### 快速io板子：

import java.io.\*;  
import java.util.\*;  
public class Main {

public static StreamTokenizer *in* = new StreamTokenizer(new BufferedReader(new InputStreamReader(System.*in*)));  
 public static BufferedReader *inString* = new BufferedReader(new InputStreamReader(System.*in*));  
 public static PrintWriter *out* = new PrintWriter(new OutputStreamWriter(System.*out*));

//前三个必要，后面可以选择需要的写  
 public static double nextDouble() throws IOException{ *in*.nextToken(); return *in*.nval; }  
 public static float nextFloat() throws IOException{ *in*.nextToken(); return (float)*in*.nval; }  
 public static int nextInt() throws IOException{ *in*.nextToken(); return (int)*in*.nval; }

public static long nextLong() throws IOException{ *in*.nextToken(); return (long)*in*.nval; }  
 public static String next() throws IOException{ return *inString*.readLine(); }

public static void main(String[] args)throws IOException {  
 String s=*next*();  
 int a=*nextInt*();  
 s=s+a;  
 *out*.println(s);  
 *out*.flush();//一定要加  
 }  
}

### **快速io2(更快但空间更多)：**

static BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));  
static StringTokenizer tokenizer = new StringTokenizer("");  
// nextLine()读取字符串  
static String nextLine() throws IOException   { return reader.readLine(); }  
// next()读取字符串  
static String next() throws IOException  
{  
    while (!tokenizer.hasMoreTokens())  tokenizer = new StringTokenizer(reader.readLine());  
    return tokenizer.nextToken();  
}  
// 读取一个int型数值  
static int nextInt() throws IOException  { return Integer.parseInt(next()); }  
// 读取一个double型数值  
static double nextDouble() throws IOException { return Double.parseDouble(next()); }  
// 读取一个long型数值  
static long nextLong() throws IOException  { return Long.parseLong(next());}  
// 读取一个BigInteger  
static BigInteger nextBigInteger() throws IOException { return new BigInteger(nextLine());}

### 遍历全排列

import java.util.Scanner;public class Main { public static void main(String[] args) { Scanner scanner=new Scanner(System.in); int n=scanner.nextInt(); int[] a=new int[n]; for (int i = 0; i < n; i++) { a[i]=i+1; } permutations(a, 0, n-1); } public static void permutations(int[] a, int m, int n) { if (m == n) { for (int i = 0; i <= n; i++) { System.out.println(a[i]); } } else { for (int i = m; i <= n; i++) { int temp = a[m]; a[m] = a[i]; a[i] = temp; permutations(a, m + 1, n); temp = a[m]; a[m] = a[i]; a[i] = temp; } } }}

### C++库与排序

#include<bits/stdc++.h>

/\*#include＜algorithm＞

#include <vector>

#include <string>\*/

using namespace std;

//排序（不加cmp为升序）

bool cmp(int a,int b){

return a>b;

//可以理解为a>b时，把a放在前面

}

int main(){

int a[5] = {3,5,2,1,4};

sort(a,a+5,cmp);

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

printf("%d",a[i]);

}

return 0;

}

//lower\_bound(a,a+a.lenth,b) 从数组a(长度为a.lenth)中找到>=b的最小指针[begin , end)

//upper\_bound(a,a+a.lenth,b) 从数组a(长度为a.lenth)中找到>b的最小指针

//长度为n的数组a中b的个数:upper\_bound(a,a+n,b)-lower\_bound(a,a+n,b);

### DFS(伪代码)：

dfs(u) {

vis[u] = 1;

for (int i = head[u]; i != 0; i = edge[i].next) {//to是往下，next是往旁边

if (vis[edge[i].to]==0) {

dfs(edge[i].to);

}

}

}

### BFS(伪代码)：

bfs(s) {

q = new queue()

q.push(s), visited[s] = true

while (!q.empty()) {

u = q.pop()

for each edge(u, v) {

if (!visited[v]) {

q.push(v)

visited[v] = true

}

}

}

}

### 归并排序板子（可以改成对类中某字段排序）：

public static void merge(long[] arry){

int n=arry.length;

long[] temArr = new long[n];

mergeDiv(arry,temArr,0,n-1);}

private static void mergeDiv(long[] arr,long[] temArr,int right,int left) {

if(right<left)

{

int mid = (right+left)/2;

mergeDiv(arr,temArr,right,mid);

mergeDiv(arr,temArr,mid+1,left);

mergeSort(arr,temArr,right,mid,left);

}

}

private static void mergeSort(long[] arry,long[] temArr,int left ,int mid,int right)

{

int leftPos=left;

int rightPos=mid+1;

int pos=left;

while(leftPos<=mid&rightPos<=right)

{

if(arry[rightPos]<arry[leftPos])

{

temArr[pos++]=arry[rightPos++];

}else {

temArr[pos++]=arry[leftPos++];

}

}

while(leftPos<=mid)

{

temArr[pos++]=arry[leftPos++];

}

while(rightPos<=right)

{

temArr[pos++]=arry[rightPos++];

}

while(left<=right)

{

arry[left]=temArr[left];

left++;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*归并排序\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### 以边建树（树链剖分）板子（LCA找树上两点公共父节点；dfs；）：

//树上多个点的LCA，就是DFS序最小的和DFS序最大的这两个点的LCA  
import java.util.\*;  
class Tree{  
 int n;  
 static class Edge{  
 int to;  
 int nx;//nx是当前边最近一条同起点的邻边的编号  
 Edge(){  
 to=0;  
 nx=0;  
 }  
 }  
 Edge[] edge;  
 int tot;//tot是边的编号  
 int[]head;  
 void init(int n){  
 this.n=n;  
 edge=new Edge[n<<1];  
 for (int i = 0; i < n << 1; i++) {  
 edge[i]= new Edge();  
 }  
 tot\_dfs=0;  
 head=new int[n+1];  
 sz=new int[n+1];son=new int[n+1];top=new int[n+1];dep=new int[n+1];Dfn=new int[n+1];Low=new int[n+1];  
 fa=new int[n+1];  
 }  
 void AddEdge(int a, int b){  
 tot++;  
 edge[tot].to=b;  
 edge[tot].nx=head[a];  
 head[a]=tot;  
 }//以a为起点的最后一条边没有nx指向，用的是head[a]去指向  
 int[] sz,son,top,dep,Dfn,Low;  
 int tot\_dfs;  
 int[] fa;  
 void dfs(int now){  
 Dfn[now]=++tot\_dfs;//Dfn是深度遍历的节点序号,也是其在线段树中的编号。  
 sz[now]=1;  
 son[now]=0;  
 for (int i = head[now]; i != 0 ; i=edge[i].nx) {  
 int nxt=edge[i].to;  
 if(nxt==fa[now])continue;//向上遍历了，跳过  
 fa[nxt]=now;//设置子节点的父亲为当前节点  
 dep[nxt]=dep[now]+1;//设置子节点的深度  
 dfs(nxt);//嵌套遍历子节点  
 sz[now]+=sz[nxt];//该点的大小：有多少子孙  
 if(sz[son[now]]<sz[nxt])son[now]=nxt;//最有出息的儿子（子树大小最大的儿子）  
 }  
 Low[now]=tot\_dfs;//以当前节点为根，深度遍历最后的序号  
 }  
 void dfs\_top(int now){  
 if(son[now]!=0){  
 top[son[now]]=top[now];  
 dfs\_top(son[now]);  
 }  
 for (int i = head[now]; i !=0 ; i=edge[i].nx) {  
 int nxt=edge[i].to;  
 if(nxt==fa[now]||nxt==son[now])continue;//不去遍历父亲和最有出息的儿子，最有出息的儿子已经在if语句中特判了  
 top[nxt]=nxt;  
 dfs\_top(nxt);  
 }  
 }  
  
 int LCA(int a, int b){//确定ab的最近公共父节点（LCA+dfs\_top两个函数共同作用）  
 while(top[a]!=top[b]){//如果//top[a]==top[b],说明它们在同一条直线路径上  
 if(dep[top[a]]<dep[top[b]])b=fa[top[b]];  
 else a=fa[top[a]];  
 }  
 return dep[a]<dep[b]?a:b;//返回的是ab的最近公共父节点  
 }  
 void bfs(int now){  
 ArrayDeque<Integer> arrayDeque=new ArrayDeque<>();  
 for (int i = head[now]; i != 0 ; i=edge[i].nx) {  
 int nxt=edge[i].to;  
 if(nxt==fa[now]) continue;//向上遍历了，跳过  
 arrayDeque.add(nxt);  
 }  
 for (int a: arrayDeque) {  
 bfs(a);  
 }  
 }  
 boolean In(int x,int y){  
 return Dfn[y]<=Dfn[x]&&Dfn[x]<=Low[y];//x节点在以y为根的子树中  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 Scanner scanner=new Scanner(System.*in*);  
 Tree tree=new Tree();  
 int n=scanner.nextInt();  
 tree.init(n);  
 int qi=0;  
 int zh=0;  
 int[] p=new int[n+1];  
 for (int i = 0; i < n; i++) {  
 int a=scanner.nextInt();  
 int b=scanner.nextInt();  
 if(p[a]==1&&p[b]==1){  
 qi=a;  
 zh=b;  
 break;  
 }  
 p[a]=1;  
 p[b]=1;  
 tree.AddEdge(a,b);  
 tree.AddEdge(b,a);  
 }  
 tree.dfs(1);  
 tree.top[1]=1;  
 tree.dfs\_top(1);  
 int c=tree.LCA(qi,zh);  
 int total=tree.dep[qi]-tree.dep[c]+tree.dep[zh]-tree.dep[c]+1;  
 System.*out*.println(total);  
 }  
}

