**package** danji;

**import** java.util.\*;

**public** **class** Cascadei {

//密钥总长度

**int** n;

//误码率

**double** p;

//块长参数（目的：设置合适的分组长度使得每组包含的错误个数不大于1），alpha取值在0.5~1之间

**double** alpha = 0.58;

//块长增长参数，beta大于等于2

**double** beta = 2.5;

//Alice的密钥

**int**[] aliceKeys;

//Bob的密钥

**int**[] bobKeys;

//第二轮置乱后的Alice密钥

**int**[] alieceKeysShuffled;

//第二轮置乱后Bob的密钥

**int**[] bobKeysShuffled;

//第一轮Alice的分块，如aliceBlock[0]表示第一个分块，也是个数组 //编号，第一轮Alice的分块，如aliceBlock[0]表示第一个分块，也是个数组，第一个【m】表示第m个块，第二个【n】表示第n位数据

**int**[][] aliceBlocks1;

//第一轮Bob的分块

**int**[][] bobBlocks1;

//第二轮Alice的分块

**int**[][] aliceBlocks2;

//第二轮Bob的分块

**int**[][] bobBlocks2;

//第一轮分块长度//第一轮分块长度（一般取1/2P<blockLength1<1/p)

**int** blockLength1 = (**int**) Math.*rint*(alpha/p);

//第二轮分块长度

**int** blockLength2 = (**int**) Math.*rint*(blockLength1 \* beta);

//第三轮Alice的密钥

**int**[] aliceKeysR3;

//第三轮Bob的密钥

**int**[] bobKeysR3;

//置乱的规则

**int**[] mapRule;

//泄露的信息

**int** leakedMsg = 0;

//已纠正的错误数

**int** correctedErr = 0;

//错误分块的集合（都是第一轮的分块），即第几个分块还存在奇数个错误

List<Integer> errBlocks = **new** LinkedList<>();

**public** Cascadei(**int**[] aliceKeys, **int**[] bobKeys, **int** n, **double** p){

**this**.aliceKeys = Arrays.*copyOf*(aliceKeys, aliceKeys.length);

**this**.bobKeys = Arrays.*copyOf*(bobKeys, bobKeys.length);

**this**.n = n;

**this**.p = p;

}

**public** Cascadei(){}

//执行cascade流程，返回纠错后的bobKeys

**public** **int**[] cascadei(){

//产生Alice和Bob的分块密钥

aliceBlocks1 = divIntoBlocks(aliceKeys, blockLength1);//二维数据

bobBlocks1 = divIntoBlocks(bobKeys, blockLength1);

//第一轮纠错

**for** (**int** i=0; i<aliceBlocks1.length; i++){//aliceBlocks1和blockLength1可能不一致，获取了行数的长度

**if** (!parityCheck(aliceBlocks1[i], bobBlocks1[i])){//aliceBlocks1的第i行和bobBlocks1的第i行，if(true or flase）

//奇偶值不一样继续进行

**int** index = binary(aliceBlocks1[i], bobBlocks1[i]);

bobKeys[blockLength1\*i + index] ^= 1;//在Bob的密钥上进行比特翻转

} //第一轮纠错后所有分组都只存在偶数个错误，或没有错误

}

//这里开始第二轮，先获取随机置乱规则

mapRule = getMapRule();//获得一个数据顺序是乱序，长度为n的一维数组

//获取Alice和Bob置乱后的密钥

alieceKeysShuffled = getShuffledKeys(aliceKeys);

bobKeysShuffled = getShuffledKeys(bobKeys);

//将置乱后的密钥分块

aliceBlocks2 = divIntoBlocks(alieceKeysShuffled, blockLength2);

bobBlocks2 = divIntoBlocks(bobKeysShuffled, blockLength2);

