LU ICPC kladīte ;)

Contents

C++ programming language

1.1. Input/Output disable sync

```
ios_base::sync_with_stdio(false);
cin.tie(NULL); cout.tie(NULL);
```

1.2. Optimization pragmas

```
// change to O3 to disable fast-math for geometry problems
#pragma GCC optimize("Ofast,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt,tune=native")
```

1.3. Printing structs

```
ostream& operator<<(ostream& os, const pair<int, int>& p) {
    return os << "(" << p.first << ", " << p.second << ")";
}
```

1.4. Lambda func for sorting

```
using ii = pair<int,int>;
vector<ii> fracs = {{1, 2}, {3, 4}, {1, 3}};
// sort positive rational numbers
sort(fracs.begin(), fracs.end(),
    [](const ii& a, const ii& b) {
    return a.fi*b.se < b.fi*a.se;
});
```

Algebra

2.1. Binary exponentiation

```
ll m_pow(ll base, ll exp, ll mod) {
    base %= mod; ll result = 1;
    while (exp > 0) {
        if (exp & 1) result = (result * base) % mod;
        base = (base * base) % mod; exp >>= 1;
    }
    return result;
}
```

2.2. Extended euclidean

Find integers x and y such that: $a \cdot x + b \cdot y = \gcd(a, b)$

```
int gcd_ext(int a, int b, int& x, int& y) {
    if (b == 0) { x = 1; y = 0; return a; }
    int x1, y1; int d = gcd_ext(b, a % b, x1, y1);
    x = y1; y = x1 - y1 * (a / b); return d;
}
```

2.3. Modular inversion & division

Mod inverse exists iff number is coprime with mod.

$$\exists x(a \cdot x \equiv 1 (\bmod m)) \Leftrightarrow \gcd(a, m) = 1$$

```
int mod_inv(int b, int m) {
    int x, y; int g = gcd_ext(b, m, &x, &y);
    if (g != 1) return -1;
    return (x%m + m) % m;
}
int m_divide(ll a, ll b, ll m) {
    int inv = mod_inv(b, m); assert(inv != -1);
    return (inv * (a % m)) % m;
}
```

2.4. Linear Diophantine equation

$$\{(x, y) \in \mathbb{Z}^2 \mid a \cdot x + b \cdot y = c\} = \{x_0 + k \cdot (b/g), y_0 - k \cdot (a/g) \mid k \in \mathbb{Z}\}$$

```
bool find_x0_y0(int a, int b, int c, int &x0, int &y0, int &g) {
    g = gcd_ext(abs(a), abs(b), x0, y0);
    if (c % g) return false;
    x0 *= c / g; y0 *= c / g;
    if (a < 0) x0 = -x0; if (b < 0) y0 = -y0;
    return true;
}
```

2.5. Linear sieve

```
const int N=10000000; vector<int> lp(N+1), pr;
for(int i=2;i<=N;i++){
    if(!lp[i]){ lp[i]=i; pr.push_back(i); }
    for(int j=0;j<pr.size() && i*pr[j]<=N;j++){
        lp[i*pr[j]]=pr[j];
        if(pr[j]==lp[i]) break;
    }
}
```

2.6. Matrix multiplication

Inherit vector's constructor to allow brace initialization.

```
struct Matrix:vector<vector<int>>{
    using vector::vector;
    Matrix operator *(const Matrix& other){
        int rows = size(); int cols = other[0].size();
        Matrix res(rows, vector<int>(cols));
        for(int i=0;i<rows;i++) for(int j=0;j<other.size();j++)
            for(int k=0;k<cols;k++)
                res[i][k]+=at(i).at(j)*other[j][k];
        return res;
    }
};
```

Usage example (prints `403 273 234 442`):

```
Matrix A = {{19,7},{6, 20}}, B = {{19,7},{6, 20}}, C = A*B;
for(auto rows: C) for(int cell: rows) cout<<cell<<" ";
```

2.7. Euler's totient function

```
int phi(int n){
    int res=n; for(int i=2;i*i<=n;i++) if(n%i==0)
        { while(n%i==0)n/=i; res-=res/i; }
    if(n>1) res-=res/n; return res;
}
void phi_1_to_n(int n){
    vector<int> phi(n+1); for(int i=0;i<=n;i++) phi[i]=i;
    for(int i=2;i<=n;i++) if(phi[i]==i)
        for(int j=i;j<=n;j+=i) phi[j]-=phi[j]/i;
}
```

2.8. Gauss method

System of n linear algebraic equations (SLAE) with m variables.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1m}x_m = b_1 \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nm}x_m = b_n \end{cases}$$

Matrix representation: $Ax = b$. Gauss-Jordan elimination impl.:

```
const double EPS=1e-9; const int INF=2;
int gauss(vector<vector<double>> a, vector<double> &ans){
    // last column of matrix a is vector b
    int n=a.size(), m=a[0].size()-1; vector<int> where(m,-1);
    for(int col=0, row=0; col<m && row<n; ++col){
        int sel=row; for(int i=row;i<n;i++)
            if(abs(a[i][col])>abs(a[sel][col])) sel=i;
        if(abs(a[sel][col])<EPS) continue;
        for(int i=col;i<=m;i++) swap(a[sel][i],a[row][i]);
        where[col]=row; for(int i=0;i<n;i++) if(i!=row){
            double c=a[i][col]/a[row][col];
            for(int j=col;j<=m;j++) a[i][j]-=a[row][j]*c;} row++;
    }
    ans.assign(m,0); for(int i=0;i<m;i++)
        if(where[i]!=-1) ans[i]=a[where[i]][m]/a[where[i]][i];
    for(int i=0;i<n;i++){
        double sum=0; for(int j=0;j<m;j++) sum+=ans[j]*a[i][j];
        if(abs(sum-a[i][m])>EPS) return 0;}
    for(int i=0;i<m;i++) if(where[i]==-1) return INF; return 1;
}
```

2.9. FFT

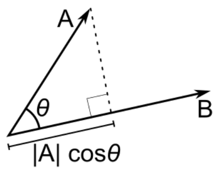
```
const int N=1<<18;
const ld PI=acos(-1.0);
struct T{
    ld x,y;
    T():x(0),y(0){}
    T(ld a, ld b=0):x(a),y(b){}
    T operator/=(ld k){x/=k;y/=k;return *this;}
    T operator*(const T&a) const {
        return T(x*a.x-y*a.y, x*a.y+y*a.x);}
    T operator+(const T&a) const {return T(x+a.x, y+a.y);}
    T operator-(const T&a) const {return T(x-a.x, y-a.y);}
};
void fft(T*a,int n,int s){
    for(int i=0,j=0;i<n;i++){
        if(i>j) swap(a[i],a[j]);
        for(int l=n/2;(j^=l)<l;l>>=1);
    }
    for(int i=1;(1<<i)<=n;i++){
        int M=1<<i, K=M>>1;
        T wn= T(cos(s*2*PI/M), sin(s*2*PI/M));
        for(int j=0;j<n;j+=M){
            T w=1;
            for(int l=j;l<j+K;l++){
                T t=w*a[l+K];
                a[l+K]=a[l]-t; a[l]=a[l]+t;
                w=wn*w;
            }
        }
    }
}
void multiply(T*a,T*b,int n){
    while(n&(n-1)) n++;
    fft(a,n,1); fft(b,n,1);
    for(int i=0;i<n;i++) a[i]=a[i]*b[i];
    fft(a,n,-1);
    for(int i=0;i<n;i++) a[i]/=n;
}
int main(){
    T a[10]={T(2),T(3)}, b[10]={T(1),T(-1)};
    multiply(a,b,4);
    for(int i=0;i<10;i++) std::cout<<int(a[i].x)<<" ";
}
```

