# Contents



# 1 Setting

## 1.1 Default code

```
#include<bits/stdc++.h>
using namespace std;
#pragma warning(disable:4996)
#pragma comment(linker, "/STACK:336777216")
using ll=long long;
using ull=unsigned long long;
using LL=__int128_t;
using ULL=__uint128_t;
using ld=long double;
ll gcd(ll a, ll b){return b?gcd(b,a%b):a;}
ll lcm(ll a, ll b){if(a&&b)return a*(b/gcd(a,b)); return a+b;}
ll powm(ll a, ll b, ll rem){ll p=1;for(;b;b/=2,a=(a*a)%rem)if(b&1)p=(p*a)%rem;return p
  ;}
int main(){
  ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
```

```
  ll i,j;
  return 0;
}
```

## 1.2 SIMD

```
#pragma GCC optimize ("O3,unroll-loops")
#pragma GCC target ("avx,avx2,fma")
#include <immintrin.h>
alignas(32) int A[8]{ 1, 2, 3, 1, 2, 3, 1, 2 }, B[8]{ 1, 2, 3, 4, 5, 6, 7, 8 };
alignas(32) int C[8]; // alignas(bit size of <type>) <type> var[256/(bit size)]
// Must compute "index is multiply of 256bit"(ex> short->16k, int->8k, ...)
__m256i a = _mm256_load_si256((__m256i*)A);
__m256i b = _mm256_load_si256((__m256i*)B);
__m256i c = _mm256_add_epi32(a, b);
_mm256_store_si256((__m256i*)C, c);

__m256i _mm256_abs_epi32 (__m256i a)
_mm256_set1_epi32(__m256i a, __m256i b)
__m256i _mm256_and_si256 (__m256i a, __m256i b)
__m256i _mm256_setzero_si256 (void)
_mm256_add_pd(__m256d a, __m256d b) // double precision(64-bit)
_mm256_sub_pd(__m256 a, __m256 b) // double precision(64-bit)
__m256d _mm256_andnot_pd (__m256d a, __m256d b) // (~a)&b
__m256i _mm256_avg_epu16 (__m256i a, __m256i b) // unsigned, (a+b+1)>>1
__m256d _mm256_ceil_pd (__m256d a)
__m256d _mm256_floor_pd (__m256d a)
__m256i _mm256_cmpeq_epi64 (__m256i a, __m256i b)
__m256i _mm256_cmpgt_epi16 (__m256i a, __m256i b)
__m256d _mm256_div_pd (__m256d a, __m256d b)
__m256i _mm256_max_epi32 (__m256i a, __m256i b)
__m256i _mm256_mul_epi32 (__m256i a, __m256i b)
__m256 _mm256_rcp_ps (__m256 a) // 1/a
__m256 _mm256_rsqrt_ps (__m256 a) // 1/sqrt(a)
__m256i _mm256_set1_epi64x (long long a)
__m256i _mm256_sign_epi16 (__m256i a, __m256i b) // a*(sign(b))
__m256i _mm256_sll_epi32 (__m256i a, __m128i count) // a << count
__m256d _mm256_sqrt_pd (__m256d a)
__m256i _mm256_sra_epi16 (__m256i a, __m128i count)
__m256i _mm256_xor_si256 (__m256i a, __m256i b)
void _mm256_zeroall (void)
void _mm256_zeroupper (void)
```

# 2 Math

## 2.1 Basic Arithmetic

```
typedef long long ll;
typedef unsigned long long ull;

// calculate lg2(a)
inline int lg2(ll a) {
    return 63 - __builtin_clzll(a);
}

// calculate the number of 1-bits
inline int bitcount(ll a) {
    return __builtin_popcountll(a);
}

// calculate ceil(a/b)
// |a|, |b| <= (2^63)-1 (does not dover -2^63)
ll ceildiv(ll a, ll b) {
    if (b < 0) return ceildiv(-a, -b);
    if (a < 0) return (-a) / b;
    return ((ull)a + (ull)b - 1ull) / b;
}

// calculate floor(a/b)
// |a|, |b| <= (2^63)-1 (does not cover -2^63)
ll floordiv(ll a, ll b) {
    if (b < 0) return floordiv(-a, -b);
    if (a >= 0) return a / b;
    return -(ll)(((ull)(-a) + b - 1) / b);
}

// calculate a*b % m
// x86-64 only
ll large_mod_mul(ll a, ll b, ll m) {
    return ll((__int128)a*(__int128)b%m);
}

// calculate a*b % m
// |m| < 2^62, x86 available
// O(logb)
ll large_mod_mul(ll a, ll b, ll m) {
    a %= m; b %= m; ll r = 0, v = a;
    while (b) {
        if (b&1) r = (r + v) % m;
        b >>= 1;
        v = (v << 1) % m;
    }
    return r;
}

// calculate n^k % m
ll modpow(ll n, ll k, ll m) {
    ll ret = 1;
    n %= m;
    while (k) {
        if (k & 1) ret = large_mod_mul(ret, n, m);
        n = large_mod_mul(n, n, m);
        k /= 2;
    }
    return ret;
}

// calculate gcd(a, b)
ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
```

```
}

// find a pair (c, d) s.t. ac + bd = gcd(a, b)
pair<ll, ll> extended_gcd(ll a, ll b) {
    if (b == 0) return { 1, 0 };
    auto t = extended_gcd(b, a % b);
    return { t.second, t.first - t.second * (a / b) };
}

// find x in [0,m) s.t. ax === gcd(a, m) (mod m)
ll modinverse(ll a, ll m) {
    return (extended_gcd(a, m).first % m + m) % m;
}

// calculate modular inverse for 1 ~ n
void calc_range_modinv(int n, int mod, int ret[]) {
    ret[1] = 1;
    for (int i = 2; i <= n; ++i)
        ret[i] = (ll)(mod - mod/i) * ret[mod%i] % mod;
}
```

## 2.2 Linear Sieve

```
struct sieve {
  const ll MAXN = 101010;
  vector<ll> sp, e, phi, mu, tau, sigma, primes;
  // sp : smallest prime factor, e : exponent, phi : euler phi, mu : mobius
  // tau : num of divisors, sigma : sum of divisors
  sieve(ll sz) {
    sp.resize(sz + 1), e.resize(sz + 1), phi.resize(sz + 1), mu.resize(sz + 1),
        tau.resize(sz + 1), sigma.resize(sz + 1);
    phi[1] = mu[1] = tau[1] = sigma[1] = 1;
    for (ll i = 2; i <= sz; i++) {
      if (!sp[i]) {
        primes.push_back(i), e[i] = 1, phi[i] = i - 1, mu[i] = -1, tau[i] = 2;
        sigma[i] = i + 1;
      }
      for (auto j : primes) {
        if (i * j > sz) break;
        sp[i * j] = j;
        if (i % j == 0) {
          e[i * j] = e[i] + 1, phi[i * j] = phi[i] * j, mu[i * j] = 0,
                tau[i * j] = tau[i] / e[i * j] * (e[i * j] + 1),
                sigma[i * j] = sigma[i] * (j - 1) / (powm(j, e[i * j]) - 1) *
                               (powm(j, e[i * j] + 1) - 1) / (j - 1);
          break;
        }
        e[i * j] = 1, phi[i * j] = phi[i] * phi[j], mu[i * j] = mu[i] * mu[j],
              tau[i * j] = tau[i] * tau[j], sigma[i * j] = sigma[i] * sigma[j];
      }
    }
  }
  sieve() : sieve(MAXN) {}
};
```

## 2.3 Primality Test

```
bool test_witness(ull a, ull n, ull s) {
    if (a >= n) a %= n;
    if (a <= 1) return true;
    ull d = n >> s;
    ull x = modpow(a, d, n);
    if (x == 1 || x == n-1) return true;
    while (s-- > 1) {
        x = large_mod_mul(x, x, n);
        if (x == 1) return false;
        if (x == n-1) return true;
    }
    return false;
}

// test whether n is prime
// based on miller-rabin test
// O(logn*logn)
bool is_prime(ull n) {
    if (n == 2) return true;
    if (n < 2 || n % 2 == 0) return false;

    ull d = n >> 1, s = 1;
    for(; (d&1) == 0; s++) d >>= 1;

#define T(a) test_witness(a##ull, n, s)
    if (n < 4759123141ull) return T(2) && T(7) && T(61);
    return T(2) && T(325) && T(9375) && T(28178)
        && T(450775) && T(9780504) && T(1795265022);
#undef T
}
```

## 2.4 Integer Factorization (Pollard's rho)

```
ll pollard_rho(ll n) {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<ll> dis(1, n - 1);
    ll x = dis(gen);
    ll y = x;
    ll c = dis(gen);
    ll g = 1;
    while (g == 1) {
        x = (modmul(x, x, n) + c) % n;
        y = (modmul(y, y, n) + c) % n;
        y = (modmul(y, y, n) + c) % n;
        g = gcd(abs(x - y), n);
    }
    return g;
}

// integer factorization
// O(n^0.25 * logn)
void factorize(ll n, vector<ll>& fl) {
    if (n == 1) {
```

```
        return;
    }
    if (n % 2 == 0) {
        fl.push_back(2);
        factorize(n / 2, fl);
    }
    else if (is_prime(n)) {
        fl.push_back(n);
    }
    else {
        ll f = pollard_rho(n);
        factorize(f, fl);
        factorize(n / f, fl);
    }
}
```

## 2.5 Chinese Remainder Theorem

```
// find x s.t.  x === a[0] (mod n[0])
//                === a[1] (mod n[1])
//                ...
// assumption: gcd(n[i], n[j]) = 1
ll chinese_remainder(ll* a, ll* n, int size) {
    if (size == 1) return *a;
    ll tmp = modinverse(n[0], n[1]);
    ll tmp2 = (tmp * (a[1] - a[0]) % n[1] + n[1]) % n[1];
    ll ora = a[1];
    ll tgcd = gcd(n[0], n[1]);
    a[1] = a[0] + n[0] / tgcd * tmp2;
    n[1] *= n[0] / tgcd;
    ll ret = chinese_remainder(a + 1, n + 1, size - 1);
    n[1] /= n[0] / tgcd;
    a[1] = ora;
    return ret;
}
```

## 2.6 Rational Number Class

```
struct rational {
    long long p, q;

    void red() {
        if (q < 0) {
            p = -p;
            q = -q;
        }
        ll t = gcd((p >= 0 ? p : -p), q);
        p /= t;
        q /= t;
    }

    rational(): p(0), q(1) {}
    rational(long long p_): p(p_), q(1) {}
    rational(long long p_, long long q_): p(p_), q(q_) { red(); }

    bool operator==(const rational& rhs) const {
        return p == rhs.p && q == rhs.q;
    }
    bool operator!=(const rational& rhs) const {
        return p != rhs.p || q != rhs.q;
    }
    bool operator<(const rational& rhs) const {
        return p * rhs.q < rhs.p * q;
    }
    rational operator+(const rational& rhs) const {
        ll g = gcd(q, rhs.q);
        return rational(p * (rhs.q / g) + rhs.p * (q / g), (q / g) * rhs.q);
    }
    rational operator-(const rational& rhs) const {
        ll g = gcd(q, rhs.q);
        return rational(p * (rhs.q / g) - rhs.p * (q / g), (q / g) * rhs.q);
    }
    rational operator*(const rational& rhs) const {
        return rational(p * rhs.p, q * rhs.q);
    }
    rational operator/(const rational& rhs) const {
        return rational(p * rhs.q, q * rhs.p);
    }
};
```

## 2.7 Kirchoff's Theorem

그래프의 스패닝 트리의 개수를 구하는 정리.

