# 1. 链表

## 1.1 一个单向链表中是否有环

【a】当相隔1个节点时，q的下一个落脚点是p的前面一个位置，此时q的下一步正好踩上；【b】当相隔两个节点是，q直接踩上；

public static boolean hasLoopV1(SingleNode headNode) {

if(headNode == null) {

return false;

SingleNode p = headNode;

SingleNode q = headNode;

// 快指针未能遍历完所有节点

while (q != null && q.next != null) {

p = p.next; // 遍历一个节点

q = q.next.next; // 遍历两个个节点

// 已到链表末尾

if (p == q) {

// 快慢指针相遇，存在环

return true;

}

}

return false;

}

}

## 1.1 环路的长度

假设两个指针在b点相遇。则点b必在环中，以b点为出发点，两个指针再来一次“追逐”，慢指针走一圈，快指针走2圈后他们又在b点相遇。此时慢指针走过的路程即为环周长。

    public int circleLength(LinkList node) {注：参数node为两指针在2中相遇的位置

        LinkList one = node;

        LinkList two = node;

        int i = 0;

        while (two!=null && two.next!=null) {

            one = one.next;

            two = two.next.next;

            i++;

            if (one == two)

                break;

        }

        return i;

    }

## 1.1 找出循环链表第一个节点

第一步，找环中相汇点。分别用p1，p2指向链表头部，p1每次走一步，p2每次走二步，直到p1==p2找到在环中的相汇点。

 第二步，找环的入口。接上步，当p1==p2时，p2所经过节点数为2x,p1所经过节点数为x,设环中有n个节点,**p2比p1多走一圈**有2x=n+x; n=x;可以看出**p1实际走了一个环的步数**，再让p2指向链表头部，p1位置不变，p1,p2每次走一步直到p1==p2; 此时p1指向环的入口

public class Solution {

    ListNode EntryNodeOfLoop(ListNode pHead){

        if(pHead == null || pHead.next == null)

            return null;

        ListNode p1 = pHead;

        ListNode p2 = pHead;

        while(p2 != null && p2.next != null ){

            p1 = p1.next;

            p2 = p2.next.next;

            if(p1 == p2){

**p2 = pHead;**

                while(p1 != p2){

                    p1 = p1.next;

                    p2 = p2.next;

                }

                if(p1 == p2)

**return p1;**

            }

        }

        return null;

    }

}

## 1.2 排序的链表，链表中删除重复元素？(不保留)

if (pHead==null || pHead.next==null){return pHead;}

ListNode Head = new ListNode(0);

Head.next = pHead;

**ListNode pre  = Head;**

**ListNode last = Head.next;**

while (last!=null){

    if(last.next!=null && last.val == last.next.val){

**// 找到最后的一个相同节点**

**while (last.next!=null && last.val == last.next.val){**

**last = last.next;**

**}**

        pre.next = last.next;**//把相同的抛弃**

        last = last.next;

    }else{

        pre = pre.next;

        last = last.next;

    }

}

return Head.next;

## 1.3 如何找出单链表中的倒数第k个元素？

假设，共 n 个结点，**倒数第k个结点**即为第 n-k+1 个结点。

public ListNode FindKthToTail(ListNode head, int k) {

if(head == null) return null;

ListNode node1 = head;

ListNode node2 = head;

while(node1 != null && k > 0) {//node1移动k次，还有n-k次会指空

node1 = node1.next;

k --;

}

if(k > 0) return null;//**长度不满足**

while(node1 != null) {//移动n-k次，此时node2到n-k+1个结点，即倒数第k个结点

node1 = node1.next;

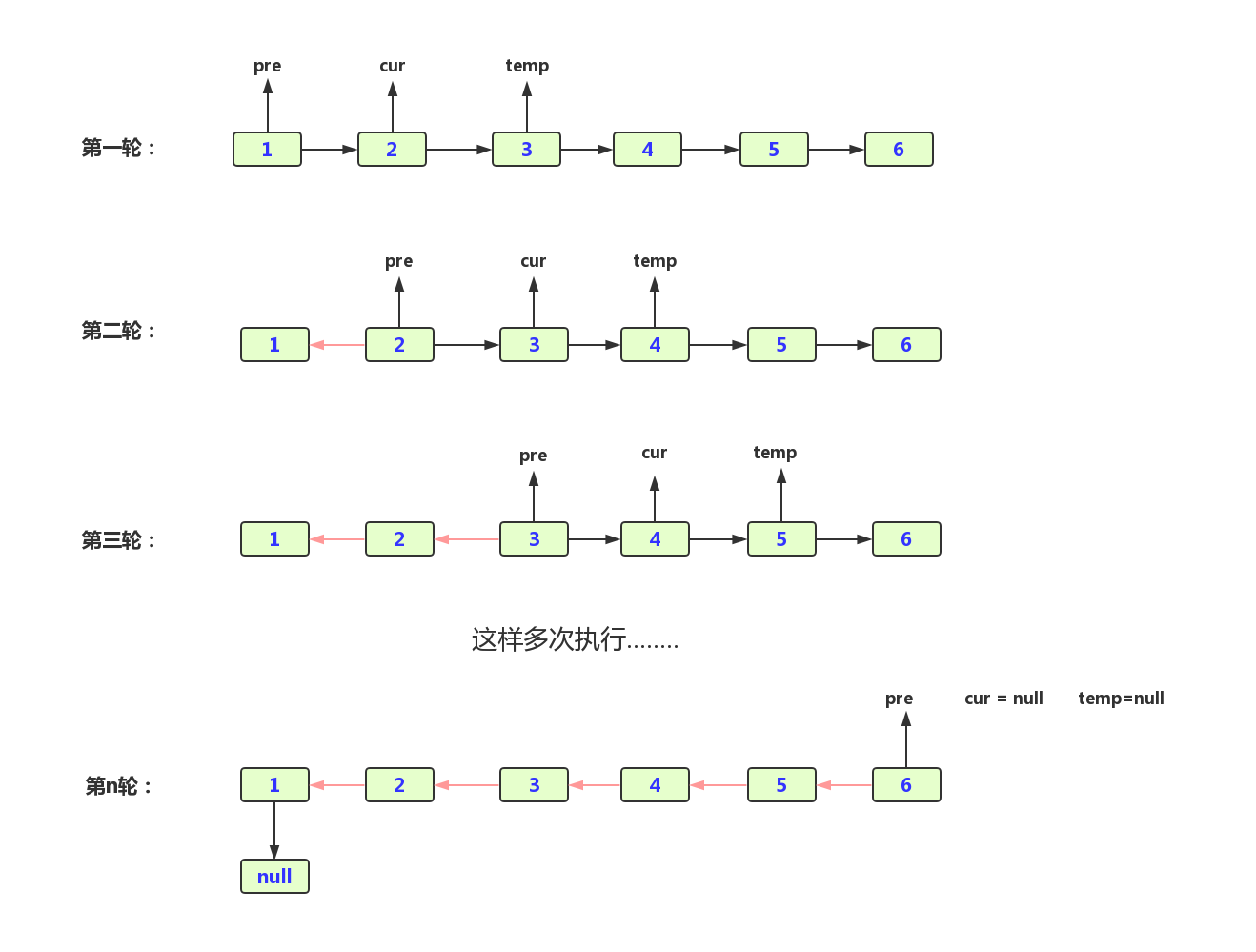
node2 = node2.next;

