DJ算法：主要是这个小根堆的写法

struct compare{

    bool operator()(const pair<int,int>&a,const pair<int,int>&b){

        return a.second>b.second;

    }

};

priority\_queue<pair<int,int>,vector<pair<int,int>>,compare>heap;

void dijkstra(){

    for(int i=0;i<=n;i++){

        dis[i]=INF;

    }

    dis[0]=0;

    heap.push({0,0});

    while(!heap.empty()){

        int u=heap.top().first;

        heap.pop();

        if(vis[u]){

            continue;

        }

        vis[u]=1;

        if(u==2\*n+5){

            break;

        }

        for(int i=head[u];i;i=nxt[i]){

            int v=to[i];

            int w=weight[i];

            if(dis[v]>dis[u]+w){

                dis[v]=dis[u]+w;

                heap.push({v,dis[v]});

            }

        }

    }

}

Floyd算法：其实本质上也就是个动态规划

// Floyd算法模版（洛谷）

// 测试链接 : https://www.luogu.com.cn/problem/P2910

int n,m,ans;

int dis[MAXN][MAXN];

int path[MAXM];

void build() {

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            dis[i][j] = INT\_MAX;

        }

    }

}

void floyd() {

    // O(N^3)的过程

    // 枚举每个跳板

    // 注意，跳板要最先枚举！跳板要最先枚举！跳板要最先枚举！

    for (int bridge = 0; bridge < n; bridge++) { // 跳板

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                // i -> .....bridge .... -> j

                // distance[i][j]能不能缩短

                // distance[i][j] = min ( distance[i][j] , distance[i][bridge] + distance[bridge][j])

                if (dis[i][bridge] != INT\_MAX

                        && dis[bridge][j] != INT\_MAX

                        && dis[i][j] > dis[i][bridge] + dis[bridge][j]) {

                    dis[i][j] = dis[i][bridge] + dis[bridge][j];

                }

            }

        }

    }

}

SPFA算法：可以处理负边权，也可以检测负权环，稀疏图

（n个点，那么最多n-1次；注意有时我们会加入一个源点，此时就是n次）

// 源点出发到每个节点的距离表

int dis[MAXN];

// 节点被松弛的次数

int updateCnt[MAXN];

bool enter[MAXN];//当前在队列中的话  就是true  否则就是false

queue<int >q; //每一轮都弹出最后面的一个  如果一个点可以被优化  那么加入队列

bool spfa() {

    dis[1] = 0;

    updateCnt[1]++;

    q.push(1) ;

    enter[1] = true;

    //这道题目中的源点都是1 保持不变   所以以上的操作都针对1

    //实际上应该理解为是针对源点的

    while (!q.empty()) {

        int u = q.front();

        q.pop();

        enter[u] = false;

        for (int ei = head[u], v, w; ei > 0; ei = Next[ei]) {

            v = to[ei];

            w = weight[ei];

            if (dis[u] + w < dis[v]) {

                //如果值变小了  那么就要改变

                dis[v] = dis[u] + w;

                if (!enter[v]) {

                    //如果不在队列中才会加入

                    // 松弛次数超过n-1就有负环

                    if (++updateCnt[v] > n - 1) {

                        return true;

                    }

                    q.push(v);

                    enter[v] = true;

                }

            }

        }

    }

    return false;

}

Tarjan——缩点

vector<int>vec[MAXN];

//dfn表示的就是dfn序  而low数组表示的是这个节点沿着边可以走到的最小的dfn序

//dfn一旦确定了 就不会改变了  而low数组可能会发生变化

int dfn[MAXN],low[MAXN];

int st[MAXN],top;

int color,Time,col[MAXN];

int in[MAXN],out[MAXN];

bool instack[MAXN];

int n,m;

void tarjan(int x){

    dfn[x]=++Time;

    low[x]=Time;

    st[++top]=x;

    instack[x]=true;

    for(int i=0;i<vec[x].size();i++){

        int v=vec[x][i];

        if(!dfn[v]){

            //表示这个节点没有被访问过

            tarjan(v);

            low[x]=min(low[x],low[v]);

        } else if(instack[v]){

            //表示这个属于是回溯了 一定是同一个环上的

            //这里之所以不用low[v],是因为这里的含义就是dfn序号,其实就算改成low也不影响

            //不过这里最好写成这样 因为其他的要求可能会导致写成low出错

            low[x]=min(low[x],dfn[v]);

        }

    }

    if(dfn[x]==low[x]){

        col[x]=++color;

        //将所有节点按照颜色分类  完成缩点

        while(st[top]!=x){

            //属于同一个强联通分量

            col[st[top]]=color;

            instack[st[top]]=false;

            top--;

        }

        instack[x]=false;

        top--;

    }

}

Tarjan——割点

// 我们判断割点的方法是：

//1.如果这个点是根节点  那么如果他有两个儿子  则他一定是割点

// 这里的两个儿子指的是在深度优先搜索的条件下的两个儿子 即这两个儿子的子树一定是互不相干的

// 否则其中一个儿子的子树一定会包括另外一个儿子

//2.如果这个节点不是根节点，并且不是叶子结点 并且low[他的任意一个孩子]>=dfn[他]则它是割点

// 如果是叶节点  那么不可能是割点

// 并且他的所有孩子不能到达它的上面  即他扼死了孩子向上的通道

vector<int>vec[MAXN];

//在割点的时候  dfn还是最早的序号 可是low却不再是这个节点可以到达的最小编号了

//和缩点的时候不同

int dfn[MAXN],low[MAXN];

int Time,n,m;

bool cut[MAXN],instack[MAXN];

// v:当前点 r：本次搜索树的root

void tarjan(int u,int r) {

    dfn[u] = low[u] = ++Time;

    int child = 0;

    for (int i = 0; i < vec[u].size(); i++) {

        int v = vec[u][i];

        if (!dfn[v]) {

            tarjan(v, r);

            low[u] = min(low[u], low[v]);

            if (low[v] >= dfn[u] && u != r)

                cut[u] = true;//不是根而且他的孩子无法跨越他回到祖先

            if (r == u){

                child++; //如果是搜索树的根，统计孩子数目

                //其实所有的节点都可以统计孩子的数量 但是非根节点统计后没有什么作用

            }

        }

        low[u] = min(low[u], dfn[v]);//这里要特别注意 不能变成low[v]

        // 举个例子 比如a-b-c-d-b  并且还存在b-e-a

        // 那么首先b是一个割点  可是如果改成low[v] 那么low[d]=dfn[a]  导致b无法成为一个割点

    }

    if (child >= 2 && u == r){

        cut[r] = true;//对应第一种情况

    }

}

Tarjan——桥

// 桥其实和tarjan没有什么本质上的区别  都是一样的

// 桥的判断方法是对于一条边  如果下面的点的low大于上面点的dfn  说明是桥

// 如果这条边是回溯的  那不算

int n,m;

vector<pair<int,int>>bridges;

int dfn[MAXN],low[MAXN];

int dfncnt;

void tarjan(int u,int id){

    //表示当前来到了点u  是通过真实编号id这条边来的

    dfn[u]=low[u]=++dfncnt;

    for(int i=head[u];i;i=nxt[i]){

        int v=to[i];

        int w=weight[i];

        if(!dfn[v]){

            tarjan(v,w);

            low[u]=min(low[u],low[v]);

            if(low[v]>dfn[u]){

                bridges.push\_back({u,v});

            }

        }

        else if(w!=id){

            //不能走回头路

            low[u]=min(low[u],dfn[v]);

        }

    }

}

拓扑排序

void topo(){

    for(int i=1;i<=n;i++){

        if(in[i]==0){

            q.push(i);

        }

    }

    while(!q.empty()){

        int u=q.front();

        q.pop();

        for(int i=head1[u];i;i=Next1[i]){

            int v=to1[i];

            in[v]--;

            len[v]=max(len[v],len[u]+weight1[i]);

            if(in[v]==0){

                q.push(v);

            }

        }

    }

}

扩展欧几里得（普通的欧几里得就是gcd）

// 同余方程

// 求关于x的同余方程 ax ≡ 1(mod b) 的最小正整数解

// 题目保证一定有解，也就是a和b互质

// 2 <= a、b <= 2 \* 10^9

// 测试链接 : https://www.luogu.com.cn/problem/P1082

#include<bits/stdc++.h>

using namespace std;

long long d, x, y, px, py;

void exgcd(long long a, long long b) {

    if (b == 0) {

        //表示已经求出了最大公约数d      x y分别是1 0

        d = a;//最大公约数设置为a

        x = 1;

        y = 0;

    } else {

        exgcd(b, a % b);

        //这个过程在向上的时候逐步更新

        //d=x'\*a'+y'\*b'

        // =x'\*b+y'\*(a-(a/b)\*b)

        // =y'\*a+(x'-(a/b)\*y')b

        // =x\*a+y\*b

        //x=y'   y=x'-(a/b)\*y'

        px = x;

        py = y;

        x = py;

        y = px - py \* (a / b);

    }

}

int main()

{

    long long a,b;

    cin>>a>>b;

    exgcd(a,b);

    cout<<((x % b + b) % b);

    return 0;

