **int** num2 = (len - num3 \* 3) / 2;  **return** (**int**)(Math.*pow*(3, num3) \* Math.*pow*(2, num2));  } |