}

return node2;

}

## 1.4 反转链表？



public Node reverse(Node head) {

if (head == null || head.next == null) {

return head;

}

// 取前面节点

Node pre = head;

// 取后面节点

Node cur = head.next;

// 临时节点

Node temp = null;

while (cur != null) {

// 1. 保存后节点的指向节点 因为要替换后节点的指向节点为他的前节点

temp = cur.next;

// 2. 把后节点的指向的节点替换成前节点

**cur.next = pre;**

// 下一轮要替换的前节点和后节点

// 第一次 pre = 1 cur =2 || 那第二次 就得 pre = 2 cur = 3

pre = cur;

cur = temp;

}

// 上述过程中未替换首节点的指向节点　这里首节点将成为尾节点　所以指向nu**ll**

// 因为循环的条件是cur是否为null 如果cur为null **那 pre将是原来链表的尾节点**

// 就是逆转后的首节点

return cur;

}

[方法二:递归](https://www.kazhikazhi.com/NiceCui/article/232)

public Node recursionNode(Node head) {

if (head == null || head.next == null) {

return head;

}

// head 1 2 3 4

Node node = reverseNode(**head.next)**;

// 展示顺序 head 4 3 2 1

// 第一轮：

// 当前指向顺序 4 -> 5

**head.next.**next = head; // 变成了 5 -> 4 但是4的指针仍然指向5 也就是双向的

// 所以 4 -> null 变成单向

head.next = null;

// node是最后一个元素 5 也就是逆转后的 第一个元素

return node;

}

## 1.5 如何从尾到头输出单链表，返回一个ArrayList？

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  
        Stack<Integer> stack=new Stack<Integer>();  
        while(listNode!=null){  
            **stack.push(listNode.val);  
            listNode=listNode.next;**     
        }  
          
        ArrayList<Integer> list=new ArrayList<Integer>();  
        **while(!stack.isEmpty()){  
            list.add(stack.pop());**  
        }  
        return list;  
    }

Collections.reverse(list);//使用Collections的reverse方法，直接将list反转

}

## 1.6 两个单链表是否相交，第一个公共字节点

解决办法是**首先比较链表两个的长度**，**然后让较长的链表的先走他们长度的差值**，**然后同一步伐一起走，知道走到的结点相同时**，就找的了第一个公共结点。

public static ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {

int count1 = 0;

int count2 = 0;

ListNode commonNode = null;

ListNode pNode1 = pHead1;

ListNode pNode2 = pHead2;

//得到链表1的长度

while(pNode1!=null){

count1++;

pNode1 = pNode1.next;

}

//得到链表2的长度

System.out.println("List1的长度为："+count1);

while(pNode2!=null){

count2++;

pNode2 = pNode2.next;

}

System.out.println("List1的长度为："+count2);

//令pNode1和pNode2重新指向头结点

pNode1 = pHead1;

pNode2 = pHead2;

int sub = count1 - count2;

System.out.println("两个List相差"+sub+"个节点");

//先在长链表上走几步，再同时在两个链表上遍历

if(sub>0){

for(int i = 0;i < sub;i++){

pNode1 = pNode1.next;

}

}else{

for(int i = 0;i < Math.abs(sub);i++){

pNode2 = pNode2.next;

}

}

System.out.println("List1从"+pNode1.val+"开始比较，List2从"+pNode2.val+"开始比较");

//得到第一个公共节点

while(pNode1!=null && pNode2!=null){

if(**pNode1!=pNode2**){

pNode1 = pNode1.next;

pNode2 = pNode2.next;

}else{

commonNode = pNode1;

return commonNode;

}

}

return commonNode;

如果从后面遍历链表，因为结点都相同，所以第一个不相同的结点就是他们第一个公共结点。但是链表只能从头开始遍历，但却要从后面开始比较，可以利用**栈**结构。**使用两个栈，分别从头开始遍历两个链表，然后出栈，如果栈顶元素不相同，那么刚出栈的元素就是第一个公共结点。**

//使用两个栈存储空间

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if(headA==null || headB==null)

return null;

Stack<ListNode> list1 = new Stack<>();

Stack<ListNode> list2 = new Stack<>();

//保存出栈结点

ListNode temp=null;

//向两个栈中添加结点

while(headA!=null){

list1.add(headA);

headA=headA.next;

}

while(headB!=null){

list2.add(headB);

headB=headB.next;

}

//比较，出栈

while(!list1.isEmpty() && !list2.isEmpty() ){

if(list1.peek() == list2.peek()){

temp=list1.pop();

list2.pop();

}else

break;

}

return temp;

}

## 1.7 合并两个有序链表

public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

if (l1 == null) return l2;

if (l2 == null) return l1;

**ListNode head = null;**

if (l1.val <= l2.val){

**head = l1;**

head.next = mergeTwoLists(l1.next, l2);

} else {

head = l2;

head.next = mergeTwoLists(l1, l2.next);

}

**return head;**

# 2. 栈与队列

## 2.1 实现队列

class MyStack<T> {

private Deque<Integer> queue = new **LinkedList**<Integer>();

// push存入元素

public void push(Integer element) {

queue.addFirst(element);

}

// pop取出元素

public Integer pop() {

return queue.removeFirst();

}

// peek获取元素，但是并不取出元素

public Integer peek() {

return queue.getFirst();

}

public String toString() {

return queue.toString();

}

## 2.2 消息队列

<https://blog.csdn.net/heyutao007/article/details/50131089>

## 2.4 如何用两个栈模拟队列操作？

public class QueueByStack {

private Stack<Integer> stack1 = new Stack<>();

private Stack<Integer> stack2 = new Stack<>();

public void push(int node) {

stack1.push(node);

}

public int pop() {

if (stack2.isEmpty()) {

//栈1的元素出栈、入栈到栈2

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}else{

return stack2.pop();}

}

}

# 3.树

## 3.1 非递归前中后序遍历二叉树

/\*\*

\* 递归创建二叉树

\*/

public void buildTree(Node node,int data){

if(root == null){

root = new Node(data);

}else{

if(data < node.data){

if(node.left == null){

node.left = new Node(data);