}

求格点公式：gcd(abs(x1 - x2), abs(y1 - y2)) + 1

中国剩余定理（CRT）、扩展中国剩余定理（exCRT）

// 给出n个同余方程，求满足同余方程的最小正数解x

// 一共n个同余方程，x ≡ ri(% mi) 1 <= n <= 10 0 <= ri、mi <= 10^5

// 所有mi一定互质 所有mi整体乘积 <= 10^18

long long m[MAXN],r[MAXN];

long long d, x, y, px, py;

long long multiply(long long a, long long b, long long mod) {

    // 既然是在%mod的意义下，那么a和b可以都转化成非负的

    // 本题不转化无所谓，但是其他题目可能需要转化

    // 尤其是b需要转化，否则while循环会跑不完

    a = (a % mod + mod) % mod;

    b = (b % mod + mod) % mod;

    long long ans = 0;

    while (b != 0) {

        if ((b & 1) != 0) {

            ans = (ans + a) % mod;

        }

        a = (a + a) % mod;

        b >>= 1;

    }

    return ans;

}

void exgcd(long long a, long long b) {

    if (b == 0) {

        d = a;

        x = 1;

        y = 0;

    } else {

        exgcd(b, a % b);

        px = x;

        py = y;

        x = py;

        y = px - py \* (a / b);

    }

}

// 中国剩余定理模版  只适用于模数互质的情况

long long crt(int n) {

    long long lcm = 1;

    for (int i = 1; i <= n; i++) {

        lcm = lcm \* m[i];

        //因为m是互质的  所以最小公倍数就是他们的累乘

    }

    long long ai, ci, ans = 0;

    for (int i = 1; i <= n; i++) {

        // ai = lcm / m[i]

        // 即 m1 \* m2 \*```\* mi-1 \* mi+1 \*```\* mn-1 \* mn

        ai = lcm / m[i];

        // ai逆元，在%m[i]意义下的逆元

        exgcd(ai, m[i]);

        // ci = (ri \*   ai \* ai逆元(即x)  ) % lcm

        //这里之所以要%lcm  是因为我们要求的是最小的符合要求的正整数  所有符合要求的可以表示为特解加上任意个lcm

        ci = multiply(r[i], multiply(ai, x, lcm), lcm);

        ans = (ans + ci) % lcm;

    }

    return ans;

}

// 扩展中国剩余定理模版 可以用于解决模数不是互质的情况  适用范围更广

long long excrt(int n) {

    long long tail = 0, lcm = 1, tmp, b, c, x0;

    // ans = lcm \* x + tail

    for (int i = 1; i <= n; i++) {

        // ans = m[i] \* y + ri

        // lcm \* x + m[i] \* y = ri - tail

        // a = lcm b = m[i] c = ri - tail

        b = m[i];

        c = ((r[i] - tail) % b + b) % b;

        exgcd(lcm, b);

        if (c % d != 0) {

            return -1;

        }

        // ax + by = gcd(a,b)，特解是，x变量

        // ax + by = c，特解是，x变量 \* (c/d)

        // ax + by = c，最小非负特解x0 = (x \* (c/d)) % (b/d) 取非负余数

        //特解是x0 通解 = x0 + (b/d) \* n

        x0 = multiply(x, c / d, b / d);

        // ans = lcm \* x + tail，带入通解

        // ans = lcm \* (x0 + (b/d) \* n) + tail

        // ans = lcm \* (b/d) \* n + lcm \* x0 + tail

        // tail' = tail' % lcm'

        tmp = lcm \* (b / d);

        tail = (tail + multiply(x0, lcm, tmp)) % tmp;

        lcm = tmp;

    }

    return tail;

}

质数筛

//埃氏筛 欧拉筛 埃氏筛很容易理解欧拉筛的扩展功能很强  可以记录每个数字的最小质因子

bool visit[MAXN];//用于埃氏筛统计是否是质数

int prime[MAXN];//欧拉筛中用于收集质数的数组

int num[MAXN];//记录每个数字的最小质因子

int ehrlich(int n) {

    // 初始时认为0~n所有数都是质数，但0和1不是质数

    for(int i=2;i\*i<=n;i++){

        if(!visit[i]){

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

                visit[j]=true;

            }

        }

    }

    int cnt=0;

    for(int i=2;i<=n;i++){

        if(!visit[i]){

            cnt++;

        }

    }

    return cnt;

}

int euler(int n){

    int cnt=0;

    for(int i=2;i<=n;i++){

        if(num[i]==0){

            prime[++cnt]=i;

        }

        //无论是不是质数  都要进行下面的过程

        for(int j=1;j<=cnt&&i\*prime[j]<=n;j++){

            num[i\*prime[j]]=prime[j];

            if(i%prime[j]==0){

                //如果i可以整除prime[j]说明一定含有这个质因子

                //那么如果继续的话  就是将接下来的数字的按照下一个质数作为他的最小质因子排除的 而不是被最小质因子排除的  所以不能继续  要立即跳出

                break; // 每个合数只被其最小的质因数筛去一次

            }

        }

    }

    return cnt;

}

逆元

//这个文件中以下三种逆元： 单个除数的逆元 连续数字的逆元 连续数字阶乘逆元

int n,m,mod;

int inv[MAXN];//连续数字的逆元

int fac[MAXN];//连续数字的阶乘

int facinv[MAXN];//连续数字阶乘的逆元

long long power(long long a,long long b,long long mod){

    long long ans=1;

    while(b){

        if(b&1){

            ans\*=a;

            ans%=mod;

        }

        b>>=1;a\*=a;a%=mod;

    }

    return ans;

}

//单个除数的逆元 费马小定理求(a/b)%mod的值

long long compute(long long a,long long b,long long mod){

    long long inv=power(b,mod-2,mod);//求b在%mod的情况下的逆元

    return (a\*inv)%mod;

}

//连续数字的逆元 预计算模逆元

void build(int n) {

    inv[1] = 1;

    for (int i = 2; i <= n; i++) {

        inv[i]=(mod - (long long)inv[mod % i] \* (mod / i) % mod + mod) % mod;

    }

}

//连续数字阶乘的逆元 初始化阶乘表和逆元表

void build() {

    //先求出阶乘表

    fac[0] = 1; // 0! = 1

    for (int i = 1; i <= n; i++) {

        fac[i] = (fac[i - 1] \* i) % mod;

    }

    // 利用线性递推优化计算逆元

    facinv[n] = power(fac[n], mod - 2,mod);//求出最后一个数字的阶乘逆元

    for (int i = n - 1; i >= 0; i--) {

        facinv[i] = (facinv[i + 1] \* (i + 1)) % mod;//线性递推

    }

}

差分（二维、等差数列）、二维前缀和

// 一开始1~n范围上的数字都是0，一共有m个操作，每次操作为(l,r,s,e,d)

// 表示在l~r范围上依次加上首项为s、末项为e、公差为d的数列

//等差数列差分需要两遍前缀和

int n,m; long long l,s,r,e,maxans,eor;

long long arr[10000005];

void Set(long long l,long long r,long long s,long long e,long long d){

    //以上参数分别表示  左起点  右终点  初始增加值  末尾增加值  公差

    arr[l]+=s;

    arr[l+1]+=(d-s);

    arr[r+1]-=(d+e);

    arr[r+2]+=e;

}

void build(){

    for(int i=1;i<=n;i++){

        arr[i]+=arr[i-1];

    }

    for(int i=1;i<=n;i++){

        arr[i]+=arr[i-1];

    }

}

二维差分

void Set(int a,int b,int c,int d,int v){

    nums[a][b]+=v;

    nums[a][d+1]-=v;

    nums[c+1][b]-=v;

    nums[c+1][d+1]+=v;

}

void build(){

    for(int i=1;i<=n;i++){

        for(int j=1;j<=n;j++){

            nums[i][j]+=nums[i-1][j]+nums[i][j-1]-nums[i-1][j-1];

        }

    }

}

二维前缀和

int fun(int X1,int Y1,int X2,int Y2){

    return sum[X2][Y2]+sum[X1-1][Y1-1]-sum[X2][Y1-1]-sum[X1-1][Y2];

}

博弈论（sg函数）

int sg[MAXN]; // 每个位置的 SG 值

bool appear[MAXV]; // 用于计算 SG 值的辅助数组

// 预处理 SG 值

void build() {

    for (int i = 1; i < MAXN; i++) {

        memset(appear, false, sizeof(appear)); // 重置 appear 数组

        for (int j = i - 1; j >= 0; j--) {

            for (int k = j; k >= 0; k--) {

                appear[sg[j] ^ sg[k]] = true; // 标记 SG 值的组合

            }

        }

        for (int s = 0; s < MAXV; s++) {

            if (!appear[s]) {

                sg[i] = s; // 找到最小的未出现的 SG 值

                break;

            }

        }

    }

}

倍增ST表、树上倍增（更常用）

// ST表查询最大值和最小值 给定一个长度为n的数组arr，一共有m次查询

// 每次查询arr[l~r]上的最大值和最小值

const int MAXN = 70005;

const int LIMIT = 16;

int stmax[MAXN][LIMIT],stmin[MAXN][LIMIT];

int arr[MAXN],Log2[MAXN],n,m;

void build(int n){

    Log2[0]=-1;

    for(int i=1;i<=n;i++){

        Log2[i]=Log2[i>>1]+1;

        stmax[i][0]=arr[i];

        stmin[i][0]=arr[i];

