# Contents


**1 Misc** **1**
- 1.1 Contest 1
  - 1.1.1 Makefile 1
- 1.2 How Did We Get Here? 2
  - 1.2.1 The Basics 2
  - 1.2.2 Linux 2
  - 1.2.3 Macros 2
- 1.3 Tools 2
  - 1.3.1 Floating Point Binary Search 2
  - 1.3.2 SplitMix64 2
  - 1.3.3 <random> 2
  - 1.3.4 x86 Stack Hack 2
- 1.4 Algorithms 2
  - 1.4.1 Bit Hacks 2
  - 1.4.2 Aliens Trick 2
  - 1.4.3 Hilbert Curve 2
  - 1.4.4 Infinite Grid Knight Distance 2
  - 1.4.5 Poker Hand 2
  - 1.4.6 Longest Increasing Subsequence 3
  - 1.4.7 Mo's Algorithm on Tree 3

**2 Data Structures** **3**
- 2.1 GNU PBDS 3
- 2.2 Segment Tree (ZKW) 3
- 2.3 Line Container 3
- 2.4 Li-Chao Tree 3
- 2.5 Heavy-Light Decomposition 4
- 2.6 Wavelet Matrix 4
- 2.7 Link-Cut Tree 5

**3 Graph** **5**
- 3.1 Floyd-Warshall 5
- 3.2 Dijkstra 5
- 3.3 Modeling 6
- 3.4 Matching/Flows 6
  - 3.4.1 Dinic's Algorithm 6
  - 3.4.2 Minimum Cost Flow 6
  - 3.4.3 Gomory-Hu Tree 7
  - 3.4.4 Global Minimum Cut 7
  - 3.4.5 Bipartite Minimum Cover 7
  - 3.4.6 Edmonds' Algorithm 7
  - 3.4.7 Minimum Weight Matching 8
  - 3.4.8 Stable Marriage 8
  - 3.4.9 Kuhn-Munkres algorithm 9
- 3.5 Shortest Path Faster Algorithm 9
- 3.6 Strongly Connected Components 9
  - 3.6.1 2-Satisfiability 10
- 3.7 Biconnected Components 10
  - 3.7.1 Articulation Points 10
  - 3.7.2 Bridges 10
  - 3.7.3 Block Cut Tree 10
- 3.8 Triconnected Components 10
- 3.9 Centroid Decomposition 11
- 3.10 Minimum Mean Cycle 11
- 3.11 Directed MST 11
- 3.12 Maximum Clique 12
- 3.13 Dominator Tree 12
- 3.14 Manhattan Distance MST 12

**4 Math** **13**
- 4.1 Number Theory 13
  - 4.1.1 Mod Struct 13
  - 4.1.2 Miller-Rabin 13
  - 4.1.3 Linear Sieve 13
  - 4.1.4 Get Factors 13
  - 4.1.5 Binary GCD 13
  - 4.1.6 Extended GCD 13
  - 4.1.7 Chinese Remainder Theorem 13
  - 4.1.8 Baby-Step Giant-Step 14
  - 4.1.9 Pollard's Rho 14
  - 4.1.10 Tonelli-Shanks Algorithm 14
  - 4.1.11 Chinese Sieve 14
  - 4.1.12 Rational Number Binary Search 14
  - 4.1.13 Farey Sequence 14
- 4.2 Combinatorics 14
  - 4.2.1 Matroid Intersection 14
  - 4.2.2 De Brujin Sequence 15
  - 4.2.3 Multinomial 15
- 4.3 Algebra 15
  - 4.3.1 Formal Power Series 15
- 4.4 Theorems 16
  - 4.4.1 Kirchhoff's Theorem 16
  - 4.4.2 Tutte's Matrix 16
  - 4.4.3 Cayley's Formula 16
  - 4.4.4 Erdős–Gallai Theorem 16
  - 4.4.5 Burnside's Lemma 16

**5 Randomisation** **16**
- 5.1 Simulated Annealing 16

**6 Numeric** **16**
- 6.1 Barrett Reduction 16
- 6.2 Long Long Multiplication 16
- 6.3 Fast Fourier Transform 17
- 6.4 Fast Walsh-Hadamard Transform 17
- 6.5 Subset Convolution 17
- 6.6 Linear Recurrences 17
  - 6.6.1 Berlekamp-Massey Algorithm 17
  - 6.6.2 Linear Recurrence Calculation 17
- 6.7 Matrices 17
  - 6.7.1 Determinant 17
  - 6.7.2 Inverse 18
  - 6.7.3 Characteristic Polynomial 18
  - 6.7.4 Solve Linear Equation 18
- 6.8 Polynomial Interpolation 19
- 6.9 Simplex Algorithm 19