}else{

buildTree(node.left,data);

}

}else{

if(node.right == null){

node.right = new Node(data);

}else{

buildTree(node.right,data);

}

}

}

/\*

\* 先序遍历二叉树（递归）

\*/

public void PrintBinaryTreePreRecur(TreeNode<T> root){

if (root!=null) {

System.out.print(root.data);

PrintBinaryTreePreRecur(root.left);

PrintBinaryTreePreRecur(root.right);

}

}

/\*

\* 中序遍历二叉树（递归）

\*/

public void PrintBinaryTreeMidRecur(TreeNode<T> root){

if (root!=null) {

PrintBinaryTreeMidRecur(root.left);

System.out.print(root.data);

PrintBinaryTreeMidRecur(root.right);

}

}

/\*

\* 后序遍历二叉树（递归）

\*/

public void PrintBinaryTreeBacRecur(TreeNode<T> root){

if (root!=null) {

PrintBinaryTreeBacRecur(root.left);

PrintBinaryTreeBacRecur(root.right);

System.out.print(root.data);

}

}

/\*

\* 先序遍历二叉树（非递归）

\* 思路：对于任意节点T，访问这个节点并压入栈中，然后访问节点的左子树，

\* 遍历完左子树后，取出栈顶的节点T，再先序遍历T的右子树

\*/

public void PrintBinaryTreePreUnrecur(TreeNode<T> root){

TreeNode<T> p=root;//p为当前节点

LinkedList<TreeNode> stack=new LinkedList<>();

//栈不为空时，或者p不为空时循环

while(p!=null || !stack.isEmpty()){

//当前节点不为空。访问并压入栈中。并将当前节点赋值为左儿子

if (p!=null) {

stack.push(p);

**System.out.print(p.data);**

p=p.left;

}

//当前节点为空：

// 1、当p指向的左儿子时，此时栈顶元素必然是它的父节点

// 2、当p指向的右儿子时，此时栈顶元素必然是它的爷爷节点

//取出栈顶元素，赋值为right

else{

p=stack.pop();

p=p.right;

}

}

}

/\*

\* 中序遍历二叉树（非递归）

\*

\* 思路：先将T入栈，遍历左子树；遍历完左子树返回时，栈顶元素应为T，

\* 出栈，访问T->data，再中序遍历T的右子树。

\*/

public void PrintBinaryTreeMidUnrecur(TreeNode<T> root){

TreeNode<T> p=root;//p为当前节点

LinkedList<TreeNode> stack=new LinkedList<>();

//栈不为空时，或者p不为空时循环

while(p!=null || !stack.isEmpty()){

//当前节点不为空。压入栈中。并将当前节点赋值为左儿子

if (p!=null) {

stack.push(p);

p=p.left;

}

else

{

p=stack.pop();

**System.out.print(p.data);**

p=p.right;

}

}

}

## 3.2 查找两个节点最低祖先节点

public class LowestAncestor {

public static class Node {

public int value;

public Node left;

public Node right;

public Node(int value) {

this.value = value;

}

}

public static Node getLowestAncestor(Node head, Node o1, Node o2) {

if (head == null || head == o1 || head == o2) {

return head;

}

Node left = getLowestAncestor(head.left, o1, o2);

Node right = getLowestAncestor(head.right, o1, o2);

if (left != null && right != null) {

return head;

}

return left == null ? right : left;

}

## 3.3 二叉树深度

public class Solution {

public int TreeDepth(TreeNode root) {

if(root==null){

return 0;

}else{

return Math.max(TreeDepth(root.left),TreeDepth(root.right))+1;}

**二叉树中叶子节点的个数**

return numsOfNodeTreeNode(root.left)+numsOfNodeTreeNode(root.right)

}

}

## 3.4 二叉树结点是关键字，给出一个target数字，找出一条从根到叶子结点的和为target的路径？

public class Solution {

    private ArrayList<ArrayList<Integer>> listAll = new ArrayList<ArrayList<Integer>>();

    private ArrayList<Integer> list = new ArrayList<Integer>();

    public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {

        if(root == null) return listAll;

        list.add(root.val);

        target -= root.val;

        if(target == 0 && root.left == null && root.right == null)

            listAll.add(new ArrayList<Integer>(list));

        FindPath(root.left, target);

        FindPath(root.right, target);

        list.remove(list.size()-1);

        return listAll;

    }

}

## 3.5 二叉搜索树第k大的数字？

//思路：二叉搜索树按照中序遍历的顺序打印出来正好就是排序好的顺序。

//     所以，按照中序遍历顺序找到第k个结点就是结果。

public class Solution {

   int index = 0; //计数器

    TreeNode KthNode(TreeNode root, int k)

    {

        if(root != null){ //中序遍历寻找第k个

            TreeNode node = KthNode(root.left,k);

            if(node != null)

                return node;

            index ++;

            if(index == k)

                return root;

            node = KthNode(root.right,k);

            if(node != null)

                return node;

        }

        return null;

    }

}

## 3.6 删除二叉树的叶子结点

static Node removeLeaves( Node t )

{

if (t == null || ( t.left == null && t.right == null ))

return null;

t.left = removeLeaves( t.left );

t.right = removeLeaves( t.right );

return t;

}

## 3.7 判断一个树是否是平衡二叉树

**检查一个树是否是二叉搜索树可以使用中序遍历，根据递增排序的序列生成二权搜索树也可以使用中序遍历。**

## 3.8 二叉树的任意两节点间的最大距离

public class Test {

private int maxLen = 0;

// 该方法返回从root节点出发，向左或向右所能走的最远距离（该方法的返回值并非是整个树的最远距离，而是它的左子树最远距离和右子树最远距离两者中的较大值）

// maxLen用于保存整个二叉树的最远距离

public int findMaxLen(Node root) {

if (root == null) {

return 0;

}

if (root.left == null && root.right == null) {

return 0;

}

int leftMaxLen = findMaxLen(root.left) + 1;

int rightMaxLen = findMaxLen(root.right) + 1;

int maxTemp = leftMaxLen + rightMaxLen;

if (maxTemp > maxLen) {

maxLen = maxTemp;

}

return leftMaxLen > rightMaxLen ? leftMaxLen : rightMaxLen;

}

## 3.9 镜像二叉树

/\* 先前序遍历这棵树的每个结点，如果遍历到的结点有子结点，就交换它的两个子节点，

当交换完所有的非叶子结点的左右子结点之后，就得到了树的镜像 \*/

public class Solution {  
    public void Mirror(TreeNode root) {  
        TreeNode tmp = null;  
        if (root != null)  
            {  
            tmp = root.left;  
            root.left = root.right;  
            root.right = tmp;  
            if (root.left != null)  
                Mirror(root.left);  
            if (root.right != null)  
                Mirror(root.right);  
        }  
    }  
}