    }

    for(int i=1;i<=Log2[n];i++){

        for(int j=1;j<=n;j++){

            stmax[j][i]=max(stmax[j][i-1],stmax[j+(1<<(i-1))][i-1]);

            stmin[j][i]=min(stmin[j][i-1],stmin[j+(1<<(i-1))][i-1]);

        }

    }

}

int query(int l,int r){

    int p=Log2[r-l+1];

    int maxans=max(stmax[l][p],stmax[r-(1<<p)+1][p]);

    int minans=min(stmin[l][p],stmin[r-(1<<p)+1][p]);

    return maxans-minans;

}

树上倍增

void dfs(int u, int f) {

    deep[u] = deep[f] + 1;

    stjump[u][0] = f;

    for (int p = 1; p <= log\_2[n]; p++) {

        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];

    }

    //完成u的deep  stjump

    for (int e = head[u]; e != 0; e = Next[e]) {

        if (to[e] != f) {

            dfs(to[e], u); //向下递归

        }

    }

}

带权并查集

//寻找i节点的父亲 扁平化处理  并在该过程中更新距离

int find(int i) {

    if (i != fa[i]) {

        int tmp = fa[i];

        fa[i] = find(tmp);

        dist[i] += dist[tmp];//将距离更新正确

    }

    return fa[i];

}

void un(int l, int r, long long v) {

    int lf = find(l), rf = find(r);

    if (lf != rf) {

        fa[lf] = rf;//强制要求后面的节点作为父亲

        dist[lf] = v + dist[r] - dist[l];

    }

}

//加减关系：

//寻找i节点的父亲 扁平化处理  并在该过程中更新距离

int find(int i) {

    if (i != fa[i]) {

        int tmp = fa[i];

        fa[i] = find(tmp);

        dist[i] += dist[tmp];//将距离更新正确

    }

    return fa[i];

}

void un(int l, int r) {

    int lf = find(l), rf = find(r);

    if (lf != rf) {

        fa[lf] = rf;//强制要求后面的节点作为父亲

        dist[lf] += sz[rf];

        sz[rf] += sz[lf];

    }

}

//种类关系

int find(int i) {

    if (i != fa[i]) {

        int tmp = fa[i];

        fa[i] = find(tmp);

        dist[i] = (dist[i] + dist[tmp]) % 3;

        //更新关系比较特殊

    }

    return fa[i];

}

// op == 1, 1 l r，l和r是同类

// op == 2, 2 l r，l吃r

void un(int op, int l, int r) {

    int lf = find(l), rf = find(r), v = op == 1 ? 0 : 1;//判断二者的关系

    if (lf != rf) {

        fa[lf] = rf;

        dist[lf] = (dist[r] - dist[l] + v + 3) % 3;

    }

}

//异或关系

int find(int i){

    if(i!=fa[i]){

        int tmp=fa[i];

        fa[i]=find(tmp);

        dis[i]^=dis[tmp];

    }

    return fa[i];

}

可撤销并查集

//利用数组完成栈的功能

int rollback[MAXN][2];

int opsize = 0;//表示栈的大小

void undo() {

    //撤销一步操作

    int fx = rollback[opsize][0];

    int fy = rollback[opsize--][1];

    father[fy] = fy;

    siz[fx] -= siz[fy];

    edgeCnt[fx] -= edgeCnt[fy] + 1;

}

矩阵乘法

vector<vector<int>> multiply(vector<vector<int>>& a,const vector<vector<int>>& b) {

    int n = a.size();

    int m = b[0].size();

    int k = a[0].size();

    vector<vector<int>> ans(n, vector<int>(m, 0));

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < m; j++) {

            for (int c = 0; c < k; c++) {

                ans[i][j] += a[i][c] \* b[c][j];

                ans[i][j]=((ans[i][j]%MOD)+MOD)%MOD;

            }

        }

    }

    return ans;

}

// 矩阵快速幂

vector<vector<int>> power(vector<vector<int>>& m, int p) {

    int n = m.size();

    vector<vector<int>> ans(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++) {

        ans[i][i] = 1;//单位矩阵 相当于乘法快速幂中的1

    }

    for (; p != 0; p >>= 1) {

        if ((p & 1) != 0) {

            ans = multiply(ans, m);

        }

        m = multiply(m, m);//每次都是倍增

    }

    return ans;

}

高精度

// 移除前导零

void removeLeadingZeros(vector<int>& num) {

    while (num.size() > 1 && num.back() == 0) {

        num.pop\_back();

    }

}

// 比较两个大整数的大小，返回1表示a>b，0表示a==b，-1表示a<b

int compare(const vector<int>& a, const vector<int>& b) {

    if (a.size() != b.size()) {

        return a.size() > b.size() ? 1 : -1;

    }

    for (int i = a.size() - 1; i >= 0; --i) {

        if (a[i] != b[i]) {

            return a[i] > b[i] ? 1 : -1;

        }

    }

    return 0;

}

// 大整数加法

vector<int> add(const vector<int>& a, const vector<int>& b) {

    vector<int> res;

    int carry = 0;

    for (int i = 0; i < (int)a.size() || i <(int) b.size() || carry; ++i) {

        if (i < (int)a.size()) carry += a[i];

        if (i < (int)b.size()) carry += b[i];

        res.push\_back(carry % 10);

        carry /= 10;

    }

    return res;

}

// 大整数减法 (假设a >= b)

vector<int> sub(const vector<int>& a, const vector<int>& b) {

    vector<int> res;

    int carry = 0;

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

        carry = a[i] - carry;

        if (i < (int)b.size()) carry -= b[i];

        res.push\_back((carry + 10) % 10);

        carry = carry < 0 ? 1 : 0;//如果carry<0  说明要向上借位

    }

    removeLeadingZeros(res);

    return res;

}

// 大整数乘以小整数

vector<int> mul(const vector<int>& a, int b) {

    vector<int> res;

    int carry = 0;

    for (int i = 0; i <(int) a.size() || carry; ++i) {

        if (i <(int64\_t) a.size()) carry += a[i] \* b;

        res.push\_back(carry % 10);

        carry /= 10;

    }

    removeLeadingZeros(res);

    return res;

}

// 大整数乘以大整数

vector<int> mul(const vector<int>& a, const vector<int>& b) {

    vector<int> res(a.size() + b.size(), 0);

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

        for (int j = 0; j < b.size(); ++j) {

            res[i + j] += a[i] \* b[j];

            res[i + j + 1] += res[i + j] / 10;

            res[i + j] %= 10;

        }

    }

    removeLeadingZeros(res);

    return res;

}

// 大整数除以小整数，返回商和余数

pair<vector<int>, int> div(const vector<int>& a, int b) {

    vector<int> res;

    int remainder = 0;

    for (int i = a.size() - 1; i >= 0; --i) {

        remainder = remainder \* 10 + a[i];

        res.push\_back(remainder / b);

        remainder %= b;

    }

    reverse(res.begin(), res.end());

    removeLeadingZeros(res);

    return make\_pair(res, remainder);

}

// 大整数除以大整数 (简单实现，效率不高)

vector<int> div(const vector<int>& a, const vector<int>& b) {

    vector<int> res;

    vector<int> current;

    for (int i = a.size() - 1; i >= 0; --i) {

        current.insert(current.begin(), a[i]);

        removeLeadingZeros(current);

        int quotient = 0;

        while (compare(current, b) >= 0) {

            current = sub(current, b);

            quotient++;

        }

        res.push\_back(quotient);

    }

    reverse(res.begin(), res.end());

    removeLeadingZeros(res);

    return res;

}

// 字符串转大整数 这里有点问题  就是没有考虑负数

vector<int> strToNum(const string& s) {

    vector<int> num;

    for (int i =(int) s.size() - 1; i >= 0; --i) {

        if(s[i]>='0'&&s[i]<='9')

            num.push\_back(s[i] - '0');

    }

    removeLeadingZeros(num);

    return num;

}

// 大整数转字符串

string numToStr(const vector<int>& num) {

    string s;

    for (int i = num.size() - 1; i >= 0; --i) {

        s += to\_string(num[i]);

    }

    return s;

}

树上启发式合并

// 一共有n个节点，编号1~n，给定n-1条边，所有节点连成一棵树，1号节点为树头

// 每个节点给定一种颜色值，一共有m条查询，每条查询给定参数x

// 每条查询打印x为头的子树上，一共有多少种不同的颜色

// 树链剖分

int fa[MAXN];

int siz[MAXN];

int son[MAXN];

// 树上启发式合并 colorCnt[i] = j，表示i这种颜色出现了j次

int colorCnt[MAXN],ans[MAXN];

int diffColors = 0;

// 重链剖分

void dfs1(int u, int f) {

    fa[u] = f;

    siz[u] = 1;

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            dfs1(v, u);

        }

    }

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            siz[u] += siz[v];

            if (son[u] == 0 || siz[son[u]] < siz[v]) {

                son[u] = v;