무향 그래프의 Laplacian matrix $L$를 만든다. 이것은 (정점의 차수 대각 행렬) - (인접행렬)이다. $L$에서 행과 열을 하나씩 제거한 것을 $L'$라 하자. 어느 행/열이든 관계 없다. 그래프의 스패닝 트리의 개수는 $det(L')$이다.

## 2.8 Lucas Theorem

```
// calculate nCm % p  when p is prime
int lucas_theorem(const char *n, const char *m, int p) {
    vector<int> np, mp;
    int i;
    for (i = 0; n[i]; i++) {
        if (n[i] == '0' && np.empty()) continue;
        np.push_back(n[i] - '0');
    }
    for (i = 0; m[i]; i++) {
        if (m[i] == '0' && mp.empty()) continue;
        mp.push_back(m[i] - '0');
    }

    int ret = 1;
    int ni = 0, mi = 0;
    while (ni < np.size() || mi < mp.size()) {
        int nmod = 0, mmod = 0;
        for (i = ni; i < np.size(); i++) {
            if (i + 1 < np.size())
```

```
                np[i + 1] += (np[i] % p) * 10;
            else
                nmod = np[i] % p;
            np[i] /= p;
        }
        for (i = mi; i < mp.size(); i++) {
            if (i + 1 < mp.size())
                mp[i + 1] += (mp[i] % p) * 10;
            else
                mmod = mp[i] % p;
            mp[i] /= p;
        }
        while (ni < np.size() && np[ni] == 0) ni++;
        while (mi < mp.size() && mp[mi] == 0) mi++;
        // implement binomial. binomial(m,n) = 0 if m < n
        ret = (ret * binomial(nmod, mmod)) % p;
    }
    return ret;
}
```

## 2.9 Fast Fourier Transform

```
void fft(int sign, int n, double *real, double *imag) {
  double theta = sign * 2 * pi / n;
  for (int m = n; m >= 2; m >>= 1, theta *= 2) {
    double wr = 1, wi = 0, c = cos(theta), s = sin(theta);
    for (int i = 0, mh = m >> 1; i < mh; ++i) {
      for (int j = i; j < n; j += m) {
        int k = j + mh;
        double xr = real[j] - real[k], xi = imag[j] - imag[k];
        real[j] += real[k], imag[j] += imag[k];
        real[k] = wr * xr - wi * xi, imag[k] = wr * xi + wi * xr;
      }
      double _wr = wr * c - wi * s, _wi = wr * s + wi * c;
      wr = _wr, wi = _wi;
    }
  }
  for (int i = 1, j = 0; i < n; ++i) {
    for (int k = n >> 1; k > (j ^= k); k >>= 1)
      ;
    if (j < i) swap(real[i], real[j]), swap(imag[i], imag[j]);
  }
}
// Compute Poly(a)*Poly(b), write to r; Indexed from 0
// O(n*logn)
int mult(int *a, int n, int *b, int m, int *r) {
  const int maxn = 100;
  static double ra[maxn], rb[maxn], ia[maxn], ib[maxn];
  int fn = 1;
  while (fn < n + m) fn <<= 1;  // n + m: interested length
  for (int i = 0; i < n; ++i) ra[i] = a[i], ia[i] = 0;
  for (int i = n; i < fn; ++i) ra[i] = ia[i] = 0;
  for (int i = 0; i < m; ++i) rb[i] = b[i], ib[i] = 0;
  for (int i = m; i < fn; ++i) rb[i] = ib[i] = 0;
  fft(1, fn, ra, ia);
  fft(1, fn, rb, ib);
  for (int i = 0; i < fn; ++i) {
    double real = ra[i] * rb[i] - ia[i] * ib[i];
    double imag = ra[i] * ib[i] + rb[i] * ia[i];
    ra[i] = real, ia[i] = imag;
  }
  fft(-1, fn, ra, ia);
  for (int i = 0; i < fn; ++i) r[i] = (int)floor(ra[i] / fn + 0.5);
  return fn;
}
```

## 2.10 NTT

```
void ntt(poly& f, bool inv = 0) {
  int n = f.size(), j = 0;
  vector<ll> root(n >> 1);
  for (int i = 1; i < n; i++) {
    int bit = (n >> 1);
    while (j >= bit) {
      j -= bit;
      bit >>= 1;
    }
    j += bit;
    if (i < j) swap(f[i], f[j]);
  }
  ll ang = pw(w, (mod - 1) / n);
  if (inv) ang = pw(ang, mod - 2);
  root[0] = 1;
  for (int i = 1; i < (n >> 1); i++) root[i] = root[i - 1] * ang % mod;
  for (int i = 2; i <= n; i <<= 1) {
    int step = n / i;
    for (int j = 0; j < n; j += i) {
      for (int k = 0; k < (i >> 1); k++) {
        ll u = f[j | k], v = f[j | k | i >> 1] * root[step * k] % mod;
        f[j | k] = (u + v) % mod;
        f[j | k | i >> 1] = (u - v) % mod;
        if (f[j | k | i >> 1] < 0) f[j | k | i >> 1] += mod;
      }
    }
  }
  ll t = pw(n, mod - 2);
  if (inv)
    for (int i = 0; i < n; i++) f[i] = f[i] * t % mod;
}

vector<ll> multiply(poly& _a, poly& _b) {
  vector<ll> a(all(_a)), b(all(_b));
  int n = 2;
  while (n < a.size() + b.size()) n <<= 1;
  a.resize(n);
  b.resize(n);
  ntt(a);
  ntt(b);
  for (int i = 0; i < n; i++) a[i] = a[i] * b[i] % mod;
  ntt(a, 1);
  return a;
}
```

## 2.11 Matrix Operations

```
const int MATSZ = 100;

inline bool is_zero(double a) { return fabs(a) < 1e-9; }

// out = A^(-1), returns det(A)
// A becomes invalid after call this
// O(n^3)
double inverse_and_det(int n, double A[][MATSZ], double out[][MATSZ]) {
    double det = 1;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) out[i][j] = 0;
        out[i][i] = 1;
    }
    for (int i = 0; i < n; i++) {
        if (is_zero(A[i][i])) {
            double maxv = 0;
            int maxid = -1;
            for (int j = i + 1; j < n; j++) {
                auto cur = fabs(A[j][i]);
                if (maxv < cur) {
                    maxv = cur;
                    maxid = j;
                }
            }
            if (maxid == -1 || is_zero(A[maxid][i])) return 0;
            for (int k = 0; k < n; k++) {
                A[i][k] += A[maxid][k];
                out[i][k] += out[maxid][k];
            }
        }
        det *= A[i][i];
        double coeff = 1.0 / A[i][i];
        for (int j = 0; j < n; j++) A[i][j] *= coeff;
        for (int j = 0; j < n; j++) out[i][j] *= coeff;
        for (int j = 0; j < n; j++) if (j != i) {
            double mp = A[j][i];
            for (int k = 0; k < n; k++) A[j][k] -= A[i][k] * mp;
            for (int k = 0; k < n; k++) out[j][k] -= out[i][k] * mp;
        }
    }
    return det;
}
```

## 2.12 Gaussian Elimination

```
const double EPS = 1e-10;
typedef vector<vector<double>> VVD;

// Gauss-Jordan elimination with full pivoting.
// solving systems of linear equations (AX=B)
// INPUT:    a[][] = an n*n matrix
//           b[][] = an n*m matrix
// OUTPUT:   X      = an n*m matrix (stored in b[][])
//           A^{-1} = an n*n matrix (stored in a[][])
// O(n^3)
bool gauss_jordan(VVD& a, VVD& b) {
    const int n = a.size();
    const int m = b[0].size();
    vector<int> irow(n), icol(n), ipiv(n);

    for (int i = 0; i < n; i++) {
        int pj = -1, pk = -1;
        for (int j = 0; j < n; j++) if (!ipiv[j])
            for (int k = 0; k < n; k++) if (!ipiv[k])
                if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
        if (fabs(a[pj][pk]) < EPS) return false; // matrix is singular
        ipiv[pk]++;
        swap(a[pj], a[pk]);
        swap(b[pj], b[pk]);
        irow[i] = pj;
        icol[i] = pk;

        double c = 1.0 / a[pk][pk];
        a[pk][pk] = 1.0;
        for (int p = 0; p < n; p++) a[pk][p] *= c;
        for (int p = 0; p < m; p++) b[pk][p] *= c;
        for (int p = 0; p < n; p++) if (p != pk) {
            c = a[p][pk];
            a[p][pk] = 0;
            for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
            for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
        }
    }
    for (int p = n - 1; p >= 0; p--) if (irow[p] != icol[p]) {
        for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
    }
    return true;
}
```

## 2.13 Simplex Algorithm

```
// Two-phase simplex algorithm for solving linear programs of the form
//     maximize     c^T x
//     subject to   Ax <= b
//                   x >= 0
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
// OUTPUT: value of the optimal solution (infinity if unbounded
//          above, nan if infeasible)
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).
typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
const double EPS = 1e-9;

struct LPSolver {
    int m, n;
```

```
    VI B, N;
    VVD D;

    LPSolver(const VVD& A, const VD& b, const VD& c) :
        m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
        for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
        for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[i][n + 1] = b[i];
         }
        for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
        N[n] = -1; D[m + 1][n] = 1;
    }

    void pivot(int r, int s) {
        double inv = 1.0 / D[r][s];
        for (int i = 0; i < m + 2; i++) if (i != r)
            for (int j = 0; j < n + 2; j++) if (j != s)
                D[i][j] -= D[r][j] * D[i][s] * inv;
        for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] *= inv;
        for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] *= -inv;
        D[r][s] = inv;
        swap(B[r], N[s]);
    }

    bool simplex(int phase) {
        int x = phase == 1 ? m + 1 : m;
        while (true) {
            int s = -1;
            for (int j = 0; j <= n; j++) {
                if (phase == 2 && N[j] == -1) continue;
                if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s])
                   s = j;
            }
            if (D[x][s] > -EPS) return true;
            int r = -1;
            for (int i = 0; i < m; i++) {
                if (D[i][s] < EPS) continue;
                if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
                    (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r])
                       r = i;
            }
            if (r == -1) return false;
            pivot(r, s);
        }
    }

    double solve(VD& x) {
        int r = 0;
        for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
        if (D[r][n + 1] < -EPS) {
            pivot(r, n);
            if (!simplex(1) || D[m + 1][n + 1] < -EPS)
                return -numeric_limits<double>::infinity();
            for (int i = 0; i < m; i++) if (B[i] == -1) {
                int s = -1;
                for (int j = 0; j <= n; j++)
                    if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N
                        [s]) s = j;
                pivot(i, s);
            }
        }
        if (!simplex(2))
            return numeric_limits<double>::infinity();
        x = VD(n);
        for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
        return D[m][n + 1];
    }
};
```

## 2.14 Nim Game

Nim Game의 해법 : 모두 XOR했을 때 0이 아니면 첫번째, 0이면 두번째 플레이어가 승리.

Grundy Number : XOR(MEX(next state grundy))

Subtraction Game : 한 번에 $k$개까지의 돌만 가져갈 수 있는 경우, 각 더미의 돌의 개수를 $k+1$로 나눈 나머지를 XOR 합하여 판단한다.