### 并查集板子：

class UnionFind {  
 private final int[] parent;  
 private final int[] rank;  
 public UnionFind(int n) {  
 parent = new int[n];  
 rank = new int[n];  
 for (int i = 0; i < n; i++) {  
 parent[i] = i;  
 }  
 }  
  
 public int find(int x) {  
 if (parent[x] != x) {  
 parent[x] = find(parent[x]);  
 }  
 return parent[x];  
 }  
  
 public void union(int x, int y) {  
 int rootX = find(x);  
 int rootY = find(y);  
 if (rootX == rootY) {  
 return;  
 }  
 if (rank[rootX] < rank[rootY]) {  
 parent[rootX] = rootY;  
 } else if (rank[rootX] > rank[rootY]) {  
 parent[rootY] = rootX;  
 } else {  
 parent[rootY] = rootX;  
 rank[rootX]++;  
 }  
 }  
}  
  
public class Main{  
 public static void main(String[] args) {  
 UnionFind uf = new UnionFind(7);  
 uf.union(0, 1);  
 uf.union(1, 2);  
 uf.union(3, 4);  
 System.*out*.println(uf.find(2) == uf.find(0)); // true  
 System.*out*.println(uf.find(3) == uf.find(4)); // true  
 System.*out*.println(uf.find(2) == uf.find(4)); // false  
 }  
}

### ST表：（区间最大和最小值询问）

import java.util.Scanner;  
class ST{  
 int n;  
 int m;  
 int[][] st;  
 void init(int n,int m,int[][] st){  
 this.n=n;  
 this.m=m;  
 this.st=st;  
 }  
 public void prepare(){  
 // 初始化ST表  
 // 应该选循环j，再循环i  
 for(int j = 1; j < m; j++){  
 for(int i = 0; i < n; i++){  
 if((i + (1 << (j - 1))) < n){  
 st[i][j] = Math.*max*(st[i][j-1], st[i + (1 << (j - 1))][j-1]);  
 }else{  
 st[i][j] = st[i][j-1];  
 }  
 }  
 }  
 }  
 public int solve(int l, int r){//查询  
 int j = (int)(Math.*log*(r - l + 1) / Math.*log*(2));  
 return Math.*max*(st[l][j], st[r + 1 - (1 << (j))][j]);  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 ST stb=new ST();  
 int n = scanner.nextInt();  
 int M = scanner.nextInt();  
 int m = (int) Math.*ceil*(Math.*log*(n) / Math.*log*(2)) + 1;//ceil向上取整  
 int[][] st = new int[n][m];  
 for (int i = 0; i < n; i++) {  
 st[i][0] = scanner.nextInt();//st[i][j]是[i,i+2^j]的最大值  
 }  
 stb.init(n,m,st);  
 // 获花O(nlogn)时间预处理  
 stb.prepare();  
 for (int i = 0; i < M; i++) {  
 int l = scanner.nextInt();  
 int r = scanner.nextInt();  
 // O(1)时间回答果询  
 System.*out*.println(stb.solve(l - 1, r - 1));  
 }  
 }  
}

### 线段树区间加法模板：

import java.util.Scanner;  
class SegmentTree {  
int MAX=10000001;  
int[] sum;  
int[] a; //线段树维护的序列  
int[] lazy;  
void init(int[] a){  
 this.a=a;  
 sum =new int[MAX];  
 lazy=new int[MAX];  
}

//QJJ()区间修改，push表示对lazy标记的转移  
 void push\_up(int id) {  
 sum[id] = sum[id \* 2] + sum[id \* 2 + 1];  
 }  
 void push\_down(int id, int l, int r) {  
 if (lazy[id]!=0)//如果id有lazy标记  
 {  
 int mid = (l + r) / 2;  
 lazy[id \* 2] += lazy[id];//将它的左孩子的lazy加上它的lazy  
 lazy[id \* 2 + 1] += lazy[id];//将它的右孩子的lazy加上它的lazy  
 sum[id \* 2] += lazy[id] \* (mid - l + 1);//左孩子的Q+它下放的Q\*区间长度  
 sum[id \* 2 + 1] += lazy[id] \* (r - mid);  
 lazy[id] = 0;//清空lazy标记  
 }  
 }  
 void bui(int id, int l, int r) {  
 if (l == r)//叶子节点  
 {  
 sum[id] = a[l];  
 return;  
 }  
 int mid = (l + r) / 2;  
 bui(id \* 2, l, mid);//递归创建左子树  
 bui(id \* 2 + 1, mid + 1, r);//递归创建右子树  
 sum[id] = sum[id \* 2] + sum[id \* 2 + 1];//左子树和+右子树和  
 }  
 void QJJ(int id, int l, int r, int x, int y, int v)//id:目前查到的节点编号 目前区间为[l,r] 目标是将[x,y]的所有数+v  
 {  
 if (l >= x && r <= y)//[l,r]被[x,y]包含了  
 {  
 lazy[id] += v;//暂时不下放v,加进lazy标记中，不下放v但是值要变  
 sum[id] += v \* (r - l + 1);//将v收入囊中  
 return;//不下放，直接返回  
 }  
 push\_down(id, l, r);//要来更新下面节点了,赶紧下放v  
 int mid = (l + r) / 2;  
 if (x <= mid) QJJ(id \* 2, l, mid, x, y, v);//因为只有x<=mid(即[l,mid]有一部分是被[x,y]覆盖了的)才需要去更新[l,mid]  
 if (y > mid) QJJ(id \* 2 + 1, mid + 1, r, x, y, v);  
 push\_up(id);//子节点更新完之后父节点当然也要更新(上升操作)  
 }  
 int find(int id, int l, int r, int x, int y)//id:目前查到的节点编号 目前区间为[l,r] 目标是求出[x,y]的和  
 {  
 if (x <= l && r <= y) return sum[id];//[l,r]被[x,y]包含了  
 push\_down(id, l, r);//要查到id的子节点了,赶紧下放  
 int mid = (l + r) / 2, ans = 0;  
 if (x <= mid) ans += find(id \* 2, l, mid, x, y);//ans+=左孩子与[x,y]区间交区间和  
 if (y > mid) ans += find(id \* 2 + 1, mid + 1, r, x, y);//ans+=右孩子与[x,y]区间交区间和  
 return ans;  
 }  
}  
public class Main{  
 public static void main(String[] args) {  
 SegmentTree ST=new SegmentTree();  
 int n;  
 int m;  
 Scanner scanner=new Scanner(System.*in*);  
 n=scanner.nextInt();//n个点  
 m=scanner.nextInt();//m次操作，p==1修改，p==2查询  
 int[] a=new int[n+1];  
 for (int i = 1; i <= n; i++) a[i]=scanner.nextInt();  
 ST.init(a);  
 ST.bui(1, 1, n);  
 while (m-- != 0) {  
 int k, x, y, p;  
 p=scanner.nextInt();  
 x=scanner.nextInt();  
 y=scanner.nextInt();  
 if (p == 1) {  
 k=scanner.nextInt();  
 ST.QJJ(1, 1, n, x, y, k);//id:目前查到的节点编号 目前区间为[l,r] 目标是将[x,y]的所有数\*v  
 } else {  
 System.*out*.println(ST.find(1, 1, n, x, y));  
 }  
 }  
 }  
}

### 线段树区间乘法模板：

import java.util.Scanner;  
class SegmentTree {  
 int MAX=10000001;  
 int[] sum;  
 int[] a;//线段树维护的序列  
 int[] lazy;  
 void init(int[] a){  
 this.a=a;

sum =new int[MAX];  
 lazy=new int[MAX];  
 for (int i = 0; i < MAX; i++) {  
 lazy[i]=1;  
 }  
 }  
 //QJC()区间修改，push表示对lazy标记的转移  
 void push\_up(int id) {  
 sum[id] = sum[id \* 2] + sum[id \* 2 + 1];  
 }  
 void push\_down(int id) {  
 if (lazy[id]!=1)//如果id有lazy标记  
 {  
 lazy[id \* 2] \*= lazy[id];//将它的左孩子的lazy乘上它的lazy  
 lazy[id \* 2 + 1] \*= lazy[id];//将它的右孩子的lazy乘上它的lazy  
 sum[id \* 2] \*= lazy[id] ;//左孩子\*lazy  
 sum[id \* 2 + 1] \*= lazy[id];  
 lazy[id] = 1;//清空lazy标记  
 }  
 }  
 void bui(int id, int l, int r) {  
 if (l == r)//叶子节点  
 {  
 sum[id] = a[l];  
 return;  
 }  
 int mid = (l + r) / 2;  
 bui(id \* 2, l, mid);//递归创建左子树  
 bui(id \* 2 + 1, mid + 1, r);//递归创建右子树  
 sum[id] = sum[id \* 2] + sum[id \* 2 + 1];//左子树和+右子树和  
 }  
 void QJC(int id, int l, int r, int x, int y, int v)//id:目前查到的节点编号 目前区间为[l,r] 目标是将[x,y]的所有数\*v  
 {  
 if (l >= x && r <= y)//[l,r]被[x,y]包含了  
 {  
 lazy[id] \*= v;//暂时不下放v,加进lazy标记中，不下放v但是值要变  
 sum[id] = v \* sum[id];//将v收入囊中  
 return;//不下放，直接返回  
 }  
 push\_down(id);//要来更新下面节点了,赶紧下放v  
 int mid = (l + r) / 2;  
 if (x <= mid) QJC(id \* 2, l, mid, x, y, v);//因为只有x<=mid(即[l,mid]有一部分是被[x,y]覆盖了的)才需要去更新[l,mid]  
 if (y > mid) QJC(id \* 2 + 1, mid + 1, r, x, y, v);  
 push\_up(id);//子节点更新完之后父节点当然也要更新(上升操作)  
 }  
 int find(int id, int l, int r, int x, int y)//id:目前查到的节点编号 目前区间为[l,r] 目标是求出[x,y]的和  
 {  
 if (x <= l && r <= y) return sum[id];//[l,r]被[x,y]包含了  
 push\_down(id);//要查到id的子节点了,赶紧下放  
 int mid = (l + r) / 2, ans = 0;  
 if (x <= mid) ans += find(id \* 2, l, mid, x, y);//ans+=左孩子与[x,y]区间交区间和  
 if (y > mid) ans += find(id \* 2 + 1, mid + 1, r, x, y);//ans+=右孩子与[x,y]区间交区间和  
 return ans;  
 }  
  
