Data Structures

// 2D Fenwick Tree (point updates)

struct FWT

{

    int v[N][N];

    FWT() {}

    void update(int x, int y, int val)

    {

        x++; y++;

        while (x<=n)

        {

            int ty=y;

            while (ty<=n)

            {

                v[x][ty]+=val;

                ty+=(ty&-ty);

            }

            x+=(x&-x);

        }

    }

    int query(int x, int y)

    {

        int ans=0;

        while (x>0)

        {

            int ty=y;

            while (ty>0)

            {

                ans+=v[x][ty];

                ty-=(ty&-ty);

            }

            x-=(x&-x);

        }

        return ans;

    }

    int query(int x1, int y1, int x2, int y2)

    {

        x2++; y2++;

        return query(x2, y2)-query(x2, y1)-query(x1, y2)+query(x1, y1);

    }

} \*fwt;

// Convex Hull Trick (deque)

struct line

{

    int a=0, b=-1e18;

    int operator()(int x) { return a\*x+b; }

};

struct CHT

{

    deque<line> q;

    CHT() {}

    void insert(line x)

    {

        while (q.size()>=2 && intersect(x, q[q.size()-2])>=intersect(x, q[q.size()-1])) q.pop\_back();

        q.push\_back(x);

    }

    int query(int x)

    {

        if (q.size()==1) return q[0](x);

        if (intersect(q[0], q[1])>x) return q[0](x);

        int l=1, r=q.size();

        while (l+1<r)

        {

            int mid=(l+r)>>1;

            if (intersect(q[mid-1], q[mid])<=x) l=mid;

            else r=mid;

        }

        return q[l](x);

    }

    double intersect(line x, line y)

    {

        if (x.a==y.a) return -1e18;

        else return (double)(x.b-y.b)/(double)(y.a-x.a);

    }

} \*cht;

// Convex Hull Trick (LC)

struct line

{

    mutable int a, b, p;

    int operator()(int x) { return a\*x+b; }

    bool operator<(line x) const { return a<x.a; }

    bool operator<(int x) const { return p<x; }

};

struct CHT:multiset<line, less<>>

{

    CHT() {}

    void add(line l)

    {

        auto z=insert(l);

        auto y=z++, x=y;

        while (isect(y, z)) z=erase(z);

        if (x!=begin() && isect(--x, y)) isect(x, y=erase(y));

        while ((y=x)!=begin() && ((--x)->p)>=(y->p)) isect(x, erase(y));

    }

    int query(int x)

    {

        auto l=\*lower\_bound(x);

        return (int)l(x);

    }

    int div(int x, int y) { return x/y-((x^y)<0 && x%y); }

    bool isect(iterator x, iterator y)

    {

        if (y==end()) return x->p=1e18, 0;

        if (x->a==y->a) x->p=(x->b)>(y->b)?1e18:-1e18;

        else x->p=div((y->b)-(x->b), (x->a)-(y->a));

        return (x->p)>=(y->p);

    }

} \*cht;

// Fenwick Tree (point updates)

struct FWT

{

    int v[N]={0};

    FWT() {}

    void update(int pos, int val)

    {

        pos++;

        while (pos<=n)

        {

            v[pos]+=val;

            pos+=(pos&-pos);

        }

    }

    int query(int pos)

    {

        int ans=0;

        while (pos>0)

        {

            ans+=v[pos];

            pos-=(pos&-pos);

        }

        return ans;

    }

    int query(int l, int r)

    {

        l++; r++;

        return query(r)-query(l-1);

    }

} \*fwt;

// Fenwick Tree

struct FWT

{

    int v1[N], v2[N];

    FWT() {}

    void update(int v[N], int pos, int val)

    {

        while (pos<=n)

        {

            v[pos]+=val;

            pos+=(pos&-pos);

        }

    }

    void update(int l, int r, int val)

    {

        l++; r++;

        update(v1, l, val); update(v1, r+1, -val);

        update(v2, l, l\*val); update(v2, r+1, -(r+1)\*val);

    }

    int query(int v[N], int pos)

    {

        int ans=0;

        while (pos>0)

        {

            ans+=v[pos];

            pos-=(pos&-pos);