**7 Geometry** **20**
- 7.1 Point 20
  - 7.1.1 Quarternion 20
  - 7.1.2 Spherical Coordinates 20
- 7.2 Segments 20
- 7.3 Convex Hull 20
  - 7.3.1 3D Hull 20
- 7.4 Angular Sort 21
- 7.5 Convex Polygon Minkowski Sum 21
- 7.6 Point In Polygon 21
  - 7.6.1 Convex Version 21
  - 7.6.2 Offline Multiple Points Version 21
- 7.7 Closest Pair 22
- 7.8 Minimum Enclosing Circle 22
- 7.9 Delaunay Triangulation 22
  - 7.9.1 Slower Version 23
- 7.10 Half Plane Intersection 23

**8 Strings** **23**
- 8.1 Knuth-Morris-Pratt Algorithm 23
- 8.2 Aho-Corasick Automaton 23
- 8.3 Suffix Array 24
- 8.4 Suffix Tree 24
- 8.5 Z Value 24
- 8.6 Manacher's Algorithm 24
- 8.7 Minimum Rotation 25
- 8.8 Palindromic Tree 25

**9 Debug List** **25**


# 1. Misc

## 1.1. Contest

### 1.1.1. Makefile

```
.PRECIOUS: ./p%

%: p%
  ulimit -s unlimited && ./$<
p%: p%.cpp
  g++ -o $@ $< -std=c++17 -Wall -Wextra -Wshadow \
    -fsanitize=address,undefined
```

## 1.2. How Did We Get Here?

### 1.2.1. The Basics

```cpp
#include <bits/stdc++.h>
#define ll    long long
#define sz(x) (int)(x).size()
using namespace std;
// mt19937
// rng(chrono::steady_clock::now().time_since_epoch().count());
// uniform_int_distribution<int>(1000,10000)(rng)

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  return 0;
}
```

### 1.2.2. Linux

- g++ test.cpp
- ./test
- ./test < input.txt > output.txt (read/write to file)

### 1.2.3. Macros

Use vectorizations and math optimizations at your own peril.
For gcc≥9, there are `[[likely]]` and `[[unlikely]]` attributes.
Call gcc with `-fopt-info-optimized-missed-optall` for optimization info.

```cpp
#define _GLIBCXX_DEBUG           1 // for debug mode
#define _GLIBCXX_SANITIZE_VECTOR 1 // for asan on vectors
#pragma GCC optimize("O3", "unroll-loops")
#pragma GCC optimize("fast-math")
#pragma GCC target("avx,avx2,abm,bmi,bmi2") // tip: `lscpu`
// before a loop
#pragma GCC unroll 16 // 0 or 1 -> no unrolling
#pragma GCC ivdep
```

## 1.3. Tools

### 1.3.1. Floating Point Binary Search

```cpp
union di {
  double d;
  ull i;
};
bool check(double);
// binary search in [L, R) with relative error 2^-eps
double binary_search(double L, double R, int eps) {
  di l = {L}, r = {R}, m;
  while (r.i - l.i > 1LL << (52 - eps)) {
    m.i = (l.i + r.i) >> 1;
    if (check(m.d))
      r = m;
    else
      l = m;
  }
  return l.d;
}
```

### 1.3.2. SplitMix64

```cpp
using ull = unsigned long long;
inline ull splitmix64(ull x) {
  // change to `static ull x = SEED;` for DRBG
  ull z = (x += 0x9E3779B97F4A7C15);
  z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9;
  z = (z ^ (z >> 27)) * 0x94D049BB133111EB;
  return z ^ (z >> 31);
}
```

### 1.3.3. <random>

```cpp
#ifdef __unix__
random_device rd;
mt19937_64 RNG(rd());
#else
const auto SEED = chrono::high_resolution_clock::now()
                    .time_since_epoch()
                    .count();
mt19937_64 RNG(SEED);
#endif
// random uint_fast64_t: RNG();
// uniform random of type T (int, double, ...) in [l, r]:
// uniform_int_distribution<T> dist(l, r); dist(RNG);
```

### 1.3.4. x86 Stack Hack

```cpp
constexpr size_t size = 200 << 20; // 200MiB
int main() {
  register long rsp asm("rsp");
  char *buf = new char[size];
  asm("movq %0, %%rsp\n" ::"r"(buf + size));
  // do stuff
  asm("movq %0, %%rsp\n" ::"r"(rsp));
  delete[] buf;
}
```

## 1.4. Algorithms

### 1.4.1. Bit Hacks

```cpp
// next permutation of x as a bit sequence
ull next_bits_permutation(ull x) {
  ull c = __builtin_ctzll(x), r = x + (1ULL << c);
  return (r ^ x) >> (c + 2) | r;
}
// iterate over all (proper) subsets of bitset s
void subsets(ull s) {
  for (ull x = s; x;) { --x &= s; /* do stuff */ }
}
```