2.10. Fast binomial coefficient

```
int MAX_CHOOSE=3e5;
vector<ll> inv_fact(MAX_CHOOSE+5), fact(MAX_CHOOSE+5);
ll fast_nCr(ll n, ll r){
    if(n<r || r<0) return 0;
    return fact[n]*inv_fact[r]%mod*inv_fact[n-r]%mod;
}
void precalc_fact(int n){
    fact[0]=fact[1]=1;
    for(ll i=2;i<=n;i++) fact[i]=(fact[i-1]*i)%mod;
    inv_fact[0]=inv_fact[1]=1;
    for(ll i=2;i<=n;i++)
    inv_fact[i]=(mod_inv(i,mod)*inv_fact[i-1])%mod;
}
```

Geometry

3.1. Dot product



$$a \cdot b = |a|\ |b| \cos(\theta)$$

$$a \cdot b = a_x b_x + a_y b_y$$

$$\theta = \arccos\left(\frac{a_x b_x + a_y b_y}{|a|\ |b|}\right)$$

Projection of a onto b: $\frac{a \cdot b}{|b|}$

3.2. Cross product

$$a \times b = |a|\ |b| \sin(\theta) = a_x b_y - a_y b_x$$

θ is positive if a is clockwise from b

3.3. Line-point distance

Line given by $ax + by + c = 0$ and point (x_0, y_0).

$$\text{distance} = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

The coordinates of this point are:

$$x = \frac{b(bx_0 - ay_0) - ac}{a^2 + b^2} \qquad y = \frac{a(-bx_0 + ay_0) - bc}{a^2 + b^2}$$

3.4. Shoelace formula

$$2A = \sum_{i=1}^{n} \begin{vmatrix} x_i & y_i \\ x_{i+1} & y_{i+1} \end{vmatrix}, \text{where} \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

3.5. Circumradius

Let a, b, c be the sides of a triangle and A the area of the triangle. Then the circumradius $R = abc/(4A)$. Alternatively, using the Law of Sines:

$$R = \frac{a}{2\sin(\alpha)} = \frac{b}{2\sin(\beta)} = \frac{c}{2\sin(\gamma)}$$

where α, β, and γ are the angles opposite sides a, b, and c respectively.

3.6. Law of Sines

In any triangle with sides a, b, c and opposite angles α, β, γ respectively:

$$\frac{a}{\sin(\alpha)} = \frac{b}{\sin(\beta)} = \frac{c}{\sin(\gamma)} = 2R$$

where R is the circumradius of the triangle. This can be rearranged to find any side or angle: $a = 2R\sin(\alpha)$ and $\sin(\alpha) = \frac{a}{2R}$

3.7. Law of Cosines

In any triangle with sides a, b, c and opposite angles α, β, γ respectively:

$$c^2 = a^2 + b^2 - 2ab\cos(\gamma)$$

3.8. Median Length Formulas

In any triangle with sides a, b, c, the lengths of the medians m_a, m_b, m_c from the respective vertices are given by:

$$m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$$

These formulas can be derived using the Apollonius's theorem.

3.9. Segment to line linear equation

Converting segment $\big((P_x, P_y), (Q_x, Q_y)\big)$ to $Ax + By + C = 0$:

$$(P_y - Q_y)x + (Q_x - P_x)y + (P_x Q_y - P_y Q_x) = 0$$

3.10. Three point orientation

```
int orientation(Point p1, Point p2, Point p3){
    int val = (p2.y-p1.y)*(p3.x-p2.x)-(p2.x-p1.x)*(p3.y-p2.y);
    if (val == 0) return 0; // collinear
    return (val > 0) ? 1 : 2; // clock or counterclock
}
```

3.11. Line-line intersection

From system of linear equations derived Cramer's rule:

$$\begin{cases} a_1x + b_1y = c_1 \\ a_2x + b_2y = c_2 \end{cases} \Rightarrow \begin{cases} x = (c_1b_2 - c_2b_1)/(a_1b_2 - a_2b_1) \\ y = (a_1c_2 - a_2c_1)/(a_1b_2 - a_2b_1) \end{cases}$$

If the denominator equals zero, the lines are parallel or coincident.

3.12. Check if two segments intersect

```
bool on_seg(Point p, Point q, Point r) {
    return (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
}
bool do_intersect(Point p1, Point q1, Point p2, Point q2){
    int o1 = orient(p1, q1, p2), o2 = orient(p1, q1, q2);
    int o3 = orient(p2, q2, p1), o4 = orient(p2, q2, q1);
    if (o1 != o2 && o3 != o4) return true;
    return (o1==0&&on_seg(p1,p2,q1))||(o2==0&&on_seg(p1,q2,q1))||
           (o3==0&&on_seg(p2,p1,q2))||(o4==0&&on_seg(p2,q1,q2));
}
```

3.13. Heron's formula

Let a, b, c - sides of a triangle. Then the area A is:

$$A = \frac{1}{4}\sqrt{(a+b+c)(-a+b+c)(a-b+c)(a+b-c)}$$

Numerically stable version:

$$a \ge b \ge c, A = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}$$

3.14. Graham's scan

Constructs convex hull of a set of points.

```
void convex_hull(vector<pt>&a, bool coll=false){
    pt p = *min_element(all(a), [](const pt&x, const pt&y){
        return make_pair(x.y, x.x) < make_pair(y.y, y.x);
    });
    sort(all(a), [](const pt&x, const pt&y){
        int ori = orientation(p, x, y);
        if(ori == 0)
            return (p.x-x.x)*(p.x-x.x) + (p.y-x.y)*(p.y-x.y) <
                   (p.x-y.x)*(p.x-y.x) + (p.y-y.y)*(p.y-y.y);
        return ori < 0;
    });
    if(coll){
        int i = a.size()-1; while(i>=0 &&
            collinear(a[i], a.back(), p)) i--;
        reverse(a.b()+i+1, a.e());
    }
    vector<pt> s;
    for(auto &p : a){
        while(s.size() > 1 &&
            !cw(s[s.size()-2], s.back(), p, coll)) s.pop_back();
        s.push_back(p);
    }
    a = s;
}
```

3.15. Closest pair of points

Finds pair of points with minimum euclidean distance.

```
struct pt { ld x, y; int id; };
ld md; pair<int,int> bp; vector<pt> a, t;
void ua(const pt&a1, const pt&b1){ // update answer
    ld d=sqrtl((a1.x-b1.x)*(a1.x-b1.x)+(a1.y-b1.y)*(a1.y-b1.y));
    if(d<md){md=d; bp={a1.id,b1.id};}
}
void rec(int l, int r){ // recursive function
    if(r - l <= 3){
        rep(i, l, r) rep(j, i+1, r) ua(a[i], a[j]);
        sort(a.b()+l, a.b()+r,
            [](const pt&x, const pt&y){return x.y<y.y;});
        return;}
    int m = (l + r) >> 1, midx = a[m].x;
    rec(l, m); rec(m, r);
    merge(a.b()+l, a.b()+m, a.b()+m, a.b()+r, t.b(),
        [](const pt&x, const pt&y){return x.y<y.y;});
    copy(t.b(), t.b()+r-l, a.b()+l);
    int ts = 0; rep(i, l, r) if(abs(a[i].x - midx) < md){
        for(int j=ts-1;j>=0&&a[i].y-t[j].y<md;j--)ua(a[i],t[j]);
        t[ts++] = a[i];
    }
}
```