}  
public class Main {  
 public static void main(String[] args) {  
 SegmentTree ST=new SegmentTree();  
 int n;  
 int m;  
 Scanner scanner=new Scanner(System.*in*);  
 n=scanner.nextInt();//n个点  
 m=scanner.nextInt();//m次操作，p==1修改，p==2查询  
 int[] a=new int[n+1];  
 for (int i = 1; i <= n; i++) a[i]=scanner.nextInt();  
 ST.init(a);  
 ST.bui(1, 1, n);  
 while (m-- != 0) {  
 int k, x, y, p;  
 p=scanner.nextInt();  
 x=scanner.nextInt();  
 y=scanner.nextInt();  
 if (p == 1) {  
 k=scanner.nextInt();  
 ST.QJC(1, 1, n, x, y, k);//id:目前查到的节点编号 目前区间为[l,r] 目标是将[x,y]的所有数\*v  
 } else {  
 System.*out*.println(ST.find(1, 1, n, x, y));  
 }  
 }  
 }  
}

### 线段树最值板子：

import java.util.Scanner;  
public class Main {  
 static int []*c*;  
 static Node []*tree*;  
 static class Node{  
 int l, r;  
 int min;  
 public Node(int l, int r, int min) {  
 this.l = l;  
 this.r = r;  
 this.min = min;//线段的最大值  
 }  
 }  
 public static void pushUp(int u) {  
 *tree*[u].min = Math.*min*(*tree*[u << 1].min, *tree*[u << 1 | 1].min);  
 }  
  
 public static void build(int u, int l, int r) {  
 if(l == r) *tree*[u] = new Node(l, r, *c*[l]);//创建叶子结点，结点中只有一个值时，最大值就是这个值  
 else {  
 *tree*[u] = new Node(l, r, Integer.*MIN\_VALUE*);//非叶子结点，最大值回溯时判断  
 int mid = l + r >> 1;//划分左右儿子  
 if(l <= mid) *build*(u << 1, l, mid);//递归创建左子树  
 if(r > mid) *build*(u << 1 | 1, mid + 1, r);  
 *pushUp*(u);//回溯，求最大值  
 }  
 }  
 public void change(int u,int id,int z){//把c[id]改为z  
 if(*tree*[u].l==*tree*[u].r) *tree*[u].min=z;  
 else{  
 int mid=(*tree*[u].l+*tree*[u].r)/2;  
 if(id<=mid) change(u\*2,id,z);  
 else change(u\*2+1,id,z);  
 *tree*[u].min=Math.*min*(*tree*[u\*2].min,*tree*[u\*2+1].min);  
 }  
 }  
  
 public static int query(int u, int l, int r) {  
 if(l <= *tree*[u].l && *tree*[u].r <= r) return *tree*[u].min;//结点中线段的区间被查询的区间完全覆盖，直接返回最大值  
 int mid = *tree*[u].l + *tree*[u].r >> 1;//该结点中线段（区间）的中点  
 int min = Integer.*MAX\_VALUE*;  
 if(l <= mid) min = Math.*min*(min, *query*(u << 1, l, r));//做子树中与带查询区间有交集  
 if(r > mid) min = Math.*min*(min, *query*(u << 1 | 1, l, r));  
 return min;  
 }  
  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 int T = scanner.nextInt();  
 for (int i = 0; i < T; i++) {  
 int n=scanner.nextInt();  
 int m=scanner.nextInt();  
 int[] k=new int[n];  
 *c*=new int[m+1];  
 for (int j = 0; j < n; j++) {  
 k[j]=scanner.nextInt();  
 }  
 for (int j = 1; j <= m; j++) {  
 *c*[j]=scanner.nextInt();  
 }  
 long ans=0;  
 *tree* = new Node[m \* 4];  
 *build*(1, 1, m);  
 for (int j = 0; j < n; j++) {  
 ans+=*query*(1,1,k[j]);  
 }  
  
 System.*out*.println(ans);  
 }  
 }  
}

### **莫队算法模板**：

import java.util.\*;  
public class Main{  
 final static int *maxn* = 1010000;  
 static int[] *aa*=new int[*maxn*], *cnt*=new int[*maxn*], *belong*=new int[*maxn*];//aa是序列，cnt是桶，记录区间某数的出现数量，belong[i]是询问i的块号  
 static int *n*, *m*, *size*, *bnum*;//n是序列长度，m是询问次数，size是分块大小（根号n）,bnum是块号（分块数量）  
 static long *curAns*;  
 static long[] *ans*;//询问答案  
 static query[] *queries*;//询问类  
 public static class query{  
 int l, r, id;//id是询问的编号  
 }  
 public static void main(String[] args) {//数询问区间不同数的个数  
 Scanner in = new Scanner(System.*in*);  
 *n* = in.nextInt(); // 序列长度  
 *size* = (int) Math.*sqrt*(*n*);  
 *bnum* = (int)Math.*ceil*((double)*n*/*size*); // block size 块数  
  
 for(int i = 1; i <= *n*; i++) *aa*[i] = in.nextInt(); // 整个序列读入  
 *m* = in.nextInt(); // 询问次数  
 *ans* = new long[*m*+1];  
 *queries* = new query[*m*+1];  
 for(int i = 1; i <= *m*; i++){  
 *queries*[i] = new query();  
 *queries*[i].l = in.nextInt();  
 *queries*[i].r = in.nextInt();  
 *queries*[i].id = i;  
 //System.out.println(queries[i].l + "-" + queries[i].r + "-" + queries[i].id);  
 }  
  
 for(int i = 1; i <= *bnum*; ++i) // 预处理序列索引分属的块  
 for(int j = (i - 1) \* *size* + 1; j <= i \* *size*; ++j) {  
 *belong*[j] = i;  
 }  
 // 对查询操作进行快速排序  
 // 但在此之上，我们还可以进行常数优化：奇偶化排序。意为：如果belong[l] 是奇数，则将r顺序排序，否则将r逆序排序。  
 Arrays.*sort*(*queries*,1,*m*+1,((a, b) ->  
 (*belong*[a.l] ^ *belong*[b.l])!=0 ? *belong*[a.l] - *belong*[b.l] : ((*belong*[a.l] & 1)!=0 ? a.r - b.r : b.r - a.r)));  
 //for(int i = 1; i <= m; i++)System.out.println(queries[i].l + "-" + queries[i].r + "-" + queries[i].id);  
  
 int l = 1, r = 0;  
 for(int i= 1; i <= *m*; i++){  
 int qr = *queries*[i].r, ql = *queries*[i].l;  
// while(l < ql) del(l++); // 如左指针在查询区间左方，左指针向右移直到与查询区间左端点重合  
// while(l > ql) add(--l); // 如左指针在查询区间左端点右方，左指针左移  
// while(r < qr) add(++r); // 右指针在查询区间右端点左方，右指针右移  
// while(r > qr) del(r--); // 否则左移  
 while(l < ql) *curAns* -= --*cnt*[*aa*[l++]]!=0 ? 0 : 1;  
 while(l > ql) *curAns* += *cnt*[*aa*[--l]]++==0 ? 1 : 0;  
 while(r < qr) *curAns* += *cnt*[*aa*[++r]]++==0 ? 1 : 0;  
 while(r > qr) *curAns* -= --*cnt*[*aa*[r--]]!=0 ? 0 : 1;  
 *ans*[*queries*[i].id] = *curAns*;  
 }  
 for(int i = 1; i <= *m*; i++)  
 System.*out*.println(*ans*[i]);  
 }  
}

### lowerbound模板：（小于等于有序数组中某数的下标）

public static int lowerBound(long []nums,int l,int r,long target) {

while (l < r) {

int m = (l + r) / 2;

if (nums[m] >= target) r = m;

else l = m + 1;

}

return l;

}

### upperbound模板：（大于等于有序数组中某数，若找不到返回数组最大值）

public static int upperBound(int []nums,int l,int r,long target) {  
 while(l<r){  
 int mid=(l+r)/2;  
 if(nums[mid]<target){l=mid+1;}  
 else{r=mid;}  
 }  
 // System.out.println(ar[l]);  
 return nums[l];  
}

### C(n,m):

//从n个球中取m个的组合种类

public static int C(int n,int m){

if(n==m)

return 1;

else

return JC(n)/(JC(m)\*JC(n-m));

}

//求阶乘

public static int JC(int num){

if (num==1){

return 1;

}else {

return JC(num-1)\*num;

}

}

### A(n,m):

// 求排列数 A(n,m) n>m

public static int A(int n, int m) {

int result = 1;

// 循环m次,如A(6,2)需要循环2次，6\*5

for (int i = m; i > 0; i--) {

result \*= n;

n--;// 下一次减一

}

return result;