### 1.4.2. Aliens Trick

```cpp
// min dp[i] value and its i (smallest one)
pll get_dp(int cost);
ll aliens(int k, int l, int r) {
  while (l != r) {
    int m = (l + r) / 2;
    auto [f, s] = get_dp(m);
    if (s == k) return f - m * k;
    if (s < k)
      r = m;
    else
      l = m + 1;
  }
  return get_dp(l).first - l * k;
}
```

### 1.4.3. Hilbert Curve

```cpp
ll hilbert(ll n, int x, int y) {
  ll res = 0;
  for (ll s = n; s /= 2;) {
    int rx = !!(x & s), ry = !!(y & s);
    res += s * s * ((3 * rx) ^ ry);
    if (ry == 0) {
      if (rx == 1) x = s - 1 - x, y = s - 1 - y;
      swap(x, y);
    }
  }
  return res;
}
```

### 1.4.4. Infinite Grid Knight Distance

```cpp
ll get_dist(ll dx, ll dy) {
  if (++(dx = abs(dx)) > ++(dy = abs(dy))) swap(dx, dy);
  if (dx == 1 && dy == 2) return 3;
  if (dx == 3 && dy == 3) return 4;
  ll lb = max(dy / 2, (dx + dy) / 3);
  return ((dx ^ dy ^ lb) & 1) ? ++lb : lb;
}
```

### 1.4.5. Poker Hand

```cpp
using namespace std;

struct hand {
  static constexpr auto rk = [] {
    array<int, 256> x{};
    auto s = "23456789TJQKACDHS";
    for (int i = 0; i < 17; i++) x[s[i]] = i % 13;
    return x;
  }();
  vector<pair<int, int>> v;
  vector<int> cnt, vf, vs;
  int type;
  hand() : cnt(4), type(0) {}
  void add_card(char suit, char rank) {
    ++cnt[rk[suit]];
    for (auto &[f, s] : v)
      if (s == rk[rank]) return ++f, void();
    v.emplace_back(1, rk[rank]);
  }
  void process() {
```

```cpp
    sort(v.rbegin(), v.rend());
    for (auto [f, s] : v) vf.push_back(f), vs.push_back(s);
    bool str = 0, flu = find(all(cnt), 5) != cnt.end();
    if ((str = v.size() == 5))
      for (int i = 1; i < 5; i++)
        if (vs[i] != vs[i - 1] + 1) str = 0;
    if (vs == vector<int>{12, 3, 2, 1, 0})
      str = 1, vs = {3, 2, 1, 0, -1};
    if (str && flu)
      type = 9;
    else if (vf[0] == 4)
      type = 8;
    else if (vf[0] == 3 && vf[1] == 2)
      type = 7;
    else if (str || flu)
      type = 5 + flu;
    else if (vf[0] == 3)
      type = 4;
    else if (vf[0] == 2)
      type = 2 + (vf[1] == 2);
    else
      type = 1;
  }
  bool operator<(const hand &b) const {
    return make_tuple(type, vf, vs) <
           make_tuple(b.type, b.vf, b.vs);
  }
};
```

### 1.4.6. Longest Increasing Subsequence

```cpp
template <class I> vi lis(const vector<I> &S) {
  if (S.empty()) return {};
  vi prev(sz(S));
  typedef pair<I, int> p;
  vector<p> res;
  rep(i, 0, sz(S)) {
    // change 0 -> i for longest non-decreasing subsequence
    auto it = lower_bound(all(res), p{S[i], 0});
    if (it == res.end())
      res.emplace_back(), it = res.end() - 1;
    *it = {S[i], i};
    prev[i] = it == res.begin() ? 0 : (it - 1)->second;
  }
  int L = sz(res), cur = res.back().second;
  vi ans(L);
  while (L--) ans[L] = cur, cur = prev[cur];
  return ans;
}
```

### 1.4.7. Mo's Algorithm on Tree

```cpp
void MoAlgoOnTree() {
  Dfs(0, -1);
  vector<int> euler(tk);
  for (int i = 0; i < n; ++i) {
    euler[tin[i]] = i;
    euler[tout[i]] = i;
  }
  vector<int> l(q), r(q), qr(q), sp(q, -1);
  for (int i = 0; i < q; ++i) {
    if (tin[u[i]] > tin[v[i]]) swap(u[i], v[i]);
    int z = GetLCA(u[i], v[i]);
    sp[i] = z[i];
    if (z == u)
      l[i] = tin[u[i]], r[i] = tin[v[i]];
    else
      l[i] = tout[u[i]], r[i] = tin[v[i]];
    qr[i] = i;
  }
  sort(qr.begin(), qr.end(), [&](int i, int j) {
    if (l[i] / kB == l[j] / kB) return r[i] < r[j];
    return l[i] / kB < l[j] / kB;
  });
  vector<bool> used(n);
  // Add(v): add/remove v to/from the path based on used[v]
  for (int i = 0, tl = 0, tr = -1; i < q; ++i) {
    while (tl < l[qr[i]]) Add(euler[tl++]);
    while (tl > l[qr[i]]) Add(euler[--tl]);
    while (tr > r[qr[i]]) Add(euler[tr--]);
    while (tr < r[qr[i]]) Add(euler[++tr]);
    // add/remove LCA(u, v) if necessary
  }
}
```