## 3.10 前序与中序，重建二叉树并返回

public class Solution {

    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {

        TreeNode root=reConstructBinaryTree(pre,0,pre.length-1,in,0,in.length-1);

        return root;

    }

    //前序遍历{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}

    private TreeNode reConstructBinaryTree(int [] pre,int startPre,int endPre,int [] in,int startIn,int endIn) {

        if(startPre>endPre||startIn>endIn)

            return null;

        TreeNode root=new TreeNode(pre[startPre]);

        for(int i=startIn;i<=endIn;i++)

            if(in[i]==pre[startPre]){

                root.left=reConstructBinaryTree(pre,startPre+1,startPre+i-startIn,in,startIn,i-1);

                root.right=reConstructBinaryTree(pre,i-startIn+startPre+1,endPre,in,i+1,endIn);

                      break;

            }

        return root;

    }

}

# 4. 位运算

## 4.1 如何用移位操作实现乘法运算？

【注】把一个数左移n位相当于把该数乘以2的n次方

## 4.2 如何判断一个数是否为2的n次方？

public static boolean isPowerOf22(int num) {

return (num & (num - 1)) == 0;

}

1000 & 0111 =0

## 4.3 如何求二进制数中1的个数？

【注】一个数n，每进行一次n&(n-1)，其结果中都会少一位1，而且是最后一位，所以可以每进行一次该操作，就计数+1，直到该操作结果为0。

public class Solution {

    public int NumberOf1(int n) {

        int count = 0;

        while(n!= 0){

            n = n & (n - 1);

count++;

         }

        return count;

    }

}

## 4.4 数值的整数次方

|  |
| --- |
| public double Power(double base, int n) {      double res = 1,curr = base;      int exponent;      if(n>0){          exponent = n;      }else if(n<0){          if(base==0)              throw new RuntimeException("分母不能为0");          exponent = -n;      }else{// n==0          return 1;// 0的0次方      }      while(exponent!=0){  **if((exponent&1)==1)**              res\*=curr;           curr\*=curr;// 翻倍           exponent>>=1;// 右移一位，删去一位      }      return n>=0?res:(1/res);  } |

## 4.5 两个整数之和 ，不用运算符

1. **两个数异或：相当于每一位相加，而不考虑进位；**
2. **两个数相与，并左移一位：相当于求得进位；**
3. **将上述两步的结果相加**

public int Add(int num1,int num2) {

while( num2!=0 ){

int sum = num1 ^ num2;

int carray = (num1 & num2) << 1;

num1 = sum;

num2 = carray;

}

return num1;

}

## 4.6 整数m和n的二进制表达，有多少个位(bit)不同么？

public int countBitDiff(int m, int n) {

        int dif=m^n;//先将二者做异或运算，得到结果；

        int cnt=0;

        while(dif!=0){

            dif=dif&(dif-1);

            cnt++;

        }             //统计一个整数dif含有多少个1；

        return cnt;

    }

13+11 = ？;

13 的二进制      1 1 0 1                     -----a        13

11 的二进制      1 0 1 1                     -----b        11

 (a&b) <<1  ->   1 0 0 1 **0**                         -----d         18

          a^b  ->     0 1 1 **0**                   -----e          6

 (d&e) <<1  ->   0 0 1 **0 0**                       ------f         4

          d^e  ->  1 0 1 **0 0**                  -----g        20

 (f&g) <<1  ->   0 1 **0 0 0**                       ------h        8

          f^g  ->  1 0 **0 0 0**                  ------i           16

 (h&i) <<1  ->   0 **0 0 0 0**                     ------h        0       ---- --------退出循环

          h^i  ->  1 1 0 0 0                  ------i           24

## 4.7 找出数组中唯一一个未重复的数

public static int NumberOf1(int[] arr) {

int len = arr.length;

int res = -1;

if(len > 1) {

res = arr[0];

for (int i = 1; i < len; i++) {

res = res ^ arr[i];//异或处理

}

}

System.out.println(res);

return res;

}

**两个数思路：**

1、依然从头到尾异或所有的数字，这样得到的结果实际上就是两个只出现了一次的数字**异或的结果，**

2、在异或后的结果中找出其二进制中最右边为1的位，该位既然为1，说明异或的两个数字对应的该位肯定不同，**必定一个为1，一个为0**

3、根据此位是否为1来划分这两个部分，**返回为true的元素异或得到一个数字为num1，返回false的所有元素异或的到num2**

## 4.8 交换A和B的值

a = a ^ b;//此时的 a 值可以看成是变量 c (上文例子中的 a ^ b = c)

b = a ^ b;//这里的 a ^ b == c ^ b == a ，所以此时的 b 就是原来 a 的值

a = a ^ b;//同上分析

**a = 9,  b = 11;**  
a=a^b; 1001^1011=0010  
b=b^a; 1011^0010=1001  
a=a^b;  0010^1001=1011

## 4.9 求1+2+3+...+n

class Solution {

public:

    int Sum\_Solution(int n) {

        int ans = n;

        ans && (ans += Sum\_Solution(n - 1));

        return ans;

    }

};

## 4.10 IP转换

public static long ipToLong(String strIp) {

String[]ip = strIp.split("**\\.**");

return (Long.parseLong(ip[0]) **<< 24**) + (Long.parseLong(ip[1]) << 16) + (Long.parseLong(ip[2]) << 8) + Long.parseLong(ip[3]);

}

# 5. 数组

## 5.1 二维数组中查找数据，左右与上下递增。

public boolean Find(int target, int[][] array) {

if(array == null || array.length == 0 || array[0].length == 0) {

return false;

}

int m = 0, n = array[0].length - 1;//从右上角开始找，array[0][n-1]

**while(m <= array.length - 1 && n >= 0)** {

if(target == array[m][n])

return true;

else if(target > array[m][n])

m ++;

else

n --;

}

return false;

}

## 5.2 旋转数组，得到最小值

**数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。**

public class Solution {

    public int minNumberInRotateArray(int [] array) {

        int low = 0 ; int high = array.length - 1;

**while(low < high)**{

**int mid = low + (high - low) / 2;**

            if(array[mid] > array[high]){

**low = mid + 1;**

            }else if(array[mid] == array[high]){**11101**

                high = high - 1;

            }else{

                high = mid;//可能是自己，在左边312

            }

        }

        return array[low];

    }

}

## 5.3 移动零

class Solution {

public void moveZeroes(int[] nums) {

int j = 0;

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

if(0 != nums[i]){

nums[j++] = nums[i];

}

}

while(j < nums.length){

nums[j++] = 0;