}

### 快速乘（溢出取模时使用，化乘为加法）：

long mul(long a,long b,long mod)  
{  
 a=(a%mod+mod)%mod;  
 b=(b&mod+mod)%mod;  
 long res=0;  
 while(b!=0){  
 if((b&1)!=0){  
 res+=a;  
 if(res>=mod)  
 res-=mod;  
 }  
 b>>=1;  
 a<<=1;  
 if(a>=mod) a-=mod;  
 }  
 return res;  
}

### 快速幂与分数取模（费马小定理乘法逆元）：

import java.util.\*;  
//分数取模(a/b)%mod=a\*b^(mod-2)%mod，b^(mod-2)为b在%mod意义下的乘法逆元（b于mod互质）  
public class Main {  
 static int *mod*=998244353;  
 static long qmi(long x, long k)//快速幂  
 {  
 long res=1;  
 while(k!=0)  
 {  
 if((k&1)!=0) res= res \* x %*mod*;  
 k>>=1;  
 x = x \* x %*mod*;  
 }  
 return res;  
 }  
 public static void main(String[] args) {  
 Scanner scanner=new Scanner(System.*in*);  
 int N=1000006;  
 long[] f=new long[N];  
 long[] inf=new long[N];  
 f[0]=inf[0]=1;  
 for(int j=1; j<N; j++)  
 {  
 f[j]=f[j-1]\*j%*mod*;  
 inf[j]=inf[j-1]\**qmi*(j,*mod*-2)%*mod*;  
 }  
 int T=scanner.nextInt();  
 for (int i = 0; i < T; i++) {  
 int n = scanner.nextInt();  
 int m=scanner.nextInt();  
 int a=scanner.nextInt();  
 int b=scanner.nextInt();  
 long[] ji=new long[n+1];  
 long[] A=new long[n+1];  
 long[] A\_=new long[n+1];  
 long[] inB=new long[n+1];  
 inB[0]=1;  
 long \_b=*qmi*(b,*mod*-2);  
 A\_[0]=1;  
 for (int j = 1; j <= n; j++) {  
 A[j]=*qmi*(a,j);  
 A\_[j]=*qmi*(b-a,j);  
 inB[j] = inB[j - 1] \* \_b % *mod*;  
 }  
 for (int j = 1; j <= n; j++) {  
 ji[j]=ji[j-1]+*qmi*(j,m);  
 }  
 int total=0;  
 for (int j = 1; j <= n; j++) {  
 long sum;  
 sum=ji[j]%*mod*\*A[j]%*mod*\*A\_[n-j]%*mod*\*inB[n]%*mod*\*f[n]%*mod*\*inf[n-j]%*mod*\*inf[j]%*mod*;//C(n,m):f[n]%mod\*inf[n-j]%mod\*inf[j]%mod  
 total+=sum%*mod*;  
 total%=*mod*;  
 }  
 System.*out*.println(total);  
 }  
 }  
}

### Kmp模板：（字符串匹配问题，预处理模式串）

import java.util.Scanner;

public class kmp {

public static void main(String[] args) {

Scanner scanner=new Scanner(System.in);

String str1 =scanner.next();

String str2 = scanner.next();

System.out.println(kmpSearch(str1, str2, kmpNext(str2)));

}

//kmp搜索算法

/\*\*

\* @param str1 源字符串（主串）

\* @param str2 模式串

\* @param next 部分匹配表

\* @return 如果匹配到了返回第一个匹配的位置, 没有返回-1

\*/

public static int kmpSearch(String str1, String str2, int[] next) {

//遍历

for (int i = 0, j = 0; i < str1.length(); i++) {

//需要处理不相等的时候

while (j > 0 && str1.charAt(i) != str2.charAt(j)) {

j = next[j - 1];

}

if (str1.charAt(i) == str2.charAt(j)) {

j++;

}

if (j == str2.length()) {

return i - j + 1;

}

}

return -1;

}

//获取到一个字符串的部分匹配值

public static int[] kmpNext(String str) {

int[] next = new int[str.length()];

next[0] = 0; //当字符串的长度为1时

for (int i = 1, j = 0; i < str.length(); i++) {

while (j > 0 && str.charAt(i) != str.charAt(j)) {

j = next[j - 1];

}

if (str.charAt(i) == str.charAt(j)) {

j++;

}

next[i] = j;

}

return next;

}

}

### 狭义AC自动机：（每个模式串只匹配一次，返回匹配了多少个模式串）：

import java.util.\*;  
public class Main {  
 static class Tree//字典树  
 {  
 int fail;//失配指针  
 int[] vis=new int[26];//子节点的位置  
 int end;//标记有几个单词以这个节点结尾  
 }  
 static Tree[] *AC*=new Tree[1000001];//Trie树  
 static int *cnt*=0;//Trie的指针  
 static void Build(String s)  
 {  
 int l=s.length();  
 int now=0;//字典树的当前指针  
 for(int i=0;i<l;++i)//构造Trie树  
 {  
 if(*AC*[now].vis[s.charAt(i)-'a']==0)//Trie树没有这个子节点  
 *AC*[now].vis[s.charAt(i)-'a']=++*cnt*;//构造出来  
 now=*AC*[now].vis[s.charAt(i)-'a'];//向下构造  
 }  
 *AC*[now].end+=1;//标记单词结尾  
 }  
 static void Get\_fail()//构造fail指针  
 {  
 ArrayDeque<Integer> Q=new ArrayDeque<>();//队列  
 for(int i=0;i<26;++i)//第二层的fail指针提前处理一下  
 {  
 if(*AC*[0].vis[i]!=0)  
 {  
 *AC*[*AC*[0].vis[i]].fail=0;//指向根节点  
 Q.add(*AC*[0].vis[i]);//压入队列  
 }  
 }  
 while(!Q.isEmpty())//BFS求fail指针  
 {  
 int u=Q.getFirst();  
 Q.pop();  
 for(int i=0;i<26;++i)//枚举所有子节点  
 {  
 if(*AC*[u].vis[i]!=0)//存在这个子节点  
 {  
 *AC*[*AC*[u].vis[i]].fail=*AC*[*AC*[u].fail].vis[i];  
 //子节点的fail指针指向当前节点的  
 //fail指针所指向的节点的相同子节点  
 Q.add(*AC*[u].vis[i]);//压入队列  
 }  
 else//不存在这个子节点  
 *AC*[u].vis[i]=*AC*[*AC*[u].fail].vis[i];  
 //当前节点的这个子节点指向当  
 //前节点fail指针的这个子节点  
 }  
 }  
 }  
 static int AC\_Query(String s)//AC自动机匹配  
 {  
 int l=s.length();  
 int now=0,ans=0;  
 for(int i=0;i<l;++i)  
 {  
 now=*AC*[now].vis[s.charAt(i)-'a'];//向下一层  
 for(int t=now;t!=0&&*AC*[t].end!=-1;t=*AC*[t].fail)//循环求解  
 {  
 ans+=*AC*[t].end;  
 *AC*[t].end=-1;  
 }  
 }  
 return ans;  
 }  
 public static void main(String[] args) {  
 Scanner scanner=new Scanner(System.*in*);  
 int n=scanner.nextInt();  
 String s;  
 for (int i = 0; i < 1000001; i++) {  
 *AC*[i]=new Tree();  
 }  
 for(int i=1;i<=n;++i)  
 {  
 s=scanner.next();  
 *Build*(s);  
 }  
 *AC*[0].fail=0;//结束标志  
 *Get\_fail*();//求出失配指针  
 s=scanner.next();//文本串  
 System.*out*.println(*AC\_Query*(s));  
 }  
}

### AC自动机：（多模式串匹配问题）

//AC自动机，说白了就是在trie树上跑kmp。  
// 是一种多匹配串，单个主串的匹配。  
// 概括来说，就是将多个匹配串构造一个trie树，对于每个trie树的节点构造nxt指针，最后把主串放在上面跑。  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
import java.util.Collection;  
import java.util.Set;  
import java.util.Collections;  
import java.util.Queue;  
import java.util.LinkedList;  
public class Main {  
 private Boolean failureStatesConstructed = false; //是否建立了failure表  
 private Node root; //根结点  
 public Main() {  
 this.root = new Node(true);  
 }  
 private static class Node{  
 private final Map<Character, Node> map; //用于放这个Node的所有子节点，储存形式是：Map(char, Node)  
 private final List<String> PattenStrings; //该节点处包含的所有pattern string  
 private Node failure; //fail指针指向的node  
 private Boolean isRoot = false; //是否为根结点  
 public Node(){  
 map = new HashMap<>();  
 PattenStrings = new ArrayList<>();  
 }  
 public Node(Boolean isRoot) {  
 this();  
 this.isRoot = isRoot;  
 }  
 //用于build trie，如果一个字符character存在于子节点中，不做任何操作，返回这个节点的node  
 //否则，建一个node，并将map(char,node)添加到当前节点的子节点里，并返回这个node  
 public Node insert(Character character) {  
 Node node = this.map.get(character);  
 if (node == null) {  
 node = new Node();  
 map.put(character, node);  
 }  
 return node;  
 }  
 public void addPattenString(String keyword) {  
 PattenStrings.add(keyword);  
 }  
 public void addPattenString(Collection<String> keywords) {  
 PattenStrings.addAll(keywords);  
 }  
 public Node find(Character character) {  
 return map.get(character);  
 }  
 */\*\*  
 \* 利用父节点fail node来寻找子节点的fail node  
 \* or  
 \* parseText时找下一个匹配的node  
 \*/* private Node nextState(Character transition) {  
 //用于构建fail node时，这里的this是父节点的fail node  
 //首先从父节点的fail node的子节点里找有没有值和当前失败节点的char值相同的  
 Node state = this.find(transition);  
  
