

## C'est BON! Team Notes

## Contents

<b>1</b>	<b>Number theory</b>	<b>1</b>
1.1	Count primes up to N	1
1.2	Extended Euclide	1
1.3	System of linear equations	1
1.4	Pollard Rho	2
1.5	Cubic	2
1.6	PythagoreTriple	2
1.7	Tonelli-Shanks	2
<b>2</b>	<b>String</b>	<b>3</b>
2.1	Suffix Array	3
2.2	Aho Corasick	3
2.3	Z algorithm	3
2.4	Manacher	3
2.5	Suffix Automaton	3
2.6	ALCS	4
2.7	Palindromic Tree	4
2.8	Lyndon Factorization	4
<b>3</b>	<b>Geometry</b>	<b>4</b>
3.1	Geometry	4
<b>4</b>	<b>Numerical algorithms</b>	<b>6</b>
4.1	Gauss Elimination	6
4.2	Simplex Algorithm	6
4.3	NTT	7
4.4	FFT	7
4.5	Bitwise FFT	7
4.6	FFT chemthan	8
4.7	Interpolation	8
4.8	Binary vector space	9
4.9	DiophanteMod	9
4.10	Berlekamp-Massey	9
4.11	Linear Sieve	9
<b>5</b>	<b>Graph algorithms</b>	<b>9</b>
5.1	Arborescence	9
5.2	Arborescence With Trace	10
5.3	Bridges and Articulations	10
5.4	Bipartite Maximum Matching	10
5.5	General Matching	11
5.6	Dinic Flow	11
5.7	Dinic Flow With Scaling	12
5.8	Gomory Hu Tree	12
5.9	Min Cost-Max Flow	12
5.10	Min Cost Max Flow Potential	12
5.11	Bounded Feasible Flow	13
5.12	Hungarian Algorithm	13
5.13	Undirected mincut	14
5.14	Eulerian Path/Circuit	14
5.15	2-SAT	14
5.16	SPFA	14
<b>6</b>	<b>Data structures</b>	<b>15</b>
6.1	Treap	15
6.2	Big Integer	15
6.3	Convex Hull IT	16
6.4	Link Cut Tree	16
6.5	Ordered Set	17
6.6	Unordered Map	17
<b>7</b>	<b>Miscellaneous</b>	<b>17</b>
7.1	RNG	17
7.2	SQRT forloop	17

## 1 Number theory

## 1.1 Count primes up to N

```

// To initialize, call init_count_primes() first.
// Function count_primes(N) will compute the number of prime numbers lower
// than
// or equal to N.
//
// Time complexity: Around  $O(N^{0.75})$ 
//
// Constants to configure:
// - MAX is the maximum value of  $\sqrt{N} + 2$ 
bool prime[MAX];
int prec[MAX];
vector<int> P;
llint rec(llint N, int K) {
    if (N <= 1 || K < 0) return 0;
    if (N <= P[K]) return N-1;

    if (N < MAX && llint(P[K])*P[K] > N) return N-1 - prec[N] + prec[P[K]];
    const int LIM = 250;
    static int memo[LIM*LIM][LIM];
    bool ok = N < LIM*LIM;
    if (ok && memo[N][K]) return memo[N][K];
    llint ret = N/P[K] - rec(N/P[K], K-1) + rec(N, K-1);
    if (ok) memo[N][K] = ret;
    return ret;
}

llint count_primes(llint N) {
    if (N < MAX) return prec[N];
    int K = prec[(int)sqrt(N) + 1];
    return N-1 - rec(N, K) + prec[P[K]];
}

void init_count_primes() {
    prime[2] = true;
    for (int i = 3; i < MAX; i += 2) prime[i] = true;
    for (int i = 3; i < MAX; i += 2) if (prime[i])
        for (int j = i+i; j < MAX; j += i+i)
            prime[j] = false;
    REP(i, MAX) if (prime[i]) P.push_back(i);
    FOR(i, 1, MAX) prec[i] = prec[i-1] + prime[i];
}

```

## 1.2 Extended Euclide

```

int bezout(int a, int b) {
    // return x such that ax + by == gcd(a, b)
    int xa = 1, xb = 0;
    while (b) {
        int q = a / b;
        int r = a - q * b, xr = xa - q * xb;
        a = b; xa = xb;
        b = r; xb = xr;
    }
    return xa;
}

pair<int, int> solve(int a, int b, int c) {
    // solve ax + by == c
    int d = __gcd(a, b);
    int x = bezout(a, b);
    int y = (d - a * x) / b;
    c /= d;
    return make_pair(x * c, y * c);
}

int main() {
    int a = 100, b = 128;
    int c = __gcd(a, b);
    int x = bezout(a, b);
    int y = (c - a * x) / b;
    cout << x << ' ' << y << endl;
    pair<int, int> xy = solve(100, 128, 40);
    cout << xy.first << ' ' << xy.second << endl;
    return 0;
}

```

## 1.3 System of linear equations

```

// extended version, uses diophantine equation solver to solve system of
// congruent equations
pair<int, int> solve(int a, int b, int c) {
    int cc = c;
    // solve ax + by == c
    int d = __gcd(a, b);
    int x = bezout(a / d, b / d);
    int y = (d - a * x) / b;
    c /= d;
    return make_pair(x * c, y * c);
}

int lcm(int a, int b) {
    return a / __gcd(a, b) * b;
}

// use this if input is large, make sure (#define int long long)
int mul(int a, int b, int p) {
    a %= p, b %= p;
    int q = (int) ((long double) a * b / p);
    int r = a * b - q * p;
    while (r < 0) r += p;
    while (r >= p) r -= p;
    return r;
}

int solveSystem(vector<int> a, vector<int> b) {
    // xi mod bi = ai
    int A = a[0], B = b[0];
    // x mod B = A
    for (int i = 1; i < a.size(); ++i) {
        int curB = b[i], curA = a[i];

```

```

// x = Bi + A = curB * j + curA
pair<int, int> ij = solve(B, -curB, curA - A);
if (B * ij.first + A != curB * ij.second + curA) return -1;
int newB = lcm(B, curB);
int newA = (mul(B, ij.first, newB) + A) % newB;
if (newA < 0) newA += newB;
A = newA; B = newB;
if (i + 1 == a.size()) return A;
}
return -1;
}

int main() {
vector<int> a = {0, 3, 3};
vector<int> b = {3, 6, 9};
cout << solveSystem(a, b) << endl;
return 0;
}

```

## 1.4 Pollard Rho

```

#include <bits/stdc++.h>

using namespace std;

struct PollardRho {
    long long n;
    map<long long, int> ans;
    PollardRho(long long n) : n(n) {}
    long long random(long long u) {
        return abs(rand()) % u;
    }
    long long mul(long long a, long long b, long long p) {
        a %= p; b %= p;
        long long q = (long long)((long double) a * b / p);
        long long r = a * b - q * p;
        while (r < 0) r += p;
        while (r >= p) r -= p;
        return r;
    }
    long long pow(long long u, long long v, long long n) {
        long long res = 1;
        while (v) {
            if (v & 1) res = mul(res, u, n);
            u = mul(u, u, n);
            v >>= 1;
        }
        return res;
    }
    bool rabin(long long n) {
        if (n < 2) return 0;
        if (n == 2) return 1;
        long long s = 0, m = n - 1;
        while (m % 2 == 0) {
            s++;
            m >>= 1;
        }
        // 1 - 0.9 ^ 40
        for (int it = 1; it <= 40; it++) {
            long long u = random(n - 2) + 2;
            long long f = pow(u, m, n);
            if (f == 1 || f == n - 1) continue;
            for (int i = 1; i < s; i++) {
                f = mul(f, f, n);
                if (f == 1) return 0;
                if (f == n - 1) break;
            }
            if (f != n - 1) return 0;
        }
        return 1;
    }
    long long f(long long x, long long n) {
        return (mul(x, x, n) + 1) % n;
    }
    long long findfactor(long long n) {
        long long x = random(n - 1) + 2;
        long long y = x;
        long long p = 1;
        while (p == 1) {
            x = f(x, n);
            y = f(f(y, n), n);
            p = __gcd(abs(x - y), n);
        }
        return p;
    }
    void pollard_rho(long long n) {
        if (n <= 1000000) {
            for (int i = 2; i * i <= n; i++) {
                while (n % i == 0) {
                    ans[i]++;
                    n /= i;
                }
            }
            if (n > 1) ans[n]++;
            return;
        }
        if (rabin(n)) {
            ans[n]++;
            return;
        }
        long long p = 0;
        while (p == 0 || p == n) {
            p = findfactor(n);
        }
        pollard_rho(n / p);
        pollard_rho(p);
    }
};

```

```

int main() {
    long long n;
    cin >> n;
    PollardRho f(n);
    f.pollard_rho(f.n);
    for (auto x : f.ans) {
        cout << x.first << " " << x.second << endl;
    }
}

```

## 1.5 Cubic

```

const double EPS = 1e-6;
struct Result {
    int n; // Number of solutions
    double x[3]; // Solutions
};
Result solve_cubic(double a, double b, double c, double d) {
    long double al = b/a, a2 = c/a, a3 = d/a;
    long double q = (al+al - 3*a2)/9.0, sq = -2*sqrt(q);
    long double r = (2*al+al+al - 9*al+a2 + 27*a3)/54.0;
    double z = r+r-q*sq, theta;
    Result s;
    if (z <= EPS) {
        s.n = 3; theta = acos(r/sqrt(q*sq));
        s.x[0] = sq*cos(theta/3.0) - al/3.0;
        s.x[1] = sq*cos((theta+2.0*PI)/3.0) - al/3.0;
        s.x[2] = sq*cos((theta+4.0*PI)/3.0) - al/3.0;
    }
    else {
        s.n = 1; s.x[0] = pow(sqrt(z)+fabs(r), 1/3.0);
        s.x[0] += q/s.x[0]; s.x[0] += (r < 0) ? 1 : -1;
        s.x[0] -= al/3.0;
    }
    return s;
}

```

## 1.6 PythagoreTriple

```

// sinh bo 3 pytago nguyen thuy voi x, y, z <= n
vector< vector<int> > genPrimitivePytTriples(int n) {
    vector< vector<int> > ret;
    for (int r=1; r*r<=n; ++r) for (int s=(r%2==0)?1:2; s<r; s+=2) if (__gcd(
        r,s)==1) {
        vector<int> t;
        t.push_back(r*r+s*s); //z
        t.push_back(2*r*s); //y
        t.push_back(r*r-s*s); //x
        if (t[0]<=n) ret.push_back(t);
    }
    sort(ret.begin(), ret.end());
    return ret;
}
// a^2 + b^2 == c^2
// To generate all primitive triples:
// a = m^2 - n^2, b = 2mn, c = m^2 + n^2 (m > n)
// Primitive triples iff gcd(m, n) == 1 && (m - n) % 2 == 1

```

## 1.7 Tonelli-Shanks

```

long pow_mod(long x, long n, long p) {
    if (n == 0) return 1;
    if (n & 1)
        return (pow_mod(x, n-1, p) * x) % p;
    x = pow_mod(x, n/2, p);
    return (x * x) % p;
}

/* Takes as input an odd prime p and n < p and returns r
 * such that r * r = n [mod p]. */
long tonelli_shanks(long n, long p) {
    long s = 0;
    long q = p - 1;
    while ((q & 1) == 0) { q /= 2; ++s; }
    if (s == 1) {
        long r = pow_mod(n, (p+1)/4, p);
        if ((r * r) % p == n) return r;
        return 0;
    }
    // Find the first quadratic non-residue z by brute-force search
    long z = 1;
    while (pow_mod(++z, (p-1)/2, p) != p - 1);
    long c = pow_mod(z, q, p);
    long r = pow_mod(n, (q+1)/2, p);
    long t = pow_mod(n, q, p);
    long m = s;
    while (t != 1) {
        long tt = t;
        long i = 0;
        while (tt != 1) {
            tt = (tt * tt) % p;
            ++i;
            if (i == m) return 0;
        }
        long b = pow_mod(c, pow_mod(2, m-i-1, p-1), p);
        long b2 = (b * b) % p;
        r = (r * b) % p;
        t = (t * b2) % p;
        c = b2;
        m = i;
    }
    if ((r * r) % p == n) return r;
}