Data structures

4.1. Treap

```
struct Node{
    int value, cnt, pri; Node *left, *right;
    Node(int p) : value(p), cnt(1), pri(gen()),
        left(NULL), right(NULL) {};
};
typedef Node* pnode;
int get(pnode q){if(!q) return 0; return q->cnt;}
void update_cnt(pnode &q){
    if(!q) return; q->cnt=get(q->left)+get(q->right)+1;
}
void merge(pnode &T, pnode lef, pnode rig){
    if(!lef){T=rig;return;} if(!rig){T=lef;return;}
    if(lef->pri>rig->pri){merge(lef->right,lef->right,rig);T=lef;
    }else{merge(rig->left, lef, rig->left); T = rig;}
    update_cnt(T);
}
void split(pnode cur, pnode &lef, pnode &rig, int key){
    if(!cur){lef=rig=NULL;return;} int id=get(cur->left)+1;
    if(id<=key){split(cur->right,cur->right,rig,key-id);lef=cur;}
    else {split(cur->left, lef, cur->left, key); rig = cur;}
    update_cnt(cur);
}
```

4.2. Lazy segment tree

```
struct SumSegmentTree{
    vector<ll> S, O, L; // S: segment tree, O: original, L: lazy
    void build(ll ti, ll tl, ll tr){
        if(tl==tr){S[ti]=O[tl]; return;}
        build(ti*2,tl,(tl+tr)/2);build((ti*2)+1,(tl+tr)/2+1,tr);
        S[ti]=S[ti*2]+S[(ti*2)+1];
    }
    void push(ll ti, ll tl, ll tr){
        S[ti] += L[ti]*(tr-tl+1); if(tl==tr){L[ti]=0;return;}
        L[ti+ti] += L[ti],L[ti+ti+1] += L[ti]; L[ti] = 0;
    }
    ll query(ll ti, ll tl, ll tr, ll i, ll j){
        push(ti, tl, tr);
        if(i<=tl&&tr<=j) return S[ti]; if(tr<i||tl>j) return 0;
        ll a = query(ti*2, tl, (tl+tr)/2, i, j);
        ll b = query((ti*2)+1, ((tl+tr)/2)+1,tr, i, j);
        return a+b;
    }
    void update(ll ti, ll tl, ll tr, ll i, ll j, ll v){
        if(i<=tl&&tr<=j){L[ti]+=v;return;}
        if(tr<i||tl>j) return; S[ti]+=v*(i-j+1);
        update(ti*2,tl,(tl+tr)/2,i,j,v);
        update((ti*2)+1,(tl+tr)/2+1,tr,i,j,v);
    };
    ST(vector<ll> &V){
        O = V; S.resize(O.size()*4, 0); L.resize(O.size()*4, 0);
        build(1, 0, O.size()-1);
    }
};
```

4.3. Sparse table

```
const int N, M; //M=log2(N)
int sparse[N][M];
void build() {
  for(int i = 0; i < n; i++) sparse[i][0] = v[i];
  for(int j = 1; j < M; j++) for(int i = 0; i < n; i++)
    sparse[i][j] = i + (1 << j - 1) < n
      ? min(sparse[i][j - 1], sparse[i + (1 << j - 1)][j - 1])
      : sparse[i][j - 1];
}
int query(int a, int b){
  int pot = 32 - __builtin_clz(b - a) - 1;
  return min(sparse[a][pot], sparse[b - (1 << pot) + 1][pot]);
}
```

4.4. Fenwick tree

```
struct FenwickTree {
    int n;vector<ll> bit; // binary indexed tree
    FenwickTree(int n) {this->n=n;bit.assign(n, 0);}
    ll sum(int r) {
        ll ret=0;
        for(;r>=0;r=(r&(r+1))-1) ret+=bit[r];
        return ret;
    }
    ll sum(int l, int r){return sum(r)-sum(l-1);}
    void add(int idx, ll delta){
        for(;idx<n;idx=idx|(idx+1))bit[idx]+=delta;
    }
};
```

4.5. Trie

```
const int K = 26;

struct Vertex {
    int next[K];
    bool output = false;
    Vertex() {fill(begin(next), end(next), -1);}
};

vector<Vertex> t(1); // trie nodes

void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back();
        }
        v = t[v].next[c];
    }
    t[v].output = true;
}
```

4.6. Aho-Corasick

```
const int K = 26;
struct V {
    int n[K], go[K], p = -1; // next, go transitions, parent
    char ch;                 // char from parent to this node
    bool out = false;        // is end of a pattern
    int l = -1, d = -1, e = -1; // fail link, depth, exit length
    V(int parent = -1, char c = '$') : p(parent), ch(c) {
        fill(n, n + K, -1);   // initialize transitions
        fill(go, go + K, -1); // initialize go transitions
    }
};
vector<V> t(1);
void add_string(const string& s){ // Add a string to the trie
    int v = 0;
    for(char c : s){
        int ci = c - 'a';
        if(t[v].n[ci] == -1){
            t[v].n[ci] = t.size();
            t.emplace_back(v, c); // create new node
        }
        v = t[v].n[ci];
    }
    t[v].out = true; // mark end of pattern
}
int go_func(int v, char c);
int get_link(int v){ // Get the fail link for node v
    if(t[v].l == -1){
        if(v == 0 || t[v].p == 0) t[v].l = 0;
        else t[v].l = go_func(get_link(t[v].p), t[v].ch);
    } return t[v].l;
}
// Compute the transition for node v with character c
int go_func(int v, char c){
    int ci = c - 'a';
    if(t[v].go[ci] == -1){
        if(t[v].n[ci] != -1) t[v].go[ci] = t[v].n[ci];
        else t[v].go[ci] = (v==0) ? 0 : go_func(get_link(v),c);
    }
    return t[v].go[ci];
}
```

4.7. Disjoint Set Union

```
struct DSU {
    vector<int> p, r; // p: parent, r: rank
    DSU(int n) {
        p.resize(n); r.resize(n);
        for (int i = 0; i < n; i++) p[i] = i;
    }
    int f(int a){if (p[a] == a) return a; return p[a] = f(p[a]);}
    void unite(int a, int b) {
        a = f(a), b = f(b); if (a == b) return;
        if (r[a] < r[b]) p[a] = b; else if (r[a] > r[b]) p[b] = a;
        else {p[b] = a; r[a]++;}
    }
};
```

4.8. Merge sort tree

```
struct MergeSortTree{
    int size; vector<vector<ll>> values;
    void init(int n){
        size=1; while(size<n) size*=2;
        values.resize(size*2, vector<ll>());
    }
    void build(vector<ll> &arr, int x, int lx, int rx){
        if(rx-lx==1){
            if(lx<arr.size()) values[x].push_back(arr[lx]);
            else values[x].push_back(-1);
            return;
        }
        int m=(lx+rx)/2;
        build(arr,2*x+1,lx,m);
        build(arr,2*x+2,m,rx);
        int i=0, j=0, asize=values[2*x+1].size();
        while(i<asize && j<values[2*x+2].size()){
            if(values[2*x+1][i]<values[2*x+2][j])
              values[x].push_back(values[2*x+1][i++]);
            else values[x].push_back(values[2*x+2][j++]);
        }
        while(i<asize)
          values[x].push_back(values[2*x+1][i++]);
        while(j<values[2*x+2].size())
          values[x].push_back(values[2*x+2][j++]);
    }
    void build(vector<ll> &arr){ build(arr,0,0,size); }
    int calc(int l, int r, int x, int lx, int rx, int k){
        if(lx>=r || rx<=l) return 0;
        if(lx>=l && rx<=r){
            int lft=-1, rght=values[x].size();
            while(rght-lft>1){
                int mid=(lft+rght)/2;
                if(values[x][mid]<k) lft=mid;
                else rght=mid;
            }
            return lft+1;
        }
        int m=(lx+rx)/2;
        return calc(l,r,2*x+1,lx,m,k) + calc(l,r,2*x+2,m,rx,k);
    }
    int calc(int l, int r, int k){ return calc(l,r,0,0,size,k); }
};
```