Index-k Nim : 한 번에 최대 k개의 더미를 골라 각각의 더미에서 아무렇게나 돌을 제거할 수 있을 때, 각 binary digit에 대하여 합을 $k+1$로 나눈 나머지를 계산한다. 만약 이 나머지가 모든 digit에 대하여 0이라면 두번째, 하나라도 0이 아니라면 첫번째 플레이어가 승리.

## 2.15 Lifting The Exponent

For any integers $x$, $y$ a positive integer $n$, and a prime number $p$ such that $p \nmid x$ and $p \nmid y$, the following statements hold:

- When $p$ is odd:
  - If $p \mid x-y$, then $\nu_p(x^n - y^n) = \nu_p(x-y) + \nu_p(n)$.
  - If $n$ is odd and $p \mid x+y$, then $\nu_p(x^n + y^n) = \nu_p(x+y) + \nu_p(n)$.
- When $p = 2$:
  - If $2 \mid x-y$ and $n$ is even, then $\nu_2(x^n - y^n) = \nu_2(x-y) + \nu_2(x+y) + \nu_2(n) - 1$.
  - If $2 \mid x-y$ and $n$ is odd, then $\nu_2(x^n - y^n) = \nu_2(x-y)$.
  - Corollary:
    - If $4 \mid x-y$, then $\nu_2(x+y) = 1$ and thus $\nu_2(x^n - y^n) = \nu_2(x-y) + \nu_2(n)$.
- For all $p$:
  - If $\gcd(n,p) = 1$ and $p \mid x-y$, then $\nu_p(x^n - y^n) = \nu_p(x-y)$.
  - If $\gcd(n,p) = 1$, $p \mid x+y$ and $n$ odd, then $\nu_p(x^n + y^n) = \nu_p(x+y)$.

### 2.16 NTT primes

$998\,244\,353 = 119 \times 2^{23} + 1$. Primitive root: 3.

$985\,661\,441 = 235 \times 2^{22} + 1$. Primitive root: 3.

$1\,012\,924\,417 = 483 \times 2^{21} + 1$. Primitive root: 5.

# 3 Data Structure

### 3.1 Order statistic tree(Policy Based Data Structure)

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/detail/standard_policies.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <functional>
#include <iostream>
using namespace __gnu_pbds;
using namespace std;
// Ordered set is a policy based data structure in g++ that keeps the unique elements
  in
// sorted order. It performs all the operations as performed by the set data structure
// in STL in log(n) complexity and performs two additional operations also in log(n)
// complexity order_of_key (k) : Number of items strictly smaller than k
// find_by_order(k) : -Kth element in a set (counting from zero) tree<key_type,
// value_type(set if null), comparator, ...>
using ordered_set =
    tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>;
using ordered_multi_set = tree<int, null_type, less_equal<int>, rb_tree_tag,
                               tree_order_statistics_node_update>;
void m_erase(ordered_multi_set &OS, int val) {
  int index = OS.order_of_key(val);
  ordered_multi_set::iterator it = OS.find_by_order(index);
  if (*it == val) OS.erase(it);
}
int main() {
  ordered_set X;
  for (int i = 1; i < 10; i += 2) X.insert(i);  // 1 3 5 7 9
  cout << boolalpha;
  cout << *X.find_by_order(2) << endl;           // 5
  cout << *X.find_by_order(4) << endl;           // 9
  cout << (X.end() == X.find_by_order(5)) << endl;  // true
  cout << X.order_of_key(-1) << endl;            // 0
  cout << X.order_of_key(1) << endl;             // 0
  cout << X.order_of_key(4) << endl;             // 2
  X.erase(3);
  cout << X.order_of_key(4) << endl;  // 1
  for (int t : X) cout << t << ' ';   // 1 5 7 9
}
```

### 3.2 Fenwick Tree

```
struct Fenwick {
  const ll MAXN = 100000;
  vector<ll> tree;
  Fenwick(ll sz) : tree(sz + 1) {}
  Fenwick() : Fenwick(MAXN) {}
  ll query(ll p) {  // sum from index 1 to p, inclusive
    ll ret = 0;
    for (; p > 0; p -= p & -p) ret += tree[p];
    return ret;
  }
  void add(ll p, ll val) {
    for (; p <= TSIZE; p += p & -p) tree[p] += val;
  }
};
```

### 3.3 Segment Tree with Lazy Propagation

```
struct segment {
#ifdef ONLINE_JUDGE
  const int TSIZE = 1 << 20;  // always 2^k form && n <= TSIZE
#else
  const int TSIZE = 1 << 3;  // always 2^k form && n <= TSIZE
#endif
  vector<ll> segtree, prop, dat;
  segment(ll n) {
    segtree.resize(TSIZE * 2);
    prop.resize(TSIZE * 2);
    dat.resize(n);
  }
  void seg_init(int nod, int l, int r) {
    if (l == r) {
      segtree[nod] = dat[l];
    } else {
      int m = (l + r) >> 1;
      seg_init(nod << 1, l, m);
      seg_init(nod << 1 | 1, m + 1, r);
      segtree[nod] = segtree[nod << 1] + segtree[nod << 1 | 1];
    }
  }

  void seg_relax(int nod, int l, int r) {
    if (prop[nod] == 0) return;
    if (l < r) {
      int m = (l + r) >> 1;
      segtree[nod << 1] += (m - l + 1) * prop[nod];
      prop[nod << 1] += prop[nod];
      segtree[nod << 1 | 1] += (r - m) * prop[nod];
      prop[nod << 1 | 1] += prop[nod];
    }
    prop[nod] = 0;
  }

  ll seg_query(int nod, int l, int r, int s, int e) {
    if (r < s || e < l) return 0;
    if (s <= l && r <= e) return segtree[nod];
    seg_relax(nod, l, r);
    int m = (l + r) >> 1;
    return seg_query(nod << 1, l, m, s, e) +
```

```
        seg_query(nod << 1 | 1, m + 1, r, s, e);
  }

  void seg_update(int nod, int l, int r, int s, int e, int val) {
    if (r < s || e < l) return;
    if (s <= l && r <= e) {
      segtree[nod] += (r - l + 1) * val;
      prop[nod] += val;
      return;
    }
    seg_relax(nod, l, r);
    int m = (l + r) >> 1;
    seg_update(nod << 1, l, m, s, e, val);
    seg_update(nod << 1 | 1, m + 1, r, s, e, val);
    segtree[nod] = segtree[nod << 1] + segtree[nod << 1 | 1];
  }
  // usage:
  // seg_update(1, 0, n - 1, qs, qe, val);
  // seg_query(1, 0, n - 1, qs, qe);
};
```

## 3.4 Persistent Segment Tree

```
// persistent segment tree impl: sum tree
// initial tree index is 0
struct pstree {
  typedef int val_t;
  const int DEPTH = 18;
  const int TSIZE = 1 << 18;
  const int MAX_QUERY = 262144;
  struct node {
    val_t v;
    node *l, *r;
  } npoll[TSIZE * 2 + MAX_QUERY * (DEPTH + 1)], *head[MAX_QUERY + 1];
  int pptr, last_q;
  void init() {
    // zero-initialize, can be changed freely
    memset(&npoll[TSIZE - 1], 0, sizeof(node) * TSIZE);

    for (int i = TSIZE - 2; i >= 0; i--) {
      npoll[i].v = 0;
      npoll[i].l = &npoll[i * 2 + 1];
      npoll[i].r = &npoll[i * 2 + 2];
    }

    head[0] = &npoll[0];
    last_q = 0;
    pptr = 2 * TSIZE - 1;
  }
  // update val to pos
  // 0 <= pos < TSIZE
  // returns updated tree index
  int update(int pos, int val, int prev) {
    head[++last_q] = &npoll[pptr++];
    node *old = head[prev], *now = head[last_q];
    int flag = 1 << DEPTH;
    for (;;) {
      now->v = old->v + val;
      flag >>= 1;
      if (flag == 0) {
        now->l = now->r = nullptr;
        break;
      }
      if (flag & pos) {
        now->l = old->l;
        now->r = &npoll[pptr++];
        now = now->r, old = old->r;
      } else {
        now->r = old->r;
        now->l = &npoll[pptr++];
        now = now->l, old = old->l;
      }
    }
    return last_q;
  }
  val_t query(int s, int e, int l, int r, node *n) {
    if (s == l && e == r) return n->v;
    int m = (l + r) / 2;
    if (m >= e)
      return query(s, e, l, m, n->l);
    else if (m < s)
      return query(s, e, m + 1, r, n->r);
    else
      return query(s, m, l, m, n->l) + query(m + 1, e, m + 1, r, n->r);
  }
  // query summation of [s, e] at time t
  val_t query(int s, int e, int t) {
    s = max(0, s);
    e = min(TSIZE - 1, e);
    if (s > e) return 0;
    return query(s, e, 0, TSIZE - 1, head[t]);
  }
};
```

## 3.5 Splay Tree

```
// example : https://www.acmicpc.net/problem/13159
struct node {
    node* l, * r, * p;
    int cnt, min, max, val;
    long long sum;
    bool inv;
    node(int _val) :
        cnt(1), sum(_val), min(_val), max(_val), val(_val), inv(false),
        l(nullptr), r(nullptr), p(nullptr) {
    }
};
node* root;

void update(node* x) {
    x->cnt = 1;
```

```
    x->sum = x->min = x->max = x->val;
    if (x->l) {
        x->cnt += x->l->cnt;
        x->sum += x->l->sum;
        x->min = min(x->min, x->l->min);
        x->max = max(x->max, x->l->max);
    }
    if (x->r) {
        x->cnt += x->r->cnt;
        x->sum += x->r->sum;
        x->min = min(x->min, x->r->min);
        x->max = max(x->max, x->r->max);
    }
}

void rotate(node* x) {
    node* p = x->p;
    node* b = nullptr;
    if (x == p->l) {
        p->l = b = x->r;
        x->r = p;
    }
    else {
        p->r = b = x->l;
        x->l = p;
    }
    x->p = p->p;
    p->p = x;
    if (b) b->p = p;
    x->p ? (p == x->p->l ? x->p->l : x->p->r) = x : (root = x);
    update(p);
    update(x);
}

// make x into root
void splay(node* x) {
    while (x->p) {
        node* p = x->p;
        node* g = p->p;
        if (g) rotate((x == p->l) == (p == g->l) ? p : x);
        rotate(x);
    }
}

void relax_lazy(node* x) {
    if (!x->inv) return;
    swap(x->l, x->r);
    x->inv = false;
    if (x->l) x->l->inv = !x->l->inv;
    if (x->r) x->r->inv = !x->r->inv;
}

// find kth node in splay tree
void find_kth(int k) {
    node* x = root;
    relax_lazy(x);
    while (true) {
        while (x->l && x->l->cnt > k) {
            x = x->l;
            relax_lazy(x);
        }
        if (x->l) k -= x->l->cnt;
        if (!k--) break;
        x = x->r;
        relax_lazy(x);
    }
    splay(x);
}

// collect [l, r] nodes into one subtree and return its root
node* interval(int l, int r) {
    find_kth(l - 1);
    node* x = root;
    root = x->r;
    root->p = nullptr;
    find_kth(r - l + 1);
    x->r = root;
    root->p = x;
    root = x;
    return root->r->l;
}

void traverse(node* x) {
    relax_lazy(x);
    if (x->l) {
        traverse(x->l);
    }
    // do something
    if (x->r) {
        traverse(x->r);
    }
}

void uptree(node* x) {
    if (x->p) {
        uptree(x->p);
    }
    relax_lazy(x);
}
```

## 3.6 Bitset to Set