```

```
    return 0;
}
```

## 2 String

### 2.1 Suffix Array

```
#include <bits/stdc++.h>

using namespace std;

struct SuffixArray {
    static const int N = 100010;

    int n;
    char *s;
    int sa[N], tmp[N], pos[N];
    int len, cnt[N], lcp[N];

    SuffixArray(char *t) {
        s = t;
        n = strlen(s) + 1;
        buildSA();
    }

    bool cmp(int u, int v) {
        if (pos[u] != pos[v]) {
            return pos[u] < pos[v];
        }
        return (u + len <= n && v + len <= n) ? pos[u + len] < pos[v + len] :
            u > v;
    }

    void radix(int delta) {
        memset(cnt, 0, sizeof cnt);
        for (int i = 1; i <= n; i++) {
            cnt[i + delta <= n ? pos[i + delta] : 0]++;
        }
        for (int i = 1; i < n; i++) {
            cnt[i] += cnt[i - 1];
        }
        for (int i = n; i > 0; i--) {
            int id = sa[i];
            tmp[cnt[id + delta <= n ? pos[id + delta] : 0]--] = id;
        }
        for (int i = 1; i <= n; i++) {
            sa[i] = tmp[i];
        }
    }

    void buildSA() {
        for (int i = 1; i <= n; i++) {
            sa[i] = i;
            pos[i] = s[i];
        }
        len = 1;
        while (1) {
            radix(len);
            radix(0);
            tmp[1] = 1;
            for (int i = 2; i <= n; i++) {
                tmp[i] = tmp[i - 1] + cmp(sa[i - 1], sa[i]);
            }
            for (int i = 1; i <= n; i++) {
                pos[sa[i]] = tmp[i];
            }
            if (tmp[n] == n) {
                break;
            }
            len <= 1;
        }

        len = 0;
        for (int i = 1; i <= n; i++) {
            if (pos[i] == n) {
                continue;
            }
            int j = sa[pos[i] + 1];
            while (s[i + len] == s[j + len]) {
                len++;
            }
            lcp[pos[i]] = len;
            if (len) {
                len--;
            }
        }
    }
};
```

### 2.2 Aho Corasick

```
struct AhoCorasick {
    const int N = 30030;

    int fail[N];
    int to[N][26];
    int ending[N];
    int sz;

    AhoCorasick() {
        sz = 1;
    }

    int add(const string &s) {
        int node = 1;
        for (int i = 0; i < s.size(); ++i) {
```

```
            if (!to[node][s[i] - 'a']) {
                to[node][s[i] - 'a'] = ++sz;
            }
            node = to[node][s[i] - 'a'];
        }
        ending[node] = true;
        return node;
    }

    void push() {
        queue<int> Q;
        Q.push(1);
        fail[1] = 1;
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int i = 0; i < 26; ++i) {
                int &v = to[u][i];
                if (!v) {
                    v = u == 1 ? 1 : to[fail[u]][i];
                } else {
                    fail[v] = u == 1 ? 1 : to[fail[u]][i];
                    ending[v] |= ending[fail[v]];
                    Q.push(v);
                }
            }
        }
    }
};
```

### 2.3 Z algorithm

```
vector<int> calcZ(const string &s) {
    int L = 0, R = 0;
    int n = s.size();
    vector<int> Z(n);
    Z[0] = n;
    for (int i = 1; i < n; i++) {
        if (i > R) {
            L = R = i;
            while (R < n && s[R] == s[R - L]) R++;
            Z[i] = R - L; R--;
        } else {
            int k = i - L;
            if (Z[k] < R - i + 1) Z[i] = Z[k];
            else {
                L = i;
                while (R < n && s[R] == s[R - L]) R++;
                Z[i] = R - L; R--;
            }
        }
    }
    return Z;
}
```

### 2.4 Manacher

```
struct Manacher {
    int n;
    vector<int> d; //Radius of odd palindromes
    vector<int> e; //Radius of even palindromes
    int build(char* s) {
        n = strlen(s), d.resize(n), e.resize(n);
        int res = 0;
        int l = 0, r = -1;
        for (int i = 0; i < n; ++i) {
            int k = (i > r) ? 1 : min(d[l + r - i], r - i) + 1;
            while (i - k >= 0 && i + k < n && s[i - k] == s[i + k]) k++;
            d[i] = --k;
            res = max(res, k + k + 1);
            if (r < i + k) {
                l = i - k;
                r = i + k;
            }
        }
        l = 0; r = -1;
        for (int i = 0; i < n; ++i) {
            int k = (i > r) ? 1 : min(e[l + r - i + 1], r - i + 1) + 1;
            while (i - k >= 0 && i + k - 1 < n && s[i - k] == s[i + k - 1]) k++;
            ++i;
            e[i] = --k;
            res = max(res, k + k);
            if (r < i + k - 1) {
                l = i - k;
                r = i + k - 1;
            }
        }
        return res;
    }
};
```

### 2.5 Suffix Automaton

```
//set last = 0 everytime we add new string
struct SuffixAutomaton {
    static const int N = 100000;
    static const int CHARACTER = 26;
    int suf[N * 2], nxt[N * 2][CHARACTER], cnt, last, len[N * 2];

    SuffixAutomaton() {
        memset(suf, -1, sizeof suf);
    }
};
```

```

memset(nxt, -1, sizeof nxt);
memset(len, 0, sizeof len);
last = cnt = 0;

}

int getNode(int last, int u) {
    int q = nxt[last][u];
    if (len[last] + 1 == len[q]) {
        return q;
    }
    int clone = ++cnt;
    len[clone] = len[last] + 1;
    for (int i = 0; i < CHARACTER; i++) {
        nxt[clone][i] = nxt[q][i];
    }
    while (last != -1 && nxt[last][u] == q) {
        nxt[last][u] = clone;
        last = suf[last];
    }
    suf[clone] = suf[q];
    return suf[q] = clone;
}

void add(int u) {
    if (nxt[last][u] == -1) {
        int newNode = ++cnt;
        len[newNode] = len[last] + 1;
        while (last != -1 && nxt[last][u] == -1) {
            nxt[last][u] = newNode;
            last = suf[last];
        }
        if (last == -1) {
            suf[newNode] = 0;
            last = newNode;
            return;
        }
        suf[newNode] = getNode(last, u);
        last = newNode;
    } else {
        last = getNode(last, u);
    }
}
};

```

## 2.6 ALCS

```

define
a: row, b: col
c_l,i,j: largest weighted path from (0, i) to (l, j)
f_l,j: for all (l, j), 0 <= l <= n_a, 1 <= j <= n_b, f_l,j is the
smallest value i < j such that C_l,i,j = C_l,i,j-1 + 1
g_l,j: for all (l, j), 1 <= l <= n_a, 0 <= j <= n_b, g_l,j is the
smallest value i <= j such that C_l,i,j = C_l-1,i,j

const int N = 2010;
int n, m;
char a[N], b[N];
int f[N][N], g[N][N];
int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    int tc;
    cin >> tc;
    while (tc--) {
        cin >> (a + 1);
        cin >> (b + 1);
        n = strlen(a + 1);
        m = strlen(b + 1);
        for (int i = 1; i <= m; i++) f[0][i] = i;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (a[i] == b[j]) {
                    f[i][j] = g[i][j - 1];
                    g[i][j] = f[i - 1][j];
                } else {
                    f[i][j] = max(f[i - 1][j], g[i][j - 1]);
                    g[i][j] = min(f[i - 1][j], g[i][j - 1]);
                }
            }
        }
        int q;
        cin >> q;
        for (int i = 1; i <= q; i++) {
            int l, r, k;
            // substring a(l, r) and b(1, k)
            cin >> l >> r >> k;
            int res = 0;
            for (int j = 1; j <= r; j++) res += (f[k][j] < l);
            cout << res << ' ';
        }
        cout << '\n';
    }
    return 0;
}

```

## 2.7 Palindromic Tree

```

const int N = 1e5, SIZE = 26;

int s[N], len[N], link[N], to[N][SIZE], depth[N];
int n, last, sz;

void init() {
    s[n++] = -1;
    link[0] = 1;
    len[1] = -1;
    sz = 2;
}

```

```

int get_link(int v) {
    while (s[n - len[v] - 2] != s[n - 1]) v = link[v];
    return v;
}

int add_letter(int c) {
    s[n++] = c;
    last = get_link(last);
    if (!to[last][c]) {
        len[sz] = len[last] + 2;
        link[sz] = to[get_link(link[last])][c];
        to[last][c] = sz++;
    }
    last = to[last][c];
    return len[last];
}

```

## 2.8 Lyndon Factorization

```

// https://cp-algorithms.com/string/lyndon_factorization.html
vector<string> duval(string const& s) {
    int n = s.size();
    int i = 0;
    vector<string> factorization;
    while (i < n) {
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j])
                k = i;
            else
                k++;
            j++;
        }
        while (i <= k) {
            factorization.push_back(s.substr(i, j - k));
            i += j - k;
        }
    }
    return factorization;
}

string min_cyclic_string(string s) {
    s += s;
    int n = s.size();
    int i = 0, ans = 0;
    while (i < n / 2) {
        ans = i;
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j])
                k = i;
            else
                k++;
            j++;
        }
        while (i <= k)
            i += j - k;
    }
    return s.substr(ans, n / 2);
}

```

## 3 Geometry

### 3.1 Geometry

```

#define EPS 1e-6
inline int cmp(double a, double b) { return (a < b - EPS) ? -1 : ((a > b + EPS) ? 1 : 0); }
struct Point {
    double x, y;
    Point() { x = y = 0.0; }
    Point(double x, double y) : x(x), y(y) {}

    Point operator + (const Point& a) const { return Point(x+a.x, y+a.y); }
    Point operator - (const Point& a) const { return Point(x-a.x, y-a.y); }
    Point operator * (double k) const { return Point(x*k, y*k); }
    Point operator / (double k) const { return Point(x/k, y/k); }

    double operator * (const Point& a) const { return x*a.x + y*a.y; } // dot
    double operator % (const Point& a) const { return x*a.y - y*a.x; } // cross product

    double norm() { return x*x + y*y; }
    double len() { return sqrt(norm()); } // hypot(x, y);
    Point rotate(double alpha) {
        double cosa = cos(alpha), sina = sin(alpha);
        return Point(x * cosa - y * sina, x * sina + y * cosa);
    }
};

double angle(Point a, Point o, Point b) { // min of directed angle AOB & BOA
    a = a - o; b = b - o;
    return acos((a * b) / sqrt(a.norm()) / sqrt(b.norm()));
}

double directed_angle(Point a, Point o, Point b) { // angle AOB, in range [0, 2*PI)
    double t = -atan2(a.y - o.y, a.x - o.x)
        + atan2(b.y - o.y, b.x - o.x);
    while (t < 0) t += 2*PI;
    return t;
}

// Distance from p to Line ab (closest Point --> c)
double distToLine(Point p, Point a, Point b, Point &c) {
    Point ap = p - a, ab = b - a;
    double u = (ap * ab) / ab.norm();
    c = a + (ab * u);
    return (p-c).len();
}