//第二轮第一阶段，和第一轮一样

**for** (**int** i=0; i<aliceBlocks2.length; i++){

**boolean** parity = parityCheck(aliceBlocks2[i], bobBlocks2[i]);

**if** (!parity){

**int** index = binary(aliceBlocks2[i], bobBlocks2[i]); //index指的是每一个块的错误位置，errIndex指的是每个分块中错误位置在未分块中（整串数据中）的位置

**int** errIndex = blockLength2\*i + index;

//这里的纠错只是把其他需要纠正的密钥中的错误一起纠正了，真正的纠正由binary方法完成

**int** blockId1 = correctErr2(errIndex);//返回来的是第二轮错误在第一轮中第几块的第几个位置，（即blockId1是一个二维数组？）

//被纠正的错误在第一轮中处于第 blockId 个分块，将它添加到错误分块中

addToErrBlocks(blockId1);

}

}

//（因为原来一轮纠正后的奇偶性相同，存在偶数个错误，第二轮第一阶段纠正了一个错误就会导致原来的偶数个错误变成奇数个，导致奇偶性不同）

//回溯阶段 对错误集合中的分块继续二分纠错，在一二轮中交替循环寻找错误位置并纠正，实际上是一个错误码纠正三次（三个阶段中都纠正）

//System.out.println("----------回溯-------------");

//只要errBlocks不为空，就不断往返进行回溯

**while** (!errBlocks.isEmpty()){

**int** blockId1 = errBlocks.get(0); //取出错误集合中的第一个（每次取出一个错误集合做完回溯纠错就立马删除，所以每次取的都是错误集合的第一个）

//get(0)就是调用0位置处的属性值

**int** index = binary(aliceBlocks1[blockId1], bobBlocks1[blockId1]); //在第一轮也就是未置乱前将二轮找出的存在错误的块进行二分纠错

//纠正以后立马移除

errBlocks.remove(**new** Integer(blockId1));//remove()删除数组

//得到第一轮分块中的错误位置

**int** errIndex = blockId1 \* blockLength1 + index;

//blockId2是第一轮的错误位置在第二轮的第几个块

**int** blockId2 = correctErr1(errIndex);//调用correctErr1（）

//由第一轮的错误连锁纠正第二轮中的错误

index = binary(aliceBlocks2[blockId2], bobBlocks2[blockId2]);//调用binary纠错

errIndex = blockId2 \* blockLength2 + index;

//循环，返回来的是第二轮错误在第一轮中第几块的第几个位置，并在调用方法中完成纠正纠正

blockId1 = correctErr2(errIndex);

//由第二轮的错误又可以得到第一轮的错误，再次添加到错误集合中

addToErrBlocks(blockId1);

}

//当errBlocks.isEmpty，即错误集合为空时，第二轮纠错结束

//第二轮纠错完成后，一二轮的所有分组都可能存在偶数个错误

//第三轮，把第二轮之后的轮次统称为第三轮

//这里为了保证成功率循环纠错5次，某篇改进文章中循环了10次

**for** (**int** i=0; i<5; i++){

mapRule = getMapRule();

//Alice和Bob第三轮的密钥

aliceKeysR3 = getShuffledKeys(aliceKeys);

bobKeysR3 = getShuffledKeys(bobKeys);

//考虑到两轮过后误码率极低，因此块长直接固定为n/2

**int**[] aliceKeysR3Left = Arrays.*copyOfRange*(aliceKeysR3, 0, n/2);

**int**[] aliceKeysR3Right = Arrays.*copyOfRange*(aliceKeysR3, n/2, n);

**int**[] bobKeysR3Left = Arrays.*copyOfRange*(bobKeysR3, 0, n/2);

**int**[] bobKeysR3Right = Arrays.*copyOfRange*(bobKeysR3, n/2, n);

**if** (!parityCheck(aliceKeysR3Left, bobKeysR3Left)){

**int** errIndex1 = binary(aliceKeysR3Left, bobKeysR3Left);

**int** blockId = correctErr2(errIndex1);

addToErrBlocks(blockId);