```
typedef unsigned long long ull;
const int sz = 100001 / 64 + 1;
struct bset {
  ull x[sz];
  bset(){
    memset(x, 0, sizeof x);
  }
  bset operator|(const bset &o) const {
    bset a;
    for (int i = 0; i < sz; i++)a.x[i] = x[i] | o.x[i];
```

```
    return a;
  }
  bset &operator|=(const bset &o) {
    for (int i = 0; i < sz; i++)x[i] |= o.x[i];
    return *this;
  }
  inline void add(int val){
    x[val >> 6] |= (1ull << (val & 63));
  }
  inline void del(int val){
    x[val >> 6] &= ~(1ull << (val & 63));
  }
  int kth(int k){
    int i, cnt = 0;
    for (i = 0; i < sz; i++){
      int c = __builtin_popcountll(x[i]);
      if (cnt + c >= k){
        ull y = x[i];
        int z = 0;
        for (int j = 0; j < 64; j++){
          z += ((x[i] & (1ull << j)) != 0);
          if (cnt + z == k)return i * 64 + j;
        }
      }
      cnt += c;
    }
    return -1;
  }
  int lower(int z){
    int i = (z >> 6), j = (z & 63);
    if (x[i]){
      for (int k = j - 1; k >= 0; k--)if (x[i] & (1ull << k))return (i << 6) | k;
    }
    while (i > 0)
    if (x[--i])
    for (j = 63;; j--)
    if (x[i] & (1ull << j))return (i << 6) | j;
    return -1;
  }
  int upper(int z){
    int i = (z >> 6), j = (z & 63);
    if (x[i]){
      for (int k = j + 1; k <= 63; k++)if (x[i] & (1ull << k))return (i << 6) | k;
    }
    while (i < sz - 1)if (x[++i])for (j = 0;; j++)if (x[i] & (1ull << j))return (i <<
    6) | j;
    return -1;
  }
};
```

## 3.7 Li-Chao Tree

```
struct Line {
  ll a, b;
  ll get(ll x) { return a * x + b; }
};
struct Node {
  int l, r;  // child
  ll s, e;   // range
  Line line;
};
struct Li_Chao {
  vector<Node> tree;
  void init(ll s, ll e) { tree.push_back({-1, -1, s, e, {0, -INF}}); }
  void update(int node, Line v) {
    ll s = tree[node].s, e = tree[node].e, m;
    m = (s + e) >> 1;
    Line low = tree[node].line, high = v;
    if (low.get(s) > high.get(s)) swap(low, high);
    if (low.get(e) <= high.get(e)) {
      tree[node].line = high;
      return;
    }
    if (low.get(m) < high.get(m)) {
      tree[node].line = high;
      if (tree[node].r == -1) {
        tree[node].r = tree.size();
        tree.push_back({-1, -1, m + 1, e, {0, -INF}});
      }
      update(tree[node].r, low);
    } else {
      tree[node].line = low;
      if (tree[node].l == -1) {
        tree[node].l = tree.size();
        tree.push_back({-1, -1, s, m, {0, -INF}});
      }
      update(tree[node].l, high);
    }
  }
  ll query(int node, ll x) {
    if (node == -1) return -INF;
    ll s = tree[node].s, e = tree[node].e, m;
    m = (s + e) >> 1;
    if (x <= m)
      return max(tree[node].line.get(x), query(tree[node].l, x));
    else
      return max(tree[node].line.get(x), query(tree[node].r, x));
  }
  // usage : seg.init(-2e8, 2e8); seg.update(0, {-c[i], c[i] * a[i - 1]});
  // seg.query(0, a[n - 1]);
};
```

# 4 DP

## 4.1 Longest Increasing Sequence

```
// Longest increasing subsequence
// O(n*logn)
vec lis(vec& arr) {
  int n = arr.size();
  vec tmp = vec();
```

```
  vec from = vec();
  for (int x : arr) {
    int loc = lower_bound(tmp.begin(), tmp.end(), x) - tmp.begin();
    if (loc == tmp.size()) {
      tmp.push_back(x);
    } else {
      tmp[loc] = x;
    }
    from.push_back(loc);
  }
  vec lis = vec(tmp.size());
  int target = tmp.size() - 1;
  for (int i = n - 1; i >= 0; i--) {
    if (target == from[i]) {
      lis[target--] = arr[i];
    }
  }
  return lis;
}
```

## 4.2 Convex Hull Optimization

$O(n^2) \to O(n \log n)$

DP 점화식 꼴

$D[i] = \max_{j<i}(D[j] + b[j] * a[i])$ $(b[k] \leq b[k+1])$

$D[i] = \min_{j<i}(D[j] + b[j] * a[i])$ $(b[k] \geq b[k+1])$

특수조건) $a[i] \leq a[i+1]$ 도 만족하는 경우, 마지막 쿼리의 위치를 저장해두면 이분검색이 필요없어지기 때문에 amortized $O(n)$ 에 해결할 수 있음

```
struct CHTLinear {
    struct Line {
        long long a, b;
        long long y(long long x) const { return a * x + b; }
    };
    vector<Line> stk;
    int qpt;
    CHTLinear() : qpt(0) { }
    // when you need maximum :  (previous l).a < (now l).a
    // when you need minimum :  (previous l).a > (now l).a
    void pushLine(const Line& l) {
        while (stk.size() > 1) {
            Line& l0 = stk[stk.size() - 1];
            Line& l1 = stk[stk.size() - 2];
            if ((l0.b - l.b) * (l0.a - l1.a) > (l1.b - l0.b) * (l.a - l0.a)) break;
            stk.pop_back();
        }
        stk.push_back(l);
    }
    // (previous x) <= (current x)
    // it calculates max/min at x
    long long query(long long x) {
        while (qpt + 1 < stk.size()) {
            Line& l0 = stk[qpt];
            Line& l1 = stk[qpt + 1];
            if (l1.a - l0.a > 0 && (l0.b - l1.b) > x * (l1.a - l0.a)) break;
            if (l1.a - l0.a < 0 && (l0.b - l1.b) < x * (l1.a - l0.a)) break;
            ++qpt;
        }
        return stk[qpt].y(x);
    }
};
```

## 4.3 Divide & Conquer Optimization

$O(kn^2) \to O(kn \log n)$

조건 1) DP 점화식 꼴

$D[t][i] = \min_{j<i}(D[t-1][j] + C[j][i])$

조건 2) $A[t][i]$는 $D[t][i]$의 답이 되는 최소의 $j$라 할 때, 아래의 부등식을 만족해야 함

$A[t][i] \leq A[t][i+1]$

조건 2-1) 비용$C$가 다음의 사각부등식을 만족하는 경우도 조건 2)를 만족하게 됨

$C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$ $(a \leq b \leq c \leq d)$

```
//To get D[t][s...e] and range of j is [l, r]
void f(int t, int s, int e, int l, int r){
  if(s > e) return;
  int m = s + e >> 1;
  int opt = l;
  for(int i=l; i<=r; i++){
    if(D[t-1][opt] + C[opt][m] > D[t-1][i] + C[i][m]) opt = i;
  }
  D[t][m] = D[t-1][opt] + C[opt][m];
  f(t, s, m-1, l, opt);
  f(t, m+1, e, opt, r);
}
```

## 4.4 Knuth Optimization

$O(n^3) \to O(n^2)$

조건 1) DP 점화식 꼴

$D[i][j] = \min_{i<k<j}(D[i][k] + D[k][j]) + C[i][j]$

조건 2) 사각 부등식

$C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$ $(a \leq b \leq c \leq d)$

조건 3) 단조성

$C[b][c] \leq C[a][d]$ $(a \leq b \leq c \leq d)$

결론) 조건 2, 3을 만족한다면 $A[i][j]$를 $D[i][j]$의 답이 되는 최소의 $k$라 할 때, 아래의 부등식을 만족하게 됨

$A[i][j-1] \leq A[i][j] \leq A[i+1][j]$

3중 루프를 돌릴 때 위 조건을 이용하면 최종적으로 시간복잡도가 $O(n^2)$ 이 됨

```
for (i = 1; i <= n; i++) {
  cin >> a[i];
  s[i] = s[i - 1] + a[i];
  dp[i - 1][i] = 0;
  assist[i - 1][i] = i;
}
for (i = 2; i <= n; i++) {
  for (j = 0; j <= n - i; j++) {
    dp[j][i + j] = 1e9 + 7;
    for (k = assist[j][i + j - 1]; k <= assist[j + 1][i + j]; k++) {
      if (dp[j][i + j] > dp[j][k] + dp[k][i + j] + s[i + j] - s[j]) {
        dp[j][i + j] = dp[j][k] + dp[k][i + j] + s[i + j] - s[j];
        assist[j][i + j] = k;
      }
    }
  }
}
```

## 4.5 Bitset Optimization

```
#define private public
#include <bitset>
#undef private
#include <x86intrin.h>
template <size_t _Nw>
void _M_do_sub(_Base_bitset<_Nw> &A, const _Base_bitset<_Nw> &B) {
  for (int i = 0, c = 0; i < _Nw; i++)
    c = _subborrow_u64(c, A._M_w[i], B._M_w[i], (unsigned long long *)&A._M_w[i]);
}
template <>
void _M_do_sub(_Base_bitset<1> &A, const _Base_bitset<1> &B) {
  A._M_w -= B._M_w;
}
template <size_t _Nb>
bitset<_Nb> &operator-=(bitset<_Nb> &A, const bitset<_Nb> &B) {
  _M_do_sub(A, B);
  return A;
}
template <size_t _Nb>
inline bitset<_Nb> operator-(const bitset<_Nb> &A, const bitset<_Nb> &B) {
  bitset<_Nb> C(A);
  return C -= B;
}
template <size_t _Nw>
void _M_do_add(_Base_bitset<_Nw> &A, const _Base_bitset<_Nw> &B) {
  for (int i = 0, c = 0; i < _Nw; i++)
    c = _addcarry_u64(c, A._M_w[i], B._M_w[i], (unsigned long long *)&A._M_w[i]);
}
template <>
void _M_do_add(_Base_bitset<1> &A, const _Base_bitset<1> &B) {
  A._M_w += B._M_w;
}
template <size_t _Nb>
bitset<_Nb> &operator+=(bitset<_Nb> &A, const bitset<_Nb> &B) {
  _M_do_add(A, B);
  return A;
}
template <size_t _Nb>
inline bitset<_Nb> operator+(const bitset<_Nb> &A, const bitset<_Nb> &B) {
  bitset<_Nb> C(A);
  return C += B;
}
```

## 4.6 Kitamasa & Berlekamp-Massey

```
// linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$
// Time: O(n^2 \log k)
ll get_nth(Poly S, Poly tr, ll k) {  // get kth term of recurrence
  int n = sz(tr);
  auto combine = [&](Poly a, Poly b) {
    Poly res(n * 2 + 1);
    rep(i, 0, n + 1) rep(j, 0, n + 1) res[i + j] = (res[i + j] + a[i] * b[j]) % mod;
    for (int i = 2 * n; i > n; --i)
      rep(j, 0, n) res[i - 1 - j] = (res[i - 1 - j] + res[i] * tr[j]) % mod;
    res.resize(n + 1);
    return res;
  };
  Poly pol(n + 1), e(pol);
  pol[0] = e[1] = 1;
  for (++k; k; k /= 2) {
    if (k % 2) pol = combine(pol, e);
    e = combine(e, e);
  }
  ll res = 0;
  rep(i, 0, n) res = (res + pol[i + 1] * S[i]) % mod;
  return res;
}