```

```

}
// Distance from p to segment ab (closest Point --> c)
double distToLineSegment(Point p, Point a, Point b, Point &c) {
    Point ap = p - a, ab = b - a;
    double u = (ap * ab) / ab.norm();
    if (u < 0.0) {
        c = Point(a.x, a.y);
        return (p - a).len();
    }
    if (u > 1.0) {
        c = Point(b.x, b.y);
        return (p - b).len();
    }
    return distToLine(p, a, b, c);
}
// NOTE: WILL NOT WORK WHEN a = b = 0.
struct Line {
    double a, b, c;
    Point A, B; // Added for polygon intersect line. Do not rely on
                // assumption that these are valid

    Line(double a, double b, double c) : a(a), b(b), c(c) {}

    Line(Point A, Point B) : A(A), B(B) {
        a = B.y - A.y;
        b = A.x - B.x;
        c = -(a * A.x + b * A.y);
    }

    Line(Point P, double m) {
        a = -m; b = 1;
        c = -(a * P.x) + (b * P.y);
    }

    double f(Point A) {
        return a*A.x + b*A.y + c;
    }
};

bool areParallel(Line l1, Line l2) {
    return cmp(11.a*12.b, 11.b*12.a) == 0;
}

bool areSame(Line l1, Line l2) {
    return areParallel(l1, l2) && cmp(11.c*12.a, 12.c*11.a) == 0
        && cmp(11.c*12.b, 11.b*12.c) == 0;
}

bool areIntersect(Line l1, Line l2, Point &p) {
    if (areParallel(l1, l2)) return false;
    double dx = 11.b*12.c - 12.b*11.c;
    double dy = 11.c*12.a - 12.c*11.a;
    double d = 11.a*12.b - 12.a*11.b;
    p = Point(dx/d, dy/d);
    return true;
}

void closestPoint(Line l, Point p, Point &ans) {
    if (fabs(11.b) < EPS) {
        ans.x = -(1.c) / 1.a; ans.y = p.y;
        return;
    }
    if (fabs(11.a) < EPS) {
        ans.x = p.x; ans.y = -(1.c) / 1.b;
        return;
    }
    Line perp(1.b, -1.a, -(1.b*p.x - 1.a*p.y));
    areIntersect(l, perp, ans);
}

void reflectionPoint(Line l, Point p, Point &ans) {
    Point b;
    closestPoint(l, p, b);
    ans = p + (b - p) * 2;
}

struct Circle : Point {
    double r;
    Circle(double x = 0, double y = 0, double r = 0) : Point(x, y), r(r) {}
    Circle(Point p, double r) : Point(p), r(r) {}
    bool contains(Point p) { return (this - p).len() <= r + EPS; }
};

// Find common tangents to 2 circles
// Tested:
// - http://codeforces.com/gym/100803/ - H
// Helper method
void tangents(Point c, double r1, double r2, vector<Line> &ans) {
    double r = r2 - r1;
    double z = sqrt(c.x) + sqrt(c.y);
    double d = z - sqrt(r);
    if (d < -EPS) return;
    d = sqrt(fabs(d));
    Line l((c.x * r + c.y * d) / z,
          (c.y * r - c.x * d) / z,
          r1);
    ans.push_back(l);
}

// Actual method: returns vector containing all common tangents
vector<Line> tangents(Circle a, Circle b) {
    vector<Line> ans; ans.clear();
    for (int i=-1; i<=1; i+=2)
        for (int j=-1; j<=1; j+=2)
            tangents(b-a, a.r+i, b.r+j, ans);
    for (int i = 0; i < ans.size(); ++i)
        ans[i].c = ans[i].a * a.x + ans[i].b * a.y;
    vector<Line> ret;
    for (int i = 0; i < (int) ans.size(); ++i) {
        bool ok = true;
        for (int j = 0; j < i; ++j)
            if (areSame(ret[j], ans[i])) {
                ok = false;
                break;
            }
        if (ok) ret.push_back(ans[i]);
    }
    return ret;
}

// Circle & line intersection
vector<Point> intersection(Line l, Circle cir) {
    double r = cir.r, a = l.a, b = l.b, c = l.c + l.a*cir.x + l.b*cir.y;
    vector<Point> res;
    double x0 = -a/c/(a*a+b*b), y0 = -b/c/(a*a+b*b);
    if (c*c > r*r*(a*a+b*b)+EPS) return res;
    else if (fabs(c*c - r*r*(a*a+b*b)) < EPS) {
        res.push_back(Point(x0, y0) + Point(cir.x, cir.y));
    }
    return res;
}

}
// helper functions for commonCircleArea
double cir_area_solve(double a, double b, double c) {
    return acos((a*a + b*b - c*c) / 2 / a / b);
}

double cir_area_cut(double a, double r) {
    double s1 = a * r * r / 2;
    double s2 = sin(a) * r * r / 2;
    return s1 - s2;
}

double commonCircleArea(Circle c1, Circle c2) { //return the common area of
    two circle
    if (c1.r < c2.r) swap(c1, c2);
    double d = (c1 - c2).len();
    if (d + c2.r <= c1.r + EPS) return c2.r*c2.r*M_PI;
    if (d >= c1.r + c2.r - EPS) return 0.0;
    double a1 = cir_area_solve(d, c1.r, c2.r);
    double a2 = cir_area_solve(d, c2.r, c1.r);
    return cir_area_cut(a1*2, c1.r) + cir_area_cut(a2*2, c2.r);
}

// Check if 2 circle intersects. Return true if 2 circles touch
bool areIntersect(Circle u, Circle v) {
    if (cmp((u - v).len(), u.r + v.r) > 0) return false;
    if (cmp((u - v).len() + v.r, u.r) < 0) return false;
    if (cmp((u - v).len() + u.r, v.r) < 0) return false;
    return true;
}

// If 2 circle touches, will return 2 (same) points
// If 2 circle are same --> be careful
vector<Point> circleIntersect(Circle u, Circle v) {
    vector<Point> res;
    if (!areIntersect(u, v)) return res;
    double d = (u - v).len();
    double alpha = acos((u.r * u.r + d*d - v.r * v.r) / 2.0 / u.r / d);

    Point p1 = (v - u).rotate(alpha);
    Point p2 = (v - u).rotate(-alpha);
    res.push_back(p1 / p1.len() * u.r + u);
    res.push_back(p2 / p2.len() * u.r + u);
    return res;
}

Point centroid(Polygon p) {
    Point c(0,0);
    double scale = 6.0 * signed_area(p);
    for (int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
    }
    return c / scale;
}

// Cut a polygon with a line. Returns one half.
// To return the other half, reverse the direction of Line l (by negating l.a, l.b)
// The line must be formed using 2 points
Polygon polygon_cut(const Polygon& P, Line l) {
    Polygon Q;
    for (int i = 0; i < P.size(); ++i) {
        Point A = P[i], B = (i == P.size()-1) ? P[0] : P[i+1];
        if (ccw(1.A, 1.B, A) != -1) Q.push_back(A);
        if (ccw(1.A, 1.B, A)*ccw(1.A, 1.B, B) < 0) {
            Point p; areIntersect(Line(A, B), l, p);
            Q.push_back(p);
        }
    }
    return Q;
}

// Find intersection of 2 convex polygons
// Helper method
bool intersect_lpt(Point a, Point b,
    Point c, Point d, Point &r) {
    double D = (b - a) % (d - c);
    if (cmp(D, 0) == 0) return false;
    double t = ((c - a) % (d - c)) / D;
    double s = -(a - c) % (b - a) / D;
    r = a + (b - a) * t;
    return cmp(t, 0) >= 0 && cmp(t, 1) <= 0 && cmp(s, 0) >= 0 && cmp(s, 1) <= 0;
}

Polygon convex_intersect(Polygon P, Polygon Q) {
    const int n = P.size(), m = Q.size();
    int a = 0, b = 0, aa = 0, ba = 0;
    enum { Pin, Qin, Unknown } in = Unknown;
    Polygon R;
    do {
        int a1 = (a+n-1) % n, b1 = (b+m-1) % m;
        double C = (P[a1] - P[a]) % (Q[b1] - Q[b]);
        double A = (P[a1] - Q[b]) % (P[a] - Q[b]);
        double B = (Q[b1] - P[a]) % (Q[b] - P[a]);
        Point r;
        if (intersect_lpt(P[a1], P[a], Q[b1], Q[b], r)) {
            if (in == Unknown) aa = ba = 0;
            R.push_back(r);
            in = B > 0 ? Pin : A > 0 ? Qin : in;
        }
        if (C == 0 && B == 0 && A == 0) {
            if (in == Pin) { b = (b+1) % m; ++ba; }
            else { a = (a+1) % n; ++aa; }
        }
        else if (C >= 0) {
            if (A > 0) { if (in == Pin) R.push_back(P[a]); a = (a+1) % n; ++aa; }
            else { if (in == Qin) R.push_back(Q[b]); b = (b+1) % m; ++ba; }
        }
        else {
            if (B > 0) { if (in == Qin) R.push_back(Q[b]); b = (b+1) % m; ++ba; }
        }
    } while (a < n && b < m);
}

```

```

    else { if (in == Pin) R.push_back(P[a]); a = (a+1)%n; ++a;
    }
} while ( (aa < n || ba < m) && aa < 2*n && ba < 2*m );
if (in == Unknown) {
    if (in_convex(Q, P[0])) return P;
    if (in_convex(P, Q[0])) return Q;
}
return R;
// Find the diameter of polygon.
// Rotating callipers
double convex_diameter(Polygon pt) {
    const int n = pt.size();
    int is = 0, js = 0;
    for (int i = 1; i < n; ++i) {
        if (pt[i].y > pt[is].y) is = i;
        if (pt[i].y < pt[js].y) js = i;
    }
    double maxd = (pt[is]-pt[js]).norm();
    int i, maxi, j, maxj;
    i = maxi = is;
    j = maxj = js;
    do {
        int jj = j+1; if (jj == n) jj = 0;
        if ((pt[i] - pt[jj]).norm() > (pt[i] - pt[j]).norm()) j = (j+1) % n;
        else i = (i+1) % n;
        if ((pt[i]-pt[jj]).norm() > maxd) {
            maxd = (pt[i]-pt[jj]).norm();
            maxi = i; maxj = j;
        }
    } while (i != is || j != js);
    return maxd; /* farthest pair is (maxi, maxj). */
}
// Check if we can form triangle with edges x, y, z.
bool isSquare(long long x) { /* */
bool isIntegerCoordinates(int x, int y, int z) {
    long long s=(long long) (x+y+z)*(x+y-z)*(x+z-y)*(y+z-x);
    return (s%4==0 && isSquare(s/4));
}

// Smallest enclosing circle:
// Given N points. Find the smallest circle enclosing these points.
// Amortized complexity: O(N)
struct SmallestEnclosingCircle {
    Circle getCircle(vector<Point> points) {
        assert(!points.empty());
        random_shuffle(points.begin(), points.end());
        Circle c(points[0], 0);
        int n = points.size();
        for (int i = 1; i < n; i++)
            if ((points[i] - c).len() > c.r + EPS) {
                c = Circle(points[i], 0);
                for (int j = 0; j < i; j++)
                    if ((points[j] - c).len() > c.r + EPS) {
                        c = Circle((points[i] + points[j]) / 2, (points[i] -
                            points[j]).len() / 2);
                        for (int k = 0; k < j; k++)
                            if ((points[k] - c).len() > c.r + EPS)
                                c = getCircle(points[i], points[j],
                                    points[k]);
                    }
            }
        return c;
    }
};
// NOTE: This code work only when a, b, c are not collinear and no 2
// points are same --> DO NOT
// copy and use in other cases.
Circle getCircle(Point a, Point b, Point c) {
    assert(a != b && b != c && a != c);
    assert(ccw(a, b, c));
    double d = 2.0 * (a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y
        - b.y));
    assert(fabs(d) > EPS);
    double x = (a.norm() * (b.y - c.y) + b.norm() * (c.y - a.y) + c.norm()
        * (a.y - b.y)) / d;
    double y = (a.norm() * (c.x - b.x) + b.norm() * (a.x - c.x) + c.norm()
        * (b.x - a.x)) / d;
    Point p(x, y);
    return Circle(p, (p - a).len());
}
bool inside(const Point &u, const vector<Point> &a) {
    for (int i = 0; i < n; i++) {
        if (cmp((a[i] - u) % (a[i] == n - 1 ? 0 : i + 1) - u), 0.0) != 0)
            continue;
        if (cmp((a[i] - u) * (a[i] == n - 1 ? 0 : i + 1) - u), 0.0) > 0)
            continue;
        return 1;
    }
    int res = 0;
    for (int i = 0; i < n; i++) {
        Point v = a[i], w = a[i == n - 1 ? 0 : i + 1];
        if (cmp(v.x, w.x) == 0) continue;
        if (v.x > w.x) swap(v, w);
        if (u.x < v.x - EPS) continue;
        if (u.x > w.x - EPS) continue;
        res ^= (cmp((u - v) % (w - v), 0) >= 0);
    }
    return res;
}