//如果左段出现错误，那么右段也必然有错误，可以直接纠错

**int** errIndex2 = binary(aliceKeysR3Right, bobKeysR3Right) + n/2;

blockId = correctErr2(errIndex2);

addToErrBlocks(blockId);

}

**while** (!errBlocks.isEmpty()){

**int** blockId1 = errBlocks.get(0);

**int** index = binary(aliceBlocks1[blockId1], bobBlocks1[blockId1]);

//纠正以后立马移除

errBlocks.remove(**new** Integer(blockId1));

//得到第一轮分块中的错误位置

**int** errIndex = blockId1 \* blockLength1 + index;

bobKeys[errIndex] ^= 1;

}

}

// System.out.println("本轮纠正:" + correctedErr);

//System.out.println(leakedMsg);

//该协议最终返回协商效率

//return 1 - (double)leakedMsg/n;

**return** bobKeys;//返回bob密钥

}

//---------以下是调用函数集合------------------------------------------------------------------

**private** **void** addToErrBlocks(**int** blockId1)

//将该block添加到错误集合中

//如果集合中已经有了这个block，说明之前该block就有奇数个错误，纠正了一个错误

//之后反倒包含偶数个错误，将该block移除

{

**if** (errBlocks.contains(blockId1)){//当且仅当此字符串包含指定的char值（blockid1）序列时，返回true

errBlocks.remove(**new** Integer(blockId1));//如果错误集合中已存在blockId1这个块，移除列表中首次出现的指定元素

}**else**{

errBlocks.add(blockId1);

}

}//如果errblocks中包括blockId1，就移除掉new Integer(blockId1）所指的块，如果没有，则添加blockId1

**private** **int** correctErr1(**int** errIndex)

//第一轮校正器

//根据第一轮错误的下标纠错，纠正三处错误（其实是一个错误）

// 1 第一轮Bob的密钥

// 2 第二轮Bob的密钥

// 3 第二轮Bob的分块密钥

{

bobKeys[errIndex] ^= 1;//errIndex= blockId1 \* blockLength1 + index;

//将第一轮密钥中的错误位置取反

//第一轮的错误在第二轮Bob分块密钥中的位置

**int**[] pos = getPosByErrIndex1to2(errIndex);//得到一个一维数组，有两个数

//将错误位置对应在第二轮的位置纠正

bobKeysShuffled[pos[0]\*blockLength2+pos[1]] ^= 1;

//将错误位置对应在第二轮分块的位置纠正

bobBlocks2[pos[0]][pos[1]] ^= 1;

//该方法顺带返回该错误在第二轮Bob分块中处于第几个分块

**return** pos[0];

}

**private** **int** correctErr2(**int** errIndex)

//根据第二轮错误下标纠错，纠正三处错误（其实是一个错误）：

// 1 第二轮Bob密钥

// 2 第一轮Bob密钥

// 3 第一轮Bob分块密钥

//第二轮Bob分块的错误已经在BINARY中纠正了

{ //将第二轮置乱密钥的错误位置取反

bobKeysShuffled[errIndex] ^= 1;

//该错误在第一轮Bob分块密钥中的位置（在块中的具体位置）在第blockId块的第 blockPos个位置

**int**[] pos = getPosByErrIndex2to1(errIndex);

//将第一轮密钥（原始密钥)错误位取反

bobKeys[mapRule[errIndex]] ^= 1;

//将第一轮每个分块里的错误位取反

bobBlocks1[pos[0]][pos[1]] ^= 1;