 //如果找到了这样的节点，那么该节点就是当前失败位置节点的fail node  
 if (state != null) {  
 return state;  
 }  
 //如果没有找到这样的节点，而父节点的fail node又是root，那么返回root作为当前失败位置节点的fail node  
 if (this.isRoot) {  
 return this;  
 }  
 //如果上述两种情况都不满足，那么就对父节点的fail node的fail node再重复上述过程，直到找到为止  
 //这个地方借鉴了KMP算法里面求解next列表的思想  
 return this.failure.nextState(transition);  
 }  
 public Collection<Node> children() {  
 return this.map.values();  
 }  
 public void setFailure(Node node) {  
 failure = node;  
 }  
 public Node getFailure() {  
 return failure;  
 }  
 //返回一个Node的所有子节点的键值，也就是这个子节点上储存的char  
 public Set<Character> getTransitions() {  
 return map.keySet();  
 }  
 public Collection<String> PattenString() {  
 return this.PattenStrings == null ? Collections.<String>*emptyList*() : this.PattenStrings;  
 }  
 }  
 private static class Patten\_String{  
 private final String keyword; //匹配到的模式串  
 private final int start; //起点  
 private final int end; //终点  
 public Patten\_String(final int start, final int end, final String keyword) {  
 this.start = start;  
 this.end = end;  
 this.keyword = keyword;  
 }  
 public String getKeyword() {  
 return this.keyword;  
 }  
 @Override  
 public String toString() {  
 return super.toString() + "=" + this.keyword;  
 }  
 }  
 */\*\*  
 \* 添加一个模式串(内部使用字典树构建)  
 \*/* public void addKeyword(String keyword) {  
 if (keyword == null || keyword.length() == 0) {  
 return;  
 }  
 Node currentState = this.root;  
 for (Character character : keyword.toCharArray()) {  
 //如果char已经在子节点里，返回这个节点的node；否则建一个node，并将map(char,node)加到子节点里去  
 currentState = currentState.insert(character);  
 }  
 //在每一个尾节点处，将从root到尾节点的整个string添加到这个叶节点的PattenString里  
 currentState.addPattenString(keyword);  
 }  
 */\*\*  
 \* 用ac自动机做匹配，返回text里包含的pattern及其在text里的起始位置  
 \*/* public Collection<Patten\_String> parseText(String text) {  
 //首先构建 fail表，如已构建则跳过  
 checkForConstructedFailureStates();  
 Node currentState = this.root;  
 List<Patten\_String> collectedPattenStrings = new ArrayList<>();  
 for (int position = 0; position < text.length(); position++) {  
 Character character = text.charAt(position);  
 //依次从子节点里找char，如果子节点没找到，就到子节点的fail node找，并返回最后找到的node；如果找不到就会返回root  
 //这一步同时也在更新currentState，如果找到了就更新currentState为找到的node，没找到currentState就更新为root，相当于又从头开始找  
 currentState = currentState.nextState(character);  
 Collection<String> PattenStrings = currentState.PattenString();  
 if (PattenStrings == null || PattenStrings.isEmpty()) {  
 continue;  
 }  
 //如果找到的node的PattenString非空，说明有pattern被匹配到了  
 for (String PattenString : PattenStrings) {  
 collectedPattenStrings.add(new Patten\_String(position - PattenString.length() + 1, position, PattenString));  
 }  
 }  
 return collectedPattenStrings;//返回匹配到的所有pattern  
 }  
 */\*\*  
 \* 建立Fail表(核心,BFS遍历)  
 \*/* private void constructFailureStates() {  
 Queue<Node> queue = new LinkedList<>();  
 //首先从把root的子节点的fail node全设为root  
 //然后将root的所有子节点加到queue里面  
 for (Node depthOneState : this.root.children()) {  
 depthOneState.setFailure(this.root);  
 queue.add(depthOneState);  
 }  
 this.failureStatesConstructed = true;  
 while (!queue.isEmpty()) {  
 Node parentNode = queue.poll();  
 //下面给parentNode的所有子节点找fail node  
 for (Character transition : parentNode.getTransitions()) { //transition是父节点的子节点的char  
 Node childNode = parentNode.find(transition); //childNode是子节点中对应上面char值的节点的Node值  
 queue.add(childNode); //将这个parentNode的所有子节点加入queue，在parentNode的所有兄弟节点都过了一遍之后，就会过这些再下一层的节点  
 Node failNode = parentNode.getFailure().nextState(transition); //利用父节点的fail node来构建子节点的fail node  
 childNode.setFailure(failNode);  
 //每个节点处的PattenString要加上它的fail node处的PattenString  
 //因为能匹配到这个位置的话，那么fail node处的PattenString一定是匹配的pattern  
 childNode.addPattenString(failNode.PattenString());  
 }  
 }  
 }  
 */\*\*  
 \* 检查是否建立了Fail表(若没建立，则建立)  
 \*/* private void checkForConstructedFailureStates() {  
 if (!this.failureStatesConstructed) {  
 constructFailureStates();  
 }  
 }  
 public static void main(String[] args) {  
 Main trie = new Main();  
 trie.addKeyword("he");  
 trie.addKeyword("she");  
 trie.addKeyword("his");  
 trie.addKeyword("hers");  
 //匹配text，并返回匹配到的pattern  
 Collection<Patten\_String> pattenStrings = trie.parseText("ushers");  
 for (Patten\_String pattenString : pattenStrings) {  
 System.*out*.println(pattenString.start + " " + pattenString.end + " " + pattenString.getKeyword());  
 }  
 }  
}

### 后缀自动机（SAM）板子（例：LCS问题（最长公共子串））：

import java.util.\*;  
class NODE{  
 int[] ch=new int[26];// 每个节点添加一个字符后到达的节点  
 int len;// 每个节点代表的状态中的最大串长  
 int fa;// 每个节点在parent树上的父亲节点, 即该节点link边指向的节点  
 int sz; // 每个节点对应的endpos集合的大小（即串的出现次数），等于所有parent树上儿子的大小  
}  
class SAM {  
 int MAXN=1000007;//字符串长度  
 NODE[] dian=new NODE[MAXN<<1];  
 int las;// las: 上一个用到的节点编号  
 int tot;// tot: 当前用到的节点编号  
 int[] b; // b: 用于基数排序  
 int[] a; //a: 用于记录点号,a的值就是parent树的拓扑序  
 String s;// 串  
 int len;  
 // 向SAM中插入一个字符c  
 public void add(int c)  
 {  
 int p=las;// 上一个状态的节点  
 int np=las=++tot;// 要加入的状态的节点  
 dian[np].sz=1; // 叶子节点endpos大小为1  
 dian[np].len=dian[p].len+1;// 新状态比上一个装填多一个首字符  
 for(;p!=0&&dian[p].ch[c]==0;p=dian[p].fa)dian[p].ch[c]=np;// 指针p沿link边回跳,直至找到一个节点包含字符c的出边，无字符c的出边则将出边指向新状态的节点  
 if(p == 0)dian[np].fa=1;//以上为case 1，指针p到SAM的起点的路径上的节点都无字符c的出边，将新节点作为SAM的起点的一个儿子节点  
 else // 节点p包含字符c的出边  
 {  
 int q=dian[p].ch[c];// 节点p的字符c的出边指向的节点  
 if(dian[q].len==dian[p].len+1)dian[np].fa=q;//以上为case,2节点p和q代表的状态的最大串长相差1  
 else// 节点p和q代表的状态的最大串长相差>1  
 {// 节点p和q代表的状态的最大串长相差>1  
 int nq=++tot; // 新建一个节点nq  
 // 节点nq克隆节点q的信息  
 dian[nq].ch=dian[q].ch.clone();  
 dian[nq].len=dian[q].len;  
 dian[nq].fa=dian[q].fa;  
 dian[nq].sz=0; // nq产生时，是一个分支节点，需要从后续儿子节点里更新获取sz  
 dian[nq].len=dian[p].len+1; // 保证节点p和nq代表的状态的最大串长相差1  
 dian[q].fa=dian[np].fa=nq;  
 for(;p!=0&&dian[p].ch[c]==q;p=dian[p].fa)dian[p].ch[c]=nq;// 以上为case 3，将节点p到SAM的起点的路径上的所有节点的字符c的出边指向的节点替换为nq  
 }  
 }  
 }  
 // 按长度基数排序，短的在前长的在后  
 // 另一种方法是用vector直接建出parent树，对parent树直接dfs，不过可能会超内存  
 void base\_sort(){  
 for(int i=1;i<=tot;++i)b[dian[i].len]++;  
 for(int i=1;i<=tot;++i)b[i]+=b[i-1];  
 for(int i=1;i<=tot;++i)a[b[dian[i].len]--]=i;  
 }  
 // 用于逆拓扑序遍历求出sz  
 void get\_sz(){  
 for(int i=tot;i>=1;--i){  
 int u=a[i],fa=dian[u].fa;  
 dian[fa].sz+=dian[u].sz;  
 }  
 }  
 void init(String s){  
 this.s=s;  
 las=1;  
 tot=1;  
 a=new int[MAXN<<1];  
 b=new int[MAXN<<1];  
 len=s.length();  
 for (int i = 0; i < MAXN<<1; i++) {  
 dian[i]=new NODE();  
 }  
 for(int i=0;i<len;i++)add(s.charAt(i)-'a');//初始化  
// base\_sort();  
// get\_sz();  
 }  
}  
public class Main {  
 public static void main(String[] args) {//解决LCS问题  
 Scanner scanner=new Scanner(System.*in*);  
 String s=scanner.next();  
 SAM sam=new SAM();  
 sam.init(s);  
 int p = 1, alen = 0;  
 String b=scanner.next();  
 int lenb=b.length();  
 int ans=0;  
 for(int i = 0; i < lenb; ++i){  
 if(sam.dian[p].ch[b.charAt(i) - 'a'] != 0) {  
 ++alen;  
 p = sam.dian[p].ch[b.charAt(i) - 'a'];  
 }  
 else{  
 while((sam.dian[p].ch[b.charAt(i) - 'a']==0) && (p!=0)) {  
 p = sam.dian[p].fa;  
 }  
 if(p==0) {p = 1; alen = 0;}  
 else {  
 alen = sam.dian[p].len + 1;  
 p = sam.dian[p].ch[b.charAt(i) - 'a'];  
 }  
 }  
 ans = Math.*max*(ans, alen);  
 }  
 System.*out*.println(ans);  
 }  
}