```

```

vector<int> where (n, -1); // corresponding row for each column
for (int row = 0, col = 0; col < n; ++col) {
    // find the maximum abs value on the current column to reduce
    // precision errors
    int maxRow = row;
    for (int i = row + 1; i < m; ++i) {
        if (abs(a[i][col]) > abs(a[maxRow][col]))
            maxRow = i;
    }
    // if cannot find anything rather than zero then forget the current
    // column
    if (abs(a[maxRow][col]) < EPS) continue;
    if (maxRow != row) swap(a[maxRow], a[row]);
    where[col] = row;
    for (int i = 0; i < m; ++i) if (i != row) {
        double coef = a[i][col] / a[row][col];
        for (int j = col; j <= n; ++j) {
            a[i][j] -= a[row][j] * coef;
        }
    }
    ++row; // only when found a non-zero element
}
ans.assign(m, 0); // default value = 0
for (int i = 0; i < n; ++i) if (where[i] != -1) {
    ans[i] = a[where[i]][n] / a[where[i]][i];
}
// recheck
for (int i = 0; i < m; ++i) {
    double sum = 0;
    for (int j = 0; j < n; ++j) {
        sum += a[i][j] * ans[j];
    }
    if (abs(sum - a[i][n]) > EPS) return 0; // no solution
}
// search for independent variables
for (int i = 0; i < n; ++i) if (where[i] == -1) return INF; // infinite
many solution
return 1; // one solution saved in vector ans
}

```

## 4.2 Simplex Algorithm

```

/**
 * minimize c^T * x
 * subject to Ax <= b
 * and x >= 0
 * The input matrix a will have the following form
 * 0 c c c c c
 * b A A A A A
 * b A A A A A
 * b A A A A A
 * Result vector will be: val x x x x x
 */
typedef long double ld;
const ld EPS = 1e-8;
struct LPSolver {
    static vector<ld> simplex(vector<vector<ld>>> a) {
        int n = (int) a.size() - 1;
        int m = (int) a[0].size() - 1;
        vector<int> left(n + 1);
        vector<int> up(m + 1);
        iota(left.begin(), left.end(), m);
        iota(up.begin(), up.end(), 0);
        auto pivot = [&](int x, int y) {
            swap(left[x], up[y]);
            ld k = a[x][y];
            a[x][y] = 1;
            vector<int> pos;
            for (int j = 0; j < m; j++) {
                a[x][j] /= k;
                if (fabs(a[x][j]) > EPS) pos.push_back(j);
            }
            for (int i = 0; i <= n; i++) {
                if (fabs(a[i][y]) < EPS || i == x) continue;
                k = a[i][y];
                a[i][y] = 0;
                for (int j : pos) a[i][j] -= k * a[x][j];
            }
        };
        while (1) {
            int x = -1;
            for (int i = 1; i <= n; i++) {
                if (a[i][0] < -EPS && (x == -1 || a[i][0] < a[x][0])) {
                    x = i;
                }
            }
            if (x == -1) break;
            int y = -1;
            for (int j = 1; j <= m; j++) {
                if (a[x][j] < -EPS && (y == -1 || a[x][j] < a[x][y])) {
                    y = j;
                }
            }
            if (y == -1) return vector<ld>(); // infeasible
            pivot(x, y);
        }
        while (1) {
            int y = -1;
            for (int j = 1; j <= m; j++) {
                if (a[0][j] > EPS && (y == -1 || a[0][j] > a[0][y])) {
                    y = j;
                }
            }
            if (y == -1) break;
            int x = -1;
            for (int i = 1; i <= n; i++) {
                if (a[i][y] > EPS && (x == -1 || a[i][0] / a[i][y] < a[x][0] /
                    a[x][y])) {
                    x = i;
                }
            }
            if (x == -1) return vector<ld>(); // unbounded
        }
    }
};

```

## 4 Numerical algorithms

### 4.1 Gauss Elimination

```

const int INF = 1e9;
const double EPS = 1e-9;
int gauss(vector<vector<double>> &a, vector<double> &ans) {
    int m = a.size(), n = a[0].size() - 1;

```

```

        pivot(x, y);
    }
    vector<ld> ans(m + 1);
    for (int i = 1; i <= n; i++) {
        if (left[i] <= m) ans[left[i]] = a[i][0];
    }
    ans[0] = -a[0][0];
    return ans;
}
};

```

### 4.3 NTT

```

//Poly Invert:  $R(2n) = 2R(n) - R(n) \cdot 2 \cdot F$  where  $R(z) = \text{invert } F(z)$ 
//Poly Sqrt:  $2 \cdot S(2n) = S(n) + F \cdot S(n) \cdot 2^{-1}$ 
const int MOD = 998244353;
struct NTT {
    int base = 1;
    int maxBase = 0;
    int root = 2;
    vector<int> w = {0, 1};
    vector<int> rev = {0, 1};
    NTT() {
        int u = MOD - 1;
        while (u % 2 == 0) {
            u >>= 1;
            maxBase++;
        }
        while (power(root, 1 << maxBase) != 1 || power(root, 1 << (maxBase - 1)) == 1) root++;
    }
    void ensure(int curBase) {
        assert(curBase <= maxBase);
        if (curBase <= base) return;
        rev.resize(1 << curBase);
        for (int i = 0; i < (1 << curBase); i++) {
            rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (curBase - 1));
        }
        w.resize(1 << curBase);
        for (; base < curBase; base++) {
            int wc = power(root, 1 << (maxBase - base - 1));
            for (int i = 1 << (base - 1); i < (1 << base); i++) {
                w[i << 1] = w[i];
                w[i << 1 | 1] = mul(w[i], wc);
            }
        }
    }
    void fft(vector<int> &a) {
        int n = a.size();
        int curBase = 0;
        while ((1 << curBase) < n) curBase++;
        int shift = base - curBase;
        for (int i = 0; i < n; i++) {
            if (i < (rev[i] >> shift)) swap(a[i], a[rev[i] >> shift]);
        }
        for (int k = 1; k < n; k <<= 1) {
            for (int i = 0; i < k; i++) {
                for (int j = i; j < n; j += k * 2) {
                    int foo = a[j];
                    int bar = mul(a[j + k], w[i + k]);
                    a[j] = add(foo, bar);
                    a[j + k] = sub(foo, bar);
                }
            }
        }
    }
    vector<int> mult(vector<int> a, vector<int> b) {
        int nResult = a.size() + b.size() - 1;
        int curBase = 0;
        while ((1 << curBase) < nResult) curBase++;
        ensure(curBase);
        int n = 1 << curBase;
        a.resize(n), b.resize(n);
        fft(a);
        fft(b);
        int invN = inv(n);
        for (int i = 0; i < n; i++) {
            a[i] = mul(mul(a[i], b[i]), invN);
        }
        reverse(a.begin() + 1, a.end());
        fft(a);
        a.resize(nResult);
        return a;
    }
    vector<int> polyInv(vector<int> r, vector<int> f) {
        vector<int> foo = mult(r, f);
        foo.resize(f.size());
        foo[0] = sub(2, foo[0]);
        for (int i = 1; i < foo.size(); i++) {
            foo[i] = sub(0, foo[i]);
        }
        vector<int> res = mult(r, foo);
        res.resize(f.size());
        return res;
    }
    vector<int> polySqrt(vector<int> s, vector<int> invS, vector<int> f) {
        vector<int> res = mult(f, invS);
        res.resize(f.size());
        for (int i = 0; i < s.size(); i++) {
            res[i] = add(res[i], s[i]);
        }
        for (int i = 0; i < res.size(); i++) {
            res[i] = mul(res[i], INV_2);
        }
        return res;
    }
    vector<int> getSqrt(vector<int> c, int sz) {
        vector<int> sqrtC = {1}, invSqrtC = {1}; //change this if c[0] != 1
        for (int k = 1; k < (1 << sz); k <<= 1) {
            vector<int> foo(c.begin(), c.begin() + (k * 2));
            vector<int> bar = sqrtC;
            bar.resize(bar.size() * 2, 0);
            vector<int> tempInv = polyInv(invSqrtC, bar);
        }
    }
};

```

## 4.4 FFT

```

typedef complex<double> cplx_t;
typedef vector<complex<double> > VC;
const double PI = acos(-1);
struct FFT {
    static void fft(VC &u, int sign) {
        int n = u.size();
        double theta = 2. * PI * sign / n;
        for (int m = n; m >= 2; m >= 1, theta *= 2.) {
            cplx_t w(1, 0), wDelta = polar(1., theta);
            for (int i = 0, mh = m >> 1; i < mh; i++) {
                for (int j = i; j < n; j += m) {
                    int k = j + mh;
                    cplx_t temp = u[j] - u[k];
                    u[j] += u[k];
                    u[k] = w * temp;
                }
                w *= wDelta;
            }
        }
        for (int i = 1, j = 0; i < n; i++) {
            for (int k = n >> 1; k > (j ^= k); k >>= 1);
            if (j < i) {
                swap(u[i], u[j]);
            }
        }
    }

    static vector<int> mul(const vector<int> &a, const vector<int> &b) {
        int newSize = a.size() + b.size() - 1;
        int fftSz = 1;
        while (fftSz < newSize) {
            fftSz <<= 1;
        }
        VC aa(fftSz, 0.), bb(fftSz, 0.);
        for (int i = 0; i < a.size(); i++) {
            aa[i] = a[i];
        }
        for (int i = 0; i < b.size(); i++) {
            bb[i] = b[i];
        }
        fft(aa, 1);
        fft(bb, 1);
        for (int i = 0; i < fftSz; i++) {
            aa[i] *= bb[i];
        }
        fft(aa, -1);
        vector<int> res(newSz);
        for (int i = 0; i < newSz; i++) {
            res[i] = (int)(aa[i].real() / fftSz + 0.5);
        }
        return res;
    }
};

```

## 4.5 Bitwise FFT

```

/*
 * matrix:
 * +1 +1
 * +1 -1
 */
void XORFFT(int a[], int n, int p, int invert) {
    for (int i = 1; i < n; i <= 1) {
        for (int j = 0; j < n; j += i < 1) {
            for (int k = 0; k < i; k++) {
                int u = a[j + k], v = a[i + j + k];
                a[j + k] = u + v;
                if (a[j + k] >= p) a[j + k] -= p;
                a[i + j + k] = u - v;
                if (a[i + j + k] < 0) a[i + j + k] += p;
            }
        }
    }
    if (invert) {
        long long inv = fpow(n, p - 2, p);
        for (int i = 0; i < n; i++) a[i] = a[i] * inv % p;
    }
}

/*
 * Matrix:
 * +1 +1
 * +1 +0
 */
void ORFFT(int a[], int n, int p, int invert) {
    for (int i = 1; i < n; i <= 1) {
        for (int j = 0; j < n; j += i < 1) {
            for (int k = 0; k < i; k++) {
                int u = a[j + k], v = a[i + j + k];
                if (!invert) {
                    a[j + k] = u + v;
                    a[i + j + k] = u;
                    if (a[j + k] >= p) a[j + k] -= p;
                }
            }
        }
    }
}

```





```

    IP::a[i] = ((long long) 3111 * i * i * i - (long long) 54 * i * i +
               13 * i) % mod;
}
assert(IP::calc(1234, 4) == IP::a[1234]);
cerr << "\nTime elapsed: " << 1000 * clock() / CLOCKS_PER_SEC << "ms\n";
return 0;
}

```

## 4.8 Binary vector space

```

int basis[d]; // basis[i] keeps the mask of the vector whose f value is i
int sz; // Current size of the basis

void insertVector(int mask) {
    for (int i = 0; i < d; i++) {
        if ((mask & 1 << i) == 0) continue; // continue if i != f(mask)

        if (!basis[i]) { // If there is no basis vector with the i'th bit set,
                        // then insert this vector into the basis
            basis[i] = mask;
            ++sz;
            return;
        }

        mask ^= basis[i]; // Otherwise subtract the basis vector from this vector
    }
}

```

## 4.9 DiophanteMod

```

// 1 <= a * k <= r (k > 0) (1 <= r) (mod)
long long solve(long long l, long long r, long long a, long long mod) {
    if (a == 0) return INF;
    if (a * 2 > mod) return solve(mod - r, mod - l, mod - a, mod);
    long long firstVal = getCeil(l, a);
    if (a * firstVal <= r) return firstVal;
    if (mod % a == 0) return INF;
    long long kPrime = solve(l % a, r % a, a - mod % a, a);
    if (kPrime == INF) return INF;
    long long res = getCeil(kPrime * mod + 1, a);
    return getCeil(kPrime * mod + 1, a);
}