## 2. Data Structures

### 2.1. GNU PBDS

```cpp
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/priority_queue.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

// most std::map + order_of_key, find_by_order, split, join
template <typename T, typename U = null_type>
using ordered_map = tree<T, U, std::less<>, rb_tree_tag,
                         tree_order_statistics_node_update>;
// useful tags: rb_tree_tag, splay_tree_tag

template <typename T> struct myhash {
  size_t operator()(T x) const; // splitmix, bswap(x*R), ...
};
// most of std::unordered_map, but faster (needs good hash)
template <typename T, typename U = null_type>
using hash_table = gp_hash_table<T, U, myhash<T>>;

// most std::priority_queue + modify, erase, split, join
using heap = priority_queue<int, std::less<>>;
// useful tags: pairing_heap_tag, binary_heap_tag,
//              (rc_)?binomial_heap_tag, thin_heap_tag
```

### 2.2. Segment Tree (ZKW)

```cpp
struct segtree {
  using T = int;
  T f(T a, T b) { return a + b; } // any monoid operation
  static constexpr T ID = 0;      // identity element
  int n;
  vector<T> v;
  segtree(int n_) : n(n_), v(2 * n, ID) {}
  segtree(vector<T> &a) : n(a.size()), v(2 * n, ID) {
    copy_n(a.begin(), n, v.begin() + n);
    for (int i = n - 1; i > 0; i--)
      v[i] = f(v[i * 2], v[i * 2 + 1]);
  }
  void update(int i, T x) {
    for (v[i += n] = x; i /= 2;)
      v[i] = f(v[i * 2], v[i * 2 + 1]);
  }
  T query(int l, int r) {
    T tl = ID, tr = ID;
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
      if (l & 1) tl = f(tl, v[l++]);
      if (r & 1) tr = f(v[--r], tr);
    }
    return f(tl, tr);
  }
};
```

### 2.3. Line Container

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

### 2.4. Li-Chao Tree

```cpp
constexpr ll MAXN = 2e5, INF = 2e18;
struct Line {
```

```
  ll m, b;
  Line() : m(0), b(-INF) {}
  Line(ll _m, ll _b) : m(_m), b(_b) {}
  ll operator()(ll x) const { return m * x + b; }
};
struct Li_Chao {
  Line a[MAXN * 4];
  void insert(Line seg, int l, int r, int v = 1) {
    if (l == r) {
      if (seg(l) > a[v](l)) a[v] = seg;
      return;
    }
    int mid = (l + r) >> 1;
    if (a[v].m > seg.m) swap(a[v], seg);
    if (a[v](mid) < seg(mid)) {
      swap(a[v], seg);
      insert(seg, l, mid, v << 1);
    } else
      insert(seg, mid + 1, r, v << 1 | 1);
  }
  ll query(int x, int l, int r, int v = 1) {
    if (l == r) return a[v](x);
    int mid = (l + r) >> 1;
    if (x <= mid)
      return max(a[v](x), query(x, l, mid, v << 1));
    else
      return max(a[v](x), query(x, mid + 1, r, v << 1 | 1));
  }
};
```