4.9. janY mass operations segment tree

```
struct item { ll x; item(ll x=0) : x(x) {} };
struct segtree {
    int size; vector<item> values, ops;
    item NEUTRAL=0, DEFAULT=0, NOOP=0;
    item modify_op(item a, item b, ll len) {
        a.x += b.x*len; return a; }
    void apply_mod_op(item &a, item b, ll len) {
        a = modify_op(a, b, len); }
    item calc_op(item a, item b) { return item(a.x + b.x); }
    void init(int n) {
        size=1; while(size<n) size<<=1;
        values.assign(size<<1, DEFAULT);
        ops.assign(size<<1, NOOP);
    }
    void build(vector<item> &arr, int x=0, int lx=0, int rx=-1) {
        if(rx==-1) rx = size;
        if(rx - lx ==1) {
            values[x] =
                (lx < arr.size()) ? arr[lx] : NEUTRAL; return;
        }
        int m=(lx+rx)/2;
        build(arr,2*x+1,lx,m);
        build(arr,2*x+2,m,rx);
        values[x] = calc_op(values[2*x+1], values[2*x+2]);
    }
    void propagate(int x, int lx, int rx) {
        if(rx - lx ==1) return;
        int m=(lx+rx)/2;
        apply_mod_op(ops[2*x+1], ops[x],1);
        apply_mod_op(values[2*x+1], ops[x],m-lx);
        apply_mod_op(ops[2*x+2], ops[x],1);
        apply_mod_op(values[2*x+2], ops[x],rx-m);
        ops[x] = NOOP;
    }
    void set(int l, int r, ll v, int x=0, int lx=0, int rx=-1) {
        if(rx==-1) rx = size; propagate(x, lx, rx);
        if(lx >= r || rx <= l) return;
        if(lx >= l && rx <= r) {
            apply_mod_op(ops[x], item(v),1);
            apply_mod_op(values[x], item(v), rx-lx); return;
        }
        int m=(lx+rx)/2;
        set(l,r,v,2*x+1,lx,m); set(l,r,v,2*x+2,m,rx);
        values[x] = calc_op(values[2*x+1], values[2*x+2]);
    }
    item calc(int l, int r, int x=0, int lx=0, int rx=-1) {
        if(rx==-1) rx = size; propagate(x, lx, rx);
        if(lx >= r || rx <= l) return NEUTRAL;
        if(lx >= l && rx <= r) return values[x];
        int m=(lx+rx)/2;
        return
            calc_op(calc(l,r,2*x+1,lx,m), calc(l,r,2*x+2,m,rx));
    }
};
```

4.10. janY fenwick tree range update

```
struct fenwick { // range update
    ll *bit1, *bit2; int fsize;
    void init(int n){
        fsize=n; bit1=new ll[n+1](); bit2=new ll[n+1]();
    }
    ll getSum(ll BIT[], int i){
        ll s=0; i++; while(i>0){ s += BIT[i]; i -= i & -i; }
        return s;
    }
    void updateBIT(ll BIT[], int i, ll v){
        i++; while(i <= fsize){ BIT[i] += v; i += i & -i; }
    }
    ll sum(int x){
        return getSum(bit1,x)*x - getSum(bit2,x);
    }
    void add(int l, int r, ll v){
        updateBIT(bit1,l,v); updateBIT(bit1,r+1,-v);
        updateBIT(bit2,l,v*(l-1)); updateBIT(bit2,r+1,-v*r);
    }
    ll calc(int l, int r){
        return sum(r) - sum(l-1);
    }
};
```

4.11. Persistent segment tree

```
#define V struct Vertex
struct Vertex { V *l, *r; ll sum;
    Vertex(ll val){l=r=nullptr; sum=val;}
    Vertex(V* le, V* ri){l=le;r=ri;sum=(l?l->sum:0)+(r?r->sum:0);}
};
int siz;
vector<V*> start_nodes;
V* build(int lx, int rx, vl &a){
    if (lx == rx-1) return new V(a[lx]);
    return new V(build(lx,(lx+rx)/2,a),build((lx+rx)/2,rx,a));
}
V* build(vl &a){ siz = a.size(); return build(0, siz, a); }
ll calc(V* v, int lx, int rx, int l, int r){
    if(lx >= r || rx <= l) return 0;
    if(lx >= l && rx <= r) return v->sum;
    int m = (lx + rx) / 2;
    return calc(v->l, lx, m, l, r) + calc(v->r, m, rx, l, r);
}
ll calc(V* v, int l, int r){
    if (l>r) return 0;
    return calc(v,0,siz,l,r);
}
V* upd(V* v, int lx, int rx, int i, ll val){
    if(lx == rx-1) return new V(val);
    int m = (lx + rx) / 2;
    if (i < m) return new V(upd(v->l, lx, m, i, val), v->r);
    else return new V(v->l, upd(v->r, m, rx, i, val));
}
V* upd(V* v, int i, ll val){ return upd(v, 0, siz, i, val); }
```

Graph algorithms

5.1. Bellman-Ford

```
void solve()
{
    vector<int> d(n, INF);
    d[v] = 0;
    for (;;) {
        bool any = false;

        for (Edge e : edges)
            if (d[e.a] < INF)
                if (d[e.b] > d[e.a] + e.cost) {
                    d[e.b] = d[e.a] + e.cost;
                    any = true;
                }

        if (!any)
            break;
    }
    // display d, for example, on the screen
}
```

5.2. Dijkstra

```
vector<int> adj[N], adjw[N];
int dist[N];

memset(dist, 63, sizeof(dist));
priority_queue<pii> pq;
pq.push(mp(0,0));

while (!pq.empty()) {
  int u = pq.top().nd;
  int d = -pq.top().st;
  pq.pop();

  if (d > dist[u]) continue;
  for (int i = 0; i < adj[u].size(); ++i) {
    int v = adj[u][i];
    int w = adjw[u][i];
    if (dist[u] + w < dist[v])
      dist[v] = dist[u]+w, pq.push(mp(-dist[v], v));
  }
}
```

5.3. Floyd-Warshall

```
int adj[N][N]; // no-edge = INF

for (int k = 0; k < n; ++k)
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      adj[i][j] = min(adj[i][j], adj[i][k]+adj[k][j]);
```

5.4. Bridges & articulations

```
// Articulation points and Bridges O(V+E)
int par[N], art[N], low[N], num[N], ch[N], cnt;
```

```
void articulation(int u) {
  low[u] = num[u] = ++cnt;
  for (int v : adj[u]) {
    if (!num[v]) {
      par[v] = u; ch[u]++;
      articulation(v);
      if (low[v] >= num[u]) art[u] = 1;
      if (low[v] >  num[u]) { /* u-v bridge */ }
      low[u] = min(low[u], low[v]);
    }
    else if (v != par[u]) low[u] = min(low[u], num[v]);
  }
}

for (int i = 0; i < n; ++i) if (!num[i])
  articulation(i), art[i] = ch[i]>1;
```

5.5. Dinic's max flow / matching

Time complexity:

- generally: $O(EV^2)$
- small flow: $O(F(V+E))$
- bipartite graph or unit flow: $O\left(E\sqrt{V}\right)$

Usage:

- dinic()
- add_edge(from, to, capacity)
- recover() (optional)

```
const ll N=1e5+5, INF=1e9;
struct edge{ll v, c, f;};

ll src=0, snk=N-1, h[N], ptr[N];
vector<edge> edgs;

vector<ll> g[N];

void add_edge(ll u, ll v, ll c) {
    edgs.push_back({v,c,0}), edgs.push_back({u,0,0});
    ll k=edgs.size();
    g[u].push_back(k), g[v].push_back(k+1);
}

bool bfs() {
    memset(h, 0, sizeof(h));
    queue<ll> q;
    h[src]=1;
    q.push(src);
    while(!q.empty()){
        ll u=q.front();q.pop();
        for(ll i:g[u]){
            ll v=edgs[i].v;
            if(!h[v]&&edgs[i].f<edgs[i].c)
                q.push(v),h[v]=h[u]+1;
        }
    }
    return h[snk];
}

ll dfs(ll u, ll flow){
    if(!flow or u==snk) return flow;
    for(ll &i=ptr[u];i<g[u].size();i++){
        edge &dir=edgs[g[u][i]],&rev=edgs[g[u][i]^1];
        if(h[dir.v]!=h[u]+1) continue;
        ll inc=min(flow,dir.c-dir.f);
        inc=dfs(dir.v,inc);
        if(inc){ dir.f+=inc,rev.f-=inc; return inc;}
    }
    return 0;
}

ll dinic(){
    ll flow=0;
    while(bfs()){
        memset(ptr,0,sizeof(ptr));
        while(ll inc=dfs(src,INF)) flow += inc;
    }
    return flow;
}

vector<pair<ii,ll>> recover() {
    vector<pair<ii,ll>> res;
    for(ll i=0;i<edgs.size();i+=2){
        if(edgs[i].f>0){
            ll v=edgs[i].v, u=edgs[i^1].v;
            res.push_back({{u,v},edgs[i].f});
        }
    }
    return res;
}
```

5.6. Flow with demands

Finding an arbitrary flow

- Assume a network with $[L;R]$ on edges (some may have $L=0$), let's call it old network.
- Create a New Source and New Sink (this will be the src and snk for Dinic).
- Modelling network:
  1. Every edge from the old network will have cost $R-L$
  2. Add an edge from New Source to every vertex v with cost:
     - $S(L)$ for every (u,v). (sum all L that LEAVES v)
  3. Add an edge from every vertex v to New Sink with cost:
     - $S(L)$ for every (v,w). (sum all L that ARRIVES v)
  4. Add an edge from Old Source to Old Sink with cost INF (circulation problem)
- The Network will be valid if and only if the flow saturates the network (max flow == $S(L)$)

Finding Min Flow

- To find min flow that satisfies just do a binary search in the (Old Sink -> Old Source) edge
- The cost of this edge represents all the flow from old network
- Min flow = $S(L)$ that arrives in Old Sink + flow that leaves (Old Sink -> Old Source)

5.7. Kosaraju's algorithm

```
const int N = 2e5 + 5;

vector<int> adj[N], adjt[N];
int n, ordn, scc_cnt, vis[N], ord[N], scc[N];

//Directed Version
void dfs(int u) {
  vis[u] = 1;
  for (auto v : adj[u]) if (!vis[v]) dfs(v);
  ord[ordn++] = u;
}

void dfst(int u) {
  scc[u] = scc_cnt, vis[u] = 0;
  for (auto v : adjt[u]) if (vis[v]) dfst(v);
}

// add edge: u -> v
void add_edge(int u, int v){
  adj[u].push_back(v);
  adjt[v].push_back(u);
}

// run kosaraju
void kosaraju(){
  for (int i = 1; i <= n; ++i) if (!vis[i]) dfs(i);
  for (int i = ordn - 1; i >= 0; --i) if (vis[ord[i]]) scc_cnt++,
dfst(ord[i]);
}
```

5.8. Lowest Common Ancestor

```
const int N = 1e6, M = 25;
int anc[M][N], h[N], rt;

// TODO: Calculate h[u] and set anc[0][u] = parent of node u for
each u

// build (sparse table)
anc[0][rt] = rt; // set parent of the root to itself
for (int i = 1; i < M; ++i)
  for (int j = 1; j <= n; ++j)
    anc[i][j] = anc[i-1][anc[i-1][j]];

// query
int lca(int u, int v) {
  if (h[u] < h[v]) swap(u, v);
  for (int i = M-1; i >= 0; --i) if (h[u]-(1<<i) >= h[v])
    u = anc[i][u];

  if (u == v) return u;

  for (int i = M-1; i >= 0; --i) if (anc[i][u] != anc[i][v])
    u = anc[i][u], v = anc[i][v];
  return anc[0][u];
}
```

5.9. General matching in a graph

```
vector<int> Blossom(vector<vector<int>> graph){
  int n = graph.size();
  int timer = -1;
  vector<int> mate(n, -1), label(n), parent(n),
              orig(n), aux(n, -1), q;
  auto lca = [&](int x, int y) {
    for (timer++; ; swap(x, y)) {
      if (x == -1) continue;
      if (aux[x] == timer) return x;
      aux[x] = timer;
      x = (mate[x] == -1 ? -1 : orig[parent[mate[x]]]);
    }
  };
  auto blossom = [&](int v, int w, int a) {
    while (orig[v] != a) {
      parent[v] = w; w = mate[v];
      if (label[w] == 1) label[w] = 0, q.push_back(w);
      orig[v] = orig[w] = a; v = parent[w];
    }
  };
  auto augment = [&](int v) {
    while (v != -1) {
      int pv = parent[v], nv = mate[pv];
      mate[v] = pv; mate[pv] = v; v = nv;
    }
  };
  auto bfs = [&](int root) {
    fill(label.begin(), label.end(), -1);
    iota(orig.begin(), orig.end(), 0);
    q.clear();
    label[root] = 0; q.push_back(root);
    for (int i = 0; i < (int)q.size(); ++i) {
      int v = q[i];
      for (auto x : graph[v]) {
        if (label[x] == -1) {
          label[x] = 1; parent[x] = v;
          if (mate[x] == -1)
            return augment(x), 1;
          label[mate[x]] = 0; q.push_back(mate[x]);
        } else if (label[x] == 0 && orig[v] != orig[x]) {
          int a = lca(orig[v], orig[x]);
          blossom(x, v, a); blossom(v, x, a);
        }
      }
    }
    return 0;
  };
  for (int i = 0; i < n; i++)
    if (mate[i] == -1)
      bfs(i);
  return mate;
}
```

String Processing

6.1. Knuth-Morris-Pratt (KMP)

```
char s[N], p[N]; int b[N], n, m; // n = strlen(s), m = strlen(p);
void kmppre() {
  b[0] = -1; for (int i = 0, j = -1; i < m; b[++i] = ++j)
    while (j >= 0 and p[i] != p[j]) j = b[j];
}
void kmp() {
  for (int i = 0, j = 0; i < n;) {
    while (j >= 0 and s[i] != p[j]) j=b[j];
    i++, j++; if (j == m) {j = b[j];}
  }
}
```

6.2. Suffix Array

```
vector<int> suffix_array(string &s){
    int n = s.size(), alph = 256;
    vector<int> cnt(max(n, alph)), p(n), c(n);
    rep(i,0,n) cnt[s[i]]++;
    rep(i,1,alph) cnt[i]+=cnt[i-1];
    rep(i,0,n) p[--cnt[s[i]]] = i;
    c[p[0]]=0; rep(i,1,n)
    c[p[i]] = (s[p[i]] != s[p[i-1]]) ? c[p[i-1]] + 1 : c[p[i-1]];
    vector<int> p2(n), c2(n);
    for(int k=0; (1<<k)<n; k++){
        rep(i,0,n) p2[i]=(p[i]-(1<<k)+n)%n;
        fill(cnt.begin(), cnt.begin()+c[p[n-1]]+1, 0);
        rep(i,0,n) cnt[c[p2[i]]]++;
        rep(i,1,c[p[n-1]]+1) cnt[i]+=cnt[i-1];
        for(int i=n-1;i>=0;i--) p[--cnt[c[p2[i]]]] = p2[i];
        c2[p[0]]=0; rep(i,1,n) {
            pair<int,int> a1 = {c[p[i]], c[(p[i]+(1<<k))%n]};
            pair<int,int> a2 = {c[p[i-1]], c[(p[i-1]+(1<<k))%n]};
            c2[p[i]] = (a1 != a2) ? c2[p[i-1]] + 1 : c2[p[i-1]];
        }
        c.swap(c2);
    }
    return p;
}
```