### 后缀自动机2：（以边建立真正的parent树，然后DFS确定sz，可能会爆栈，因为加边可能还会爆内存）

import java.util.\*;  
public class Main {  
 static int *MAXN*=1000001;//字符串长度  
 static Scanner *scanner*=new Scanner(System.*in*);  
 static int *las*;// las: 上一个用到的节点编号  
 static int *tot*;// tot: 当前用到的节点编号  
 static class NODE  
 {  
 int[] ch=new int[26];// 每个节点添加一个字符后到达的节点  
 int len;// 每个节点代表的状态中的最大串长(以该点结尾的前缀串长)  
 int fa;// 每个节点在parent树上的父亲节点, 即该节点link边指向的节点  
 long sz; // 每个节点对应的endpos集合的大小（即串的出现次数），等于所有parent树上儿子的大小  
 }  
 static NODE[] *dian*;  
 static class Edge{  
 int t,next;  
 }  
 static Edge[] *edge*;  
 static int[] *head*;  
 static int *cnt*;  
 static void jia(int a, int b)  
 {  
 *cnt*++;  
 *edge*[*cnt*]=new Edge();  
 *edge*[*cnt*].t=b;  
 *edge*[*cnt*].next=*head*[a];  
 *head*[a]=*cnt*;  
 }  
 // 向SAM中插入一个字符c  
 public static void add(int c)  
 {  
 int p=*las*;// 上一个状态的节点  
 int np=*las*=++*tot*;// 要加入的状态的节点  
 *dian*[np].sz=1; // 叶子节点endpos大小为1  
 *dian*[np].len=*dian*[p].len+1;// 新状态比上一个装填多一个首字符  
 for(;p!=0&&*dian*[p].ch[c]==0;p=*dian*[p].fa)*dian*[p].ch[c]=np;// 指针p沿link边回跳,直至找到一个节点包含字符c的出边，无字符c的出边则将出边指向新状态的节点  
 if(p == 0)*dian*[np].fa=1;//以上为case 1，指针p到SAM的起点的路径上的节点都无字符c的出边，将新节点作为SAM的起点的一个儿子节点  
 else // 节点p包含字符c的出边  
 {  
 int q=*dian*[p].ch[c];// 节点p的字符c的出边指向的节点  
 if(*dian*[q].len==*dian*[p].len+1)*dian*[np].fa=q;//以上为case,2节点p和q代表的状态的最大串长相差1  
 else// 节点p和q代表的状态的最大串长相差>1  
 {// 节点p和q代表的状态的最大串长相差>1  
 int nq=++*tot*; // 新建一个节点nq  
 *dian*[nq].ch = *dian*[q].ch.clone();  
 *dian*[nq].len = *dian*[q].len;  
 *dian*[nq].fa = *dian*[q].fa;// 节点nq克隆节点q的信息  
 *dian*[nq].sz=0; // nq产生时，是一个分支节点，需要从后续儿子节点里更新获取sz  
 *dian*[nq].len=*dian*[p].len+1; // 保证节点p和nq代表的状态的最大串长相差1  
 *dian*[q].fa=nq;  
 *dian*[np].fa=nq;  
 for(;p!=0&&*dian*[p].ch[c]==q;p=*dian*[p].fa)*dian*[p].ch[c]=nq;// 以上为case 3，将节点p到SAM的起点的路径上的所有节点的字符c的出边指向的节点替换为nq  
 }  
 }  
 }  
 static String *s*;// 串  
 static int *len*;  
 static long *ans*;  
 static void dfs(int node)  
 {  
 for(int i=*head*[node];i!=0;i=*edge*[i].next)  
 {  
 *dfs*(*edge*[i].t);  
 *dian*[node].sz+=*dian*[*edge*[i].t].sz;  
 }  
 if(*dian*[node].sz!=1)*ans*=Math.*max*(*ans*,*dian*[node].sz\**dian*[node].len);  
 }  
 static void init(){  
 *s*= *scanner*.next();  
 *las*=1;  
 *tot*=1;  
 *cnt*=0;  
 *ans*=0;  
 *len*=*s*.length();  
 *head*=new int[*len*<<1];  
 *dian*=new NODE[*len*<<1];  
 *edge*=new Edge[*MAXN*<<1];  
 for (int i = 0; i < *len*<<1; i++)*dian*[i]=new NODE();  
 for(int i=0;i<*len*;i++)*add*(*s*.charAt(i)-'a');//初始化  
 for(int i=2;i<=*tot*;i++)*jia*(*dian*[i].fa,i);  
 *dfs*(1);  
 }  
 public static void main(String[] args) {  
 *init*();  
 System.*out*.println(*ans*);  
 }  
}

### 广义后缀自动机（exsam:多个串中本质不同的字串）：

import java.util.\*;  
class Trie {  
 int M=1000001;  
 int tot=1;  
 int[] col = new int[M];  
 int[] fa= new int[M];  
 int[][] ch= new int[M][26];  
 void insert(char[] s) {  
 int now = 1;  
 for (char value : s) {  
 int c = value - 'a';  
 if (ch[now][c] == 0) {  
 ch[now][c] = ++tot;  
 fa[tot] = now;  
 col[tot] = c;  
 }  
 now = ch[now][c];  
 }  
 }  
}  
class SuffixAuto {  
 int N=2000001;  
 int tot= 1;  
 int[] pos = new int[N];  
 int[] fa = new int[N];  
 int[] len= new int[N];  
 // int[] b=new int[N]; // b: 用于基数排序  
// int[] a=new int[N]; //a: 用于记录点号,a的值就是parent树的拓扑序  
// int[] sz=new int[N];  
 int[][] ch= new int[N][26];  
 Queue<Integer> Q= new LinkedList<>();  
 int insert(int c, int last) {  
 int p = last;  
 int q = ++tot;  
// sz[q]=1;  
 len[q] = len[p] + 1;  
 while (p != 0 && ch[p][c] == 0) {  
 ch[p][c] = q;  
 p = fa[p];  
 }  
 if (p == 0) {  
 fa[q] = 1;  
 } else {  
 int np = ch[p][c];  
 if (len[p] + 1 == len[np]) {  
 fa[q] = np;  
 } else {  
 int nq = ++tot;  
 len[nq] = len[p] + 1;  
 System.*arraycopy*(ch[np], 0, ch[nq], 0, 26);  
 while (p != 0 && ch[p][c] == np) {  
 ch[p][c] = nq;  
 p = fa[p];  
 }  
 fa[nq] = fa[np];  
 fa[q] = fa[np] = nq;  
// sz[nq]=0;  
 }  
 }  
 return q;  
 }  
 void build(Trie T1) {  
 for (int i = 0; i < 26; i++) {  
 if (T1.ch[1][i] != 0) {  
 Q.add(T1.ch[1][i]);  
 }  
 }  
 pos[1] = 1;  
 while (!Q.isEmpty()) {  
 int x = Q.poll();  
 pos[x] = insert(T1.col[x], pos[T1.fa[x]]);  
 for (int i = 0; i < 26; i++) {  
 if (T1.ch[x][i] != 0) {  
 Q.add(T1.ch[x][i]);  
 }  
 }  
 }  
// base\_sort();  
// get\_sz();  
 }  
 void query() {  
 long ans = 0;  
 for (int i = 2; i <= tot; i++) {  
 ans += len[i] - len[fa[i]];  
 }  
 System.*out*.println(ans);  
 }  
// void base\_sort(){  
// for(int i=1;i<=tot;++i)b[len[i]]++;  
// for(int i=1;i<=tot;++i)b[i]+=b[i-1];  
// for(int i=1;i<=tot;++i)a[b[len[i]--]]=i;  
// }  
// // 用于逆拓扑序遍历求出sz  
// void get\_sz(){  
// for(int i=tot;i>=1;--i){  
// int u=a[i],fat=fa[u];  
// sz[fat]+=sz[u];  
// }  
// }  
}  
public class Main {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 int n = scanner.nextInt();  
 Trie T1 = new Trie();  
 for (int i = 0; i < n; i++) {  
 String s = scanner.next();  
 T1.insert(s.toCharArray());  
 }  
 SuffixAuto SAM = new SuffixAuto();  
 SAM.build(T1);  
 SAM.query();  
 }  
}