```

## 4.10 Berlekamp-Massey

```

#include <bits/stdc++.h>
using namespace std;
// linear recurrence: a[i] = sum(a[i - j] * p[j]) (sum from 1->m)
// calculate a[k] in O(m^2 log(k)) (better than matrix multiplication O(m^3 log(k)))
// Usage:
// calculate a[0], ..., a[m + m] (2 * m elements is enough) and call calc(
// vector<int>, int)
template<const int maxn, const int mod>
struct linear_solver {
    static const long long sqmod = (long long) mod * mod;
    int n;
    int a[maxn], h[maxn], s[maxn], t[maxn];
    long long t_[maxn];

    inline int fpow(int a, long long b) {
        int res = 1;
        while (b) {
            if (b & 1) res = (long long) res * a % mod;
            a = (long long) a * a % mod;
            b >>= 1;
        }
        return res;
    }

    inline vector<int> BM(vector<int> x) {
        vector<int> ls, cur;
        int lf, ld;
        for (int i = 0; i < (int) x.size(); i++) {
            long long t = 0;
            for (int j = 0; j < (int) cur.size(); j++) {
                t += (long long) x[i - j - 1] * cur[j];
                t -= sqmod <= t ? sqmod : 0;
            }
            t %= mod;
            if ((t - x[i]) % mod == 0) continue;
            if (!cur.size()) {
                cur.resize(i + 1);
                lf = i;
                ld = t - x[i];
                ld += ld < 0 ? mod : 0;
                continue;
            }
            int k = (long long) (t - x[i] + mod) * fpow(ld, mod - 2) % mod;
            vector<int> c(i - lf - 1);
            c.push_back(k);
            for (int j = 0; j < (int) ls.size(); j++) {
                c.push_back((long long) k * (mod - ls[j]) % mod);
            }
            if (c.size() < cur.size()) c.resize(cur.size());
            for (int j = 0; j < (int) cur.size(); j++) {
                c[j] += cur[j];
                c[j] -= mod <= c[j] ? mod : 0;
            }
            if (i - lf + (int) ls.size() >= (int) cur.size()) {
                ls = cur, lf = i;
                ld = t - x[i];
                ld += ld < 0 ? mod : 0;
            }
        }
    }
}

```

```

    }
    cur = c;
    for (int i = 0; i < (int) cur.size(); i++) {
        cur[i] = (cur[i] % mod + mod) % mod;
    }
    return cur;
}

inline void mult(int* p, int* q) {
    for (int i = 0; i < n + n; i++) t_[i] = 0;
    for (int i = 0; i < n; i++) if (p[i]) {
        for (int j = 0; j < n; j++) {
            t_[i + j] += (long long) p[i] * q[j];
            t_[i + j] -= sqmod <= t_[i + j] ? sqmod : 0;
        }
    }
    for (int i = n + n - 1; n <= i; i--) if (t_[i]) {
        t_[i] %= mod;
        for (int j = n - 1; ~j; j--) {
            t_[i - j - 1] += t_[i] * h[j];
            t_[i - j - 1] -= sqmod <= t_[i - j - 1] ? sqmod : 0;
        }
    }
    for (int i = 0; i < n; i++) p[i] = t_[i] % mod;
}

inline long long calc(long long k) {
    for (int i = n; ~i; i--) {
        s[i] = t[i] = 0;
    }
    s[0] = 1;
    if (n != 1) {
        t[1] = 1;
    }
    else {
        t[0] = h[0];
    }
    while (k) {
        if (k & 1) mult(s, t);
        mult(t, t); k >>= 1;
    }
    long long sum = 0;
    for (int i = 0; i < n; i++) {
        sum = (sum + (long long) s[i] * a[i]) % mod;
    }
    return (sum % mod + mod) % mod;
}

inline int calc(vector<int> x, long long k) {
    if (k < (int) x.size()) return x[k];
    vector<int> v = BM(x);
    n = v.size();
    if (!n) return 0;
    for (int i = 0; i < n; i++) h[i] = v[i], a[i] = x[i];
    return calc(k);
}
}

```

```

};
linear_solver<1 << 18, (int) 1e9 + 7> ls;

```

```

const int mod = (int) 1e9 + 7;
const int maxn = 1 << 20;
int a[maxn];
int sa[maxn];

int main() {
    a[1] = 1;
    for (int i = 2; i < maxn; i++) {
        a[i] = ((long long) a[i - 1] * 3 + (long long) a[i - 2] * 5) % mod;
    }
    vector<int> sample;
    for (int i = 0; i < maxn; i++) {
        if (i) sa[i] = sa[i - 1];
        sa[i] = (sa[i] + a[i]) % mod;
        if (i < 2 * 1 + 4) {
            sample.push_back(sa[i]);
        }
    }
    cerr << ls.calc(sample, 50000) << " " << sa[50000] << "\n";
    cerr << "\nTime elapsed: " << 1000 * clock() / CLOCKS_PER_SEC << "ms\n";
    return 0;
}

```

## 4.11 Linear Sieve

```

vector<int> prime;
bool is_composite[MXN];
void sieve(int n) {
    fill(is_composite, is_composite + n, false);
    for (int i = 2; i < n; ++i) {
        if (!is_composite[i]) prime.push_back(i);
        for (int j = 0; j < prime.size() && i * prime[j] < n; ++j) {
            is_composite[i * prime[j]] = true;
            if (i % prime[j] == 0) break;
        }
    }
}

```

## 5 Graph algorithms

### 5.1 Arborescence

```

namespace Arborescence {
    const int maxv = 2550;
    const int maxe = maxv * maxv;
    const long long INF = (long long) 1e18;

    struct edge_t {
        int u, v;
    };
}

```

```

    long long w;
    edge_t(int u = 0, int v = 0, long long w = 0) : u(u), v(v), w(w) {}
} edge[maxe];

int ec;
int id[maxv], pre[maxv];
long long in[maxv];
int vis[maxv];

void init() {
    ec = 0;
}

void add(int u, int v, int w) {
    // 1-indexed
    edge[++ec] = edge_t(u, v, w);
}

long long mst(int n, int rt) {
    long long res = 0;
    int idx;
    while (1) {
        for (int i = 1; i <= n; i++) {
            in[i] = INF, vis[i] = -1, id[i] = -1;
        }
        for (int i = 1; i <= ec; i++) {
            int u = edge[i].u, v = edge[i].v;
            if (u == v || in[v] <= edge[i].w) continue;
            in[v] = edge[i].w, pre[v] = u;
        }
        pre[rt] = rt, in[rt] = 0;
        for (int i = 1; i <= n; i++) {
            res += in[i];
            if (in[i] == INF) return -1;
        }

        idx = 0;
        for (int i = 1; i <= n; i++) {
            if (vis[i] != -1) continue;
            int u = i, v;
            while (vis[u] == -1) {
                vis[u] = i;
                u = pre[u];
            }
            if (vis[u] != i || u == rt) continue;
            for (v = u, u = pre[u], idx++; u != v; u = pre[u]) id[u] =
                idx;
            id[v] = idx;
        }
        if (idx == 0) return res;
        for (int i = 1; i <= n; i++) if (id[i] == -1) id[i] = ++idx;
        for (int i = 1; i <= ec; i++) {
            int u = edge[i].u, v = edge[i].v;
            edge[i].u = id[u], edge[i].v = id[v];
            edge[i].w -= in[v];
        }
        n = idx, rt = id[rt];
    }
    return res;
}

```

## 5.2 Arborescence With Trace

```

namespace Arborescence {
    static const int maxv = 2555 + 5;
    static const int maxe = maxv * maxv;
    int n, m, root;
    int pre[maxv], node[maxv], vis[maxv], best[maxv];
    struct Cost;
    vector<Cost> costlist;
    struct Cost {
        int id;
        long long val;
        int used, a, b, pos;
        Cost() { val = -1; used = 0; }
        Cost(int id, long long val, bool temp) : id(id), val(val) {
            a = b = -1, used = 0;
            pos = costlist.size();
            costlist.push_back(*this);
        }
        Cost(int a, int b) : a(a), b(b) {
            id = -1;
            val = costlist[a].val - costlist[b].val;
            used = 0; pos = costlist.size();
            costlist.push_back(*this);
        }
        void push() {
            if (id == -1) {
                costlist[a].used += used;
                costlist[b].used -= used;
            }
        }
    };
    struct Edge {
        int u, v;
        Cost cost;
        Edge() {}
        Edge(int id, int u, int v, long long c) : u(u), v(v) {
            cost = Cost(id, c, 0);
        }
    } edge[maxe];

    void init(int _n) {
        n = _n; m = 0;
        costlist.clear();
    }

    void add(int id, int u, int v, long long c) {
        // zero indexed
        edge[m++] = Edge(id, u, v, c);
    }

    long long mst(int root) {
        long long res = 0;

```

```

        while (1) {
            for (int i = 0; i < n; i++) best[i] = -1;
            for (int e = 0; e < m; e++) {
                int u = edge[e].u, v = edge[e].v;
                if ((best[v] == -1 || edge[e].cost.val < costlist[best[v]].
                    val) && u != v) {
                    pre[v] = u;
                    best[v] = edge[e].cost.pos;
                }
            }
            for (int i = 0; i < n; i++) if (i != root && best[i] == -1)
                return -1;
            int cntnode = 0;
            memset(node, -1, sizeof(node));
            memset(vis, -1, sizeof(vis));
            for (int i = 0; i < n; i++) if (i != root) {
                res += costlist[best[i]].val;
                costlist[best[i]].used++;
                int v = i;
                while (vis[v] != i && node[v] == -1 && v != root) {
                    vis[v] = i;
                    v = pre[v];
                }
                if (v != root && node[v] == -1) {
                    for (int u = pre[v]; u != v; u = pre[u]) node[u] =
                        cntnode;
                    node[v] = cntnode++;
                }
            }
            if (cntnode == 0) break;
            for (int i = 0; i < n; i++) if (node[i] == -1) node[i] = cntnode
                ++;
            for (int e = 0; e < m; e++) {
                int v = edge[e].v;
                edge[e].u = node[edge[e].u];
                edge[e].v = node[edge[e].v];
                if (edge[e].u != edge[e].v) edge[e].cost = Cost(edge[e].cost.
                    pos, best[v]);
            }
            n = cntnode;
            root = node[root];
        }
        return res;
    }

    vector<int> trace() {
        vector<int> res;
        for (int i = costlist.size() - 1; i >= 0; i--) costlist[i].push();
        for (int i = 0; i < costlist.size(); i++) {
            Cost cost = costlist[i];
            if (cost.id != -1 && cost.used > 0) res.push_back(cost.id);
        }
        return res;
    }
}

```

## 5.3 Bridges and Articulations

```

void dfs(int u, int p) {
    num[u] = low[u] = ++cnt;
    st.push_back(u);
    for (auto v : adj[u]) {
        if (!num[v]) {
            if (u == 1) nChild++; // root = 1
            dfs(v, u);
            if (low[v] >= num[u]) { // u is articulation point, EXCEPT for 1
                isArticulation[u] = 1;
                numComp++;
                while (!st.empty()) {
                    int x = st.back();
                    st.pop_back();
                    ls[numComp].push_back(x);
                    if (x == v) break;
                }
                ls[numComp].push_back(u);
            }
            if (low[v] > num[u]) { // u - v bridge
                nBridge++;
            }
            low[u] = min(low[u], low[v]);
        } else if (v != p) {
            low[u] = min(low[u], num[v]);
        }
    }
}

if (nChild <= 1) isArticulation[1] = 0;

```

## 5.4 Bipartite Maximum Matching

```

struct BipartiteGraph {
    vector< vector<int> > a;
    vector<int> match;
    vector<bool> was;
    int m, n;

    BipartiteGraph(int m, int n) {
        // zero-indexed
        this->m = m; this->n = n;
        a.resize(m);
        match.assign(n, -1);
        was.assign(n, false);
    }

    void addEdge(int u, int v) {
        a[u].push_back(v);
    }

    bool dfs(int u) {
        for (int v : a[u]) if (!was[v]) {