6.3. Longest common prefix (LCP) with SA

```
vector<int> lcp(string &s, vector<int> &p){
    int n = s.size(); vector<int> pi(n), ans(n-1);
    rep(i,0,n) pi[p[i]] = i;
    int lst = 0;
    rep(i,0,n-1){
        if(pi[i] == n-1){ lst = 0; continue; }
        int j = p[pi[i]+1];
        while(i+lst<n && j+lst<n && s[i+lst] == s[j+lst]) lst++;
        ans[pi[i]] = lst; lst = max(lst-1, 0);
    }
    return ans;
}
```

6.4. Rabin-Karp pattern match with hashing

```
const int B = 31;
const int MOD = 1e9+7, B = 31;
void rabin(string s, string p){
    int n = s.size(), m = p.size(); if(n<m) return;
    vector<ull> power(max(n, m), 1);
    rep(i,1,power.size()) power[i] = (power[i-1]*B)%MOD;
    ull hp=0, hs=0;
    rep(i,0,m){ hp=(hp*B + p[i])%MOD; hs=(hs*B + s[i])%MOD; }
    if(hs == hp) { /* match at 0 */ }
    rep(i,m,n){
        hs = (hs*B + s[i])%MOD;
        hs = (hs + MOD - (s[i-m]*power[m])%MOD)%MOD;
        if(hs == hp) { /* match at i-m+1 */ }
    }
}
```

6.5. Z-function

The Z-function of a string s is an array z where z_i is the length of the longest substring starting from s_i which is also a prefix of s.

Examples:

- "aaaaa": $[0, 4, 3, 2, 1]$
- "aaabaab": $[0, 2, 1, 0, 2, 1, 0]$
- "abacaba": $[0, 0, 1, 0, 3, 0, 1]$

```
vector<int> zfunction(const string& s){
  vector<int> z (s.size());
  for (int i = 1, l = 0, r = 0, n = s.size(); i < n; i++){
    if (i <= r) z[i] = min(z[i-l], r - i + 1);
    while (i + z[i] < n and s[z[i]] == s[z[i] + i]) z[i]++;
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

6.6. Manacher's longest palindromic substring

```
int manacher(string s){
    int n = s.size(); string p = "^#";
    rep(i,0,n) p += string(1, s[i]) + "#";
    p += "$"; n = p.size(); vector<int> lps(n, 0);
    int C=0, R=0, m=0;
    rep(i,1,n-1){
        int mirr = 2*C - i;
        if(i < R) lps[i] = min(R-i, lps[mirr]);
        while(p[i + 1 + lps[i]] == p[i - 1 - lps[i]]) lps[i]++;
        if(i + lps[i] > R){ C = i; R = i + lps[i]; }
        m = max(m, lps[i]);
    }
    return m;
}
```

Dynamic programming

7.1. Convex hull trick

```
// Convex Hull Trick

// ATTENTION: This is the maximum convex hull. If you need the minimum
// CHT use {-b, -m} and modify the query function.

// In case of floating point parameters swap long long with long double
typedef long long type;
struct line { type b, m; };

line v[N]; // lines from input
int n; // number of lines
// Sort slopes in ascending order (in main):
sort(v, v+n, [](line s, line t){
     return (s.m == t.m) ? (s.b < t.b) : (s.m < t.m); });

// nh: number of lines on convex hull
// pos: position for linear time search
// hull: lines in the convex hull
int nh, pos;
line hull[N];

bool check(line s, line t, line u) {
  // verify if it can overflow. If it can just divide using long double
  return (s.b - t.b)*(u.m - s.m) < (s.b - u.b)*(t.m - s.m);
}

// Add new line to convex hull, if possible
// Must receive lines in the correct order, otherwise it won't work
void update(line s) {
  // 1. if first lines have the same b, get the one with bigger m
  // 2. if line is parallel to the one at the top, ignore
  // 3. pop lines that are worse
  // 3.1 if you can do a linear time search, use
  // 4. add new line

  if (nh == 1 and hull[nh-1].b == s.b) nh--;
  if (nh > 0  and hull[nh-1].m >= s.m) return;
  while (nh >= 2 and !check(hull[nh-2], hull[nh-1], s)) nh--;
  pos = min(pos, nh);
  hull[nh++] = s;
}

type eval(int id, type x) { return hull[id].b + hull[id].m * x; }

// Linear search query - O(n) for all queries
// Only possible if the queries always move to the right
type query(type x) {
  while (pos+1 < nh and eval(pos, x) < eval(pos+1, x)) pos++;
  return eval(pos, x);
  // return -eval(pos, x);    ATTENTION: Uncomment for minimum CHT
}
```

7.2. Online Convex Hull Trick

```
// Source: KTH notebook

struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) return x->p = inf, 0;
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), 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));
  }
  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

7.3. Longest Increasing Subsequence

```
memset(dp, 63, sizeof dp);
for (int i = 0; i < n; ++i) {
  // increasing: lower_bound
  // non-decreasing: upper_bound
  int j = lower_bound(dp, dp + lis, v[i]) - dp;
  dp[j] = min(dp[j], v[i]); lis = max(lis, j + 1);
}
```

7.4. SOS DP (Sum over Subsets)

Sum over subsets (SOS) DP trick efficiently computes the sum of all the subsets of an array with time complexity $O(\text{bits} \cdot 2^{\text{bits}})$.

```
const int bits = 20;
vector<int> a(1<<bits); // initial value of each subset
vector<int> f(1<<bits); // sum over all subsets
// (at f[011] = a[011]+a[001]+a[010]+a[000])
for (int i = 0; i<(1<<bits); i++){f[i] = a[i];}
for (int i = 0; i < bits; i++) {
  for(int mask = 0; mask < (1<<bits); mask++){
    if(mask & (1<<i)){f[mask] += f[mask^(1<<i)];}
  }
}
```

8. I'm running out of time

8.1. Simulated annealing

```
const ld T = (ld)2000;
const ld alpha = 0.999999;
// (new_score - old_score) / (temperature_final) ~ 10 works well

const ld L = (ld)1e6;
ld small_rand(){
  return ((ld)gen(L))/L;
}

ld P(ld old, ld nw, ld temp){
  if(nw > old)
    return 1.0;
  return exp((nw-old)/temp);
}

{
  auto start = chrono::steady_clock::now();
  ld time_limit = 2000;
  ld temperature = T;
  ld max_score = -1;

  while(elapsed_time < time_limit){
    auto cur = chrono::steady_clock::now();
    elapsed_time = chrono::duration_cast<chrono::milliseconds>(cur - start).count();
    temperature *= alpha;

    // try a neighboring state
    // ....
    // ....

    old_score = score(old_state);
    new_score = score(new_state);
    if(P(old_score, new_score, temperature) >= small_rand()){
      old_state = new_state;
      old_score = new_score;
    }
    if(old_score > max_score){
      max_score = old_score;
      max_state = old_state;
    }
  }
}
```

]

8.2. Eulerian Path

A Eulerian path is a path in a graph that passes through all of its edges exactly once. A Eulerian cycle is a Eulerian path that is a cycle.