// Usage: berlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}
// Time: O(N^2)
vector<ll> berlekampMassey(vector<ll> s) {
  ll n = s.size(), L = 0, m = 0, d, coef;
  vector<ll> C(n), B(n), T;
  C[0] = B[0] = 1;
  ll b = 1;
  for (ll i = 0; i < n; i++) {
    ++m, d = s[i] % mod;
    for (ll j = 1; j <= L; j++) d = (d + C[j] * s[i - j]) % mod;
    if (!d) continue;
    T = C, coef = d * modpow(b, mod - 2) % mod;
    for (j = m; j < n; j++) C[j] = (C[j] - coef * B[j - m]) % mod;
    if (2 * L > i) continue;
    L = i + 1 - L, B = T, b = d, m = 0;
```

```
  }
  C.resize(L + 1), C.erase(C.begin());
  for (ll& x : C) x = (mod - x) % mod;
  return C;
}
ll guess_nth_term(vector<ll> x, lint n) {
  if (n < x.size()) return x[n];
  vector<ll> v = berlekamp_massey(x);
  if (v.empty()) return 0;
  return get_nth(v, x, n);
}
```

# 5 Graph

## 5.1 SCC

```
const int MAXN = 100;
vector<int> graph[MAXN];
int up[MAXN], visit[MAXN], vtime;
vector<int> stk;
int scc_idx[MAXN], scc_cnt;

void dfs(int nod) {
    up[nod] = visit[nod] = ++vtime;
    stk.push_back(nod);
    for (int next : graph[nod]) {
        if (visit[next] == 0) {
            dfs(next);
            up[nod] = min(up[nod], up[next]);
        }
        else if (scc_idx[next] == 0)
            up[nod] = min(up[nod], visit[next]);
    }
    if (up[nod] == visit[nod]) {
        ++scc_cnt;
        int t;
        do {
            t = stk.back();
            stk.pop_back();
            scc_idx[t] = scc_cnt;
        } while (!stk.empty() && t != nod);
    }
}

// find SCCs in given directed graph
// O(V+E)
// the order of scc_idx constitutes a reverse topological sort
void get_scc() {
    vtime = 0;
    memset(visit, 0, sizeof(visit));
    scc_cnt = 0;
    memset(scc_idx, 0, sizeof(scc_idx));
    for (int i = 0; i < n; ++i)
        if (visit[i] == 0) dfs(i);
}
```

## 5.2 2-SAT

boolean variable $b_i$ 마다 $b_i$를 나타내는 정점, $\neg b_i$를 나타내는 정점 2개를 만듦. 각 clause $b_i \vee b_j$ 마다 $\neg b_i \to b_j$, $\neg b_j \to b_i$ 이렇게 edge를 이어줌. 그렇게 만든 그래프에서 SCC를 다 구함. 어떤 SCC 안에 $b_i$ 와 $\neg b_i$가 같이 포함되어있다면 해가 존재하지 않음. 아니라면 해가 존재함. 해가 존재할 때 구체적인 해를 구하는 방법. 위에서 SCC를 구하면서 SCC DAG를 만들어준다. 거기서 위상정렬을 한 후, 앞에서부터 SCC를 하나씩 봐준다. 현재 보고있는 SCC에 $b_i$가 속해있는데 얘가 $\neg b_i$보다 먼저 등장했다면 $b_i$ = false, 반대의 경우라면 $b_i$ = true, 이미 값이 assign되었다면 pass.

## 5.3 BCC, Cut vertex, Bridge

```
const int MAXN = 100;
vector<pair<int, int>> graph[MAXN];  // { next vertex id, edge id }
int up[MAXN], visit[MAXN], vtime;
vector<int> stk;

int is_cut[MAXN];             // v is cut vertex if  is_cut[v] > 0
vector<int> bridge;           // list of edge ids
vector<int> bcc_edges[MAXN];  // list of edge ids in a bcc
int bcc_cnt;

void dfs(int nod, int par_edge) {
    up[nod] = visit[nod] = ++vtime;
    int child = 0;
    for (const auto& e : graph[nod]) {
        int next = e.first, eid = e.second;
        if (eid == par_edge) continue;
        if (visit[next] == 0) {
            stk.push_back(eid);
            ++child;
            dfs(next, eid);
            if (up[next] == visit[next]) bridge.push_back(eid);
            if (up[next] >= visit[nod]) {
                ++bcc_cnt;
                do {
                    auto lasteid = stk.back();
                    stk.pop_back();
                    bcc_edges[bcc_cnt].push_back(lasteid);
                    if (lasteid == eid) break;
                } while (!stk.empty());
                is_cut[nod]++;
            }
            up[nod] = min(up[nod], up[next]);
        }
        else if (visit[next] < visit[nod]) {
            stk.push_back(eid);
            up[nod] = min(up[nod], visit[next]);
        }
    }
    if (par_edge == -1 && is_cut[nod] == 1)
        is_cut[nod] = 0;
}

// find BCCs & cut vertexs & bridges in undirected graph
```

```
// O(V+E)
void get_bcc() {
    vtime = 0;
    memset(visit, 0, sizeof(visit));
    memset(is_cut, 0, sizeof(is_cut));
    bridge.clear();
    for (int i = 0; i < n; ++i) bcc_edges[i].clear();
    bcc_cnt = 0;
    for (int i = 0; i < n; ++i) {
        if (visit[i] == 0)
            dfs(i, -1);
    }
}
```

## 5.4 Block-cut Tree

각 BCC 및 cut vertex가 block-cut tree의 vertex가 되며, BCC와 그 BCC에 속한 cut vertex 사이에 edge를 이어주면 된다.

## 5.5 Dijkstra

```
// O(ElogV)
vector<ll> dijk(ll n, ll s){
  vector<ll>dis(n,INF);
  priority_queue<pll, vector<pll>, greater<pll> > q; // pair(dist, v)
  dis[s] = 0;
  q.push({dis[s], s});
  while (!q.empty()){
    while (!q.empty() && visit[q.top().second]) q.pop();
    if (q.empty()) break;
    ll next = q.top().second; q.pop();
    visit[next] = 1;
    for (ll i = 0; i < adj[next].size(); i++)
      if (dis[adj[next][i].first] > dis[next] + adj[next][i].second){
        dis[adj[next][i].first] = dis[next] + adj[next][i].second;
        q.push({dis[adj[next][i].first], adj[next][i].first});}}
  for(ll i=0;i<n;i++)if(dis[i]==INF)dis[i]=-1;
  return dis;
}
```

## 5.6 Shortest Path Faster Algorithm

```
// shortest path faster algorithm
// average for random graph : O(E) , worst : O(VE)

const int MAXN = 20001;
const int INF = 100000000;
int n, m;
vector<pair<int, int>> graph[MAXN];
bool inqueue[MAXN];
int dist[MAXN];

void spfa(int st) {
    for (int i = 0; i < n; ++i) {
        dist[i] = INF;
    }
    dist[st] = 0;

    queue<int> q;
    q.push(st);
    inqueue[st] = true;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inqueue[u] = false;
        for (auto& e : graph[u]) {
            if (dist[u] + e.second < dist[e.first]) {
                dist[e.first] = dist[u] + e.second;
                if (!inqueue[e.first]) {
                    q.push(e.first);
                    inqueue[e.first] = true;
                }
            }
        }
    }
}
```

## 5.7 Centroid Decomposition

```
int get_siz(int v, int p = -1) {
  siz[v] = 1;
  for (auto [nxt, w] : g[v])
    if (ok(nxt)) siz[v] += get_siz(nxt, v);
  return siz[v];
}
int get_cent(int v, int p, int S) {
  for (auto [nxt, w] : g[v])
    if (ok(nxt) && siz[nxt] * 2 > S) return get_cent(nxt, v, S);
  return v;
}
void dfs(int v, int p, int depth, int len, vector<pii>& t) {
  if (len > k) return;
  t.eb(depth, len);
  for (auto [nxt, w] : g[v])
    if (ok(nxt)) dfs(nxt, v, depth + 1, len + w, t);
}
void dnc(int v) {
  int cent = get_cent(v, -1, get_siz(v));
  vector<pii> t;
  vector<int> reset;
  for (auto [nxt, w] : g[cent]) {
    if (vis[nxt]) continue;
    t.clear();
    dfs(nxt, cent, 1, w, t);
    for (auto [d, l] : t) ans = min(ans, A[k - l] + d);
    for (auto [d, l] : t) {
      if (d < A[l]) {
        A[l] = d;
        reset.pb(l);
      }
```

```
    }
  }
  for (auto l : reset) A[l] = inf;
  vis[cent] = 1;
  for (auto [nxt, w] : g[cent])
    if (!vis[nxt]) dnc(nxt);
}
void solve() {
  cin >> n >> k;
  for (int i = 1; i <= k; i++) A[i] = inf;
  rep(i, n - 1) {
    int a, b, w;
    cin >> a >> b >> w;
    g[a].eb(b, w);
    g[b].eb(a, w);
  }
  dnc(0);
  if (ans == inf) ans = -1;
  cout << ans << nl;
}
```

## 5.8 Lowest Common Ancestor

```
const int MAXN = 100;
const int MAXLN = 9;
vector<int> tree[MAXN];
int depth[MAXN];
int par[MAXLN][MAXN];

void dfs(int nod, int parent) {
    for (int next : tree[nod]) {
        if (next == parent) continue;
        depth[next] = depth[nod] + 1;
        par[0][next] = nod;
        dfs(next, nod);
    }
}

void prepare_lca() {
    const int root = 0;
    dfs(root, -1);
    par[0][root] = root;
    for (int i = 1; i < MAXLN; ++i)
        for (int j = 0; j < n; ++j)
            par[i][j] = par[i - 1][par[i - 1][j]];
}

// find lowest common ancestor in tree between u & v
// assumption : must call 'prepare_lca' once before call this
// O(logV)
int lca(int u, int v) {
    if (depth[u] < depth[v]) swap(u, v);
    if (depth[u] > depth[v]) {
        for (int i = MAXLN - 1; i >= 0; --i)
            if (depth[u] - (1 << i) >= depth[v])
                u = par[i][u];
    }
    if (u == v) return u;
    for (int i = MAXLN - 1; i >= 0; --i) {
        if (par[i][u] != par[i][v]) {
            u = par[i][u];
            v = par[i][v];
        }
    }
    return par[0][u];
}
```

## 5.9 Heavy-Light Decomposition

```
// heavy-light decomposition
//
// hld h;
// insert edges to tree[0~n-1];
// h.init(n, root);
// h.decompose(root);
// h.hldquery(u, v); // edges from u to v
struct hld {
    static const int MAXLN = 18;
    static const int MAXN = 1 << (MAXLN - 1);
    vector<int> tree[MAXN];
    int subsize[MAXN], depth[MAXN], pa[MAXLN][MAXN];

    int chead[MAXN], cidx[MAXN];
    int lchain;
    int flatpos[MAXN + 1], fptr;

    void dfs(int u, int par) {
        pa[0][u] = par;
        subsize[u] = 1;
        for (int v : tree[u]) {
            if (v == pa[0][u]) continue;
            depth[v] = depth[u] + 1;
            dfs(v, u);
            subsize[u] += subsize[v];
        }
    }

    void init(int size, int root)
    {
        lchain = fptr = 0;
        dfs(root, -1);
        memset(chead, -1, sizeof(chead));

        for (int i = 1; i < MAXLN; i++) {
            for (int j = 0; j < size; j++) {
                if (pa[i - 1][j] != -1) {
                    pa[i][j] = pa[i - 1][pa[i - 1][j]];
                }
            }
        }
    }
```