```

```

        was[v] = true;
        if (match[v] == -1 || dfs(match[v])) {
            match[v] = u;
            return true;
        }
    }
    return false;
}

int maximumMatching() {
    vector<int> buffer;
    for (int i = 0; i < m; ++i) buffer.push_back(i);
    bool stop = false;
    int ans = 0;
    do {
        stop = true;
        for (int i = 0; i < n; ++i) was[i] = false;
        for (int i = (int)buffer.size() - 1; i >= 0; --i) {
            int u = buffer[i];
            if (dfs(u)) {
                ++ans;
                stop = false;
                buffer[i] = buffer.back();
                buffer.pop_back();
            }
        }
    } while (!stop);
    return ans;
}

vector<int> konig() {
    // returns minimum vertex cover, run this after maximumMatching()
    vector<bool> matched(m);
    for (int i = 0; i < n; ++i) {
        if (match[i] != -1) matched[match[i]] = true;
    }
    queue<int> Q;
    was.assign(m + n, false);
    for (int i = 0; i < m; ++i) {
        if (!matched[i]) {
            was[i] = true;
            Q.push(i);
        }
    }

    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int v : a[u]) if (!was[m + v]) {
            was[m + v] = true;
            if (match[v] != -1 && !was[match[v]]) {
                was[match[v]] = true;
                Q.push(match[v]);
            }
        }
    }

    vector<int> res;
    for (int i = 0; i < m; ++i) {
        if (!was[i]) res.push_back(i);
    }
    for (int i = m; i < m + n; ++i) {
        if (was[i]) res.push_back(i);
    }

    return res;
}
};

```

## 5.5 General Matching

```

/*
 * Complexity:  $O(E \cdot \sqrt{V})$ 
 * Indexing from 1
 */
struct Blossom {
    static const int MAXV = 1e3 + 5;
    static const int MAXE = 1e6 + 5;
    int n, E, lst[MAXV], next[MAXE], adj[MAXE];
    int nxt[MAXV], mat[MAXV], dad[MAXV], col[MAXV];
    int que[MAXV], qh, qt;
    int vis[MAXV], act[MAXV];
    int tag, total;

    void init(int n) {
        this->n = n;
        for (int i = 0; i <= n; ++i) {
            lst[i] = nxt[i] = mat[i] = vis[i] = 0;
        }
        E = 1, tag = total = 0;
    }

    void add(int u, int v) {
        if (!mat[u] && !mat[v]) mat[u] = v, mat[v] = u, total++;
        E++, adj[E] = v, next[E] = lst[u], lst[u] = E;
        E++, adj[E] = u, next[E] = lst[v], lst[v] = E;
    }

    int lca(int u, int v) {
        tag++;
        for (; ; swap(u, v)) {
            if (u) {
                if (vis[u] == tag) return u;
                vis[u] = tag;
                u = nxt[mat[u]];
            }
        }
    }

    void blossom(int u, int v, int g) {
        while (dad[u] != g) {
            nxt[u] = v;
            if (col[mat[u]] == 2) {
                col[mat[u]] = 1;
                que[++qt] = mat[u];
            }
        }
    }
}

```

```

    }
    if (u == dad[u]) dad[u] = g;
    if (mat[u] == dad[mat[u]]) dad[mat[u]] = g;
    v = mat[u];
    u = nxt[v];
}

int augment(int s) {
    for (int i = 1; i <= n; ++i) {
        col[i] = 0;
        dad[i] = i;
    }
    qh = 0; que[qt = 1] = s; col[s] = 1;
    for (int u, v, i; qh < qt; ) {
        act[u = que[++qh]] = 1;
        for (i = lst[u]; i; i = next[i]) {
            v = adj[i];
            if (col[v] == 0) {
                nxt[v] = u;
                col[v] = 2;
                if (!mat[v]) {
                    for (; v; v = u) {
                        u = mat[nxt[v]];
                        mat[v] = nxt[v];
                        mat[nxt[v]] = v;
                    }
                    return 1;
                }
                col[mat[v]] = 1;
                que[++qt] = mat[v];
            }
            else if (dad[u] != dad[v] && col[v] == 1) {
                int g = lca(u, v);
                blossom(u, v, g);
                blossom(v, u, g);
                for (int j = 1; j <= n; ++j) {
                    dad[j] = dad[dad[j]];
                }
            }
        }
    }
    return 0;
}

int maxmat() {
    for (int i = 1; i <= n; ++i) {
        if (!mat[i]) {
            total += augment(i);
        }
    }
    return total;
}
}

```

## 5.6 Dinic Flow

```

/*
 * U = max capacity
 * Complexity:  $O(V^2 \cdot E)$ 
 *  $O(\min(E^{1/2}, V^{2/3}) \cdot E)$  if  $U = 1$ 
 *  $O(V^{1/2} \cdot E)$  for bipartite matching.
 */
template <typename flow_t = int>
struct DinicFlow {
    const flow_t INF = numeric_limits<flow_t>::max() / 2; // 1e9

    int n, s, t;
    vector<vector<int>> adj;
    vector<int> d, cur;
    vector<int> to;
    vector<flow_t> c, f;

    DinicFlow(int n, int s, int t) : n(n), s(s), t(t), adj(n, vector<int>()),
        d(n, -1), cur(n, 0) {}

    int addEdge(int u, int v, flow_t _c) {
        adj[u].push_back(to.size());
        to.push_back(v); f.push_back(0); c.push_back(_c);
        adj[v].push_back(to.size());
        to.push_back(u); f.push_back(0); c.push_back(0);
        return (int)to.size() - 2;
    }

    bool bfs() {
        fill(d.begin(), d.end(), -1);
        d[s] = 0;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (auto edgeId : adj[u]) {
                int v = to[edgeId];
                if (d[v] == -1 && f[edgeId] < c[edgeId]) {
                    d[v] = d[u] + 1;
                    if (v == t) return 1;
                    q.push(v);
                }
            }
        }
        return d[t] != -1;
    }

    flow_t dfs(int u, flow_t res) {
        if (u == t || !res) return res;
        for (int &i = cur[u]; i < adj[u].size(); ++i) {
            int edgeId = adj[u][i];
            int v = to[edgeId];
            if (d[v] == d[u] + 1 && f[edgeId] < c[edgeId]) {
                flow_t foo = dfs(v, min(res, c[edgeId] - f[edgeId]));
                if (foo) {
                    f[edgeId] += foo;
                    f[edgeId ^ 1] -= foo;
                    return foo;
                }
            }
        }
        return 0;
    }
}

```

```

    }
    return 0;
}

flow_t maxFlow() {
    flow_t res = 0;
    while (bfs()) {
        fill(cur.begin(), cur.end(), 0);
        while (flow_t aug = dfs(s, INF)) res += aug;
    }
    return res;
}
};

```

## 5.7 Dinic Flow With Scaling

```

/*
    U = max capacity
    Complexity:  $O(V * E * \log(U))$ 
     $O(\min(E^{1/2}, V^{2/3}) * E)$  if  $U = 1$ 
     $O(V^{1/2} * E)$  for bipartite matching.
    Tested: https://vn.spoj.com/problems/FFLOW/
    --> CHANGE LIM TO MAX CAPACITY<--
*/
template <typename flow_t = int>
struct DinicFlow {
    const flow_t INF = numeric_limits<flow_t>::max() / 2; // 1e9

    int n, s, t;
    vector<vector<int>>> adj;
    vector<int> d, cur;
    vector<int> to;
    vector<flow_t> c, f;

    DinicFlow(int n, int s, int t) : n(n), s(s), t(t), adj(n, vector<int>()),
        d(n, -1), cur(n, 0) {}

    int addEdge(int u, int v, flow_t _c) {
        adj[u].push_back(to.size());
        to.push_back(v); f.push_back(0); c.push_back(_c);
        adj[v].push_back(to.size());
        to.push_back(u); f.push_back(0); c.push_back(0);
        return (int)to.size() - 2;
    }

    bool bfs(flow_t lim) {
        fill(d.begin(), d.end(), -1);
        d[s] = 0;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (auto edgeId : adj[u]) {
                int v = to[edgeId];
                if (d[v] == -1 && lim <= c[edgeId] - f[edgeId]) {
                    d[v] = d[u] + 1;
                    if (v == t) return 1;
                    q.push(v);
                }
            }
        }
        return d[t] != -1;
    }

    flow_t dfs(int u, flow_t res) {
        if (u == t || !res) return res;
        for (int &i = cur[u]; i < adj[u].size(); i++) {
            int edgeId = adj[u][i];
            int v = to[edgeId];
            if (d[v] == d[u] + 1 && res <= c[edgeId] - f[edgeId]) {
                flow_t foo = dfs(v, res);
                if (foo) {
                    f[edgeId] += foo;
                    f[edgeId ^ 1] -= foo;
                    return foo;
                }
            }
        }
        return 0;
    }

    flow_t maxFlow() {
        flow_t res = 0;
        flow_t lim = (flow_t)1 << (int)(round(log2(INF))); // change this to
        // max capacity
        while (lim >= 1) {
            if (!bfs(lim)) {
                lim >>= 1;
                continue;
            }
            fill(cur.begin(), cur.end(), 0);
            while (flow_t aug = dfs(s, lim)) res += aug;
        }
        return res;
    }
};

```

## 5.8 Gomory Hu Tree

```

// a weighted tree that represents the minimum s-t cuts for all s-t pairs in
// the graph
vector<pair<pair<int, int>, flow_t>> gomory_hu() {
    vector<pair<pair<int, int>, flow_t>> tree;
    vector<int> p(n);
    for (int u = 1; u < n; u++) {
        tree.push_back(make_pair(make_pair(p[u], u), maxflow(u, p[u])));
        for (int v = u + 1; v < n; ++v) {

```

```

            if (lev[v] && p[v] == p[u]) {
                p[v] = u;
            }
        }
        return tree;
    }
};

```

## 5.9 Min Cost-Max Flow

```

/*
    Complexity:  $O(V^2 * E^2)$ 
     $O(VE)$  phases,  $O(VE)$  for SPFA
    Tested: https://open.kattis.com/problems/mincostmaxflow
*/
template <typename flow_t = int, typename cost_t = int>
struct MinCostMaxFlow {
    const flow_t FLOW_INF = numeric_limits<flow_t>::max() / 2; // 1e9
    const cost_t COST_INF = numeric_limits<cost_t>::max() / 2; // 1e9

    int n, s, t;
    vector<vector<int>>> adj;
    vector<int> to;
    vector<flow_t> f, c;
    vector<cost_t> cost;

    vector<cost_t> d;
    vector<bool> inQueue;
    vector<int> prev;

    MinCostMaxFlow(int n, int s, int t) : n(n), s(s), t(t), adj(n, vector<int>
        >()), d(n, -1), inQueue(n, 0), prev(n, -1) {}

    int addEdge(int u, int v, flow_t _c, cost_t _cost) {
        adj[u].push_back(to.size());
        to.push_back(v); f.push_back(0); c.push_back(_c); cost.push_back(_
            _cost);
        adj[v].push_back(to.size());
        to.push_back(u); f.push_back(0); c.push_back(0); cost.push_back(-
            _cost);
        return (int)to.size() - 2;
    }

    pair<flow_t, cost_t> maxFlow() {
        flow_t res = 0;
        cost_t minCost = 0;
        while (1) {
            fill(prev.begin(), prev.end(), -1);
            fill(d.begin(), d.end(), COST_INF);
            d[s] = 0;
            inQueue[s] = 1;
            queue<int> q;
            q.push(s);
            while (!q.empty()) {
                int u = q.front();
                q.pop();
                inQueue[u] = 0;
                for (int id : adj[u]) {
                    int v = to[id];
                    if (d[v] > d[u] + cost[id] && f[id] < c[id]) {
                        d[v] = d[u] + cost[id];
                        prev[v] = id;
                        if (!inQueue[v]) {
                            inQueue[v] = 1;
                            q.push(v);
                        }
                    }
                }
            }
            if (prev[t] == -1) break;
            int x = t;
            flow_t now = FLOW_INF;
            while (x != s) {
                int id = prev[x];
                now = min(now, c[id] - f[id]);
                x = to[id ^ 1];
            }
            x = t;
            while (x != s) {
                int id = prev[x];
                minCost += cost[id] * now;
                f[id] += now;
                f[id ^ 1] -= now;
                x = to[id ^ 1];
            }
            res += now;
        }
        return (res, minCost);
    }
};