            }

        }

    }

}

// 子树u每个节点贡献信息

void effect(int u) {

    if (++colorCnt[arr[u]] == 1) {

        diffColors++;

    }

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa[u]) {

            effect(v);

        }

    }

}

// 子树u每个节点取消贡献

void cancle(int u) {

    colorCnt[arr[u]] = 0; // 出现任何颜色，直接把该颜色的计数重置为0

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa[u]) {

            cancle(v);

        }

    }

}

// 树上启发式合并的过程

void dfs2(int u, int keep) {

    // 遍历轻儿子的子树，统计子树的答案，然后取消贡献

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa[u] && v != son[u]) {

            dfs2(v, 0);

        }

    }

    // 遍历重儿子的子树，统计子树的答案，然后保留贡献

    if (son[u] != 0) {

        dfs2(son[u], 1);

    }

    // 当前节点贡献信息

    if (++colorCnt[arr[u]] == 1) {

        diffColors++;

    }

    // 遍历轻儿子的子树，重新贡献一遍

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa[u] && v != son[u]) {

            effect(v);

        }

    }

    // 记录子树u的答案

    ans[u] = diffColors;

    // 如果u是上级节点的轻儿子，子树u的贡献取消，否则保留

    if (keep == 0) {

        diffColors = 0;

        cancle(u);

    }

}

树链剖分（点权）

int fa[MAXN];

int dep[MAXN];

int siz[MAXN];//子树大小

int son[MAXN];//重儿子节点编号

int top[MAXN];//所在重链头节点

int dfn[MAXN];//dfn序号

int seg[MAXN];//dfn序号对应的原始节点编号

int cntd = 0;

// 来到节点u，节点u树上的父节点是f dfs1的过程去设置 fa dep siz son

void dfs1(int u, int f) {

    fa[u] = f;

    dep[u] = dep[f] + 1;

    siz[u] = 1;

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            dfs1(v, u);

        }

    }

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            siz[u] += siz[v];

            if (son[u] == 0 || siz[son[u]] < siz[v]) {

                son[u] = v;

            }

        }

    }

}

// 来到节点u，节点u所在重链的头节点是t dfs2的过程去设置 top dfn seg

void dfs2(int u, int t) {

    top[u] = t;

    dfn[u] = ++cntd;

    seg[cntd] = u;

    if (son[u] == 0) {

        return;

    }

    dfs2(son[u], t);//去 重儿子继续向下

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa[u] && v != son[u]) {

            //轻儿子自己开一条重链 自己做头结点

            dfs2(v, v);

        }

    }

}

// 从x到y的路径上，所有节点的值增加v

void pathAdd(int x, int y, int v) {

    while (top[x] != top[y]) {

        if (dep[top[x]] <= dep[top[y]]) {

            add(dfn[top[y]], dfn[y], v, 1, n, 1);

            y = fa[top[y]];

        } else {

            add(dfn[top[x]], dfn[x], v, 1, n, 1);

            x = fa[top[x]];

        }

    }

    add(min(dfn[x], dfn[y]), max(dfn[x], dfn[y]), v, 1, n, 1);

}

// 从x到y的路径上，查询所有节点的累加和

long long pathSum(int x, int y) {

    long long ans = 0;

    while (top[x] != top[y]) {

        if (dep[top[x]] <= dep[top[y]]) {

            ans = (ans + query(dfn[top[y]], dfn[y], 1, n, 1)) % MOD;

            y = fa[top[y]];

        } else {

            ans = (ans + query(dfn[top[x]], dfn[x], 1, n, 1)) % MOD;

            x = fa[top[x]];

        }

    }

    ans = (ans + query(min(dfn[x], dfn[y]), max(dfn[x], dfn[y]), 1, n, 1)) % MOD;

    return ans;

}

// x的子树上，查询所有节点的累加和

long long subtreeSum(int x) {

    return query(dfn[x], dfn[x] + siz[x] - 1, 1, n, 1);

}

树链剖分（边权）

int fa[MAXN];

int dep[MAXN];

int siz[MAXN];

int son[MAXN];

int top[MAXN];

int dfn[MAXN];

int cntd = 0;

void dfs1(int u, int f) {

    fa[u] = f;

    dep[u] = dep[f] + 1;

    siz[u] = 1;

    for (int e = head[u]; e > 0; e = nxt[e]) {

        int v = to[e];

        if (v != f) {

            dfs1(v, u);

        }

    }

    for (int e = head[u]; e > 0; e = nxt[e]) {

        int v = to[e];

        if (v != f) {

            siz[u] += siz[v];

            if (son[u] == 0 || siz[son[u]] < siz[v]) {

                son[u] = v;

            }

        }

    }

}

void dfs2(int u, int t) {

    top[u] = t;

    dfn[u] = ++cntd;

    if (son[u] == 0) {

        return;

    }

    dfs2(son[u], t);

    for (int e = head[u]; e > 0; e = nxt[e]) {

        int v = to[e];

        if (v != fa[u] && v != son[u]) {

            dfs2(v, v);

        }

    }

}

//更新一条边权  就是更新一个点的点权

void edgeUpdate(int ei, int val) {

    int x = arr[ei][0];

    int y = arr[ei][1];

    int downx = max(dfn[x], dfn[y]);

    update(downx, val, 1, n, 1);

}

//求x到y的之间的路径权值和

int pathSum(int x, int y) {

    int ans = 0;

    while (top[x] != top[y]) {

        if (dep[top[x]] <= dep[top[y]]) {

            ans += querySum(dfn[top[y]], dfn[y], 1, n, 1);

            y = fa[top[y]];

        } else {

            ans += querySum(dfn[top[x]], dfn[x], 1, n, 1);

            x = fa[top[x]];

        }

    }

    //lca不能动

    ans += querySum(min(dfn[x], dfn[y]) + 1, max(dfn[x], dfn[y]), 1, n, 1);

    return ans;

}

//求x到y的之间的路径权值最大值

int pathMax(int x, int y) {

    int ans = INT\_MIN;

    while (top[x] != top[y]) {

        if (dep[top[x]] <= dep[top[y]]) {

            ans = max(ans, queryMax(dfn[top[y]], dfn[y], 1, n, 1));

            y = fa[top[y]];

        } else {

            ans = max(ans, queryMax(dfn[top[x]], dfn[x], 1, n, 1));

            x = fa[top[x]];

        }

    }

    //lca不能动

    ans = max(ans, queryMax(min(dfn[x], dfn[y]) + 1, max(dfn[x], dfn[y]), 1, n, 1));

    return ans;

}

长链剖分

int stjump[MAXN][MAXH];//倍增表

int dep[MAXN];//深度表

int len[MAXN];//长度表

int son[MAXN];//长儿子

int top[MAXN];

int dfn[MAXN];

int cntd = 0;

int high[MAXN];//high[i]记录的是i的二进制最高位对应的数字是多少

int up[MAXN];//记录节点i向上走1 2 3 到达那个节点  只有长链的头结点才可以有

//并且只会填长儿子大小的个数   即使向上走还存在更远的  也不填了

int down[MAXN];//这个和上面一样  记录的是向下走几步

//设置u这个节点向上走i步 达到v节点  u一定是某条长链的头结点

void setUp(int u, int i, int v) {

    up[dfn[u] + i] = v;

}

//查询u节点向上走i步到达那个节点  u一定是某条长链的头结点

int getUp(int u, int i) {

    return up[dfn[u] + i];

}

//设置u这个节点向下走i步 达到v节点  u一定是某条长链的头结点

void setDown(int u, int i, int v) {

    down[dfn[u] + i] = v;

}

//查询u节点向下走i步到达那个节点  u一定是某条长链的头结点

int getDown(int u, int i) {

    return down[dfn[u] + i];

}

void dfs1(int u, int f) {

    stjump[u][0] = f;//没有fa数组 因为倍增表可以直接查询

    for (int p = 1; p < MAXH; p++) {

        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];

    }

    dep[u] = dep[f] + 1;

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            dfs1(v, u);

        }

    }

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            if (son[u] == 0 || len[son[u]] < len[v]) {

                son[u] = v;

            }

        }

    }

    len[u] = len[son[u]] + 1;//每个节点的大小是自己长儿子加一

}

void dfs2(int u, int t) {

    top[u] = t;

    dfn[u] = ++cntd;

    if (son[u] == 0) return;

    dfs2(son[u], t);

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != stjump[u][0] && v != son[u]) {

            dfs2(v, v);

        }

    }

}

//查询x节点向上走k步能到哪里

int query(int x, int k) {

    if (k == 0) {

        //如果是0步  那么直接就是这个节点

        return x;

    }

    if (k == (1 << high[k])) {

        //如果这个k正好是2的某次方  那么直接st表查询即可

        return stjump[x][high[k]];

    }

    //否则  我们先将其最高位的1用st表查询  找到一个中间节点

    x = stjump[x][high[k]];

    //然后k减去它的最高位

    k -= (1 << high[k]);

    //k剩下的这部分就通过来到的中间节点和他的头结点查询

    k -= dep[x] - dep[top[x]];

    x = top[x];

    return (k >= 0) ? getUp(x, k) : getDown(x, -k);