        }

        return ans;

    }

    int query(int l, int r)

    {

        l++; r++;

        return ((r+1)\*query(v1, r)-query(v2, r))-(l\*query(v1, l-1)-query(v2, l-1));

    }

} \*fwt;

// Lichao Tree

struct line

{

    int a=0, b=-1e18;

    int operator()(int x) { return a\*x+b; }

    bool operator==(line x) { return a==x.a && b==x.b; }

};

struct LCT

{

    int l, r, mid;

    line v;

    LCT \*lptr, \*rptr;

    LCT(int L, int R)

    {

        l=L; r=R; mid=(l+r)>>1;

        lptr=rptr=nullptr;

    }

    void build()

    {

        if (lptr==nullptr) lptr=new LCT(l, mid);

        if (rptr==nullptr) rptr=new LCT(mid+1, r);

    }

    void insert(line x)

    {

        bool bl=(x(l)>v(l)), bm=(x(mid)>v(mid)), br=(x(r)>v(r));

        if (bm) swap(x, v);

        if (l==r || x==v || bl==br) return;

        build();

        if (bl!=bm) lptr->insert(x);

        else rptr->insert(x);

    }

    int query(int x)

    {

        if (l==r || lptr==nullptr) return v(x);

        build();

        if (x<=mid) return max(v(x), lptr->query(x));

        else return max(v(x), rptr->query(x));

    }

} \*lct;

// Segment Tree Beats

struct SEG

{

    int l, r, mid;

    int sum, v;

    SEG \*lptr, \*rptr;

    SEG(int L, int R)

    {

        l=L; r=R; mid=(l+r)>>1;

        if (l==r)

        {

            lptr=rptr=nullptr;

            sum=v=s[l];

        }

        else

        {

            lptr=new SEG(l, mid);

            rptr=new SEG(mid+1, r);

            sum=lptr->sum+rptr->sum;

            v=max(lptr->v, rptr->v);

        }

    }

    void update(int x, int y)

    {

        if (v<10) return;

        if (l==r)

        {

            sum=v=sod(v);

            return;

        }

        if (y<=mid) lptr->update(x, y);

        else if (x>=mid+1) rptr->update(x, y);

        else

        {

            lptr->update(x, mid);

            rptr->update(mid+1, y);

        }

        sum=lptr->sum+rptr->sum;

        v=max(lptr->v, rptr->v);

    }

    int query(int pos)

    {

        if (l==r) return v;

        if (pos<=mid) return lptr->query(pos);

        else return rptr->query(pos);

    }

    int sod(int x)

    {

        string str=to\_string(x);

        int ans=0;

        for (auto i:str) ans+=(int)(i-'0');

        return ans;

    }

} \*seg;

// Segment Tree

struct SEG

{

    int l, r, mid;

    int v, lazy;

    SEG \*lptr, \*rptr;

    SEG(int L, int R)

    {

        l=L; r=R; mid=(l+r)>>1;

        lazy=0;

        if (l==r)

        {

            lptr=rptr=nullptr;

            v=s[l];

        }

        else

        {

            lptr=new SEG(l, mid);

            rptr=new SEG(mid+1, r);

            v=lptr->v+rptr->v;

        }

    }

    void update(int pos, int val)

    {

        if (l==r)

        {

            v=val;

            return;

        }

        prop();

        if (pos<=mid) lptr->update(pos, val);

        else rptr->update(pos, val);

        v=lptr->v+rptr->v;

    }

    void update(int x, int y, int val)

    {

        if (l==x && r==y)

        {

            v+=val\*(r-l+1);

            lazy+=val;

            return;

        }

        prop();

        if (y<=mid) lptr->update(x, y, val);

        else if (x>=mid+1) rptr->update(x, y, val);

        else

        {

            lptr->update(x, mid, val);

            rptr->update(mid+1, y, val);

        }

        v=lptr->v+rptr->v;

    }

    int query(int pos)

    {

        if (l==r) return v;

        prop();

        if (pos<=mid) return lptr->query(pos);

        else return rptr->query(pos);

    }

    int query(int x, int y)