### 回文自动机（PAM：O(n)解决字符串有多少不同回文字串的问题）：

import java.io.\*;  
public class Main {  
 public static BufferedReader *inString* = new BufferedReader(new InputStreamReader(System.*in*));  
 public static PrintWriter *out* = new PrintWriter(new OutputStreamWriter(System.*out*));  
 public static String next() throws IOException{ return *inString*.readLine(); }  
 static char[] *s*;  
 static int[] *len*,*num*,*fail*;  
 static int[][] *trie*;  
 static int *n*,*last*,*cur*,*pos*,*tot*;  
 static void init(){  
 *tot*=1;//tot-1为不同回文串数量  
 *len*=new int[*n*+2];  
 *num*=new int[*n*+2];  
 *fail*=new int[*n*+2];  
 *trie*=new int[*n*+2][26];  
 }  
 static int getFail(int x, int i)  
 {  
 while(i-*len*[x]-1<0||*s*[i-*len*[x]-1]!=*s*[i])x=*fail*[x];  
 return x;  
 }  
 public static void main(String[] args) throws IOException {  
 String ss=*next*();  
 *n*=ss.length();  
 *s*=ss.toCharArray();  
 *init*();  
 *fail*[0]=1;*len*[1]=-1;  
 for(int i=0;i<=*n*-1;i++){  
// if(i>=1)s[i]= (char) ((s[i]+last-97)%26+97);  
 *pos*=*getFail*(*cur*,i);  
 //找到cur的fail链中能匹配i位的最长回文后缀  
 if(*trie*[*pos*][*s*[i]-'a'] == 0){  
 *fail*[++*tot*]=*trie*[*getFail*(*fail*[*pos*],i)][*s*[i]-'a'];  
 *trie*[*pos*][*s*[i]-'a']=*tot*;  
 *len*[*tot*]=*len*[*pos*]+2;  
 *num*[*tot*]=*num*[*fail*[*tot*]]+1;  
 }//不存在建立点，存在直接走过去  
 *cur*=*trie*[*pos*][*s*[i]-'a'];  
 *last*=*num*[*cur*];  
 *out*.print(*last*+" ");  
 *out*.flush();//一定要加  
 }  
 }  
}

### Manacher/马拉车（计算最长回文子串）：

public String longestPalindrome(String s) {

// manacher算法

// 首先给字符串增加从来没有出现过的字符（这里为#号）

StringBuilder ss = new StringBuilder();

ss.append('#');

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

ss.append(s.charAt(i)).append('#');

}

// p[i]表示以i为中心的最长回文子串的回文半径

int[] p = new int[ss.length()];

// maxx表示当前已经计算过的最右边界坐标,id表示当前已经计算过的最右边界回文串的回文中心

int maxx = -1, id = 0;

// 开始计算p数组

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

// r表示回文子串的最小半径

int r = 1;

// 如果当前坐标在计算过的范围内

if(i <= maxx) {

// 取当前点距最右边界的距离和当前点的对称点的回文半径的最小值

r = Math.min(p[id] - i + id, p[2 \* id - i]);

}

// 暴力判断更后面的地方

while ((i - r) >= 0 && (i + r) < ss.length() && ss.charAt(i - r) == ss.charAt(i + r)) {

r++;

}

// 更新回文边界

if(i + r - 1 > maxx) {

maxx = i + r - 1;

id = i;

}

// 记录当前回文半径

p[i] = r;

}

int maxr = 0, pos = 0;

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

if(p[i] > maxr) {

maxr = p[i];

pos = i;

}

}

// maxr-1为最长回文子串长度

// pos是其回文中心

//下面是计算起点

pos = (pos - maxr + 1) / 2;

return s.substring(pos, pos + maxr - 1);

}

### 欧拉函数：

int euler(int n){  
 int ans=1;  
 for(int i=2;i\*i<=n;i++){  
 if(n%i==0){  
 n/=i;  
 ans\*=i-1;  
 while(n%i==0){  
 n/=i;  
 ans\*=i;  
 }  
 }  
 }  
 if(n>1){  
 ans\*=n-1;  
 }  
 return ans;  
}

### 欧拉线性筛：确定一定范围的欧拉函数值

import java.io.\*;  
import java.util.Scanner;  
public class Main {  
 public static void main(String[] args) throws IOException {  
 *phi\_table*(50);  
 for (int i = 0; i <= 50; i++) {  
 System.*out*.println(*phi*[i]); }  
 }  
 static int *phi*[]=new int[1000000];//phi[i]的值是i的欧拉函数  
 static void phi\_table(int n){  
 *phi*[1]=1;  
 for(int i=2;i<=n;i++){  
 if(*phi*[i]==0){  
 for(int j=i;j<=n;j+=i){  
 if(*phi*[j]==0){  
 *phi*[j]=j;  
 }  
 *phi*[j]=*phi*[j]/i\*(i-1);  
 }  
 }  
 }  
 }  
}

### 欧拉筛：（找区间素数）

import java.io.\*;  
public class Main {  
 static int *MAXN* = 10000005;  
 static int[] *prime*=new int[*MAXN*];//素数集  
 static boolean[] *pp*=new boolean[*MAXN*];  
 static int[] *vis*=new int[*MAXN*];  
 static void Prime()  
 { int cnt=0;  
 for(int i=2;i<=*MAXN*-5;i++)  
 {  
 if(*vis*[i]==0) {*prime*[cnt++]=i;*pp*[i]=true;}  
 for(int j=0;j<cnt&&i\**prime*[j]<=*MAXN*-5;j++)  
 {  
 *vis*[i\**prime*[j]]=i;  
 if(i%*prime*[j]==0) break;  
 }  
 }  
 }  
 public static void main(String[] args) throws IOException {  
 *Prime*();  
 for (int i = 0; i <= 50; i++) {  
 System.*out*.println(*prime*[i]+","+*pp*[i]+","+*vis*[i]);  
 }  
 }  
}

### 莫比乌斯函数（O(n^(1/2))）：

long mu(long n){  
 if(n==1) return 1;  
 int prime = 1;  
 int flag = 0;  
 for (int i = 2; (long) i \*i <= n; i++) {  
 if (n%i == 0) {  
 prime++;  
 n /= i;  
 if (n%i == 0) {  
 flag = 1;  
 break;  
 }  
 }  
 }  
 if (flag!=0)  
 return 0;  
 if (prime % 2 != 0)return -1;  
 else return 1;  
}

### 莫比乌斯函数线性筛：

import java.io.\*;  
public class Main {  
 static int *N*=5000;  
 static int[] *mu*=new int[*N*+1];//莫比乌斯函数值  
 static int[] *flg*=new int[*N*+1];//质数标记，0为质数  
 static int[] *p*=new int[*N*+1];//质数  
 static int *tot*=0;  
 static void getMu() {  
 *mu*[1] = 1;  
 for (int i = 2; i <= *N*; ++i) {  
 if (*flg*[i]==0) {*p*[++*tot*] = i; *mu*[i] = -1;}  
 for (int j = 1; j <= *tot* && i \* *p*[j] <= *N*; ++j) {  
 *flg*[i \* *p*[j]] = 1;//该质数的i倍记为合数  
 if (i % *p*[j] == 0) {//说明i中已经有p[j]这个因子，i\*p[j]不满足有不同的质因子，mu[i\*p[j]]=0  
 *mu*[i \* *p*[j]] = 0;  
 break;  
 }  
 *mu*[i \* *p*[j]] = -*mu*[i];  
 }  
 }  
 }  
 public static void main(String[] args) throws IOException {  
 *getMu*();  
 for (int i = 0; i <= *N*; i++) {  
 System.*out*.println(*mu*[i]+","+*p*[i]+","+*flg*[i]);  
 }  
 }  
}

### 矩阵相关模板类：

import java.util.Scanner;  
public class Main {  
 /\*设计的矩阵类包含以下功能:（不只是加法功能和转置功能）  
 \* 1.三种普通构造方法以及一种复制构造方法  
 \* 2.矩阵的加法、减法、数乘、求负以及矩阵乘法的运算  
 \* 3.矩阵的转置  
 \* 4.矩阵的三种判断：矩阵是否同型、矩阵是否相等以及矩阵是否为方阵  
 \* 5.计算矩阵的迹、矩阵中所有元素的和以及矩阵的乘方  
 \* 6.获取矩阵的行数和列数  
 \* 7.输出矩阵相关的所有信息  
 \* 8.获取行列数确定的单位矩阵的静态方法  
 \*/  
 public static class Basematrix  
 {  
 private double[][] matrix;//用二维数组存储矩阵，数据成员声明为私有保证封装性和安全性  
 private int column,row;//分别声明矩阵的行数和列数  
  
 //声明了三种普通构造方法和一种复制构造方法  
 public Basematrix() {}//默认构造方法，只是为了方便创建一个矩阵对象  
 public Basematrix(int column,int row)//以行数和列数为参数的构造方法，确定矩阵的类型但是还不填充元素  
 {  
 this.column=column;  
 this.row=row;  
 this.matrix=new double[column][row];//此处容易出错，尽管只给出行数和列数，但是也应该对作为引用类型的数据成员matrix进行创建  
 }  
 public Basematrix(double [][]matrix)//以一个二维数组作为参数的构造方法，由二维数组的相关性质便可以确定矩阵的行数和列数  
 {  
 this.column=matrix.length;  
 this.row=matrix[0].length;  
 this.matrix=new double[column][row];  
 for(int i=0;i<column;i++)  
 {  
 for(int j=0;j<row;j++)  
 this.matrix[i][j]=matrix[i][j];  
 }  
 }  
 public Basematrix(Basematrix Matrix)//复制构造函数  
 {  
 this.column=Matrix.column;  
 this.row=Matrix.row;  
 this.matrix=new double[column][row];//不直接使用this.matrix=Matrix.matrix是为了实现深复制  
 for(int i=0;i<column;i++)  
 {  
 for(int j=0;j<row;j++)  
 this.matrix[i][j]=matrix[i][j];  
 }  
 }  
  