    //如果k小于0  那么说明根本到不了top[x]  那么向下走一点就好

    //如果k大于0  那么说明top[x]也不够 那么向上走一点就好

    //而且可以肯定的是  top[x]一定可以解决

    //因为既然初始节点走了k的最高位的距离来到了中间节点   至少走了k/2

    //那么说明这个中间节点的头结点的深度至少要大于k/2

}

哈希

const int MAXN = 10001;

const int BASE = 499;

long long power[MAXN];

long long hash\_val[MAXN];

void build(const string& str) {

    power[0] = 1;

    for (int j = 1; j < MAXN; ++j) {

        power[j] = power[j - 1] \* BASE;

    }

    hash\_val[0] = str[0] - 'a' + 1;

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

        hash\_val[j] = hash\_val[j - 1] \* BASE + str[j] - 'a' + 1;

    }

}

// 范围是s[l,r]，左闭右开

long long getHash(int l, int r) {

    long long ans = hash\_val[r];

    if (l > 0) {

        ans -= hash\_val[l - 1] \* power[r – l + 1];

    }

    return ans;

}

// 计算一个字符串的哈希值

long long hashString(const string& str) {

    if (str.empty()) {

        return 0;

    }

    int n = str.length();

    long long ans = str[0] - 'a' + 1;

    for (int j = 1; j < n; ++j) {

        ans = ans \* BASE + str[j] - 'a' + 1;

    }

    return ans;

}

KMP

char s1[MAXN],s2[MAXN];int Next[MAXN];

void nextArray(const char \*s, int m) {

    if (m == 1) {

        Next[0] = -1;

        return ;

    }

    Next[0] = -1,Next[1] = 0;

    int i = 2, cn = 0;

    // i表示当前要求next值的位置 cn表示当前要和前一个字符比对的下标

    while (i < m) {

        if (s[i - 1] == s[cn]) {

            Next[i] = ++cn;

            if(s[Next[i]]==s[i]){

                Next[i]=Next[Next[i]];

            }i++;

        } else if (cn > 0) {

            cn = Next[cn];

        } else {

            Next[i] = 0;

            if(s[Next[i]]==s[i]){

                Next[i]=Next[Next[i]];

            }

            i++;

        }

    }

}

int kmp(const char \*s1, const char \*s2) {

    int n = strlen(s1), m = strlen(s2), x = 0, y = 0;

    nextArray(s2, m);

    // s1中当前比对的位置是x s2中当前比对的位置是y

    while (x < n && y < m) {

        if (s1[x] == s2[y]) {

            x++;y++;

        } else if (y == 0) {

            x++;//表示s2已经在第一个位置了

        } else {

            y = Next[y];//表示s2往前跳

        }

    }

    return y == m ? x - y : -1;

}

Lucas定理

long long comb(long long n,long long m){

    if(n<m){

        return 0;

    }

    m=min(m,n-m);

    long long a=1,b=1;

    for(int i=1;i<=m;i++){

        a=(a\*(n-i+1))%p;

        b=(b\*i)%p;

    }

    return a\*power(b,p-2)%p;

}

long long lucas(int n,int m){

    if(m==0){

        return 1;

    }

    else{

        return lucas(n/p,m/p)\*comb(n%p,m%p)%p;

    }

}

BSGS定理

int bsgs(int g, int h, int mod) {

    // 处理特殊情况

    h%=mod;

    g%=mod;

    if(g==0){

        if (h == 0) return 1;  // 0^1 = 0 (x=1)

        if (h == 1) return 0;  // 0^0 = 1 (x=0)

        return -1;             // 无解

    }

    if (h == 1 ) return 0;

    if (h == 0 ) return -1;  // h=0 无解

    int t = ceil(sqrt(mod));

    map<int, int> mp;

    int base = h;

    // 小步：存储 h \* g^j (j=0 到 t-1)

    for (int j = 0; j < t; j++) {

        mp[base] = j;  // 直接覆盖，保留最大j

        base = (base \* g) % mod;

    }

    // 计算大步参数

    int tmp = power(g, t);  // g^t

    int cur = 1;  // 当前大步值 (g^{i\*t})

    // 大步：i 从 0 到 t

    for (int i = 0; i <= t; i++) {

        if (mp.find(cur) != mp.end()) {

            int x = i \* t - mp[cur];

            // 调整解到[0, mod-2]范围内

            //费马小定理：a^(p−1)≡1(modp)(对于任意 a!≡0)

            //这意味着指数x在模(p−1)意义下具有周期性：

            //y^x≡y^(x%(p−1))(mod p)

            x = (x % (mod-1) + (mod-1)) % (mod-1);

            return x;

        }

        cur = (cur \* tmp) % mod;  // 下一个大步

    }

    return -1;

}

欧拉函数

欧拉降幂公式（求2^3^4^…^2023数塔）

int geteuler(int tmp){

    int n=1;

    for(int i=2;i\*i<=tmp;i++){

        if(tmp%i){

            continue;

        }

        n\*=(i-1);

        tmp/=i;

        while(tmp%i==0){

            tmp/=i;

            n\*=i;

        }

    }

    if(tmp>1){

        n\*=(tmp-1);

    }

    return n;

}

int compute(int u,int mod){

    if(u==2023){

        return u;

    }

    return power(u,compute(u+1,geteuler(mod))+geteuler(mod),mod);

}

卡特兰数

long long c(int n, int k) {

    return (((fac[n] \* inv1[k]) % MOD) \* inv1[n - k]) % MOD;

}

// 公式1

long long compute1(int n) {

    build1(2 \* n);

    return (c(2 \* n, n) - c(2 \* n, n - 1) + MOD) % MOD;

}

// 公式2

long long compute2(int n) {

    build1(2 \* n);

    return c(2 \* n, n) \* power(n + 1, MOD - 2) % MOD;

}

// 公式3

long long compute3(int n) {

    build2(n);

    long long f[n + 1];

    f[0] = f[1] = 1;

    for (int i = 2; i <= n; i++) {

        f[i] = f[i - 1] \* (4 \* i - 2) % MOD \* inv2[i + 1] % MOD;

    }

    return f[n];

}

// 公式4

long long compute4(int n) {

    long long f[n + 1];

    for(int i=0;i<=n;i++){

        f[i]=0;

    }

    f[0] = f[1] = 1;

    for (int i = 2; i <= n; i++) {

        for (int l = 0, r = i - 1; l < i; l++, r--) {

            f[i] = (f[i] + f[l] \* f[r] % MOD) % MOD;

        }

    }

    return f[n];

}

康托展开

void add(int i,int v){

    while(i<=n){

        tree[i]+=v;i+=lowbit(i);

    }

}

int sum(int i){

    int ans=0;

    while(i>0){

        ans+=tree[i];

        i-=lowbit(i);

    }

    return ans;

}

int query(int l,int r){

    return sum(r)-sum(l-1);

}

signed main(){

    //将树状数组初始化

    for(int i=1;i<=n;i++){

        add(i,1);

    }

    long long ans=0;

    for(int i=1;i<=n;i++){

        ans=(ans+(fac[n-i]\*sum(arr[i]-1))%MOD)%MOD;

        add(arr[i],-1);//将这一位的信息去除

    }

    ans=(ans+1)%MOD;

    cout<<ans<<endl;

    return 0;

}

逆过程

// 初始化线段树，单点范围的初始累加和为1，认为所有数字都可用

void build(int l, int r, int i) {

    if (l == r) {

        sum[i] = 1;

    } else {

        int mid = (l + r) >> 1;

        build(l, mid, i << 1);build(mid + 1, r, i << 1 | 1);

        sum[i] = sum[i << 1] + sum[i << 1 | 1];

    }

}

// 单点jobi上，增加jobv，因为是单点更新，所以不需要建立懒更新机制

void add(int jobi, int jobv, int l, int r, int i) {

    if (l == r) {

        sum[i] += jobv;

    } else {

        int mid = (l + r) >> 1;

        if (jobi <= mid) {

            add(jobi, jobv, l, mid, i << 1);

        } else {

            add(jobi, jobv, mid + 1, r, i << 1 | 1);

        }

        sum[i] = sum[i << 1] + sum[i << 1 | 1];

    }

}

// 查询jobl~jobr范围的累加和

int Sum(int jobl, int jobr, int l, int r, int i) {

    if (jobl <= l && r <= jobr) {

        return sum[i];

    }

    int mid = (l + r) >> 1;

    int ans = 0;

    if (jobl <= mid) {

        ans += Sum(jobl, jobr, l, mid, i << 1);

    }

    if (jobr > mid) {

        ans += Sum(jobl, jobr, mid + 1, r, i << 1 | 1);

    }

    return ans;

}

// 线段树上找到第k名的是什么，找到后删掉词频，返回的过程修改累加和

// 注意这个排名是在子树中的排名

int getAndDelete(int k, int l, int r, int i) {

    int ans;

    if (l == r) {

        //找到目标 删除词频

        sum[i]--;

        ans = l;

    } else {

        int mid = (l + r) >> 1;

        if (sum[i << 1] >= k) {

            ans = getAndDelete(k, l, mid, i << 1);

        } else {

            ans = getAndDelete(k - sum[i << 1], mid + 1, r, i << 1 | 1);