```
    void decompose(int u) {
        if (chead[lchain] == -1) chead[lchain] = u;
        cidx[u] = lchain;
        flatpos[u] = ++fptr;

        int maxchd = -1;
        for (int v : tree[u]) {
            if (v == pa[0][u]) continue;
            if (maxchd == -1 || subsize[maxchd] < subsize[v]) maxchd = v;
        }
        if (maxchd != -1) decompose(maxchd);

        for (int v : tree[u]) {
            if (v == pa[0][u] || v == maxchd) continue;
            ++lchain; decompose(v);
        }
    }

    int lca(int u, int v) {
        if (depth[u] < depth[v]) swap(u, v);

        int logu;
        for (logu = 1; 1 << logu <= depth[u]; logu++);
        logu--;

        int diff = depth[u] - depth[v];
        for (int i = logu; i >= 0; --i) {
            if ((diff >> i) & 1) u = pa[i][u];
        }
        if (u == v) return u;

        for (int i = logu; i >= 0; --i) {
            if (pa[i][u] != pa[i][v]) {
                u = pa[i][u];
                v = pa[i][v];
            }
        }
        return pa[0][u];
    }

    // TODO: implement query functions
    inline int query(int s, int e) {
        return 0;
    }

    int subquery(int u, int v) {
        int uchain, vchain = cidx[v];
        int ret = 0;
        for (;;) {
            uchain = cidx[u];
            if (uchain == vchain) {
                ret += query(flatpos[v], flatpos[u]);
                break;
            }

            ret += query(flatpos[chead[uchain]], flatpos[u]);
            u = pa[0][chead[uchain]];
        }
        return ret;
    }

    inline int hldquery(int u, int v) {
        int p = lca(u, v);
        return subquery(u, p) + subquery(v, p) - query(flatpos[p], flatpos[p]);
    }
};
```

## 5.10 Bipartite Matching (Hopcroft-Karp)

```
// in: n, m, graph
// out: match, matched
// vertex cover: (reached[0][left_node] == 0) || (reached[1][right_node] == 1)
// O(E*sqrt(V))
struct BipartiteMatching {
    int n, m;
    vector<vector<int>> graph;
    vector<int> matched, match, edgeview, level;
    vector<int> reached[2];
    BipartiteMatching(int n, int m) : n(n), m(m), graph(n), matched(m, -1), match(n,
      -1) {}

    bool assignLevel() {
        bool reachable = false;
        level.assign(n, -1);
        reached[0].assign(n, 0);
        reached[1].assign(m, 0);
        queue<int> q;
        for (int i = 0; i < n; i++) {
            if (match[i] == -1) {
                level[i] = 0;
                reached[0][i] = 1;
                q.push(i);
            }
        }
        while (!q.empty()) {
            auto cur = q.front(); q.pop();
            for (auto adj : graph[cur]) {
                reached[1][adj] = 1;
                auto next = matched[adj];
                if (next == -1) {
                    reachable = true;
                }
                else if (level[next] == -1) {
                    level[next] = level[cur] + 1;
                    reached[0][next] = 1;
                    q.push(next);
                }
            }
        }
        return reachable;
    }
```

```
    int findpath(int nod) {
        for (int &i = edgeview[nod]; i < graph[nod].size(); i++) {
            int adj = graph[nod][i];
            int next = matched[adj];
            if (next >= 0 && level[next] != level[nod] + 1) continue;
            if (next == -1 || findpath(next)) {
                match[nod] = adj;
                matched[adj] = nod;
                return 1;
            }
        }
        return 0;
    }

    int solve() {
        int ans = 0;
        while (assignLevel()) {
            edgeview.assign(n, 0);
            for (int i = 0; i < n; i++)
                if (match[i] == -1)
                    ans += findpath(i);
        }
        return ans;
    }
};
```

## 5.11 Maximum Flow (Dinic)

```
// usage:
// MaxFlowDinic::init(n);
// MaxFlowDinic::add_edge(0, 1, 100, 100); // for bidirectional edge
// MaxFlowDinic::add_edge(1, 2, 100); // directional edge
// result = MaxFlowDinic::solve(0, 2); // source -> sink
// graph[i][edgeIndex].res -> residual
//
// in order to find out the minimum cut, use `l'.
// if l[i] == 0, i is unrechable.
//
// O(V*V*E)
// with unit capacities, O(min(V^(2/3), E^(1/2)) * E)
struct MaxFlowDinic {
    typedef int flow_t;
    struct Edge {
        int next;
        size_t inv; /* inverse edge index */
        flow_t res; /* residual */
    };
    int n;
    vector<vector<Edge>> graph;
    vector<int> q, l, start;

    void init(int _n) {
        n = _n;
        graph.resize(n);
        for (int i = 0; i < n; i++) graph[i].clear();
    }
    void add_edge(int s, int e, flow_t cap, flow_t caprev = 0) {
        Edge forward{ e, graph[e].size(), cap };
        Edge reverse{ s, graph[s].size(), caprev };
        graph[s].push_back(forward);
        graph[e].push_back(reverse);
    }
    bool assign_level(int source, int sink) {
        int t = 0;
        memset(&l[0], 0, sizeof(l[0]) * l.size());
        l[source] = 1;
        q[t++] = source;
        for (int h = 0; h < t && !l[sink]; h++) {
            int cur = q[h];
            for (const auto& e : graph[cur]) {
                if (l[e.next] || e.res == 0) continue;
                l[e.next] = l[cur] + 1;
                q[t++] = e.next;
            }
        }
        return l[sink] != 0;
    }
    flow_t block_flow(int cur, int sink, flow_t current) {
        if (cur == sink) return current;
        for (int& i = start[cur]; i < graph[cur].size(); i++) {
            auto& e = graph[cur][i];
            if (e.res == 0 || l[e.next] != l[cur] + 1) continue;
            if (flow_t res = block_flow(e.next, sink, min(e.res, current))) {
                e.res -= res;
                graph[e.next][e.inv].res += res;
                return res;
            }
        }
        return 0;
    }
    flow_t solve(int source, int sink) {
        q.resize(n);
        l.resize(n);
        start.resize(n);
        flow_t ans = 0;
        while (assign_level(source, sink)) {
            memset(&start[0], 0, sizeof(start[0]) * n);
            while (flow_t flow = block_flow(source, sink, numeric_limits<flow_t>::max
              ()))
                ans += flow;
        }
        return ans;
    }
};
```

## 5.12 Maximum Flow with Edge Demands

그래프 $G = (V, E)$ 가 있고 source $s$와 sink $t$가 있다. 각 간선마다 $d(e) \le f(e) \le c(e)$ 를 만족하도록 flow $f(e)$를 흘려야 한다. 이 때의 maximum flow를 구하는 문제다.

먼저 모든 demand를 합한 값 $D$를 아래와 같이 정의한다.

$$D = \sum_{(u \to v) \in E} d(u \to v)$$

이제 $G$ 에 몇개의 정점과 간선을 추가하여 새로운 그래프 $G' = (V', E')$ 을 만들 것이다. 먼저 새로운 source $s'$ 과 새로운 sink $t'$ 을 추가한다. 그리고 $s'$ 에서 $V$ 의 모든 점마다 간선을 이어주고, $V$ 의 모든 점에서 $t'$ 로 간선을 이어준다.

새로운 capacity function $c'$ 을 아래와 같이 정의한다.

1. $V$ 의 점 $v$에 대해 $c'(s' \to v) = \sum_{u \in V} d(u \to v)$ , $c'(v \to t') = \sum_{w \in V} d(v \to w)$
2. $E$ 의 간선 $u \to v$에 대해 $c'(u \to v) = c(u \to v) - d(u \to v)$
3. $c'(t \to s) = \infty$

이렇게 만든 새로운 그래프 $G'$ 에서 maximum flow를 구했을 때 그 값이 $D$ 라면 원래 문제의 해가 존재하고, 그 값이 $D$ 가 아니라면 원래 문제의 해는 존재하지 않는다.

위에서 maximum flow를 구하고 난 상태의 residual graph 에서 $s'$ 과 $t'$ 을 떼버리고 $s$ 에서 $t$ 사이의 augument path 를 계속 찾으면 원래 문제의 해를 구할 수 있다.

```
struct MaxFlowEdgeDemands
{
    MaxFlowDinic mf;
    using flow_t = MaxFlowDinic::flow_t;

    vector<flow_t> ind, outd;
    flow_t D; int n;

    void init(int _n) {
        n = _n; D = 0; mf.init(n + 2);
        ind.clear(); outd.clear();
        ind.resize(n, 0); outd.resize(n, 0);
    }

    void add_edge(int s, int e, flow_t cap, flow_t demands = 0) {
        mf.add_edge(s, e, cap - demands);
        D += demands; ind[e] += demands; outd[s] += demands;
    }

    // returns { false, 0 } if infeasible
    // { true, maxflow } if feasible
    pair<bool, flow_t> solve(int source, int sink) {
        mf.add_edge(sink, source, numeric_limits<flow_t>::max());

        for (int i = 0; i < n; i++) {
            if (ind[i]) mf.add_edge(n, i, ind[i]);
            if (outd[i]) mf.add_edge(i, n + 1, outd[i]);
        }

        if (mf.solve(n, n + 1) != D) return{ false, 0 };

        for (int i = 0; i < n; i++) {
            if (ind[i]) mf.graph[i].pop_back();
            if (outd[i]) mf.graph[i].pop_back();
        }

        return{ true, mf.solve(source, sink) };
    }
};
```

## 5.13 Min-cost Maximum Flow

```
// precondition: there is no negative cycle.
// usage:
// MinCostFlow mcf(n);
// for(each edges) mcf.addEdge(from, to, cost, capacity);
// mcf.solve(source, sink); // min cost max flow
// mcf.solve(source, sink, 0); // min cost flow
// mcf.solve(source, sink, goal_flow); // min cost flow with total_flow >= goal_flow
  if possible
struct MinCostFlow {
    typedef int cap_t;
    typedef int cost_t;

    bool iszerocap(cap_t cap) { return cap == 0; }

    struct edge {
        int target;
        cost_t cost;
        cap_t residual_capacity;
        cap_t orig_capacity;
        size_t revid;
    };

    int n;
    vector<vector<edge>> graph;

    MinCostFlow(int n) : graph(n), n(n) {}

    void addEdge(int s, int e, cost_t cost, cap_t cap) {
        if (s == e) return;
        edge forward{ e, cost, cap, cap, graph[e].size() };
        edge backward{ s, -cost, 0, 0, graph[s].size() };
        graph[s].emplace_back(forward);
        graph[e].emplace_back(backward);
    }

    pair<cost_t, cap_t> augmentShortest(int s, int e, cap_t flow_limit) {
        auto infinite_cost = numeric_limits<cost_t>::max();
        auto infinite_flow = numeric_limits<cap_t>::max();
        vector<pair<cost_t, cap_t>> dist(n, make_pair(infinite_cost, 0));
        vector<int> from(n, -1), v(n);

        dist[s] = pair<cost_t, cap_t>(0, infinite_flow);
        queue<int> q;
        v[s] = 1; q.push(s);
        while(!q.empty()) {
            int cur = q.front();
```