}

}

}

## 5.4 数组中出现次数超过一半的数字

/\*

 \* 利用map存值，找出存在最多的数字，若大于长度一半，返回此数，否则返回0

 \*/

public class Solution {

    public int MoreThanHalfNum\_Solution(int [] array) {

        if(array.length==0||array==null)

            return 0;

        Map<Integer,Integer> map=new HashMap<Integer,Integer>();

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

            if(**map.containsKey**(array[i])){

                map.put(array[i], map.get(array[i])+1);

            }else{

                map.put(array[i], 1);

            }

        }

        for (Map.Entry<Integer, Integer> entry : map.entrySet()) {

            if(entry.getValue()>array.length/2)

                return entry.getKey();

        }

        return 0;

    }

}

## 5.5 数组中两两相加等于k的组合？乘积最小，左右夹逼法

public static void findSumK(int[] array, int k) {

//排序

Arrays.sort(array);

int left = 0;

int right = array.length - 1;

while (left < right) {

if (array[left] + array[right] < k) {

left++;

} else if (array[left] + array[right] > k) {

right--;

} else {

System.out.println(array[left] + " + " + array[right] + " = 20");

left++;

right--;

}

}

}

## 5.8 数组中只出现一次的数字？

（1）第一次使用异或运算，得到了两个只出现一次的数相异或的结果。

（2）因为两个只出现一次的数肯定不同，即他们的异或结果一定不为0，一定有一个位上有1。另外一个此位上没有1，我们可以根据此位上是否有1，将整个数组重新划分成两部分，一部分此位上一定有1，另一部分此位上一定没有1，然后分别对每部分求异或，因为划分后的两部分有这样的特点：其他数都出现两次，只有一个数只出现一次。因此，我们又可以运用异或运算，**分别得到两部分只出现一次的数。**

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];//所有数异或，结果为不同的两个数字的异或

        }

        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++){

            if((array[i]&(1<<count))==0){//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（））

                num1[0]^=array[i];

            }else{

                num2[0]^=array[i];//标志位为1的为一组

            }

        }

    }

}

## 5.9 和等于 k 的最长子数组长度

<https://blog.csdn.net/speargod/article/details/99696461>

## 5.10对日期字符串进行排序

创建dateStrs字符数组变量，里面存放三个日期字符串。

创建一个dates日期数组变量，用来存放将日期字符串转换后的日期。

要将日期字符串转换成日期类型，可以使用SimpleDateFormat类来进行转换。

创建完sdf对象后，由于sdf.parse(dateStrs【i】)

用Arrays.sort()对日期数组进行排序，sdf.format(dates [i]);

## 5.11 两个有序的List合并 并保持有序

public class SortList {

public static List<Integer> sort(List<Integer> list1 , List<Integer> list2){

List<Integer> list = new ArrayList<Integer>();

int i = 0 , j = 0;

while(i < list1.size() && j < list2.size()){

if(list1.get(i) < list2.get(j)){

list.add(list1.get(i++));

}else{

list.add(list2.get(j++));

}

}

while(i < list1.size()){**//一个数组排完了，后面不用比了**

list.add(list1.get(i++));

}

while(j < list2.size()){

list.add(list2.get(j++));

}

return list;}

## 5.12 奇数位于偶数前但奇偶数的相对位置不变

public class Solution {

public void reOrderArray(int [] array) {

if(array.length==0||array.length==1) return;

int oddCount=0,oddBegin=0;

int[] newArray=new int[array.length];

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

if((array[i]&1)==1)

oddCount++;//如果是奇数的话，最低位永远都是1

}

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

if((array[i]&1)==1)

newArray[oddBegin++]=array[i]；

}

else newArray[**oddCount++**]=array[i];//从奇数位置开始

// else {

// newArray[oddCount]=array[i];

// oddCount = oddCount+1;

}

}

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

array[i]=newArray[i];

}

}

}

## 5.13 顺时针打印矩阵

/\*\*

\* 顺时针打印数组

\*/

public class PrintMatrix {

public void printMatrixClockwisely(int[][] numbers, int rows, int columns) {

if (numbers == null || rows <= 0 || columns <= 0) {

return;

}

int start = 0;

while (start \* 2 < columns && start \* 2 < rows) {

printMatrixInCircle(numbers, rows, columns, start);

start++;

}

}

public void printMatrixInCircle(int[][] numbers, int rows, int columns,int start) {

//终止列号

int endX = columns - 1 - start;

//终止行号

int endY = rows - 1 - start;

//从左至右打印一行

for(int i = start; i <= endX; i++){

System.out.print(numbers[start][i]+" ");

}

//从上到下打印一列

if(endY > start){

for(int i = start+1; i<= endY; i++){

System.out.print(numbers[i][endX] + " ");

}

}

//从右至左打印一行

if(endY > start && endX > start){

for(int i = endX-1 ; i>= start; i--){

System.out.print(numbers[endY][i] + " ");

}

}

//从下到上打印一列

if(endY - 1 > start && endX > start){

for(int i = endY-1; i >= start+1 ; i--){

System.out.print(numbers[i][start] + " ");

}

}

}

}

## 5.17 寻找缺失的整数？

【思路】先使用O(nlogn)的排序方法将原数组排序，然后遍历排序后的数组，如果相邻元素不连续，那么就找到缺少的数。

//快排

//遍历排序后的数组，如果相邻元素不连续，说明找到缺少的数

for (int i = 0; i < array.length - 1; i++) {

if (array[i] + 1 != array[i + 1]) {

return array[i] + 1;

}

}

return -1;

}

【思路】求1~10的和，用这个和依次减去数组中元素，剩下的就是缺少的数。

## 5.18 二分查找

向下找

int bsearchWithoutRecursion(int a[], int key) {

int low = 0;

int high = a.length - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

if (a[mid] > key)

high = mid - 1;

else if (a[mid] < key)

low = mid + 1;

else

return mid;

}

return -1;

}

## 5.19 素数

void printPrime(int n){//判断n是否是质数

boolean isPrime=true;//是否是质数的标志

**int s=(int)Math.sqrt(n);//对n开根号**

for(int i=s;i>1;i—){//n除以每个比n开根号小比1大的自然数

if(n%i==0){//如果有能被整除的，则不是质数

isPrime=false;

}

}

if(isPrime){//如果是质数，则打印出来

System.out.print(n+" ");

primeNumber++;//记录质数的个数

}

}

## 5.19 数位丢弃

List<Integer> list = new ArrayList<Integer>();

for (int i = 0; i <= n; i ++ )

list.add(i);

while (list.size() != 1) {

for (int i = 0; i < list.size(); **i = i + 1)**

list.remove(i)

## 5.20 duplicates are allowed at most twice?

public class Solution {

public int removeDuplicates(int[] A) {

List<Integer> list = new ArrayList<Integer>();//记住数字

Map<Integer,Integer> hash = new HashMap<>();

for(int i:A){

Integer value=**hash.get(i);**

if(value==null){//第一次出现该数字

**hash.put(i,1);**

list.add(i);