            //要减去左侧排名的影响

        }

        sum[i] = sum[i << 1] + sum[i << 1 | 1];

        //返回的过程修改累加和

    }

    return ans;

}

void compute() {

    build(1, n, 1);

    // 当前排列转化为阶乘进制的排名

    for (int i = 1, x; i <= n; i++) {

        x = (int) arr[i];

        if (x == 1) {

            arr[i] = 0;

        } else {

            arr[i] = Sum(1, x - 1, 1, n, 1);

        }

        //arr数组表示一开始的排列在阶乘进制下的排名

        add(x, -1, 1, n, 1);

        //消除这个数字的影响

    }

    // 当前排名加上m之后，得到新的排名，用阶乘进制表示

    arr[n] += m; // 最低位获得增加的幅度

    for (int i = n; i >= 1; i--) {

        // 往上进位多少

        arr[i - 1] += arr[i] / (n - i + 1);

        // 当前位是多少

        arr[i] %= n - i + 1;

    }

    //到这里时  arr数组里面是加上m后的排名在阶乘进制下的表示

    // 根据阶乘进制转化为具体的排列

    build(1, n, 1);

    //将线段树重新复原

    for (int i = 1; i <= n; i++) {

        arr[i] = getAndDelete((int) arr[i] + 1, 1, n, 1);

        //此时这个arr数组用来存放排列

    }

}

双向广搜

//这种才是双向广搜的本体用法 即必须要全部展开 但是时间复杂度不允许  eg:2^40(10^12)

//所以我们将左右两边分别展开  最后将左右两边信息合并起来  得到汇总信息

long arr[MAXN];

long lsum[MAXM];//左边展开的信息

long rsum[MAXM];//右边展开的信息

int n;

long w;

//暴力展开 从i开始到e结束  不要超过w  因为超过w没有任何意义

//目前已经有了j个答案  暂时总和是s 递归函数  会一直递归到i==e

//返回值是自己可以得到的最终答案

int f(int i, int e, long s, long w, long\* ans, int j) {

    if (s > w) {

        return j;

    }

    // s <= w

    if (i == e) {

        //表示到了可以选择的最后一个物品了  记录可行答案的值 填入相应的答案记录数组中

        ans[j++] = s;

    } else {

        j = f(i + 1, e, s, w, ans, j); // 不要arr[i]位置的数

        j = f(i + 1, e, s + arr[i], w, ans, j); // 要arr[i]位置的数

    }

    return j;

}

long compute(){

    int lsize = f(0, n >> 1, 0, w, lsum, 0);

    int rsize = f(n >> 1, n, 0, w, rsum, 0);

    sort(lsum, lsum+lsize);

    sort(rsum, rsum+rsize);

    long ans = 0;

    //汇总左右两边信息 双指针

    for (int i = lsize - 1, j = 0; i >= 0; i--) {

        while (j < rsize && lsum[i] + rsum[j] <= w) {

            j++;

        }

        ans += j;

    }

    return ans;

}

高斯消元

加法模板

void swap(int a, int b) {

    //并且手写的时候一定要注意列的范围  不能缺少

    double tmp[MAXN + 1];

    for (int j = 0; j <= n+1; j++) {

        tmp[j] = mat[a][j];

        mat[a][j] = mat[b][j];

        mat[b][j] = tmp[j];

    }

}

//这个函数的作用是求出mat矩阵的解  n是x1 x2    x 如果无解返回0  有解返回1

int gauss(int n) {

    for (int i = 1; i <= n; i++) {

        //每次求解一个xi  也不是最终的值

        //只是将其他式子中的xi的系数消除为0

        //只保留自己的系数  最终将系数化为1

        int max = i;

        for (int j = 1; j <= n; j++) {

            if (j < i && abs(mat[j][j]) >= sml) {

                continue;

            }

            if (abs(mat[j][i]) > abs(mat[max][i])) {

                max = j;

            }

        }

        swap(i, max);//交换改行最大值

        if (abs(mat[i][i]) < sml) {

            //表示xi为0  那么不可能有解了

            return 0;

        }

        if (abs(mat[i][i]) >= sml) {

            //消去其他式子中的xi

            double tmp = mat[i][i];

            for (int j = i; j <= n + 1; j++) {

                mat[i][j] /= tmp;

            }

            for (int j = 1; j <= n; j++) {

                if (i != j) {

                    double rate = mat[j][i] / mat[i][i];

                    for (int k = i; k <= n + 1; k++) {

                        mat[j][k] -= mat[i][k] \* rate;

                    }

                }

            }

        }

    }

    return 1;

}

异或模板

// 高斯消元解决异或方程组模版 需要保证变量有n个，表达式也有n个

void gauss(int n) {

    for (int i = 1; i <= n; i++) {

        for (int j = 1; j <= n; j++) {

            if (j < i && mat[j][j] == 1) {

                continue;

            }

            if (mat[j][i] == 1) {

                swap(mat[i], mat[j]);

                break;

            }

        }

        if (mat[i][i] == 1) {

            for (int j = 1; j <= n; j++) {

                if (i != j && mat[j][i] == 1) {

                    for (int k = i; k <= n + 1; k++) {

                        mat[j][k] ^= mat[i][k];

                    }

                }

            }

        }

    }

}

同余模板

// 高斯消元解决同余方程组模版，保证初始系数没有负数

void gauss(int n) {

    for (int i = 1; i <= n; i++) {

        for (int j = 1; j <= n; j++) {

            if (j < i && mat[j][j] != 0) {

                continue; // 已经成为主元的行不参与

            }

            if (mat[j][i] != 0) {

                swap(i,j); // 找到系数不等于0的行做主元即可

                break;

            }

        }

        if (mat[i][i] != 0) {

            for (int j = 1; j <= n; j++) {

                if (i != j && mat[j][i] != 0) {

                    int gcd\_val = gcd(mat[j][i], mat[i][i]);

                    int a = mat[i][i] / gcd\_val;

                    int b = mat[j][i] / gcd\_val;

                    if (j < i && mat[j][j] != 0) {

                        // 如果j行有主元，那么从j列到i-1列的所有系数 \* a

                        // 正确更新主元和自由元之间的关系

                        for (int k = j; k < i; k++) {

                            mat[j][k] = (mat[j][k] \* a) % MOD;

                        }

                    }

                    // 正常消元

                    for (int k = i; k <= n + 1; k++) {

                        mat[j][k] = ((mat[j][k] \* a - mat[i][k] \* b) % MOD + MOD) % MOD;

                    }

                }

            }

        }

    }

    for (int i = 1; i <= n; i++) {

        if (mat[i][i] != 0) {

            // 检查当前主元是否被若干自由元影响

            // 如果当前主元不受自由元影响，那么可以确定当前主元的值

            // 否则保留这种影响，正确显示主元和自由元的关系

            bool flag = false;

            for (int j = i + 1; j <= n; j++) {

                if (mat[i][j] != 0) {

                    flag = true;break;

                }

            }

            if (!flag) {

                // 在模意义下应该求逆元，(a / b) % MOD = (a \* b的逆元) % MOD

                mat[i][n + 1] = (mat[i][n + 1] \* inv[mat[i][i]]) % MOD;

                mat[i][i] = 1;

            }

        }

    }

}

线性基

异或高斯消元法

void swap(int a, int b) {

    long long tmp = arr[a];

    arr[a] = arr[b];

    arr[b] = tmp;

}

//这个是高斯消元法  得到的是标准线性基

void compute() {

    len = 1;

    for (int i = BIT; i >= 0; i--) {

        for (int j = len; j <= n; j++) {

            if (arr[j] & (1LL << i)) {

                swap(j, len);//这里是对的 改成swap(arr[j], arr[len]);也可以通过

                break;

            }

        }

        if (arr[len] & (1LL << i)) {

            //表示找到了这一位上是1的数字

            for (int j = 1; j <= n; j++) {

                //将所有这一位置上是1的数字全部异或掉

                if (j != len && (arr[j] & (1LL << i))) {

                    arr[j] ^= arr[len];

                }

            }

            len++;

        }

    }

    len--;//表示有len位可以确定

    zero = len != n;

}

// 返回第k小的异或和

long query(long k) {

    if (zero && k == 1) {

        //如果有0  并且查询第一小的数字

        //那么直接返回0就好了

        return 0;

    }

    if (zero) {

        //如果有0  那么k就要减1

        //因为我们统计的是异或后不产生0的线性基

        k--;

    }

    if (k >= 1L << len) {

        //超出范围

        return -1;

    }

    long ans = 0;

    for (int i = len, j = 0; i >= 1; i--, j++) {

        if ((k & (1L << j)) != 0) {

            ans ^= arr[i];

        }

    }

    return ans;

}

异或普通消元法

// 线性基里插入num，如果线性基增加了返回true，否则返回false

bool insert(long num) {

    for (int i = BIT; i >= 0; i--) {

        if (num >> i == 1) {

            if (basis[i] == 0) {

                basis[i] = num;

                return true;

            }

            num ^= basis[i];

        }

    }

    return false;

}

// 普通消元

// 计算最大异或和

long compute() {

    for (int i = 1; i <= n; i++) {

        insert(arr[i]);