```
            v[cur] = 0; q.pop();
            for (const auto& e : graph[cur]) {
                if (iszerocap(e.residual_capacity)) continue;
                auto next = e.target;
                auto ncost = dist[cur].first + e.cost;
                auto nflow = min(dist[cur].second, e.residual_capacity);
                if (dist[next].first > ncost) {
                    dist[next] = make_pair(ncost, nflow);
                    from[next] = e.revid;
                    if (v[next]) continue;
                    v[next] = 1; q.push(next);
                }
            }
        }

        auto p = e;
        auto pathcost = dist[p].first;
        auto flow = dist[p].second;
        if (iszerocap(flow)|| (flow_limit <= 0 && pathcost >= 0)) return pair<cost_t,
          cap_t>(0, 0);
        if (flow_limit > 0) flow = min(flow, flow_limit);

        while (from[p] != -1) {
            auto nedge = from[p];
            auto np = graph[p][nedge].target;
            auto fedge = graph[p][nedge].revid;
            graph[p][nedge].residual_capacity += flow;
            graph[np][fedge].residual_capacity -= flow;
            p = np;
        }
        return make_pair(pathcost * flow, flow);
    }

    pair<cost_t,cap_t> solve(int s, int e, cap_t flow_minimum = numeric_limits<cap_t
      >::max()) {
        cost_t total_cost = 0;
        cap_t total_flow = 0;
        for(;;) {
            auto res = augmentShortest(s, e, flow_minimum - total_flow);
            if (res.second <= 0) break;
            total_cost += res.first;
            total_flow += res.second;
        }
        return make_pair(total_cost, total_flow);
    }
};
```

## 5.14 General Min-cut (Stoer-Wagner)

```
// implementation of Stoer-Wagner algorithm
// O(V^3)
//usage
// MinCut mc;
// mc.init(n);
// for (each edge) mc.addEdge(a,b,weight);
// mincut = mc.solve();
// mc.cut = {0,1}^n describing which side the vertex belongs to.
struct MinCutMatrix
{
    typedef int cap_t;
    int n;
    vector<vector<cap_t>> graph;

    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));
    }

    vector<int> cut;

    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;
    }
};
```

## 5.15 Hungarian Algorithm

```
int n, m;
int mat[MAX_N + 1][MAX_M + 1];

// hungarian method : bipartite min-weighted matching
// O(n^3) or O(m*n^2)
// http://e-maxx.ru/algo/assignment_hungary
// mat[1][1] ~ mat[n][m]
// matched[i] : matched column of row i
int hungarian(vector<int>& matched) {
    vector<int> u(n + 1), v(m + 1), p(m + 1), way(m + 1), minv(m + 1);
    vector<char> used(m + 1);
    for (int i = 1; i <= n; ++i) {
        p[0] = i;
        int j0 = 0;
        fill(minv.begin(), minv.end(), INF);
        fill(used.begin(), used.end(), false);
        do {
            used[j0] = true;
            int i0 = p[j0], delta = INF, j1;
            for (int j = 1; j <= m; ++j) {
                if (!used[j]) {
                    int cur = mat[i0][j] - u[i0] - v[j];
                    if (cur < minv[j]) minv[j] = cur, way[j] = j0;
                    if (minv[j] < delta) delta = minv[j], j1 = j;
                }
            }
            for (int j = 0; j <= m; ++j) {
                if (used[j])
                    u[p[j]] += delta, v[j] -= delta;
                else
                    minv[j] -= delta;
            }
            j0 = j1;
        } while (p[j0] != 0);
        do {
            int j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while (j0);
    }
    for (int j = 1; j <= m; ++j) matched[p[j]] = j;
```

```
    return -v[0];
}
```

# 6 Geometry

## 6.1 Basic Operations

```
const ld eps = 1e-12;
inline ll diff(ld lhs, ld rhs) {
  if (lhs - eps < rhs && rhs < lhs + eps) return 0;
  return (lhs < rhs) ? -1 : 1;
}
inline bool is_between(ld check, ld a, ld b) {
  return (a < b) ? (a - eps < check && check < b + eps)
                 : (b - eps < check && check < a + eps);
}
struct Point {
  ld x, y;
  bool operator==(const Point& rhs) const {
    return diff(x, rhs.x) == 0 && diff(y, rhs.y) == 0;
  }
  Point operator+(const Point& rhs) const { return Point{x + rhs.x, y + rhs.y}; }
  Point operator-(const Point& rhs) const { return Point{x - rhs.x, y - rhs.y}; }
  Point operator*(ld t) const { return Point{x * t, y * t}; }
};
struct Circle {
  Point center;
  ld r;
};
struct Line {
  Point pos, dir;
};
inline ld inner(const Point& a, const Point& b) { return a.x * b.x + a.y * b.y; }
inline ld outer(const Point& a, const Point& b) { return a.x * b.y - a.y * b.x; }
inline ll ccw_line(const Line& line, const Point& point) {
  return diff(outer(line.dir, point - line.pos), 0);
}
inline ll ccw(const Point& a, const Point& b, const Point& c) {
  return diff(outer(b - a, c - a), 0);
}
inline ld dist(const Point& a, const Point& b) { return sqrt(inner(a - b, a - b)); }
inline ld dist2(const Point& a, const Point& b) { return inner(a - b, a - b); }
inline ld dist(const Line& line, const Point& point, bool segment = false) {
  ld c1 = inner(point - line.pos, line.dir);
  if (segment && diff(c1, 0) <= 0) return dist(line.pos, point);
  ld c2 = inner(line.dir, line.dir);
  if (segment && diff(c2, c1) <= 0) return dist(line.pos + line.dir, point);
  return dist(line.pos + line.dir * (c1 / c2), point);
}
bool get_cross(const Line& a, const Line& b, Point& ret) {
  ld mdet = outer(b.dir, a.dir);
  if (diff(mdet, 0) == 0) return false;
  ld t2 = outer(a.dir, b.pos - a.pos) / mdet;
  ret = b.pos + b.dir * t2;
  return true;
```

```
}
bool get_segment_cross(const Line& a, const Line& b, Point& ret) {
  ld mdet = outer(b.dir, a.dir);
  if (diff(mdet, 0) == 0) return false;
  ld t1 = -outer(b.pos - a.pos, b.dir) / mdet;
  ld t2 = outer(a.dir, b.pos - a.pos) / mdet;
  if (!is_between(t1, 0, 1) || !is_between(t2, 0, 1)) return false;
  ret = b.pos + b.dir * t2;
  return true;
}
Point inner_center(const Point& a, const Point& b, const Point& c) {
  ld wa = dist(b, c), wb = dist(c, a), wc = dist(a, b);
  ld w = wa + wb + wc;
  return Point{(wa * a.x + wb * b.x + wc * c.x) / w,
               (wa * a.y + wb * b.y + wc * c.y) / w};
}
Point outer_center(const Point& a, const Point& b, const Point& c) {
  Point d1 = b - a, d2 = c - a;
  ld area = outer(d1, d2);
  ld dx = d1.x * d1.x * d2.y - d2.x * d2.x * d1.y + d1.y * d2.y * (d1.y - d2.y);
  ld dy = d1.y * d1.y * d2.x - d2.y * d2.y * d1.x + d1.x * d2.x * (d1.x - d2.y);
  return Point{a.x + dx / area / 2.0, a.y - dy / area / 2.0};
}
vector<Point> circle_line(const Circle& circle, const Line& line) {
  vector<Point> result;
  ld a = 2 * inner(line.dir, line.dir);
  ld b = 2 * (line.dir.x * (line.pos.x - circle.center.x) +
              line.dir.y * (line.pos.y - circle.center.y));
  ld c = inner(line.pos - circle.center, line.pos - circle.center) - circle.r * circle
    .r;
  ld det = b * b - 2 * a * c;
  ll pred = diff(det, 0);
  if (pred == 0)
    result.push_back(line.pos + line.dir * (-b / a));
  else if (pred > 0) {
    det = sqrt(det);
    result.push_back(line.pos + line.dir * ((-b + det) / a));
    result.push_back(line.pos + line.dir * ((-b - det) / a));
  }
  return result;
}
vector<Point> circle_circle(const Circle& a, const Circle& b) {
  vector<Point> result;
  ll pred = diff(dist(a.center, b.center), a.r + b.r);
  if (pred > 0) return result;
  if (pred == 0) {
    result.push_back((a.center * b.r + b.center * a.r) * (1 / (a.r + b.r)));
    return result;
  }
  ld aa = a.center.x * a.center.x + a.center.y * a.center.y - a.r * a.r;
  ld bb = b.center.x * b.center.x + b.center.y * b.center.y - b.r * b.r;
  ld tmp = (bb - aa) / 2.0;
  Point cdiff = b.center - a.center;
  if (diff(cdiff.x, 0) == 0) {
    if (diff(cdiff.y, 0) == 0) return result;
    return circle_line(a, Line{Point{0, tmp / cdiff.y}, Point{1, 0}});
  }
  return circle_line(a, Line{Point{tmp / cdiff.x, 0}, Point{-cdiff.y, cdiff.x}});
}
Circle circle_from_3pts(const Point& a, const Point& b, const Point& c) {
  Point ba = b - a, cb = c - b;
  Line p{(a + b) * 0.5, Point{ba.y, -ba.x}};
  Line q{(b + c) * 0.5, Point{cb.y, -cb.x}};
  Circle circle;
  if (!get_cross(p, q, circle.center))
    circle.r = -1;
  else
    circle.r = dist(circle.center, a);
  return circle;
}
Circle circle_from_2pts_rad(const Point& a, const Point& b, ld r) {
  ld det = r * r / dist2(a, b) - 0.25;
  Circle circle;
  if (det < 0)
    circle.r = -1;
  else {
    ld h = sqrt(det);
    // center is to the left of a->b
    circle.center = (a + b) * 0.5 + Point{a.y - b.y, b.x - a.x} * h;
    circle.r = r;
  }
  return circle;
}
```

## 6.2 Convex Hull

```
// find convex hull
// O(n*logn)
vector<Point> convex_hull(vector<Point>& dat) {
    if (dat.size() <= 3) return dat;
    vector<Point> upper, lower;
    sort(dat.begin(), dat.end(), [](const Point& a, const Point& b) {
        return (a.x == b.x) ? a.y < b.y : a.x < b.x;
    });
    for (const auto& p : dat) {
        while (upper.size() >= 2 && ccw(*++upper.rbegin(), *upper.rbegin(), p) >= 0)
          upper.pop_back();
        while (lower.size() >= 2 && ccw(*++lower.rbegin(), *lower.rbegin(), p) <= 0)
          lower.pop_back();
        upper.emplace_back(p);
        lower.emplace_back(p);
    }
    upper.insert(upper.end(), ++lower.rbegin(), --lower.rend());
    return upper;
}
```

## 6.3 Rotating Calipers

```
// get all antipodal pairs
// O(n)
void antipodal_pairs(vector<Point>& pt) {
```

```
  // calculate convex hull
  sort(pt.begin(), pt.end(), [](const Point& a, const Point& b) {
    return (a.x == b.x) ? a.y < b.y : a.x < b.x;
  });
  vector<Point> up, lo;
  for (const auto& p : pt) {
    while (up.size() >= 2 && ccw(*++up.rbegin(), *up.rbegin(), p) >= 0) up.pop_back();
    while (lo.size() >= 2 && ccw(*++lo.rbegin(), *lo.rbegin(), p) <= 0) lo.pop_back();
    up.emplace_back(p);
    lo.emplace_back(p);
  }
  for (int i = 0, j = (int)lo.size() - 1; i + 1 < up.size() || j > 0;) {
    get_pair(up[i], lo[j]);  // DO WHAT YOU WANT
    if (i + 1 == up.size()) {
      --j;
    } else if (j == 0) {
      ++i;
    } else if ((long long)(up[i + 1].y - up[i].y) * (lo[j].x - lo[j - 1].x) >
               (long long)(up[i + 1].x - up[i].x) * (lo[j].y - lo[j - 1].y)) {
      ++i;
    } else {
      --j;
    }
  }
}
```

## 6.4 Half Plane Intersection