//该方法顺带返回该错误在第一轮Bob分块中处于第几个分块

**return** pos[0];

}

**private** **int**[] getPosByErrIndex1to2(**int** index)

//根据第一轮错误的下标，它会告诉你这个错误在第二轮属于第几个分块的第几个位置

{

//index2是该错误在第二轮中的下标

**int** index2 = 0;

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

**if** (mapRule[i] == index) {//调用 mapRule = getMapRule()是一个一维数组;

//如果数组中的第i位和第一轮错误位置下标相同

index2 = i;//则第二轮错误下标替换成i

**break**;//跳出for循环继续执行下一步

}

}

**int** blockId = (**int**)Math.*floor*((**double**)index2/blockLength2);

//对blockid执行向下取整，数据是第二轮错误下标除以第二轮块长

**int** blockPos = index2 % blockLength2;

//第二轮错误下标除以第二轮下标取余数

**return** **new** **int**[]{blockId, blockPos};//返回一个一维数组，数组的第0位是商，第1位是余数

}

**private** **int**[] getPosByErrIndex2to1(**int** index)

//根据第二轮错误的下标，它会告诉你这个错误在第一轮属于第几个分块的第几个位置

{//blockId 指第一轮中的第几块

**int** blockId = (**int**)Math.*floor*((**double**)mapRule[index]/blockLength1);

//%除法取余数，blockPos指第几个位置

**int** blockPos = mapRule[index] % blockLength1;

**return** **new** **int**[]{blockId, blockPos};

}

**private** **int**[] getShuffledKeys(**int**[] origKeys)

//参数为被置乱密钥

//获取置乱后的密钥,对任何密钥都适用，给它一串密钥它就能按照定义好的规则置乱

{

**int**[] shuffledKeys = **new** **int**[origKeys.length];

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

shuffledKeys[i] = origKeys[mapRule[i]];//置乱数组的第i位为原数组的置乱数组的第i位对应的数

//相当于把密钥中随机一个位置替换到第i位达到置乱密钥的效果

}

**return** shuffledKeys;

}

**private** **int**[] getMapRule()

//获取置乱规则

{

**int**[] a = **new** **int**[n];//设置一个和密钥长度相同的一维数组a

**int** i = 0;

**while** (i < n){

a[i] = i++;//从0开始把数据写入数组a中，数据从0到n-1

}

**return** shuffleKeys(a);//调用随机置乱函数，参数为a数组，返回一个打乱顺序后的一维数组

}

**private** **int**[] shuffleKeys(**int**[] a)

//随机置乱，由于Arrays.aslist不能用于基本类型数组，所以这个方法还有点麻烦

//要么先把int数组转为Integer数组，再用Arrays.aslist，要么就逐个添加

//Python可以直接置乱List，方便很多

//置乱后再把数组变回int数组

{

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

**for** (**int** i : a) {//选定a数组循环

list.add(i);//把数组数据加入到list接口中

}

Collections.*shuffle*(list);//在接口中进行置乱，打乱数据顺序

**return** list.stream().mapToInt(Integer::*valueOf*).toArray();

//list.stream().mapToInt(Integer::valueOf).toArray()是List<Integer>转int[]的表示，转化的过程中,转换的基本数据类型要与引用数据类型保持一致