## 2.5. Heavy-Light Decomposition

```
struct heavy_light_decomposition {
  int n;
  vector<vector<int>> edges;
  vector<int> par, heavy, height, pos, head;
  heavy_light_decomposition(int n)
      : n(n), edges(n + 1), par(n + 1), heavy(n + 1),
        height(n + 1), pos(n + 1), head(n + 1) {}
  void add_edge(int x, int y) {
    edges[x].push_back(y);
    edges[y].push_back(x);
  }
  void heavy_dfs(int c, vector<int> &sub) {
    heavy[c] = -1;
    sub[c] = 1;
    int max_size = 0;
    for (int i : edges[c]) {
      if (i != par[c]) {
        par[i] = c;
        height[i] = height[c] + 1;
        heavy_dfs(i, sub);
        sub[c] += sub[i];
        if (sub[i] > max_size) {
          heavy[c] = i;
          max_size = sub[i];
        }
      }
    }
  }
  void decompose(int c, int h, int &timer) {
    pos[c] = timer++;
    head[c] = h;
    if (heavy[c] != -1) { decompose(heavy[c], h, timer); }
    for (int i : edges[c]) {
      if (i != par[c] && i != heavy[c]) {
        decompose(i, i, timer);
      }
    }
  }
  void build() {
    int timer = 0;
    vector<int> sub(n + 1);
    for (int i = 1; i <= n; ++i) {
      if (sub[i] == 0) {
        par[i] = 0;
        height[i] = 0;
        heavy_dfs(i, sub);
        decompose(i, i, timer);
      }
    }
  }
  int get_position(int x) { return pos[x]; }
  vector<array<int, 2>> path_queries(int x, int y) {
    vector<array<int, 2>> queries;
    while (head[x] != head[y]) {
      if (height[head[x]] > height[head[y]]) { swap(x, y); }
      queries.push_back({pos[head[y]], pos[y]});
      y = par[head[y]];
    }
    if (height[x] > height[y]) { swap(x, y); }
    queries.push_back({pos[x], pos[y]});
    return queries;
  }
};
```

## 2.6. Wavelet Matrix

```
#pragma GCC target("popcnt,bmi2")
#include <immintrin.h>

// T is unsigned. You might want to compress values first
template <typename T> struct wavelet_matrix {
  static_assert(is_unsigned_v<T>, "only unsigned T");
  struct bit_vector {
    static constexpr uint W = 64;
    uint n, cnt0;
    vector<ull> bits;
    vector<uint> sum;
    bit_vector(uint n_)
        : n(n_), bits(n / W + 1), sum(n / W + 1) {}
    void build() {
      for (uint j = 0; j != n / W; ++j)
        sum[j + 1] = sum[j] + _mm_popcnt_u64(bits[j]);
      cnt0 = rank0(n);
    }
    void set_bit(uint i) { bits[i / W] |= 1ULL << i % W; }
    bool operator[](uint i) const {
      return !!(bits[i / W] & 1ULL << i % W);
    }
    uint rank1(uint i) const {
      return sum[i / W] +
             _mm_popcnt_u64(_bzhi_u64(bits[i / W], i % W));
    }
    uint rank0(uint i) const { return i - rank1(i); }
  };
  uint n, lg;
  vector<bit_vector> b;
  wavelet_matrix(const vector<T> &a) : n(a.size()) {
    lg =
    __lg(max(*max_element(a.begin(), a.end()), T(1))) + 1;
    b.assign(lg, n);
    vector<T> cur = a, nxt(n);
    for (int h = lg; h--;) {
      for (uint i = 0; i < n; ++i)
        if (cur[i] & (T(1) << h)) b[h].set_bit(i);
      b[h].build();
      int il = 0, ir = b[h].cnt0;
      for (uint i = 0; i < n; ++i)
        nxt[(b[h][i] ? ir : il)++] = cur[i];
      swap(cur, nxt);
    }
  }
  T operator[](uint i) const {
    T res = 0;
    for (int h = lg; h--;)
      if (b[h][i])
        i += b[h].cnt0 - b[h].rank0(i), res |= T(1) << h;
      else
        i = b[h].rank0(i);
    return res;
  }
  // query k-th smallest (0-based) in a[l, r)
  T kth(uint l, uint r, uint k) const {
    T res = 0;
    for (int h = lg; h--;) {
      uint tl = b[h].rank0(l), tr = b[h].rank0(r);
      if (k >= tr - tl) {
        k -= tr - tl;
        l += b[h].cnt0 - tl;
        r += b[h].cnt0 - tr;
        res |= T(1) << h;
      } else
        l = tl, r = tr;
    }
    return res;
  }
  // count of i in [l, r) with a[i] < u
  uint count(uint l, uint r, T u) const {
    if (u >= T(1) << lg) return r - l;
    uint res = 0;
    for (int h = lg; h--;) {
      uint tl = b[h].rank0(l), tr = b[h].rank0(r);
      if (u & (T(1) << h)) {
        l += b[h].cnt0 - tl;
        r += b[h].cnt0 - tr;
        res += tr - tl;
      } else
        l = tl, r = tr;
    }
```

```
    return res;
  }
};
```