}else {

if(value<2){//该数字出现的次数少于2

list.add(i);

value++;

hash.put(i,value);//覆盖掉

}

}

}

for(int i=0;i<list.size();i++){

A[i]=list.get(i);

}

return list.size();

}

}

## 5.21 valid-parentheses

public static boolean isValid(String s) {

Stack<Character> stack = new Stack<>();

Character temp;

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

temp=s.charAt(i);

if(temp=='('||temp=='['||temp=='{'){//当前字符是左括号，入栈

stack.push(temp);

}else {

if(stack.isEmpty()){//当前括号是右括号，栈空说明没有左括号

return false;

}

Character top=stack.peek();//取栈顶元素

if(temp==')'){//需要判断当前元素与栈顶元素的匹配情况，例如："([)]" 返回：false

if(top=='('){

stack.pop();

}

}

else if(temp==']'){

if(top=='['){

stack.pop();

}

}else {

stack.pop();

}

}

}

return stack.isEmpty();//栈空则说明全部匹配了

}

## 5.22 search-insert-position（给一个值，大的个数）

public class Solution {

public int searchInsert(int[] A, int target) {

if(target<=A[0]){//1.target<=最小值；

return 0;

}else if(target==A[A.length-1]){//2.target==最大值；

return A.length-1;

}else if(target>A[A.length-1]){//3.target>最大值；

return A.length;

}else{//target介于最小值与最大值之间

for(int i=0;i<A.length-1;i++){

if(A[i]==target){//（1）target==当前值，直接返回当前值的下标；

return i;

}

if(target>A[i]&&target<A[i+1]){//（2）target>当前值，小于下一个值，就返回下一个值的下标

return i+1;

}

}

}

return -1;

}

}

## 5.23 对角线方向打印

import java.util.\*;

public class Printer {

public int[] arrayPrint(int[][] arr, int n) {

int [] result = new int[n\*n];

int x=0;

List<Integer> list = new ArrayList<>();

for(int y=n-1;y>=0;y--){//x=0 y--遍历

int x2=x;

int y2=y;

if(x2!=0&&y2!=0){

list.add(arr[x2][y2]);

}

if(x2!=0){

y2--;

}

while (x2<n&&(y2>=0&&y2<n)){

list.add(arr[x2][y2]);

x2++;

y2++;

}

}

int y=0;

for(int i=1;i<n;i++){//y=0 x++遍历

int x3=i;

int y3=y;

list.add(arr[x3][y3]);

x3++;

y3++;

while (x3<n&&(y3>=0&&y3<n)){

list.add(arr[x3][y3]);

x3++;

y3++;

}

}

for(int i=0;i<list.size();i++){

result[i]=list.get(i);

}

return result;

}

}

# 6. 符串

## 6.1 如何实现字符串单词的反转？

**【思路】String[] array = s.split(" ");**

再倒序for

StringBuilder sb = new StringBuilder();

for (String ss : array) {

sb.append(ss).append(" ");

}

## 6.2 如何判断两个字符串是否由相同的字符组成？

public static boolean compare1(String s1, String s2) {

//如果两个字符串长度不同，返回false

if (s1.length() != s2.length()) {

return false;

}

//对两个字符串分别排序，然后直接比较是否相等

char[] chars1 = s1.toCharArray();

char[] chars2 = s2.toCharArray();

Arrays.sort(chars1);

Arrays.sort(chars2);

return new String(chars1).equals(new String(chars2));

}

## 6.3 如何删除字符串中重复的字符？

public class StringTest {

public static String removeDuplicate(String s){

Set<Character> set = new LinkedHashSet<>();

for (char c : s.toCharArray()) {

set.add(c);

}

StringBuilder sb = new StringBuilder();

for (Character c : set) {

sb.append(c);

}

return sb.toString();

}

## 6.4 如何输出字符串的所有组合？

public static void main(String[] args){

char[] c = new char[]{'a','b','c'};  
int length = c.length;  
int max = 1<<length;  
for(int i = 1;i<=max;i++){  
 for(int j = 0;j<length;j++){  
 if(((1<<j)&i)!=0){  
 System.out.print(c[j]);}}  
 System.out.println("");}}

## 6.5 替换空格

public String replaceSpace(StringBuffer str) {

int oldLen = str.length();

for(int i = 0; i < oldLen; i ++) {

if(str.charAt(i) == ' ') {//每出现一个空格，在结尾添加两个任意字符，扩充字符串长度

str.append("12");

}

}

int i = oldLen - 1;

int j = str.length() - 1;

while(i >= 0 && j > i) {

char c = str.charAt(i);

i --;

if(c == ' ') {//每出现一个空格，替换为%20

str.setCharAt(j, '0');

j --;

str.setCharAt(j, '2');

j --;

str.setCharAt(j, '%');

j --;

} else {//否则照旧

str.setCharAt(j, c);

j --;

}

}

return str.toString();

}

## 6.6 两个字符串最长公共子序列长度

## 6.7 输个字符串中各元素

public class STring {

void count(String s)

{

int n=s.length();

char[] a=s.toCharArray();

int up=0,low=0,space=0,other=0;

for(int i=0;i<n;i++)

{

if(a[i]>='A'&&a[i]<='Z')

up++;

else if(a[i]>='a'&&a[i]<='z')

low++;

else if(a[i]==' ')

space++;

else

other++;

}

System.out.println("LowerCases:"+low+" UpperCases:"+up+"Spaces:"+space+" Others:"+other);

}

public static void main(String[] args) {

Scanner sc=new Scanner(System.in);

String s=new String();

System.out.println("Please input the String:");

s=sc.nextLine();

STring string=new STring();

string.count(s);

}

}

## 6.8 最长回文子串

public class LongestPalindrome {

//中心扩展法:总共有两种情况。第一种字符串为奇数，第二种字符串为偶数。无需判断字串的奇偶性，因为我们保存的是最大的回文字符串

//abcba

//abccba

public static String longestPalindrome(String s) {

String result="";

int leftIndex=0;

int rightIndex=0;

int max=0;

for(int i=0;i<s.length();i++){//子串为奇数

int left=i-1;//第i个字符串的左侧第一个字符

int right=i+1;//第i个字符串的右侧第一个字符

while (left>=0&&left<right&&right<s.length()){

if(s.charAt(left)==s.charAt(right)){//左侧与右侧相同

left--;

right++;

}else {//左侧与右侧不同的时候就跳出

break;

}

}

++left;//由于上边的循环都--了，该++了，恢复原来的位置

--right;//由于上边的循环最后都++了，该--了，恢复原来的位置

if((right-left+1)>=max){//该区间的回文长度大于最大的字符串长度，该记下相应的参数了

max=right-left+1;

leftIndex=left;

rightIndex=right;