    {

        if (l==x && r==y) return v;

        prop();

        if (y<=mid) return lptr->query(x, y);

        else if (x>=mid+1) return rptr->query(x, y);

        else return lptr->query(x, mid)+rptr->query(mid+1, y);

    }

    void prop()

    {

        if (lazy==0) return;

        lptr->v+=lazy\*(mid-l+1);

        lptr->lazy+=lazy;

        rptr->v+=lazy\*(r-mid);

        rptr->lazy+=lazy;

        lazy=0;

    }

} \*seg;

// Sparse Table

struct ST

{

    int st[N][L], lg[N]={0};

    ST()

    {

        for (int i=2; i<=n; i++) lg[i]=lg[i>>1]+1;

        for (int i=0; i<n; i++) st[i][0]=s[i];

        for (int j=1; (1<<j)<=n; j++)

        {

            for (int i=0; i+(1<<j)-1<n; i++)

            {

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

            }

        }

    }

    int query(int l, int r)

    {

        int i=lg[r-l+1];

        return max(st[l][i], st[r-(1<<i)+1][i]);

    }

} \*st;

// Sqrt Decomposition

struct SD

{

    int lptr[N], rptr[N];

    int v[N]={0}, lazy[N]={0};

    SD()

    {

        for (int j=0; j\*B<n; j++)

        {

            lptr[j]=j\*B;

            rptr[j]=min((j+1)\*B-1, n-1);

            for (int i=lptr[j]; i<=rptr[j]; i++) v[j]+=s[i];

        }

    }

    void update(int pos, int val)

    {

        v[pos/B]+=val-s[pos];

        s[pos]+=val-s[pos];

    }

    void update(int l, int r, int val)

    {

        int bl=l/B, br=r/B;

        if (bl==br)

        {

            for (int i=l; i<=r; i++) s[i]+=val;

            v[bl]+=val\*(r-l+1);

        }

        else

        {

            for (int j=bl+1; j<=br-1; j++)

            {

                v[j]+=val\*(rptr[j]-lptr[j]+1);

                lazy[j]+=val;

            }

            for (int i=l; i<=rptr[bl]; i++) s[i]+=val;

            v[bl]+=val\*(rptr[bl]-l+1);

            for (int i=lptr[br]; i<=r; i++) s[i]+=val;

            v[br]+=val\*(r-lptr[br]+1);

        }

    }

    int query(int pos) { return s[pos]+lazy[pos/B]; }

    int query(int l, int r)

    {

        int ans=0;

        int bl=l/B, br=r/B;

        if (bl==br) for (int i=l; i<=r; i++) ans+=s[i]+lazy[bl];

        else

        {

            for (int j=bl+1; j<=br-1; j++) ans+=v[j];

            for (int i=l; i<=rptr[bl]; i++) ans+=s[i]+lazy[bl];

            for (int i=lptr[br]; i<=r; i++) ans+=s[i]+lazy[br];

        }

        return ans;

    }

} \*sd;

// Trie

struct node

{

    int ne[N][3], idx=1;

    int sz[N];

    node() { for (int i=0; i<3; i++) fill(ne[i], ne[i]+N, -1); }

    void insert(int x)

    {

        int u=0;

        for (int i=L; i>=0; i--)

        {

            int cur=(x>>i)&1;

            if (ne[u][cur]==-1) ne[u][cur]=idx++;

            u=ne[u][cur];

            sz[u]++;

        }

    }

    int query(int x)

    {

        int u=0, ans=0;

        bool flag=1;

        for (int i=L; i>=0; i--)

        {

            int cur=(x>>i)&1, req=(m>>i)&1;

            cur^=req;

            if (req==1 && ne[u][!cur]!=-1) ans+=sz[ne[u][!cur]];

            if (ne[u][cur]==-1)

            {

                flag=0;

                break;

            }

            u=ne[u][cur];

        }

        if (flag) ans+=sz[u];

        return ans;

    }

} \*trie;

// UFDS

int find(int x)

{

    if (g[x]==x) return x;

    else return g[x]=find(g[x]);

}

void unite(int x, int y)

{

    x=find(x); y=find(y);

    if (x==y) return;

    g[y]=x;

}