### 2.7. Link-Cut Tree

```
const int MXN = 100005;
const int MEM = 100005;

struct Splay {
  static Splay nil, mem[MEM], *pmem;
  Splay *ch[2], *f;
  int val, rev, size;
  Splay() : val(-1), rev(0), size(0) {
    f = ch[0] = ch[1] = &nil;
  }
  Splay(int _val) : val(_val), rev(0), size(1) {
    f = ch[0] = ch[1] = &nil;
  }
  bool isr() {
    return f->ch[0] != this && f->ch[1] != this;
  }
  int dir() { return f->ch[0] == this ? 0 : 1; }
  void setCh(Splay *c, int d) {
    ch[d] = c;
    if (c != &nil) c->f = this;
    pull();
  }
  void push() {
    if (rev) {
      swap(ch[0], ch[1]);
      if (ch[0] != &nil) ch[0]->rev ^= 1;
      if (ch[1] != &nil) ch[1]->rev ^= 1;
      rev = 0;
    }
  }
  void pull() {
    size = ch[0]->size + ch[1]->size + 1;
    if (ch[0] != &nil) ch[0]->f = this;
    if (ch[1] != &nil) ch[1]->f = this;
  }
} Splay::nil, Splay::mem[MEM], *Splay::pmem = Splay::mem;
Splay *nil = &Splay::nil;

void rotate(Splay *x) {
  Splay *p = x->f;
  int d = x->dir();
  if (!p->isr())
    p->f->setCh(x, p->dir());
  else
    x->f = p->f;
  p->setCh(x->ch[!d], d);
  x->setCh(p, !d);
  p->pull();
  x->pull();
}

vector<Splay *> splayVec;
void splay(Splay *x) {
  splayVec.clear();
  for (Splay *q = x;; q = q->f) {
    splayVec.push_back(q);
    if (q->isr()) break;
  }
  reverse(begin(splayVec), end(splayVec));
  for (auto it : splayVec) it->push();
  while (!x->isr()) {
    if (x->f->isr())
      rotate(x);
    else if (x->dir() == x->f->dir())
      rotate(x->f), rotate(x);
    else
      rotate(x), rotate(x);
  }
}

Splay *access(Splay *x) {
  Splay *q = nil;
  for (; x != nil; x = x->f) {
    splay(x);
    x->setCh(q, 1);
    q = x;
  }
  return q;
}
void evert(Splay *x) {
  access(x);
  splay(x);
  x->rev ^= 1;
  x->push();
  x->pull();
}
void link(Splay *x, Splay *y) {
  //  evert(x);
  access(x);
  splay(x);
  evert(y);
  x->setCh(y, 1);
}
void cut(Splay *x, Splay *y) {
  //  evert(x);
  access(y);
  splay(y);
  y->push();
  y->ch[0] = y->ch[0]->f = nil;
}

int N, Q;
Splay *vt[MXN];

int ask(Splay *x, Splay *y) {
  access(x);
  access(y);
  splay(x);
  int res = x->f->val;
  if (res == -1) res = x->val;
  return res;
}

int main(int argc, char **argv) {
  scanf("%d%d", &N, &Q);
  for (int i = 1; i <= N; i++)
    vt[i] = new (Splay::pmem++) Splay(i);
  while (Q--) {
    char cmd[105];
    int u, v;
    scanf("%s", cmd);
    if (cmd[1] == 'i') {
      scanf("%d%d", &u, &v);
      link(vt[v], vt[u]);
    } else if (cmd[0] == 'c') {
      scanf("%d", &v);
      cut(vt[1], vt[v]);
    } else {
      scanf("%d%d", &u, &v);
      int res = ask(vt[u], vt[v]);
      printf("%d\n", res);
    }
  }
}
```

## 3. Graph

### 3.1. Floyd-Warshall

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

### 3.2. Dijkstra

```
struct dijkstra {
  using S =
  int; // regular edge weight type, change if needed
  using T = long long; // maximum possible distance type,
                       // change if needed
  const T INF = 1e18;
  struct Edge {
    int target;
    S weight;
  };
  struct next_shortest {
    int node;
    T dist;
    bool operator<(const next_shortest &rhs) const {
      return dist > rhs.dist;
    }
  };
  int n;
  vector<vector<Edge>> e;
  dijkstra(int n) : n(n), e(n + 1) {}
  void add_directed_edge(int x, int y, S z) {
    e[x].push_back({y, z});
  }
  void add_undirected_edge(int x, int y, S z) {
```

```
    e[x].push_back({y, z});
    e[y].push_back({x, z});
  }
  vector<T> shortest_path(int src) {
    vector<T> d(n + 1, INF);
    d[src] = 0;
    priority_queue<next_shortest> pq;
    pq.push({src, 0});
    while (!pq.empty()) {
      int cur_node = pq.top().node;
      T cur_dist = pq.top().dist;
      pq.pop();
      if (cur_dist > d[cur_node]) continue;
      for (auto i : e[cur_node])
        if (cur_dist + i.weight < d[i.target]) {
          d[i.target] = cur_dist + i.weight;
          pq.push({i.target, d[i.target]});
        }
    }
    return d;
  }
};
```