}

}

for(int i=0;i<s.length();i++){//子串为偶数

int left=i;//左边就是该字符串了

int right=i+1;//右边第一个需要与left位置的比较

while (left>=0&&left<right&&right<s.length()){

if(s.charAt(left)==s.charAt(right)){

left--;

right++;

}else {

break;

}

}

++left;//理由均同上

--right;

if((right-left+1)>=max){

max=right-left+1;

leftIndex=left;

rightIndex=right;

}

}

result=s.substring(leftIndex,rightIndex+1);//需要注意，substring函数是左闭右开的，需要rightIndex+1

return result;

}

public static void main(String[] args) {

System.out.println(longestPalindrome("abb"));//测试下

}

}

## 6.9 最大连续子串长度

## 6.10 字符串对称

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

String s = sc.nextLine();

System.out.println(s.length());

int count = (s.length()-1)/2;

for(int x = 0;x <= count;x++){

**if(s.charAt(x) != s.charAt(s.length()-1-x)){**

System.out.println("该字符串不对称");

break;

}else{

System.out.println("该字符串对称");

break;

}

}

}

# . 其他

## 8.0 文件名获取

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

while(sc.hasNext()) {

String line = sc.nextLine();

int index = line.lastIndexOf('.') + 1;

String result = (index == 0) ? "null" : line.substring(index);

System.out.println(result);

}

sc.close();

}

}

## 8.1 如何不使用比较运算就求出两个数哪个大哪个小？

【思路】max(a, b) = (a + b + |a - b|) / 2，min(a, b) = (a + b - |a - b|) / 2。

## 8.2 如何求出最大公约数？

**[思路]**两个正整数a和b（a > b），它们的最大公约数等于a % b和b的最大公约数。

public static int getGreatestCommonDivisor1(int a, int b) {

int big = a > b ? a : b;

int small = a < b ? a : b;

if (big % small == 0) {

return small;

}

return getGreatestCommonDivisor1(big % small, small);

}

## 8.3 斐波那契数列

**递归**

if(n<=2) return n;

else return Fibonacci(n-1)+Fibonacci(n-2);

**动态规划**

if(n<=2){

    return n;

}

        int first = 1, second = 2, third = 0;

        for (int i = 3; i <= n； i++) {

            third = first + second;

            first = second;

            second = third;

        }

        return third;

## 8.4 找到一个文件夹下所有文件，包括子文件夹

 public static void listFileDic(File file){

        File[] listFiles = file.listFiles();

        for (File file2 : listFiles) {

           if(file2.isDirectory()){

                System.out.println(file2.getName() +"：是一个文件夹");

                listFileDic(file2);

            }

            if(file2.isFile()){

                System.out.println(file2.getName() +"：是一个文件");

            }

        }

    }

## 8.5 那一瓶有毒

给1000个瓶分别标上如下标签（10位长度）：

0000000001 （第1瓶）

0000000010 （第2瓶）

0000000011 （第3瓶）

……

1111101000 （第1000瓶）

从编号最后1位是1的所有的瓶子里面取出1滴混在一起（比如从第一瓶，第三瓶，。。。里分别取出一滴混在一起）并标上记号为1。以此类推，从编号第一位是1的所有的瓶子里面取出1滴混在一起并标上记号为10。现在得到有10个编号的混合液，小白鼠排排站，分别标上10，9，。。。1号，并分别给它们灌上对应号码的混合液。24小时过去了，过来验尸吧：

从左到右，死了的小白鼠贴上标签1，没死的贴上0，最后得到一个序号，把这个序号换成10进制的数字，就是有毒的那瓶水的编号。

检验一下：假如第一瓶有毒，按照0000000001 （第1瓶），说明第1号混合液有毒，因此小白鼠的生死符为0000000001（编号为1的小白鼠挂了），0000000001二进制标签转换成十进制=1号瓶有毒；假如第三瓶有毒，0000000011 （第3瓶），第1号和第2号混合液有毒，因此小白鼠的生死符为00000011（编号为1，2的鼠兄弟挂了），0000000011二进制标签转换成十进制=3号瓶有毒。

## 8.6 LRU（最简单）

因为显然 cache 必须有**顺序之分**，以区分最近使用的和久未使用的数据；而且我们要在 cache 中查找键是否已存在；如果容量满了要删除最后一个数据；**每次访问还要把数据插入到队头。**

哈希表查找快，但是数据无固定顺序；链表有顺序之分，插入删除快，但是查找慢。所以结合一下，形成一种新的数据结构：哈希链表。

到这里就能回答刚才“为什么必须要用双向链表”的问题了，因为我们需要删除操作。删除一个链表节点不光要得到该节点本身的指针，也需要操作其前驱节点的指针，而双向链表才能支持直接查找前驱，保证操作的时间复杂度 *O*(1)。

<http://ifeve.com/%E5%8A%A8%E6%89%8B%E5%AE%9E%E7%8E%B0%E4%B8%80%E4%B8%AA-lru-cache/>

**LinkedHashMap是有序的，且默认为插入顺序。只有头结点的双向链表**

class LRUCache<K, V> extends **LinkedHashMap**<K, V> {

private final int CACHE\_SIZE;

/\*\*

\* 传递进来最多能缓存多少数据

\*

\* @param cacheSize 缓存大小

\*/

public LRUCache(int cacheSize) {

//public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)

// true 表示让 linkedHashMap 按照访问顺序来进行排序，最近访问的放在头部，最老访问的放在尾部。

super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);

CACHE\_SIZE = cacheSize;

}

@Override //LinkedHashMap自带的判断是否删除最老的元素方法，默认返回false，即不删除老数据

protected boolean removeEldestEntry(**Map.Entry<K, V> eldest**) {

// 当 map中的数据量大于指定的缓存个数的时候，就自动删除最老的数据。

return size() > CACHE\_SIZE;

}

}

## 8.7 如何确保三个线程顺序执行？

public class Main implements Runnable {

private Integer num;

private Object lock;

public Main(Integer num, Object lock) {

this.num = num;

this.lock = lock;

}

public static void main(String[] args) {

Object lock = new Object();

Thread thread1 = new Thread(new Main(1, lock));

Thread thread2 = new Thread(new Main(2, lock));

thread1.start();

thread2.start();

}//main

@Override

public void run() {

while (true) {

synchronized (lock) {//得到锁

try {

lock.notify();//唤醒其他的线程，交替唤醒

lock.wait();//自己等待，释放锁,交替等待

System.out.println(num+","+Thread.currentThread().getName());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}//run

}

//1,Thread-0

//2,Thread-1

//1,Thread-0

//2,Thread-1

//1,Thread-0

//2,Thread-1

<https://blog.csdn.net/Evankaka/article/details/80800081>

<http://wingjay.com/2017/04/09/Java%E9%87%8C%E5%A6%82%E4%BD%95%E5%AE%9E%E7%8E%B0%E7%BA%BF%E7%A8%8B%E9%97%B4%E9%80%9A%E4%BF%A1%EF%BC%9F/>

## 两个线程，一个打印奇数，一个打印偶数

/\*\*

\* 两个线程，一个打印奇数，一个打印偶数，连续输出1-100

\* Created by vendetta on 2018/8/31.