```

## 5.10 Min Cost Max Flow Potential

```

/*
    Complexity:  $O(VE * E \log N + VE)$ 
     $O(VE)$  phases,  $O(E \log N)$  for Dijkstra,  $O(VE)$  for the initial SPFA
    Tested: https://open.kattis.com/problems/mincostmaxflow
    https://codeforces.com/problemset/problem/164/C (92ms vs 936ms)
    --> RUN INIT BEFORE MAXFLOW IF WE HAVE NEG-EDGES <--
*/
template <typename flow_t = int, typename cost_t = int>
struct MinCostMaxFlow {
    const flow_t FLOW_INF = numeric_limits<flow_t>::max() / 2; // 1e9
    const cost_t COST_INF = numeric_limits<cost_t>::max() / 2; // 1e9

    int n, s, t;
    vector<vector<int>>> adj;

```

```

vector<int> to;
vector<flow_t> f, c;
vector<cost_t> cost;

vector<cost_t> pot;
vector<cost_t> d;
vector<int> prev;
vector<bool> used;

MinCostMaxFlow(int n, int s, int t) : n(n), s(s), t(t), adj(n, vector<int>
    >()), d(n, -1), prev(n, -1), pot(n, 0), used(n, 0) {}

int addEdge(int u, int v, flow_t _c, cost_t _cost) {
    adj[u].push_back(to.size());
    to.push_back(v); f.push_back(0); c.push_back(_c); cost.push_back(
        _cost);
    adj[v].push_back(to.size());
    to.push_back(u); f.push_back(0); c.push_back(0); cost.push_back(-
        _cost);
    return (int)to.size() - 2;
}

bool dijkstra() {
    fill(prev.begin(), prev.end(), -1);
    fill(d.begin(), d.end(), COST_INF);
    fill(used.begin(), used.end(), 0);
    d[s] = 0;
    set<pair<cost_t, int>> ss;
    ss.insert({0, s});
    while (!ss.empty()) {
        int u = ss.begin()->second; ss.erase(ss.begin());
        if (used[u]) continue;
        used[u] = 1;
        for (int id : adj[u]) {
            int v = to[id];
            cost_t now = d[u] + pot[u] - pot[v] + cost[id];
            if (!used[v] && d[v] > now && f[id] < c[id]) {
                d[v] = now;
                prev[v] = id;
                ss.insert({d[v], v});
            }
        }
    }
    for (int i = 0; i < n; i++) pot[i] += d[i];
    return prev[t] != -1;
}

pair<flow_t, cost_t> maxFlow() {
    flow_t res = 0;
    cost_t minCost = 0;
    while (dijkstra()) {
        int x = t;
        flow_t now = FLOW_INF;
        while (x != s) {
            int id = prev[x];
            now = min(now, c[id] - f[id]);
            x = to[id ^ 1];
        }
        x = t;
        while (x != s) {
            int id = prev[x];
            minCost += cost[id] * now;
            f[id] += now;
            f[id ^ 1] -= now;
            x = to[id ^ 1];
        }
        res += now;
    }
    return {res, minCost};
}

void init() {
    fill(pot.begin(), pot.end(), COST_INF);
    pot[s] = 0;
    bool changed = 1;
    while (changed) { // be careful for NEG cycle
        changed = 0;
        for (int i = 0; i < n; i++) if (pot[i] < COST_INF) {
            for (int id : adj[i]) {
                int v = to[id];
                if (pot[v] > pot[i] + cost[id] && f[id] < c[id]) {
                    pot[v] = pot[i] + cost[id];
                    changed = 1;
                }
            }
        }
    }
}
};

```

## 5.11 Bounded Feasible Flow

```

struct BoundedFlow {
    int low[N][N], high[N][N];
    int c[N][N];
    int f[N][N];
    int n, s, t;

    void reset() {
        memset(low, 0, sizeof low);
        memset(high, 0, sizeof high);
        memset(c, 0, sizeof c);
        memset(f, 0, sizeof f);
        n = s = t = 0;
    }

    void addEdge(int u, int v, int d, int c) {
        low[u][v] = d; high[u][v] = c;
    }

    int flow;
    int trace[N];

    bool findPath() {

```

```

        memset(trace, 0, sizeof trace);
        queue<int> Q;
        Q.push(s);
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v = 1; v <= n; ++v) if (c[u][v] > f[u][v] && !trace[v]) {
                trace[v] = u;
                if (v == t) return true;
                Q.push(v);
            }
        }
        return false;
    }

    void incFlow() {
        int delta = INF;
        for (int v = t; v != s; v = trace[v])
            delta = min(delta, c[trace[v]][v] - f[trace[v]][v]);
        for (int v = t; v != s; v = trace[v])
            f[trace[v]][v] += delta, f[v][trace[v]] -= delta;
        flow += delta;
    }

    int maxFlow() {
        flow = 0;
        while (findPath()) incFlow();
        return flow;
    }

    bool feasible() {
        c[t][s] = INF;
        s = n + 1; t = n + 2;
        int sum = 0;
        for (int u = 1; u <= n; ++u) for (int v = 1; v <= n; ++v) {
            c[s][v] += low[u][v];
            c[u][t] += low[u][v];
            c[u][v] += high[u][v] - low[u][v];
            sum += low[u][v];
        }
        n += 2;
        return maxFlow() == sum;
    }
};

```

## 5.12 Hungarian Algorithm

```

struct BipartiteGraph {
    const int INF = 1e9;

    vector<vector<int>> c; // cost matrix
    vector<int> fx, fy; // potentials
    vector<int> matchX, matchY; // corresponding vertex
    vector<int> trace; // last vertex from the left side
    vector<int> d, arg; // distance from the tree && the corresponding node
    queue<int> Q; // queue used for BFS

    int n; // assume that |L| = |R| = n
    int start; // current root of the tree
    int finish; // leaf node of the augmenting path

    BipartiteGraph(int n) {
        this->n = n;
        c = vector<vector<int>>(n + 1, vector<int>(n + 1, INF));
        fx = fy = matchX = matchY = trace = d = arg = vector<int>(n + 1);
    }

    void addEdge(int u, int v, int cost) { c[u][v] = min(c[u][v], cost); }
    int cost(int u, int v) { return c[u][v] - fx[u] - fy[v]; }

    void initBFS(int root) {
        start = root;
        Q = queue<int>(); Q.push(start);
        for (int i = 1; i <= n; ++i) {
            trace[i] = 0;
            d[i] = cost(start, i);
            arg[i] = start;
        }
    }

    int findPath() {
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v = 1; v <= n; ++v) if (trace[v] == 0) {
                int w = cost(u, v);
                if (w == 0) {
                    trace[v] = u;
                    if (matchY[v] == 0) return v;
                    Q.push(matchY[v]);
                }
                if (d[v] > w) d[v] = w, arg[v] = u;
            }
        }
        return 0;
    }

    void enlarge() {
        for (int y = finish, next; y; y = next) {
            int x = trace[y];
            next = matchX[x];
            matchX[x] = y;
            matchY[y] = x;
        }
    }

    void update() {
        int delta = INF;
        for (int i = 1; i <= n; ++i) if (trace[i] == 0) delta = min(delta, d[i]);
        fx[start] += delta;
        for (int i = 1; i <= n; ++i) {
            if (trace[i] != 0) {
                fx[matchY[i]] += delta;
                fy[i] -= delta;
            }
        }
    }
};

```

```

    } else {
        d[i] -= delta;
        if (d[i] == 0) {
            trace[i] = arg[i];
            if (matchY[i] == 0)
                finish = i;
            else
                Q.push(matchY[i]);
        }
    }
}

void hungarian() {
    for (int i = 1; i <= n; ++i) {
        initBFS(i);
        do {
            finish = findPath();
            if (finish == 0) update();
        } while (finish == 0);
        enlarge();
    }
}

void show() {
    int ans = 0;
    for (int i = 1; i <= n; ++i) if (matchX[i]) ans += c[i][matchX[i]];
    cout << ans << endl;
    for (int i = 1; i <= n; ++i) cout << i << ' ' << matchX[i] << endl;
}
};

```

## 5.13 Undirected mincut

```

/*
 * Find minimum cut in undirected weighted graph
 * Complexity: O(V^3)
 */
#define SW StoerWagner
#define cap_t int
namespace StoerWagner {
    int n;
    vector<vector<cap_t>> > graph;
    vector<int> cut;

    void init(int _n) {
        n = _n;
        graph = vector<vector<cap_t>>(n, vector<cap_t>(n, 0));
    }

    void addEdge(int a, int b, cap_t w) {
        if (a == b) return;
        graph[a][b] += w;
        graph[b][a] += w;
    }

    pair<cap_t, pair<int, int>> stMinCut(vector<int> &active) {
        vector<cap_t> key(n);
        vector<int> v(n);
        int s = -1, t = -1;
        for (int i = 0; i < active.size(); i++) {
            cap_t maxv = -1;
            int cur = -1;
            for (auto j : active) {
                if (v[j] == 0 && maxv < key[j]) {
                    maxv = key[j];
                    cur = j;
                }
            }
            t = s;
            s = cur;
            v[cur] = 1;
            for (auto j : active) key[j] += graph[cur][j];
        }
        return make_pair(key[s], make_pair(s, t));
    }

    cap_t solve() {
        cap_t res = numeric_limits<cap_t>::max();
        vector<vector<int>> grps;
        vector<int> active;
        cut.resize(n);
        for (int i = 0; i < n; i++) grps.emplace_back(1, i);
        for (int i = 0; i < n; i++) active.push_back(i);
        while (active.size() >= 2) {
            auto stcut = stMinCut(active);
            if (stcut.first < res) {
                res = stcut.first;
                fill(cut.begin(), cut.end(), 0);
                for (auto v : grps[stcut.second.first]) cut[v] = 1;
            }
            int s = stcut.second.first, t = stcut.second.second;
            if (grps[s].size() < grps[t].size()) swap(s, t);
            active.erase(find(active.begin(), active.end(), t));
            grps[s].insert(grps[s].end(), grps[t].begin(), grps[t].end());
            for (int i = 0; i < n; i++) {
                graph[i][s] += graph[i][t];
                graph[i][t] = 0;
            }
            for (int i = 0; i < n; i++) {
                graph[s][i] += graph[t][i];
                graph[t][i] = 0;
            }
            graph[s][s] = 0;
        }
        return res;
    }
}

```

```

struct EulerianGraph {
    vector< vector< pair<int, int> > > a;
    int num_edges;

    EulerianGraph(int n) {
        a.resize(n + 1);
        num_edges = 0;
    }

    void add_edge(int u, int v, bool undirected = true) {
        a[u].push_back(make_pair(v, num_edges));
        if (undirected) a[v].push_back(make_pair(u, num_edges));
        num_edges++;
    }

    vector<int> get_eulerian_path() {
        vector<int> path, s;
        vector<bool> was(num_edges);

        s.push_back(1);
        // start of eulerian path
        // directed graph: deg_out - deg_in == 1
        // undirected graph: odd degree
        // for eulerian cycle: any vertex is OK

        while (!s.empty()) {
            int u = s.back();
            bool found = false;
            while (!a[u].empty()) {
                int v = a[u].back().first;
                int e = a[u].back().second;
                a[u].pop_back();
                if (was[e]) continue;
                was[e] = true;
                s.push_back(v);
                found = true;
                break;
            }
            if (!found) {
                path.push_back(u);
                s.pop_back();
            }
        }
        reverse(path.begin(), path.end());
        return path;
    }
};

```

## 5.15 2-SAT

```

inline int pos(int u) { return u << 1; }
inline int neg(int u) { return u << 1 | 1; }
// ZERO-indexed
// color[i] = 1 means we choose i
struct TwoSAT {
    int n;
    int numComp;
    vector<int> adj[V];
    int low[V], num[V], root[V], cntTarjan;
    vector<int> stTarjan;
    int color[V];

    TwoSAT(int n) : n(n + 2) {
        memset(root, -1, sizeof root);
        memset(low, -1, sizeof low);
        memset(num, -1, sizeof num);
        memset(color, -1, sizeof color);
        cntTarjan = 0;
        stTarjan.clear();
    }

    // u | v
    void addEdge(int u, int v) {
        adj[u ^ 1].push_back(v);
        adj[v ^ 1].push_back(u);
    }

    void tarjan(int u) {
        stTarjan.push_back(u);
        num[u] = low[u] = cntTarjan++;
        for (int v : adj[u]) {
            if (root[v] != -1) continue;
            if (low[v] == -1) tarjan(v);
            low[u] = min(low[u], low[v]);
        }
        if (low[u] == num[u]) {
            while (1) {
                int v = stTarjan.back();
                stTarjan.pop_back();
                root[v] = numComp;
                if (u == v) break;
            }
            numComp++;
        }
    }

    bool solve() {
        for (int i = 0; i < n; i++) if (root[i] == -1) tarjan(i);
        for (int i = 0; i < n; i += 2) {
            if (root[i] == root[i ^ 1]) return 0;
            color[i >> 1] = (root[i] < root[i ^ 1]);
        }
        return 1;
    }
};