## 3.3. Modeling

- Maximum/Minimum flow with lower bound / Circulation problem
  1. Construct super source $S$ and sink $T$.
  2. For each edge $(x, y, l, u)$, connect $x \to y$ with capacity $u - l$.
  3. For each vertex $v$, denote by $in(v)$ the difference between the sum of incoming lower bounds and the sum of outgoing lower bounds.
  4. If $in(v) > 0$, connect $S \to v$ with capacity $in(v)$, otherwise, connect $v \to T$ with capacity $-in(v)$.
     - To maximize, connect $t \to s$ with capacity $\infty$ (skip this in circulation problem), and let $f$ be the maximum flow from $S$ to $T$. If $f \neq \sum_{v \in V, in(v)>0} in(v)$, there's no solution. Otherwise, the maximum flow from $s$ to $t$ is the answer.
     - To minimize, let $f$ be the maximum flow from $S$ to $T$. Connect $t \to s$ with capacity $\infty$ and let the flow from $S$ to $T$ be $f'$. If $f + f' \neq \sum_{v \in V, in(v)>0} in(v)$, there's no solution. Otherwise, $f'$ is the answer.
  5. The solution of each edge $e$ is $l_e + f_e$, where $f_e$ corresponds to the flow of edge $e$ on the graph.
- Construct minimum vertex cover from maximum matching $M$ on bipartite graph $(X, Y)$
  1. Redirect every edge: $y \to x$ if $(x, y) \in M$, $x \to y$ otherwise.
  2. DFS from unmatched vertices in $X$.
  3. $x \in X$ is chosen iff $x$ is unvisited.
  4. $y \in Y$ is chosen iff $y$ is visited.
- Minimum cost cyclic flow
  1. Consruct super source $S$ and sink $T$
  2. For each edge $(x, y, c)$, connect $x \to y$ with $(cost, cap) = (c, 1)$ if $c > 0$, otherwise connect $y \to x$ with $(cost, cap) = (-c, 1)$
  3. For each edge with $c < 0$, sum these cost as $K$, then increase $d(y)$ by 1, decrease $d(x)$ by 1
  4. For each vertex $v$ with $d(v) > 0$, connect $S \to v$ with $(cost, cap) = (0, d(v))$
  5. For each vertex $v$ with $d(v) < 0$, connect $v \to T$ with $(cost, cap) = (0, -d(v))$
  6. Flow from $S$ to $T$, the answer is the cost of the flow $C + K$
- Maximum density induced subgraph
  1. Binary search on answer, suppose we're checking answer $T$
  2. Construct a max flow model, let $K$ be the sum of all weights
  3. Connect source $s \to v$, $v \in G$ with capacity $K$
  4. For each edge $(u, v, w)$ in $G$, connect $u \to v$ and $v \to u$ with capacity $w$
  5. For $v \in G$, connect it with sink $v \to t$ with capacity $K + 2T - (\sum_{e \in E(v)} w(e)) - 2w(v)$
  6. $T$ is a valid answer if the maximum flow $f < K|V|$
- Minimum weight edge cover
  1. For each $v \in V$ create a copy $v'$, and connect $u' \to v'$ with weight $w(u, v)$.
  2. Connect $v \to v'$ with weight $2\mu(v)$, where $\mu(v)$ is the cost of the cheapest edge incident to $v$.
  3. Find the minimum weight perfect matching on $G'$.
- Project selection problem
  1. If $p_v > 0$, create edge $(s, v)$ with capacity $p_v$; otherwise, create edge $(v, t)$ with capacity $-p_v$.
  2. Create edge $(u, v)$ with capacity $w$ with $w$ being the cost of choosing $u$ without choosing $v$.
  3. The mincut is equivalent to the maximum profit of a subset of projects.
- 0/1 quadratic programming

$$\sum_x c_x x + \sum_y c_y \bar{y} + \sum_{xy} c_{xy} x\bar{y} + \sum_{xyx'y'} c_{xyx'y'}(x\bar{y} + x'\bar{y'})$$

  can be minimized by the mincut of the following graph:
  1. Create edge $(x, t)$ with capacity $c_x$ and create edge $(s, y)$ with capacity $c_y$.
  2. Create edge $(x, y)$ with capacity $c_{xy}$.
  3. Create edge $(x, y)$ and edge $(x', y')$ with capacity $c_{xyx'y'}$.