\*/

public class OddAndEven {

//state==1表示线程1开始打印，state==2表示线程2开始打印

private static int state = 1;

private static int odd = 1; private static int even = 0;

public static void main(String[] args) {

OddAndEven oddAndEven = new OddAndEven();

new Thread(() -> {

while (odd <= 100){

synchronized (oddAndEven){

if (state !=1){

try {

oddAndEven.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(odd);

odd += 2;

state = 2;

oddAndEven.notifyAll();

}

}

}).start();

new Thread(() -> {

while (even <= 100){

synchronized (oddAndEven){

if (state !=2){

try {

oddAndEven.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(even);

even += 2;

state = 1;

oddAndEven.notifyAll();

}

}

}).start();

}}

## 8.8 TopN问题

<https://blog.csdn.net/nuowei_senlin/article/details/86734931>

## 8.9 正则表达式

IPV4

(((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))\.){3}((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))

## 8.10 解决生产者-消费者问题？

**用lock、synchronized、阻塞队列三种方法实现生产者消费者模式**

<https://blog.csdn.net/antony9118/article/details/51500278>

public class ProducerConsumerSolution {

public static void main(String args[]) {

Vector sharedQueue = new Vector();

int size = 4;

Thread prodThread = new Thread(new Producer(sharedQueue, size), "Producer");

Thread consThread = new Thread(new Consumer(sharedQueue, size), "Consumer");

prodThread.start();

consThread.start();

}

}

class Producer implements Runnable {

private final Vector sharedQueue;

private final int SIZE;

public Producer(Vector sharedQueue, int size) {

this.sharedQueue = sharedQueue;

this.SIZE = size;

}

@Override

public void run() {

for (int i = 0; i < 7; i++) {

System.out.println("Produced: " + i);

try {

produce(i);

} catch (InterruptedException ex) {

Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

private void produce(int i) throws InterruptedException {

// wait if queue is full

while (sharedQueue.size() == SIZE) {

synchronized (sharedQueue) {

System.out.println("Queue is full " + Thread.currentThread().getName()

+ " is waiting , size: " + sharedQueue.size());

**sharedQueue.wait();**

}

}

// producing element and notify consumers

synchronized (sharedQueue) {

sharedQueue.add(i);

sharedQueue.notifyAll();

}

}

}

class Consumer implements Runnable {

private final Vector sharedQueue;

private final int SIZE;

public Consumer(Vector sharedQueue, int size) {

this.sharedQueue = sharedQueue;

this.SIZE = size;

}

@Override

public void run() {

while (true) {

try {

System.out.println("Consumed: " + consume());

**Thread.sleep(50);**

} catch (InterruptedException ex) {

Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

private int consume() throws InterruptedException {

// wait if queue is empty

while (sharedQueue.isEmpty()) {

synchronized (sharedQueue) {

System.out.println("Queue is empty " + Thread.currentThread().getName()

+ " is waiting , size: " + sharedQueue.size());

**sharedQueue.wait();**

}

}

// Otherwise consume element and notify waiting producer

synchronized (sharedQueue) {

sharedQueue.notifyAll();

return (Integer) sharedQueue.remove(0);

}

}

}

## 8.11 如何从海量日志中提取访问最多的10个IP

算法思想：**分而治之+Hash**

分而治之/hash映射 + hash统计 + 堆/快速/归并排序，说白了，就是先映射，而后统计，最后排序；

1、IP地址最多有2^32=4G种取值情况，所以不能完全加载到内存中处理；

2、可以考虑采用分而治之的思想，按照IP地址的Hash(IP) %1024的值，把海量IP日志分别存储到**1024个小文件中**，这样，每个小文件最多**包含4MB个IP地址；**

Hash(IP)%1024值，这样的话，通过计算IP的Hash值，**相同IP肯定会放到一个文件**中，当然不同的IP的Hash值也可能相同，就存在一个小文件中。

3、对于每一个小文件，可以构建一个IP为key，**出现的次数为value的HashMap**，同时记录当前出现次数最多的那个IP地址；

4、可以得到1024个小文件中的出现次数最多的那个IP，再依据常规的排序算法得出总体上出现次数最多的IP。

## 8.12 大数相加

public static void main(String[] args) {

String str1 = "123459";

String str2 = "123";

System.out.println(add(str1, str2));//123582

}

private static String add(String str1, String str2) {

//任何一个字符串为null或空字符串，都不需要相加了

if (str1 == null || "".equals(str1)) {

return str2;

}

if (str2 == null || "".equals(str2)) {

return str1;

}

int maxLength = Math.max(str1.length(), str2.length());

//定义一个存贮结果的字符串，长度要比最大长度字符串**还长一位**，用于存储可能出现的进位

StringBuffer result = new StringBuffer(maxLength + 1);

//翻转两个字符串

str1 = new StringBuffer(str1).reverse().toString();

str2 = new StringBuffer(str2).reverse().toString();

//反转后的结果分别为：

//954321

//321

int minLength = Math.min(str1.length(), str2.length());

//进位

int carry = 0;

//当前位上的数值

int currentNum = 0;

//循环变量

int i = 0;

for (; i < minLength; i++) {

//分别获取两个字符对应的字面数值，然后相加，再加上进位

currentNum = str1.charAt(i) + str2.charAt(i) - 2 \* '0' **+ carry;**

//获取进位

**carry = currentNum / 10;**

//处理当前位的最终值

currentNum %= 10;

//保存当前位的值到最终的字符缓冲区中

result.append(String.valueOf(currentNum));

}

if (str1.length() < str2.length()) {

//选择

str1 = str2;

}

for (; i < str1.length(); i++) {

//分别获取两个字符对应的字面数值，然后相加，再加上进位

currentNum = str1.charAt(i) - '0' + carry;

//获取进位

carry = currentNum / 10;

//处理当前位的最终值

currentNum %= 10;

//保存当前位的值到最终的字符缓冲区中

result.append(String.valueOf(currentNum));

}

//处理最后一个的进位(当循环结束后，是不是还可能会有一个进位)

if (carry > 0) {

result.append(String.valueOf(carry));

}

//最后翻转恢复字符串，再返回

return result.reverse().toString();

}