The problem is to find the Eulerian path in an **undirected multigraph with loops**.

Algorithm

First we can check if there is an Eulerian path. We can use the following theorem. An Eulerian cycle exists if and only if the degrees of all vertices are even. And an Eulerian path exists if and only if the number of vertices with odd degrees is two (or zero, in the case of the existence of a Eulerian cycle). In addition, of course, the graph must be sufficiently connected (i.e., if you remove all isolated vertices from it, you should get a connected graph).

To find the Eulerian path / Eulerian cycle we can use the following strategy: We find all simple cycles and combine them into one - this will be the Eulerian cycle. If the graph is such that the Eulerian path is not a cycle, then add the missing edge, find the Eulerian cycle, then remove the extra edge.

Looking for all cycles and combining them can be done with a simple recursive procedure:

```
procedure FindEulerPath(V)
  1. iterate through all the edges outgoing from vertex V;
       remove this edge from the graph,
       and call FindEulerPath from the second end of this edge;
  2. add vertex V to the answer.
```

The complexity of this algorithm is obviously linear with respect to the number of edges.

But we can write the same algorithm in the non-recursive version:

```
stack St;
put start vertex in St;
until St is empty
  let V be the value at the top of St;
  if degree(V) = 0, then
    add V to the answer;
    remove V from the top of St;
  otherwise
    find any edge coming out of V;
    remove it from the graph;
    put the second end of this edge in St;
```

It is easy to check the equivalence of these two forms of the algorithm. However, the second form is obviously faster, and the code will be much more efficient.

8.3. Flows with demands

Finding an arbitrary flow

We make the following changes in the network. We add a new source s' and a new sink t', a new edge from the source s' to every other vertex, a new edge for every vertex to the sink t', and one edge from t to s. Additionally we define the new capacity function c' as:

- $c'((s',v)) = \sum_{u \in V} d((u,v))$ for each edge (s',v).
- $c'((v,t')) = \sum_{w \in V} d((v,w))$ for each edge (v,t').
- $c'((u,v)) = c((u,v)) - d((u,v))$ for each edge (u,v) in the old network.
- $c'((t,s)) = \infty$

If the new network has a saturating flow (a flow where each edge outgoing from s' is completely filled, which is equivalent to every edge incoming to t' is completely filled), then the network with demands has a valid flow, and the actual flow can be easily reconstructed from the new network. Otherwise there doesn't exist a flow that satisfies all conditions. Since a saturating flow has to be a maximum flow, it can be found by any maximum flow algorithm, like the Edmonds-Karp algorithm or the Push-relabel algorithm.

The correctness of these transformations is more difficult to understand. We can think of it in the following way: Each edge $e = (u,v)$ with $d(e) > 0$ is originally replaced by two edges: one with the capacity $d(i)$, and the other with $c(i) - d(i)$. We want to find a flow that saturates the first edge (i.e. the flow along this edge must be equal to its capacity). The second edge is less important - the flow along it can be anything, assuming that it doesn't exceed its capacity. Consider each edge that has to be saturated, and we perform the following operation: we draw the edge from the new source s' to its end v, draw the edge from its start u to the new sink t', remove the edge itself, and from the old sink t to the old source s we draw an edge of infinite capacity. By these actions we simulate the fact that this edge is saturated - from v there will be an additionally $d(e)$ flow outgoing (we simulate it with a new source that feeds the right amount of flow to v), and u will also push $d(e)$ additional flow (but instead along the old edge, this flow will go directly to the new sink t'). A flow with the value $d(e)$, that originally flowed along the path $s - \dots - u - v - \dots t$ can now take the new path $s' - v - \dots - t - s - \dots - u - t'$. The only thing that got simplified in the definition of the new network, is that if procedure created multiple edges between the same pair of vertices, then they are combined to one single edge with the summed capacity.

Minimal flow

Note that along the edge (t,s) (from the old sink to the old source) with the capacity ∞ flows the entire flow of the corresponding old network. I.e. the capacity of this edge effects the flow value of the old network. By giving this edge a sufficient large capacity (i.e. ∞), the flow of the old network is unlimited. By limiting this edge by smaller capacities, the flow value will decrease. However if we limit this edge by a too small value, than the network will not have a saturated solution, e.g. the corresponding solution for the original network will not satisfy the demand of the edges. Obviously here can use a binary search to find the lowest value with which all constraints are still satisfied. This gives the minimal flow of the original network.

8.4. Point in convex polygon $O(\log n)$

Algorithm

Let's pick the point with the smallest x-coordinate. If there are several of them, we pick the one with the smallest y-coordinate. Let's denote it as p_0. Now all other points $p_1, \ldots, p_n$ of the polygon are ordered by their polar angle from the chosen point (because the polygon is ordered counter-clockwise).

If the point belongs to the polygon, it belongs to some triangle p_0, p_i, p_{i+1} (maybe more than one if it lies on the boundary of triangles). Consider the triangle p_0, p_i, p_{i+1} such that p belongs to this triangle and i is maximum among all such triangles.

There is one special case. p lies on the segment (p_0, p_n). This case we will check separately. Otherwise all points p_j with $j \leq i$ are counter-clockwise from p with respect to p_0, and all other points are not counter-clockwise from p. This means that we can apply binary search for the point p_i, such that p_i is not counter-clockwise from p with respect to p_0, and i is maximum among all such points. And afterwards we check if the points is actually in the determined triangle.

The sign of $(a - c) \times (b - c)$ will tell us, if the point a is clockwise or counter-clockwise from the point b with respect to the point c. If $(a - c) \times (b - c) > 0$, then the point a is to the right of the vector going from c to b, which means clockwise from b with respect to c. And if $(a - c) \times (b - c) < 0$, then the point is to the left, or counter clockwise. And it is exactly on the line between the points b and c.

Back to the algorithm: Consider a query point p. Firstly, we must check if the point lies between p_1 and p_n. Otherwise we already know that it cannot be part of the polygon. This can be done by checking if the cross product $(p_1 - p_0) \times (p - p_0)$ is zero or has the same sign with $(p_1 - p_0) \times (p_n - p_0)$, and $(p_n - p_0) \times (p - p_0)$ is zero or has the same sign with $(p_n - p_0) \times (p_1 - p_0)$. Then we handle the special case in which p is part of the line (p_0, p_1). And then we can binary search the last point from $p_1, \ldots p_n$ which is not counter-clockwise from p with respect to p_0. For a single point p_i this condition can be checked by checking that $(p_i - p_0) \times (p - p_0) \leq 0$. After we found such a point p_i, we must test if p lies inside the triangle p_0, p_i, p_{i+1}. To test if it belongs to the triangle, we may simply check that $|(p_i - p_0) \times (p_{i+1} - p_0)| = |(p_0 - p) \times (p_i - p)| + |(p_i - p) \times (p_{i+1} - p)| + |(p_{i+1} - p) \times (p_0 - p)|$. This checks if the area of the triangle p_0, p_i, p_{i+1} has to exact same size as the sum of the sizes of the triangle p_0, p_i, p, the triangle p_0, p, p_{i+1} and the triangle p_i, p_{i+1}, p. If p is outside, then the sum of those three triangle will be bigger than the size of the triangle. If it is inside, then it will be equal.

```
bool pointInTriangle(pt a, pt b, pt c, pt point) {
    long long s1 = abs(a.cross(b, c));
    long long s2 = abs(point.cross(a, b)) + abs(point.cross(b, c)) + abs(point.cross(c, a));
    return s1 == s2;
}

void prepare(vector<pt> &points) {
    n = points.size();
    int pos = 0;
    for (int i = 1; i < n; i++) {
        if (lexComp(points[i], points[pos]))
            pos = i;
    }
    rotate(points.begin(), points.begin() + pos, points.end());

    n--;
    seq.resize(n);
    for (int i = 0; i < n; i++)
        seq[i] = points[i + 1] - points[0];
    translation = points[0];
}
```

```
bool pointInConvexPolygon(pt point) {
    point = point - translation;
    if (seq[0].cross(point) != 0 &&
            sgn(seq[0].cross(point)) != sgn(seq[0].cross(seq[n - 1])))
        return false;
    if (seq[n - 1].cross(point) != 0 &&
            sgn(seq[n - 1].cross(point)) != sgn(seq[n - 1].cross(seq[0])))
        return false;

    if (seq[0].cross(point) == 0)
        return seq[0].sqrLen() >= point.sqrLen();

    int l = 0, r = n - 1;
    while (r - l > 1) {
        int mid = (l + r) / 2;
        int pos = mid;
        if (seq[pos].cross(point) >= 0)
            l = mid;
        else
            r = mid;
    }
    int pos = l;
    return pointInTriangle(seq[pos], seq[pos + 1], pt(0, 0), point);
}
```