    }

    long ans = 0;

    for (int i = BIT; i >= 0; i--) {

        ans = max(ans, ans ^ basis[i]);

    }

    //如果可以使答案增加  那么就异或上这个数字

    return ans;

}

向量线性基

//向量线性基模版

bool insert(int i) {

    for (int j = 1; j <= m; j++) {

        if (fabs(mat[i][j]) >= sml) {

            if (basis[j] == 0) {

                basis[j] = i;

                return true;

            }

            double rate = mat[i][j] / mat[basis[j]][j];

            for (int k = j; k <= m; k++) {

                mat[i][k] -= rate \* mat[basis[j]][k];

            }

        }

    }

    return false;

}

void compute() {

    cnt = cost = 0;//记录答案

    vector<pair<double, int>> prices;

    for (int i = 1; i <= n; i++) {

        prices.emplace\_back(mat[i][m + 1], i);

    }

    sort(prices.begin(), prices.end());

    //根据价格排序  价格小的优先

    for (const auto& p : prices) {

        int i = p.second;

        if (insert(i)) {

            //如果这个物品可以插入 那么就插入

            cnt++;

            cost += static\_cast<int>(mat[i][m + 1]);

        }

    }

}

前缀树

int cnt=1;//用于赋值新编号   全部从1开始使用

int tree[MAXN][26];//表示最多只有26个字符

//tree[i][j]:表示编号为i的节点通过j（代表一个字符）这条路径到达下一个节点

//如果这个位置上是0  表示没有这条路

int en[MAXN];

int pass[MAXN];

//end  pass表示这个编号上的经过信息和结尾信息

int path;

void insert(string word){

    int cur=1;

    pass[cur]++;

    for (char ch : word){

        path=ch-'a';

        if(tree[cur][path]==0){

            tree[cur][path]=++cnt;

            //没有路就要新建一条路  并赋予编号

        }

        cur=tree[cur][path];

        pass[cur]++;

    }

    en[cur]++;

}

int search(string word){

    int cur=1;

    for (char ch : word){

        path=ch-'a';

        if(tree[cur][path]==0){

            return 0;

            //没有路说明走不通

        }

        cur=tree[cur][path];

    }

    return en[cur];

}

int preword(string word){

    int cur=1;

    for (char ch : word){

        path=ch-'a';

        if(tree[cur][path]==0){

            return 0;

            //没有路说明走不通

        }

        cur=tree[cur][path];

    }

    return pass[cur];

}

void erase(string word){

    if(search(word)>0){

        int cur=1;

        for (char ch : word){

            path=ch-'a';

            if(--pass[tree[cur][path]]==0){

                tree[cur][path]=0;

                //将这个节点设置为0  那么这个节点所连着的路全部失效

                //因为即使后来有结点重新连接上  但是会赋予这个新节点另外一个编号

                return ;

            }

            cur=tree[cur][path];

        }

        en[cur]--;

    }

}

void clear(){

    for(int i=1;i<=cnt;i++){

        pass[i]=0;

        en[i]=0;

        for(int j=0;j<26;j++){

            tree[i][j]=0;

        }

    }

}

可持久化前缀树（01trie）

// 最大异或和，java版

// 非负序列arr的初始长度为n，一共有m条操作，每条操作是如下两种类型中的一种

// A x     : arr的末尾增加数字x，arr的长度n也增加1

// Q l r x : l~r这些位置中，选一个位置p，现在希望

//           arr[p] ^ arr[p+1] ^ .. ^ arr[n] ^ x 这个值最大

//           打印这个最大值

// 1 <= n、m <= 3 \* 10^5

// 0 <= arr[i]、x <= 10^7

// 因为练的就是可持久化前缀树，所以就用在线算法，不要使用离线算法

const int MAXN = 600001;

const int MAXT = MAXN \* 22;

const int BIT = 25;

int n, m, eor;

int root[MAXN];

int tree[MAXT][2];//01trie只有两条路可以走 一条是0 一条是1

int pass[MAXT];//记录这个节点走过了几次

int cnt = 0;

//当前走到了i节点  插入了num  更新并返回节点编号

int insert(int num, int i) {

    int rt = ++cnt;

    tree[rt][0] = tree[i][0];

    tree[rt][1] = tree[i][1];

    pass[rt] = pass[i] + 1;

    for (int b = BIT, path, pre = rt, cur; b >= 0; b--, pre = cur) {

        path = (num >> b) & 1;

        i = tree[i][path];

        cur = ++cnt;

        tree[cur][0] = tree[i][0];

        tree[cur][1] = tree[i][1];

        pass[cur] = pass[i] + 1;

        tree[pre][path] = cur;

    }

    return rt;

}

int query(int num, int u, int v) {

    int ans = 0;

    for (int b = BIT, path, best; b >= 0; b--) {

        path = (num >> b) & 1;

        best = path ^ 1;

        if (pass[tree[v][best]] > pass[tree[u][best]]) {

            //表示可以是best

            ans += 1 << b;

            u = tree[u][best];

            v = tree[v][best];

        } else {

            u = tree[u][path];

            v = tree[v][path];

        }

    }

    return ans;

}

int main() {

    cin >> n >> m;

    eor = 0;//作为所有值的异或和

    root[0] = insert(eor, 0);

    for (int i = 1, num; i <= n; i++) {

        cin >> num;

        eor ^= num;

        root[i] = insert(eor, root[i - 1]);

    }

    string op;

    int x, y, z;

    for (int i = 1; i <= m; i++) {

        cin >> op;

        if (op == "A") {

            cin >> x;

            eor ^= x;

            n++;

            root[n] = insert(eor, root[n - 1]);

        } else {

            cin >> x >> y >> z;

            //首先将要查询的变为eor^z

            if (x == 1) {

                cout << query(eor ^ z, 0, root[y - 1]) << "\n";

            } else {

                cout << query(eor ^ z, root[x - 2], root[y - 1]) << "\n";

            }

        }

    }

    return 0;

}

Manacher

// Manacher算法模版

// 求字符串s中最长回文子串的长度

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 11000001;

char ss[MAXN << 1];//这个数组中间添加了#

char a[MAXN];//原始数组

int p[MAXN << 1];//半径数组

int n;

void manacherss() {

    n = strlen(a) \* 2 + 1;

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

        ss[i] = (i & 1) == 0 ? '#' : a[j++];

    }

}

int manacher() {

    manacherss();

    int maxans = 0;

    for (int i = 0, c = 0, r = 0, len; i < n; i++) {//i是此时来到的中心位置

        len = r > i ? min(p[2 \* c - i], r - i) : 1;//这个值是基本值  如果包住了  那么基本值就是最终值  否则后续再加

        while (i + len < n && i - len >= 0 && ss[i + len] == ss[i - len]) {

            len++;

            //如果被包住了  根本就进不来

        }

        //此时已经求出了以i为中心的回文半径大小

        if (i + len > r) {

            r = i + len;

            c = i;//c表示的是如今的r是以那个点为中心的右边界

            //如果更新了右边界  那么c就是i  r就是i+len

        }

        maxans = max(maxans, len);

        p[i] = len;

        //得到回文半径

    }

    return maxans - 1;

}

可持久化线段树

// 给定一个长度为n的数组arr，下标1~n，一共有m条查询

// 每条查询 l r k : 打印arr[l..r]中第k小的数字

// 1 <= n、m <= 2 \* 10^5 0 <= arr[i] <= 10^9

int n, m, s;

int arr[MAXN]; // 原始数组

int sorted[MAXN]; // 收集权值排序并且去重做离散化

// root[i] : 插入arr[i]之后形成新版本的线段树，记录头节点编号

// 0号版本的线段树代表一个数字也没有时，每种名次的数字出现的次数

// i号版本的线段树代表arr[1..i]范围内，每种名次的数字出现的次数

int root[MAXN],ls[MAXT],rs[MAXT];

// 排名范围内收集了多少个数字

int size[MAXT],cnt;

// 返回num在所有值中排名多少

int kth(int num) {

    int left = 1, right = s, mid, ans = 0;

    while (left <= right) {

        mid = (left + right) / 2;

        if (sorted[mid] <= num) {

            ans = mid;left = mid + 1;

        } else {

            right = mid - 1;

        }

    }

    return ans;

}

// 排名范围l~r，建立线段树，返回头节点编号

int build(int l, int r) {

    int rt = ++cnt;

    size[rt] = 0;

    if (l < r) {

        int mid = (l + r) / 2;

        ls[rt] = build(l, mid);

        rs[rt] = build(mid + 1, r);

    }

    return rt;

}

// 排名范围l~r，信息在i号节点，增加一个排名为jobi的数字 返回新的头节点编号

int insert(int jobi, int l, int r, int i) {

    int rt = ++cnt;

    ls[rt] = ls[i];

    rs[rt] = rs[i];

    size[rt] = size[i] + 1;//新加一个元素  大小加一

    if (l < r) {

        int mid = (l + r) / 2;

        if (jobi <= mid) {

            ls[rt] = insert(jobi, l, mid, ls[rt]);

        } else {

            rs[rt] = insert(jobi, mid + 1, r, rs[rt]);

        }

    }

    return rt;