```

## 5.14 Eulerian Path/Circuit

## 5.16 SPFA

```

struct Graph {

```

```

vector< vector< pair<int, int> > > a;
vector<int> d;
int n;

Graph(int n) {
    this->n = n;
    a.resize(n);
}

void add_edge(int u, int v, int c) {
    // x[u] - x[v] <= c
    a[v].push_back(make_pair(u, c));
}

bool spfa(int s) {
    // return false if found negative cycle from s
    queue<int> Q;
    vector<bool> inqueue(n);
    d.assign(n, INF);
    d[s] = 0;
    Q.push(s); inqueue[s] = 1;

    vector<int> cnt(n);
    cnt[s] = 1;

    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); inqueue[u] = 0;
        for (auto e : a[u]) {
            int v = e.first;
            int c = e.second;
            if (d[v] > d[u] + c) {
                d[v] = d[u] + c;
                cnt[v]++;
                if (cnt[v] >= n) return false;
                if (!inqueue[v]) {
                    Q.push(v);
                    inqueue[v] = 1;
                }
            }
        }
    }

    return true;
}

int spfa(int s, int t) {
    assert(spfa(s));
    return d[t];
}
};

```

## 6 Data structures

### 6.1 Treap

```

class Treap {
    struct Node {
        int key;
        uint32_t prior;
        bool rev_lazy;
        int size;
        Node *l, *r;
        Node(int key : key(key), prior(rand()), rev_lazy(false), size(1), l(
            nullptr), r(nullptr)) {}
        ~Node() { delete l; delete r; }
    };

    inline int size(Node *x) { return x ? x->size : 0; }

    void push(Node *x) {
        if (x && x->rev_lazy) {
            x->rev_lazy = false;
            swap(x->l, x->r);
            if (x->l) x->l->rev_lazy ^= true;
            if (x->r) x->r->rev_lazy ^= true;
        }
    }

    inline void update(Node *x) {
        if (x) {
            x->size = size(x->l) + size(x->r) + 1;
        }
    }

    void join(Node *t, Node *l, Node *r) {
        push(l); push(r);
        if (!l || !r)
            t = l ? l : r;
        else if (l->prior < r->prior)
            join(l->r, l->r, r, t = l;
        else
            join(r->l, l, r->l, t = r;
        update(t);
    }

    void splitByKey(Node *v, int x, Node* &l, Node* &r) {
        if (!v) return void(l = r = nullptr);
        push(v);
        if (v->key < x)
            splitByKey(v->r, x, v->r, r), l = v;
        else
            splitByKey(v->l, x, l, v->l), r = v;
        update(v);
    }

    void splitByIndex(Node *v, int x, Node* &l, Node* &r) {
        if (!v) return void(l = r = nullptr);
        push(v);
        int index = size(v->l) + 1;
    }

```

### 6.2 Big Integer

```

typedef vector<int> bigInt;
const int BASE = 1000;
const int LENGTH = 3;

// * Refine function
bigInt & fix(bigInt &a) {
    a.push_back(0);
    for (int i = 0; i + 1 < a.size(); ++i) {
        a[i + 1] += a[i] / BASE; a[i] %= BASE;
        if (a[i] < 0) a[i] += BASE, --a[i + 1];
    }
    while (a.size() > 1 && a.back() == 0) a.pop_back();
    return a;
}

// * Constructors
bigInt big(int x) {
    bigInt result;
    while (x > 0) {
        result.push_back(x % BASE);
        x /= BASE;
    }
    return result;
}

```

```

}

bigInt big(string s) {
    bigInt result(s.size() / LENGTH + 1);
    for (int i = 0; i < s.size(); ++i) {
        int pos = (s.size() - i - 1) / LENGTH;
        result[pos] = result[pos] * 10 + s[i] - '0';
    }
    return fix(result), result;
}

// * Compare operators

int compare(bigInt &a, bigInt &b) {
    if (a.size() != b.size()) return (int)a.size() - (int)b.size();
    for (int i = (int)a.size() - 1; i >= 0; --i)
        if (a[i] != b[i]) return a[i] - b[i];
    return 0;
}

#define DEFINE_OPERATOR(x) bool operator x (bigInt &a, bigInt &b) { return
    compare(a, b) x 0; }
DEFINE_OPERATOR(==)
DEFINE_OPERATOR(!=)
DEFINE_OPERATOR(>)
DEFINE_OPERATOR(<)
DEFINE_OPERATOR(>=)
DEFINE_OPERATOR(<=)
#undef DEFINE_OPERATOR

// * Arithmetic operators

void operator += (bigInt &a, bigInt b) {
    a.resize(max(a.size(), b.size()));
    for (int i = (int)a.size() - 1; i < b.size(); ++i)
        a[i] += b[i];
    fix(a);
}

void operator -= (bigInt &a, bigInt b) {
    for (int i = 0; i < b.size(); ++i)
        a[i] -= b[i];
    fix(a);
}

void operator *= (bigInt &a, int b) {
    for (int i = 0; i < a.size(); ++i)
        a[i] *= b;
    fix(a);
}

void divide(bigInt a, int b, bigInt &q, int &r) {
    for (int i = int(a.size()) - 1; i >= 0; --i) {
        r = r * BASE + a[i];
        q.push_back(r / b); r %= b;
    }
    reverse(q.begin(), q.end());
    fix(q);
}

bigInt operator + (bigInt a, bigInt b) { a += b; return a; }
bigInt operator - (bigInt a, bigInt b) { a -= b; return a; }
bigInt operator * (bigInt a, int b) { a *= b; return a; }

bigInt operator / (bigInt a, int b) {
    bigInt q; int r = 0;
    divide(a, b, q, r);
    return q;
}

int operator % (bigInt a, int b) {
    bigInt q; int r = 0;
    divide(a, b, q, r);
    return r;
}

bigInt operator * (bigInt a, bigInt b) {
    bigInt result(a.size() + b.size());
    for (int i = 0; i < a.size(); ++i)
        for (int j = 0; j < b.size(); ++j)
            result[i + j] += a[i] * b[j];
    return fix(result);
}

// * I/O routines

istream& operator >> (istream& cin, bigInt &a) {
    string s; cin >> s;
    a = big(s);
    return cin;
}

ostream& operator << (ostream& cout, const bigInt &a) {
    cout << a.back();
    for (int i = (int)a.size() - 2; i >= 0; --i)
        cout << setw(LENGTH) << setfill('0') << a[i];
    return cout;
}

```

## 6.3 Convex Hull IT

```

struct Line {
    long long a, b; // y = ax + b
    Line(long long a = 0, long long b = -INF) : a(a), b(b) {}
    long long eval(long long x) {
        return a * x + b;
    }
};

struct Node {
    Line line;
    int l, r;
    Node *left, *right;
};

```

```

Node(int l, int r) : l(l), r(r), left(NULL), right(NULL), line() {}

void update(int i, int j, Line newLine) {
    if (r < i || j < l) return;
    if (i <= l && r <= j) {
        Line AB = line, CD = newLine;
        if (AB.eval(valueX[l]) < CD.eval(valueX[l])) swap(AB, CD);
        if (AB.eval(valueX[r]) >= CD.eval(valueX[r])) {
            line = AB;
            return;
        }
        int mid = valueX[l + r >> 1];
        if (AB.eval(mid) < CD.eval(mid))
            line = CD, left->update(i, j, AB);
        else
            line = AB, right->update(i, j, CD);
        return;
    }
    left->update(i, j, newLine);
    right->update(i, j, newLine);
}

long long getMax(int i) {
    if (l == r) return line.eval(valueX[i]);
    if (i <= (l + r >> 1)) return max(line.eval(valueX[i]), left->getMax(i));
    return max(line.eval(valueX[i]), right->getMax(i));
}

Node* build(int l, int r) {
    Node *x = new Node(l, r);
    if (l == r) return x;
    x->left = build(l, l + r >> 1);
    x->right = build(l + r >> 1 + 1, r);
    return x;
}

```

## 6.4 Link Cut Tree

```

// treequery returns sum weight of child in subtree
// to change it to sum weight of child in root->u
// comment all update on w and return x->s instead
struct node_t {
    node_t *p, *l, *r;
    int size, rev;
    int s, w;
    node_t() : p(0), l(0), r(0), size(1), rev(0), s(1), w(1) {}
};

int isrt(node_t* x) {
    return !(x->p) || (x->p->l != x && x->p->r != x);
}

int left(node_t* x) {
    return x->p->l == x;
}

void setchild(node_t* x, node_t* p, int l) {
    (l ? p->l : p->r) = x;
    if (x) x->p = p;
}

void push(node_t* x) {
    node_t* u = x->l;
    node_t* v = x->r;
    if (x->rev) {
        if (u) swap(u->l, u->r), u->rev ^= 1;
        if (v) swap(v->l, v->r), v->rev ^= 1;
        x->rev = 0;
    }
}

int size(node_t* x) {
    return x ? x->size : 0;
}

int sum(node_t* x) {
    return x ? x->s : 0;
}

void pull(node_t* x) {
    x->size = size(x->l) + 1 + size(x->r);
    x->s = sum(x->l) + x->w + sum(x->r);
}

void rotate(node_t* x) {
    node_t *p = x->p, *g = p->p;
    int l = left(x);
    setchild(l ? x->r : x->l, p, l);
    if (!isrt(p)) setchild(x, g, left(p));
    else x->p = g;
    setchild(p, x, !l);
    pull(p);
}

node_t* splay(node_t* x) {
    push(x);
    while (!isrt(x)) {
        node_t *p = x->p, *g = p->p;
        if (g) push(g);
        push(p), push(x);
        if (!isrt(p)) rotate(left(x) != left(p) ? x : p);
        rotate(x);
    }
    pull(x);
    return x;
}

node_t* access(node_t* x) {
    node_t* z = 0;
    for (node_t* y = x; y; y = y->p) {
        splay(y);

```



```

        y->w += sum(y->r);
        y->r = z;
        y->w -= sum(y->r);
        pull(z = y);
    }
    splay(x);
    return z;
}

void link(node_t* x, node_t* p) {
    access(x); access(p);
    x->p = p;
    p->w += sum(x);
}

void cut(node_t* x) {
    access(x);
    x->l->p = 0, x->l = 0;
    pull(x);
}

void makeroot(node_t* x) {
    access(x);
    x->rev ^= 1;
    swap(x->l, x->r);
}

node_t* findroot(node_t* x) {
    access(x);
    while (x->l) push(x), x = x->l;
    push(x);
    return splay(x);
}

node_t* lca(node_t* x, node_t* y) {
    if (findroot(x) != findroot(y)) return 0;
    access(x);
    return access(y);
}

int connect(node_t* x, node_t* y) {
    if (x == y) return 1;
    access(x), access(y);
    return x->p != 0;
}

int treequery(node_t* x) {
    access(x);
    return x->w;
}

```

## 6.5 Ordered Set

```

#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
/*
    change null_type to int if we want to use map instead

```

```

    find_by_order(k) returns an iterator to the k-th element (0-indexed)
    order_of_key(k) returns numbers of item being strictly smaller than k
*/
template<typename T = int>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

```

## 6.6 Unordered Map

```

struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049b133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().
            time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
unordered_map<long long, int, custom_hash> safe_map;
gp_hash_table<long long, int, custom_hash> safe_hash_table;

```

## 7 Miscellaneous

### 7.1 RNG

```

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
//use mt19937_64 if we want 64-bit number

```

### 7.2 SQRT forloop

```

for (int i = 1, la; i <= n; i = la + 1) {
    la = n / (n / i);
    //n / x yields the same value for i <= x <= la.
}

```