 //声明了矩阵的加法、减法、数乘和乘法运算  
 public Basematrix add(Basematrix Matrix)//矩阵的加法运算  
 {  
 Basematrix temp=new Basematrix(column,row);//两个矩阵是同型矩阵时，对应位置的元素相加  
 for(int i=0;i<column;i++)  
 {  
 for(int j=0;j<row;j++)  
 {  
 temp.matrix[i][j]=this.matrix[i][j]+Matrix.matrix[i][j];  
 }  
 }  
 return temp;  
 }  
 public Basematrix substract(Basematrix Matrix)//矩阵的减法运算，与加法运算类似  
 {  
 if(!this.SameKind(Matrix))  
 {  
 System.*out*.println("You can't substract two kinds of matrix");  
 return (new Basematrix(1,1));  
 }  
 Basematrix temp=new Basematrix(column,row);  
 for(int i=0;i<column;i++)  
 {  
 for(int j=0;j<row;j++)  
 temp.matrix[i][j]=this.matrix[i][j]-Matrix.matrix[i][j];  
 }  
 return temp;  
 }  
 public Basematrix multiple(double n)//矩阵的数乘运算，每个元素分别数乘即可  
 {  
 Basematrix temp=new Basematrix(this.matrix);  
 for(int i=0;i<column;i++)  
 {  
 for(int j=0;j<row;j++)  
 temp.matrix[i][j]\*=n;  
 }  
 return temp;  
 }  
 public Basematrix multiple(Basematrix Matrix)//矩阵的乘法运算，是矩阵运算中较为复杂的运算，使用了方法重载  
 {  
 if(this.row!=Matrix.column)//判断两个矩阵是否能进行乘法运算  
 {  
 System.*out*.println("You can't mutiple these two kinds of matrixs");  
 return (new Basematrix(1,1));  
 }  
 Basematrix temp=new Basematrix(this.column,Matrix.row);//能进行乘法运算时，执行下列语句  
 for(int i=0;i<temp.column;i++)  
 {  
 for(int j=0;j<temp.row;j++)  
 {  
 double sum=0;  
 for(int k=0;k<this.row;k++)  
 sum+=(this.matrix[i][k]\*Matrix.matrix[k][j]);  
 temp.matrix[i][j]=sum;//通过上述算法计算每一个元素的值  
 }  
 }  
 return temp;  
 }  
 //声明了求负矩阵的方法  
 public Basematrix negative()  
 {  
 Basematrix temp=new Basematrix(this.matrix);  
 for(int i=0;i<column;i++)  
 {  
 for(int j=0;j<row;j++)  
 temp.matrix[i][j]=-this.matrix[i][j];  
 }  
 return temp;  
 }  
  
 //声明了矩阵的转置方法  
 public Basematrix reverse()  
 {  
 Basematrix temp=new Basematrix(row,column);//新建一个行数与列数与原矩阵相反的矩阵  
 for(int i=0;i<column;i++)  
 {  
 for(int j=0;j<row;j++)  
 temp.matrix[j][i]=this.matrix[i][j];//填充新矩阵中的元素  
 }  
 return temp;  
 }  
  
 //声明了判断矩阵同型、矩阵相等和是否为方阵的方法，返回值均为boolean类型  
 public boolean SameKind(Basematrix Matrix)//判断两个矩阵是否同型  
 {  
 if(this.column==Matrix.column&&this.row==Matrix.row)  
 return true;  
 else  
 return false;  
 }  
 public boolean equal(Basematrix Matrix)//判断两个矩阵是否相等  
 {  
 if(!this.SameKind(Matrix))//矩阵相等的前提条件是两个矩阵同型  
 return false;  
 for(int i=0;i<column;i++)  
 {  
 for(int j=0;j<row;j++)  
 if(this.matrix[i][j]!=Matrix.matrix[i][j])  
 return false;  
 }  
 return true;  
 }  
 public boolean phalanx()//判断矩阵是否为方阵  
 {  
 if(column==row)  
 return true;  
 else  
 return false;  
 }  
  
 //声明了计算矩阵的迹和所有元素之和的方法  
 public double trail()//计算矩阵的迹  
 {  
 if(!phalanx())//矩阵有迹的前提是矩阵是一个方阵  
 {  
 System.*out*.println("Not a phalanx!");  
 return 0;  
 }  
 else//对角线上的元素相加  
 {  
 double Trail=0;  
 for(int i=0;i<column;i++)  
 Trail+=this.matrix[i][i];  
 return Trail;  
 }  
 }  
 public double sum()//计算矩阵中所有元素的和  
 {  
 double Sum=0;  
 for(int i=0;i<column;i++)  
 {  
 for(int j=0;j<row;j++)  
 Sum+=this.matrix[i][j];  
 }  
 return Sum;  
 }  
 public Basematrix pow(int n)//计算矩阵的乘方  
 {  
 if(!this.phalanx())//首先判断矩阵是否是方阵  
 {  
 System.*out*.println("You can't calculate the pow of a non-phalanx!");  
 return (new Basematrix(1,1));  
 }  
 Basematrix temp=new Basematrix(this);  
 Basematrix base=new Basematrix(this);  
 n--;  
 while(n!=0) {  
 if((n&1)!=0){  
 temp=temp.multiple(base);  
 }  
 base = base.multiple(base);  
 n>>=1;  
 }  
 return temp;  
 }  
 //获取行列数确定的单位矩阵的方法，由于该方法不属于类中的某一个对象，因此设置为静态方法  
 public static Basematrix unit(int n)  
 {  
 Basematrix unit=new Basematrix(n,n);  
 for(int i=0;i<unit.column;i++)  
 unit.matrix[i][i]=1;  
 return unit;  
 }  
 //声明了矩阵的两个对外公有接口，分别获取行数和列数  
 public int getColumn() {return column;}  
 public int getRow() {return row;}  
 //声明了获取矩阵所有信息并输出的方法  
 public void show()  
 {  
 //首先输出行数和列数  
 System.*out*.println("This is a matrix with "+column+" columns and "+row+" rows.");  
 System.*out*.println("The elements of the matrix:");  
 for(int i=0;i<column;i++)  
 {  
 for(int j=0;j<row;j++)  
 System.*out*.printf("%5.2f", matrix[i][j]);//以格式化方法输出矩阵中每一个元素的值  
 System.*out*.println(" ");  
 }  
 }  
 }  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 double[][] b=new double[2][2];  
 b[0][0]=1;  
 b[0][1]=1;  
 b[1][0]=1;  
 b[1][1]=0;  
 Basematrix basematrix=new Basematrix(b);  
 basematrix=basematrix.pow(2);  
 basematrix.show();  
 }  
}

### 数论分块：

for(int l=1,r;l<=n;l=r+1){//已知n求解[n/i]（取整）的前n项和  
 r=n/(n/l);  
 ans+=(n/l)\*(r-l+1);  
}

### 欧几里得：

int gcd(int a,int b){  
 if(b==0) return a;  
 return gcd(b,a%b);  
}

### 拓展欧几里得：（求解ax+by=gcd(a,b)的解）

static int *x*, *y*;//求解a\*x+b\*y=gcd(a,b)的一组解(x0,y0)，其它解为(x0+k\*b/gcd,y0-k\*a/gcd);  
public static int exgcd(int a, int b){

if(b == 0){  
 *x* = 1;  
 *y* = 0;  
 return a;  
 }  
 int d = *exgcd*(b, a%b);  
 int t = *y*;  
 *y* = *x* - a/b\**y*;  
 *x* = t;  
 return d;  
}

//最后x=(x%n+n)%n;(防止x是负数)

//乘法逆元是(a\*d)%mod==1%mod则称d是a的逆元，则a\*d-k\*mod=1，设d=x,k=-y,则a的逆元是a\*x+mod\*y=1的x，  
//由于a,b互质的充分必要条件是存在整数x,y使ax+by=1，所以逆元存在则gcd(a,b)=1(特判)  
约瑟夫问题求解：

static public int findTheWinner(int n, int k) {  
 int pos = 0;  
 for (int i = 2; i < n + 1; ++i) {  
 pos = (pos + k) % i;  
 }  
 return pos + 1;  
}

### SG函数递归写法：

class SG{//sg全部异或，为0必败，为1必胜，初始为0  
 int N=10000007;  
 int M=10000007;  
 int[] sg=new int[N];//初始化为-1  
 int[] lim=new int[M];//限取的石子数  
 int n,m;//石子数，可取石子集合大小  
 int mex(int pos){  
 if(sg[pos]!=-1) return sg[pos];  
 boolean[] flag=new boolean[M];  
 int i;  
 for (i = 0; i < m; i++) {  
 if(pos-lim[i]<0) continue;  
 flag[mex(pos-lim[i])]=true;  
 }  
 for(i=0;flag[i];i++);  
 return sg[pos]=i;  
 }  
}

### SG函数非递归写法：

class SG{//sg全部异或，为0必败，为1必胜，初始为0  
 int N=10000007;  
 int M=10000007;  
 int[] sg=new int[N];//初始化为-1  
 int[] lim=new int[M];//限取的石子数  
 int[] flag=new int[M];//初始化为-1  
 int i,j;  
 void init(){  
 for (i = 0; i < N; i++) {  
 for (j = 0; j < M; j++) {  
 if (i - lim[j] < 0) continue;  
 flag[sg[i - lim[j]]] = i;//当前标记  
 }  
 for (j = 0; flag[j] == i; j++) ;  
 sg[i] = j;  
 }  
 }  
}