}

// 排名范围l~r，老版本信息在u号节点，新版本信息在v号节点 返回，第jobk小的数字，排名多少

int query(int jobk, int l, int r, int u, int v) {

    if (l == r) {

        return l;

    }

    int lsize = size[ls[v]] - size[ls[u]];

    int mid = (l + r) / 2;

    if (lsize >= jobk) {

        return query(jobk, l, mid, ls[u], ls[v]); //如果左半边排名足够 那么就在左半边找

    } else {

        return query(jobk - lsize, mid + 1, r, rs[u], rs[v]);

    }

}

// 权值做离散化并且去重 + 生成各版本的线段树

void prepare() {

    cnt = 0;

    for (int i = 1; i <= n; i++) {

        sorted[i] = arr[i];

    }

    sort(sorted + 1, sorted + n + 1);

    s = 1;

    for (int i = 2; i <= n; i++) {

        if (sorted[s] != sorted[i]) {

            sorted[++s] = sorted[i];

        }

    }

    root[0] = build(1, s);//建立第一棵树

    for (int i = 1, x; i <= n; i++) {

        x = kth(arr[i]);

        root[i] = insert(x, 1, s, root[i - 1]);

        //每次基于上一个树插入一个元素  建立新树

    }

}

树上差分

点差分

// 有n个节点形成一棵树，一开始所有点权都是0

// 给定很多操作，每个操作(a,b)表示从a到b路径上所有点的点权增加1

// 所有操作完成后，返回树上的最大点权

//点差分中将从a到b的路径上所有点权加v  相当于a+v b+v  lca-v  lcafa-v

//最后统计的时候 来到一个节点  先遍历他的子节点  然后这个节点加上子节点的点权

//最终得到自己的点权

int deep[MAXN],stjump[MAXN][LIMIT];

int power;

int num[MAXN];//记录点权

//这个函数的作用是建立deep  st的信息

void dfs1(int u, int f) {

    deep[u] = deep[f] + 1;

    stjump[u][0] = f;

    for (int p = 1; p <= power; p++) {

        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];

    }

    //完成u的deep  stjump

    for (int e = head[u]; e != 0; e = Next[e]) {

        if (to[e] != f) {

            dfs1(to[e], u);

        }

        //向下递归

    }

}

int lca(int a, int b) {

    if (deep[a] < deep[b]) {

        int tmp = a;

        a = b;

        b = tmp;

    }

    //确定大小关系

    for (int p = power; p >= 0; p--) {

        if (deep[stjump[a][p]] >= deep[b]) {

            a = stjump[a][p];

        }

    }

    //首先将两者变为同一高度

    if (a == b) {

        return a;

    }

    //如果相同说明就是祖先关系

    for (int p = power; p >= 0; p--) {

        if (stjump[a][p] != stjump[b][p]) {

            a = stjump[a][p];

            b = stjump[b][p];

        }

        //判断跳完后是否符合规则

    }

    return stjump[a][0];

    //我们将头结点的祖先设置为0  实际上没有0

}

//更新修改后的信息

void dfs2(int u,int f){

    for(int i=head[u];i>0;i=Next[i]){

        int v=to[i];

        if(v!=f){

            dfs2(v,u);

        }

    }

    for(int i=head[u];i>0;i=Next[i]){

        int v=to[i];

        if(v!=f){

            num[u]+=num[v];

        }

    }

}

边差分

平衡树

FHQ平衡树

// 实现一种结构，支持如下操作，要求单次调用的时间复杂度O(log n)

// 1，增加x，重复加入算多个词频

// 2，删除x，如果有多个，只删掉一个

// 3，查询x的排名，x的排名为，比x小的数的个数+1

// 4，查询数据中排名为x的数

// 5，查询x的前驱，x的前驱为，小于x的数中最大的数，不存在返回整数最小值

// 6，查询x的后继，x的后继为，大于x的数中最小的数，不存在返回整数最大值

// 所有操作的次数 <= 10^5

// -10^7 <= x <= +10^7

//词频不压缩的一个巨大优势是在删除和加入节点时非常方便

#include <bits/stdc++.h>

using namespace std;

const int MAXN = 100001;

int head = 0;

int cnt = 0;

int key[MAXN];

int ls[MAXN];

int rs[MAXN];

int size[MAXN];

double priority[MAXN];

//没有词频压缩  所以出现次数就是1

void up(int i) {

    size[i] = size[ls[i]] + size[rs[i]] + 1;

}

void split(int l, int r, int i, int num) {

    if (i == 0) {

        rs[l] = ls[r] = 0;

    } else {

        if (key[i] <= num) {

            rs[l] = i;

            split(i, r, rs[i], num);

        } else {

            ls[r] = i;

            split(l, i, ls[i], num);

        }

        up(i);

    }

}

int merge(int l, int r) {

    if (l == 0 || r == 0) {

        return l + r;

    }

    if (priority[l] >= priority[r]) {

        rs[l] = merge(rs[l], r);

        up(l);

        return l;

    } else {

        ls[r] = merge(l, ls[r]);

        up(r);

        return r;

    }

}

void add(int num) {

    split(0, 0, head, num);

    key[++cnt] = num;

    size[cnt] = 1;

    priority[cnt] = (double)rand() / RAND\_MAX;

    head = merge(merge(rs[0], cnt), ls[0]);

}

//删除节点的时候 是将树按照num分裂  然后将<=num 的按照 num-1 分裂

//将那么 >num-1 的树头结点一定是num  只要将这个节点忽略就是删除节点

void remove(int num) {

    split(0, 0, head, num);

    int lm = rs[0];

    int r = ls[0];

    split(0, 0, lm, num - 1);

    int l = rs[0];

    int m = ls[0];

    head = merge(merge(l, merge(ls[m], rs[m])), r);

}

int getRank(int num) {

    split(0, 0, head, num - 1);

    int ans = size[rs[0]] + 1;

    head = merge(rs[0], ls[0]);

    return ans;

}

int index(int i, int x) {

    if (size[ls[i]] >= x) {

        return index(ls[i], x);

    } else if (size[ls[i]] + 1 < x) {

        return index(rs[i], x - size[ls[i]] - 1);

    } else {

        return key[i];

    }

}

int index(int x) {

    return index(head, x);

}

int pre(int i, int num) {

    if (i == 0) {

        return INT\_MIN;

    }

    if (key[i] >= num) {

        return pre(ls[i], num);

    } else {

        return max(key[i], pre(rs[i], num));

    }

}

int pre(int num) {

    return pre(head, num);

}

int post(int i, int num) {

    if (i == 0) {

        return INT\_MAX;

    }

    if (key[i] <= num) {

        return post(rs[i], num);

    } else {

        return min(key[i], post(ls[i], num));

    }

}

int post(int num) {

    return post(head, num);

}

快读

inline int read(){

    int x=0,f=1;

    char ch=getchar();

    while(ch<'0'||ch>'9'){

        if(ch=='-')

            f=-1;

        ch=getchar();

    }

    while(ch>='0' && ch<='9')

        x=x\*10+ch-'0',ch=getchar();

    return x\*f;

}

int n,m,s,t;

int dep[MAXN];//记录深度数组

int iter[MAXN];//当前弧数组 记录这个节点有效访问的第一条边的编号

//这里编号从2开始 为了正反边编号寻找方便

int cnt=2;

int head[MAXN];

int nxt[MAXM];

int to[MAXM];

int now[MAXM];//表示现在的流量

int cap[MAXM];//表示流量限制

void addedge(int u,int v,int w){

    nxt[cnt]=head[u];

    to[cnt]=v;

    cap[cnt]=w;

    head[u]=cnt++;

    nxt[cnt]=head[v];

    to[cnt]=u;

    cap[cnt]=0;

    head[v]=cnt++;

}

//BFS构建分层图，并判断是否存在增广路径

bool bfs(){

    for(int i=1;i<=n;i++){

        dep[i]=-1;

    }

    queue<int>q;

    dep[s]=0;

    q.push(s);

    while(!q.empty()){

        int u=q.front();

        q.pop();

        for(int i=head[u];i;i=nxt[i]){

            int v=to[i];

            int w=cap[i];

            int k=now[i];

            if(dep[v]<0&&(k<w)){

                dep[v]=dep[u]+1;

                q.push(v);

            }

        }

    }

    return dep[t]>=0;// 如果汇点未被访问到，说明无增广路径

}

int dfs(int u,int f){

    if(u==t){

        return f;

    }

    int flow=0;

    for(int &i=iter[u];i;i=nxt[i]){

        //注意这里是引用  iter会随着i发生变化

        int v=to[i];

        int w=cap[i];

        int k=now[i];

        if(dep[u]+1==dep[v]&&k<w){

            int d=dfs(v,min(f,w-k));

            if(d>0){

                now[i]+=d;

                now[i^1]-=d;//更新反向边

                flow+=d;

                f-=d;

                if(f==0){

                    break;

                }

            }

        }

    }

    return flow;

}

// Dinic算法主函数

int maxflow(){

    int flow=0;

    while(bfs()){

        //当前弧全部初始化为最初值

        for(int i=1;i<=n;i++){

            iter[i]=head[i];

        }

        int maxflow;

        while((maxflow=dfs(s,INF))>0){

            flow+=maxflow;

        }

    }

    return flow;

}