```
typedef pair<long double, long double> pi;
bool z(long double x) { return fabs(x) < eps; }
struct line {
  long double a, b, c;
  bool operator<(const line &l) const {
    bool flag1 = pi(a, b) > pi(0, 0);
    bool flag2 = pi(l.a, l.b) > pi(0, 0);
    if (flag1 != flag2) return flag1 > flag2;
    long double t = ccw(pi(0, 0), pi(a, b), pi(l.a, l.b));
    return z(t) ? c * hypot(l.a, l.b) < l.c * hypot(a, b) : t > 0;
  }
  pi slope() { return pi(a, b); }
};
pi cross(line a, line b) {
  long double det = a.a * b.b - b.a * a.b;
  return pi((a.c * b.b - a.b * b.c) / det, (a.a * b.c - a.c * b.a) / det);
}
bool bad(line a, line b, line c) {
  if (ccw(pi(0, 0), a.slope(), b.slope()) <= 0) return false;
  pi crs = cross(a, b);
  return crs.first * c.a + crs.second * c.b >= c.c;
}
bool solve(vector<line> v, vector<pi> &solution) {  // ax + by <= c;
  sort(v.begin(), v.end());
  deque<line> dq;
  for (auto &i : v) {
    if (!dq.empty() && z(ccw(pi(0, 0), dq.back().slope(), i.slope()))) continue;
    while (dq.size() >= 2 && bad(dq[dq.size() - 2], dq.back(), i)) dq.pop_back();
    while (dq.size() >= 2 && bad(i, dq[0], dq[1])) dq.pop_front();
    dq.push_back(i);
  }
  while (dq.size() > 2 && bad(dq[dq.size() - 2], dq.back(), dq[0])) dq.pop_back();
  while (dq.size() > 2 && bad(dq.back(), dq[0], dq[1])) dq.pop_front();
  vector<pi> tmp;
  for (int i = 0; i < dq.size(); i++) {
    line cur = dq[i], nxt = dq[(i + 1) % dq.size()];
    if (ccw(pi(0, 0), cur.slope(), nxt.slope()) <= eps) return false;
    tmp.push_back(cross(cur, nxt));
  }
  solution = tmp;
  return true;
}
```

## 6.5 Point in Polygon Test

```
typedef double coord_t;
inline coord_t is_left(Point p0, Point p1, Point p2) {
  return (p1.x - p0.x) * (p2.y - p0.y) - (p2.x - p0.x) * (p1.y - p0.y);
}
// point in polygon test
bool is_in_polygon(Point p, vector<Point>& poly) {
  int wn = 0;
  for (int i = 0; i < poly.size(); ++i) {
    int ni = (i + 1 == poly.size()) ? 0 : i + 1;
    if (poly[i].y <= p.y) {
      if (poly[ni].y > p.y) {
        if (is_left(poly[i], poly[ni], p) > 0) {
          ++wn;
        }
      }
    } else {
      if (poly[ni].y <= p.y) {
        if (is_left(poly[i], poly[ni], p) < 0) {
          --wn;
        }
      }
    }
  }
  return wn != 0;
}
```

## 6.6 Polygon Cut

```
// left side of a->b
vector<Point> cut_polygon(const vector<Point>& polygon, Line line) {
    if (!polygon.size()) return polygon;
    typedef vector<Point>::const_iterator piter;
    piter la, lan, fi, fip, i, j;
    la = lan = fi = fip = polygon.end();
    i = polygon.end() - 1;
    bool lastin = diff(ccw_line(line, polygon[polygon.size() - 1]), 0) > 0;
    for (j = polygon.begin() ; j != polygon.end() ; j++) {
        bool thisin = diff(ccw_line(line, *j), 0) > 0;
```

```
        if (lastin && !thisin) {
            la = i;
            lan = j;
        }
        if (!lastin && thisin) {
            fi = j;
            fip = i;
        }
        i = j;
        lastin = thisin;
    }
    if (fi == polygon.end()) {
        if (!lastin) return vector<Point>();
        return polygon;
    }
    vector<Point> result;
    for (i = fi ; i != lan ; i++) {
        if (i == polygon.end()) {
            i = polygon.begin();
            if (i == lan) break;
        }
        result.push_back(*i);
    }
    Point lc, fc;
    get_cross(Line{ *la, *lan - *la }, line, lc);
    get_cross(Line{ *fip, *fi - *fip }, line, fc);
    result.push_back(lc);
    if (diff(dist2(lc, fc), 0) != 0) result.push_back(fc);
    return result;
}
```

## 6.7 Pick's theorem

격자점으로 구성된 simple polygon에 대해 $i$는 polygon 내부의 격자수, $b$는 polygon 선분 위 격자수, $A$는 polygon 넓이라고 할 때 $A = i + \frac{b}{2} - 1$.

# 7 String

## 7.1 KMP

```
typedef vector<int> seq_t;

void calculate_pi(vector<int>& pi, const seq_t& str) {
    pi[0] = -1;
    for (int i = 1, j = -1; i < str.size(); i++) {
        while (j >= 0 && str[i] != str[j + 1]) j = pi[j];
        if (str[i] == str[j + 1])
            pi[i] = ++j;
        else
            pi[i] = -1;
    }
}

// returns all positions matched
// O(|text|+|pattern|)
vector<int> kmp(const seq_t& text, const seq_t& pattern) {
    vector<int> pi(pattern.size()), ans;
    if (pattern.size() == 0) return ans;
    calculate_pi(pi, pattern);
    for (int i = 0, j = -1; i < text.size(); i++) {
        while (j >= 0 && text[i] != pattern[j + 1]) j = pi[j];
        if (text[i] == pattern[j + 1]) {
            j++;
            if (j + 1 == pattern.size()) {
                ans.push_back(i - j);
                j = pi[j];
            }
        }
    }
    return ans;
}
```

## 7.2 Z Algorithm

```
// Z[i] : maximum common prefix length of &s[0] and &s[i]
// O(|s|)
using seq_t = string;
vector<int> z_func(const seq_t &s) {
    vector<int> z(s.size());
    z[0] = s.size();
    int l = 0, r = 0;

    for (int i = 1; i < s.size(); i++) {
        if (i > r) {
            int j;
            for (j = 0; i + j < s.size() && s[i + j] == s[j]; j++) ;
            z[i] = j; l = i; r = i + j - 1;
        } else if (z[i - l] < r - i + 1) {
            z[i] = z[i - l];
        } else {
            int j;
            for (j = 1; r + j < s.size() && s[r + j] == s[r - i + j]; j++) ;
            z[i] = r - i + j; l = i; r += j - 1;
        }
    }

    return z;
}
```

## 7.3 Aho-Corasick

```
struct aho_corasick_with_trie {
  const ll MAXN = 100005, MAXC = 26;
  ll trie[MAXN][MAXC], fail[MAXN], term[MAXN], piv = 0;
  void init(vector<string> &v) {
    memset(trie, 0, sizeof(trie));
    memset(fail, 0, sizeof(fail));
    memset(term, 0, sizeof(term));
```

```
    piv = 0;
    for (auto &i : v) {
      ll p = 0;
      for (auto &j : i) {
        if (!trie[p][j]) trie[p][j] = ++piv;
        p = trie[p][j];
      }
      term[p] = 1;
    }
    queue<ll> que;
    for (ll i = 0; i < MAXC; i++) {
      if (trie[0][i]) que.push(trie[0][i]);
    }
    while (!que.empty()) {
      ll x = que.front();
      que.pop();
      for (ll i = 0; i < MAXC; i++) {
        if (trie[x][i]) {
          ll p = fail[x];
          while (p && !trie[p][i]) p = fail[p];
          p = trie[p][i];
          fail[trie[x][i]] = p;
          if (term[p]) term[trie[x][i]] = 1;
          que.push(trie[x][i]);
        }
      }
    }
  }
  bool query(string &s) {
    ll p = 0;
    for (auto &i : s) {
      while (p && !trie[p][i]) p = fail[p];
      p = trie[p][i];
      if (term[p]) return 1;
    }
    return 0;
  }
};
```

## 7.4 Suffix Array with LCP

```
typedef char T;

// calculates suffix array.
// O(n*logn)
vector<int> suffix_array(const vector<T>& in) {
    int n = (int)in.size(), c = 0;
    vector<int> temp(n), pos2bckt(n), bckt(n), bpos(n), out(n);
    for (int i = 0; i < n; i++) out[i] = i;
    sort(out.begin(), out.end(), [&](int a, int b) { return in[a] < in[b]; });
    for (int i = 0; i < n; i++) {
        bckt[i] = c;
        if (i + 1 == n || in[out[i]] != in[out[i + 1]]) c++;
    }
    for (int h = 1; h < n && c < n; h <<= 1) {
        for (int i = 0; i < n; i++) pos2bckt[out[i]] = bckt[i];
        for (int i = n - 1; i >= 0; i--) bpos[bckt[i]] = i;
        for (int i = 0; i < n; i++)
            if (out[i] >= n - h) temp[bpos[bckt[i]]++] = out[i];
        for (int i = 0; i < n; i++)
            if (out[i] >= h) temp[bpos[pos2bckt[out[i] - h]]++] = out[i] - h;
        c = 0;
        for (int i = 0; i + 1 < n; i++) {
            int a = (bckt[i] != bckt[i + 1]) || (temp[i] >= n - h)
                    || (pos2bckt[temp[i + 1] + h] != pos2bckt[temp[i] + h]);
            bckt[i] = c;
            c += a;
        }
        bckt[n - 1] = c++;
        temp.swap(out);
    }
    return out;
}

// calculates lcp array. it needs suffix array & original sequence.
// O(n)
vector<int> lcp(const vector<T>& in, const vector<int>& sa) {
    int n = (int)in.size();
    if (n == 0) return vector<int>();
    vector<int> rank(n), height(n - 1);
    for (int i = 0; i < n; i++) rank[sa[i]] = i;
    for (int i = 0, h = 0; i < n; i++) {
        if (rank[i] == 0) continue;
        int j = sa[rank[i] - 1];
        while (i + h < n && j + h < n && in[i + h] == in[j + h]) h++;
        height[rank[i] - 1] = h;
        if (h > 0) h--;
    }
    return height;
}
```

## 7.5 Manacher's Algorithm

```
// find longest palindromic span for each element in str
// O(|str|)
void manacher(const string& str, int plen[]) {
    int r = -1, p = -1;
    for (int i = 0; i < str.length(); ++i) {
        if (i <= r)
            plen[i] = min((2 * p - i >= 0) ? plen[2 * p - i] : 0, r - i);
        else
            plen[i] = 0;
        while (i - plen[i] - 1 >= 0 && i + plen[i] + 1 < str.length()
                && str[i - plen[i] - 1] == str[i + plen[i] + 1]) {
            plen[i] += 1;
        }
        if (i + plen[i] > r) {
            r = i + plen[i];
            p = i;
        }
    }
}
```