}

**private** **int** binary(**int**[] a, **int**[] b)

//BINARY纠错，先找出一个错误位置，再实现纠错

{

**int** index = binaryRecursion(a, b, 0);

b[index] ^= 1;//比特翻转

correctedErr += 1;//纠正个数+1

**return** index;

}

**private** **int** binaryRecursion(**int**[] a, **int**[] b, **int** n)

//BINARY纠错,返回的是出现错误的下标，并没有真正纠正错误

{//n=k+r，分组码（n,k），码组长度n。信息位k，r个监督关系式（r为校正子数），可以指明一个错码的（2^r-1)个位置

//汉明码（7,4）码的最小码距是3，可以检测两个错码或纠正一个错码

**if** (a.length <= 7){ //二分到块长小于7就不往下分了，这里n等于7，校正子有3个

**return** (7-errIndex(Arrays.*copyOf*(a, 7), Arrays.*copyOf*(b, 7)))+n;//这里n=0

}

//再div处将数组一分为二，并进行递归

**int** div = (**int**) Math.*floor*(a.length/2.0);//数组长度除以2，向下取整，div是分开的位置，奇数位，前短后长

**if** (parityCheck(Arrays.*copyOfRange*(a, 0, div), Arrays.*copyOfRange*(b, 0, div)))

//对前一块分块进行奇偶检验

{

**return** binaryRecursion(Arrays.*copyOfRange*(a, div, a.length), Arrays.*copyOfRange*(b, div, b.length), n+div);

}**else**{

**return** binaryRecursion(Arrays.*copyOfRange*(a, 0, div), Arrays.*copyOfRange*(b, 0, div), n);

}//二分递归，在大于7的时候会递归至小于等于7寻找错误位置

}//二分到a的长度小于等于7的时候不再二分，转到校正子寻找错误位置

**private** **int** errIndex(**int**[] a, **int**[] b)

//返回错误码的位置

{

**int**[] as = getCorrectI(a);//读取校正子

**int**[] bs = getCorrectI(b);//读取校正子

**return** 4\*(as[0]^bs[0])+2\*(as[1]^bs[1])+as[2]^bs[2];//对ab校正子做异或运算，相同为0，相反为一，可以得到最终的监督码

//s1s2s3的8421码-1对应的a的下标就是错误所在位置

//错误关系表

//s[0]s[1]s[2]为001时，错误位置在a[0];010时在a[1];100时在a[2]；

//011时在a[3]；101时在a[4]；110在a[5]；111在a[6]

}

**private** **int**[] getCorrectI(**int**[] a)

//计算校正子

{//s[]为校正子，监督位数r=3；3个校正子能够指明7个错码的位置

leakedMsg++;//Alice通信校正子会让Eve知道码组里面的一个位置

**int**[] s = **new** **int**[3];//s数组

s[0] = a[3]^a[2]^a[1]^a[0];//当校正子的值为1时，四个码元位置上有错码，构成偶数监督关系。

s[1] = a[5]^a[4]^a[1]^a[0];

s[2] = a[6]^a[4]^a[2]^a[0];

**return** s;

}

**private** **boolean** parityCheck(**int**[] a, **int**[] b)

//奇偶检验

{

**int** parity1 = 0;

**int** parity2 = 0;

**for** (**int** i=0; i<a.length; i++){//一维数组长，blocks的列数长

parity1 ^= a[i];

parity2 ^= b[i];

}//异或运算，for循环，最后得到0对一维数组里面的数的异或结果

//泄露的信息加一

leakedMsg += 1;//每次发送

**return** parity1 == parity2;//奇偶值相同返回

}

**private** **int**[][] divIntoBlocks(**int**[] keys, **int** blockLength)

//将密钥按指定的分块大小分块，得到的是一个二维数组，每个元素是一个分块（也是一个01数组）

{

**int** blockNum = (**int**) Math.*ceil*((**double**)keys.length/blockLength);//指密钥长度/块长

**int**[][] blocks = **new** **int**[blockNum][];//二维数组，行数是分块编号，行长是分块数；列数是读入密钥所在为止下标，列长是分块长blockLength

**for** (**int** i=0; i<blockNum; i++){

**if** (i != blockNum-1){

blocks[i] = Arrays.*copyOfRange*(keys, i\*blockLength, (i+1)\*blockLength);

//Arrays.copyOfRange(T[ ] original,int from,int to)

//将一个原始的数组original，从下标from开始复制，复制到上标to，生成一个新的数组。注意这里包括下标from，不包括上标to。

}**else**{

blocks[i] = Arrays.*copyOfRange*(keys, i\*blockLength, keys.length);

}//写入每个分块的数据

}

**return** blocks;

}//分块函数

**int** getLeakedMsg(){

**return** leakedMsg;

}

**int** getcorrectedErr(){

**return** correctedErr;

}

**public** **int** Simulation()

//仿真过程

{

**int**[] a = aliceKeys;

**int**[] b = bobKeys;

**int** x = 0;

**for** (**int** i = 0; i < a.length; i=i+100) {

**if** (!parityCheck(Arrays.*copyOfRange*(a, i, i+100), Arrays.*copyOfRange*(b, i, i+100))) {//进行奇偶校验

x++;

}

}

**return** x;

}

}