8.5. Minkowski sum of convex polygons

Definition

Consider two sets A and B of points on a plane. Minkowski sum $A + B$ is defined as $\{a + b | a \in A, b \in B\}$. Here we will consider the case when A and B consist of convex polygons P and Q with their interiors. Throughout this article we will identify polygons with ordered sequences of their vertices, so that notation like $|P|$ or P_i makes sense. It turns out that the sum of convex polygons P and Q is a convex polygon with at most $|P| + |Q|$ vertices.

Algorithm

Here we consider the polygons to be cyclically enumerated, i. e. $P_{|P|} = P_0,\ Q_{|Q|} = Q_0$ and so on.

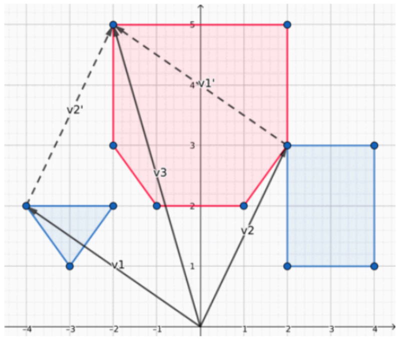
Since the size of the sum is linear in terms of the sizes of initial polygons, we should aim at finding a linear-time algorithm. Suppose that both polygons are ordered counter-clockwise. Consider sequences of edges $\{\overrightarrow{P_iP_{i+1}}\}$ and $\{\overrightarrow{Q_jQ_{j+1}}\}$ ordered by polar angle. We claim that the sequence of edges of $P + Q$ can be obtained by merging these two sequences preserving polar angle order and replacing consecutive co-directed vectors with their sum. Straightforward usage of this idea results in a linear-time algorithm, however, restoring the vertices of $P + Q$ from the sequence of sides requires repeated addition of vectors, which may introduce unwanted precision issues if we're working with floating-point coordinates, so we will describe a slight modification of this idea.

Firstly we should reorder the vertices in such a way that the first vertex of each polygon has the lowest y-coordinate (in case of several such vertices pick the one with the smallest x-coordinate). After that the sides of both polygons will become sorted by polar angle, so there is no need to sort them manually. Now we create two pointers i (pointing to a vertex of P) and j (pointing to a vertex of Q), both initially set to 0. We repeat the following steps while $i < |P|$ or $j < |Q|$.

1. Append $P_i + Q_j$ to $P + Q$.
2. Compare polar angles of $\overrightarrow{P_iP_{i+1}}$ and $\overrightarrow{Q_jQ_{j+1}}$.
3. Increment the pointer which corresponds to the smallest angle (if the angles are equal, increment both).

Visualization

Here is a nice visualization, which may help you understand what is going on.



Distance between two polygons

One of the most common applications of Minkowski sum is computing the distance between two convex polygons (or simply checking whether they intersect). The distance between two convex polygons P and Q is defined as $\min_{a\in P, b\in Q} ||a-b||$. One can note that the distance is always attained between two vertices or a vertex and an edge, so we can easily find the distance in $O(|P||Q|)$. However, with clever usage of Minkowski sum we can reduce the complexity to $O(|P|+|Q|)$.

If we reflect Q through the point $(0,0)$ obtaining polygon $-Q$, the problem boils down to finding the smallest distance between a point in $P+(-Q)$ and $(0,0)$. We can find that distance in linear time using the following idea. If $(0,0)$ is inside or on the boundary of polygon, the distance is 0, otherwise the distance is attained between $(0,0)$ and some vertex or edge of the polygon. Since Minkowski sum can be computed in linear time, we obtain a linear-time algorithm for finding the distance between two convex polygons.

```
void reorder_polygon(vector<pt> & P){
    size_t pos = 0;
    for(size_t i = 1; i < P.size(); i++){
        if(P[i].y < P[pos].y || (P[i].y == P[pos].y && P[i].x < P[pos].x))
            pos = i;
    }
    rotate(P.begin(), P.begin() + pos, P.end());
}

vector<pt> minkowski(vector<pt> P, vector<pt> Q){
    // the first vertex must be the lowest
    reorder_polygon(P); reorder_polygon(Q);
    // we must ensure cyclic indexing
    P.push_back(P[0]); P.push_back(P[1]);
    Q.push_back(Q[0]); Q.push_back(Q[1]);
    vector<pt> result; size_t i = 0, j = 0;
    while(i < P.size() - 2 || j < Q.size() - 2){
        result.push_back(P[i] + Q[j]);
        auto cross = (P[i + 1] - P[i]).cross(Q[j + 1] - Q[j]);
        if(cross >= 0 && i < P.size() - 2) ++i;
        if(cross <= 0 && j < Q.size() - 2) ++j;
    }
    return result
}
```

8.6. janY template

```
#include<bits/stdc++.h>
using namespace std;
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,popcnt,lzcnt")
#define fo(i,n) for(i=0;i<n;i++)
#define Fo(i,k,n) for(i=k;k<n?i<n:i>n;k<n?i+=1:i-=1)
#define ll long long
#define ld long double
#define all(x) x.begin(),x.end()
#define sortall(x) sort(all(x))
#define rev(x) reverse(x.begin(),x.end())
#define fi first
#define se second
#define pb push_back
#define PI 3.14159265359
typedef pair<int,int> pii;
typedef pair<ll,ll> pl;
typedef vector<int> vi;
typedef vector<ll> vl;
typedef vector<pii> vpii;
typedef vector<pl> vpl;
typedef vector<vi> vvi;
typedef vector<vl> vvl;
bool sortbysec(const pair<int,int> &a,const pair<int,int> &b){return a.second<b.second;}
#define sortpairbysec(x) sort(all(x), sortbysec)
bool sortcond(const pair<int,int> &a,const pair<int,int> &b){
    if(a.fi!=b.fi) return a.fi<b.fi;
    return a.se>b.se;
}
struct myComp {
    constexpr bool operator()(pii const& a, pii const& b) const noexcept{
        if(a.first!=b.first) return a.first<b.first;
        return a.second>b.second;
    }
};
const int mod=1000000007, N=3e5, M=N;
// & - AND; | - OR; ^ - XOR
vl a;
ll n,m,k,q;
void solve(int tc){
    int i,j;
    cin>>n;
}
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0); cout.tie(0);
    int t=1; cin>>t; int i;
    fo(i,t) solve(i+1);
    return 0;
}
```