## 3.4. Matching/Flows

### 3.4.1. Dinic's Algorithm

```
struct Dinic {
  struct edge {
    int to, cap, flow, rev;
  };
  static constexpr int MAXN = 1000, MAXF = 1e9;
  vector<edge> v[MAXN];
  int top[MAXN], deep[MAXN], side[MAXN], s, t;
  void make_edge(int s, int t, int cap) {
    v[s].push_back({t, cap, 0, (int)v[t].size()});
    v[t].push_back({s, 0, 0, (int)v[s].size() - 1});
  }
  int dfs(int a, int flow) {
    if (a == t || !flow) return flow;
    for (int &i = top[a]; i < v[a].size(); i++) {
      edge &e = v[a][i];
      if (deep[a] + 1 == deep[e.to] && e.cap - e.flow) {
        int x = dfs(e.to, min(e.cap - e.flow, flow));
        if (x) {
          e.flow += x, v[e.to][e.rev].flow -= x;
          return x;
        }
      }
    }
    deep[a] = -1;
    return 0;
  }
  bool bfs() {
    queue<int> q;
    fill_n(deep, MAXN, 0);
    q.push(s), deep[s] = 1;
    int tmp;
    while (!q.empty()) {
      tmp = q.front(), q.pop();
      for (edge e : v[tmp])
        if (!deep[e.to] && e.cap != e.flow)
          deep[e.to] = deep[tmp] + 1, q.push(e.to);
    }
    return deep[t];
  }
  int max_flow(int _s, int _t) {
    s = _s, t = _t;
    int flow = 0, tflow;
    while (bfs()) {
      fill_n(top, MAXN, 0);
      while ((tflow = dfs(s, MAXF))) flow += tflow;
    }
    return flow;
  }
  void reset() {
    fill_n(side, MAXN, 0);
    for (auto &i : v) i.clear();
  }
};
```

### 3.4.2. Minimum Cost Flow

```
struct MCF {
  struct edge {
    ll to, from, cap, flow, cost, rev;
  } * fromE[MAXN];
  vector<edge> v[MAXN];
  ll n, s, t, flows[MAXN], dis[MAXN], pi[MAXN], flowlim;
  void make_edge(int s, int t, ll cap, ll cost) {
    if (!cap) return;
    v[s].pb(edge{t, s, cap, 0LL, cost, v[t].size()});
    v[t].pb(edge{s, t, 0LL, 0LL, -cost, v[s].size() - 1});
  }
  bitset<MAXN> vis;
  void dijkstra() {
    vis.reset();
    __gnu_pbds::priority_queue<pair<ll, int>> q;
    vector<decltype(q)::point_iterator> its(n);
    q.push({0LL, s});
    while (!q.empty()) {
      int now = q.top().second;
      q.pop();
      if (vis[now]) continue;
      vis[now] = 1;
      ll ndis = dis[now] + pi[now];
      for (edge &e : v[now]) {
        if (e.flow == e.cap || vis[e.to]) continue;
        if (dis[e.to] > ndis + e.cost - pi[e.to]) {
          dis[e.to] = ndis + e.cost - pi[e.to];
          flows[e.to] = min(flows[now], e.cap - e.flow);
          fromE[e.to] = &e;
          if (its[e.to] == q.end())
            its[e.to] = q.push({-dis[e.to], e.to});
```

```
        else
          q.modify(its[e.to], {-dis[e.to], e.to});
      }
    }
  }
}
bool AP(ll &flow) {
  fill_n(dis, n, INF);
  fromE[s] = 0;
  dis[s] = 0;
  flows[s] = flowlim - flow;
  dijkstra();
  if (dis[t] == INF) return false;
  flow += flows[t];
  for (edge *e = fromE[t]; e; e = fromE[e->from]) {
    e->flow += flows[t];
    v[e->to][e->rev].flow -= flows[t];
  }
  for (int i = 0; i < n; i++)
    pi[i] = min(pi[i] + dis[i], INF);
  return true;
}
pll solve(int _s, int _t, ll _flowlim = INF) {
  s = _s, t = _t, flowlim = _flowlim;
  pll re;
  while (re.F != flowlim && AP(re.F))
    ;
  for (int i = 0; i < n; i++)
    for (edge &e : v[i])
      if (e.flow != 0) re.S += e.flow * e.cost;
  re.S /= 2;
  return re;
}
void init(int _n) {
  n = _n;
  fill_n(pi, n, 0);
  for (int i = 0; i < n; i++) v[i].clear();
}
void setpi(int s) {
  fill_n(pi, n, INF);
  pi[s] = 0;
  for (ll it = 0, flag = 1, tdis; flag && it < n; it++) {
    flag = 0;
    for (int i = 0; i < n; i++)
      if (pi[i] != INF)
        for (edge &e : v[i])
          if (e.cap && (tdis = pi[i] + e.cost) < pi[e.to])
            pi[e.to] = tdis, flag = 1;
  }
}
};
```

### 3.4.3. Gomory-Hu Tree

Requires: Dinic's Algorithm

```
int e[MAXN][MAXN];
int p[MAXN];
Dinic D; // original graph
void gomory_hu() {
  fill(p, p + n, 0);
  fill(e[0], e[n], INF);
  for (int s = 1; s < n; s++) {
    int t = p[s];
    Dinic F = D;
    int tmp = F.max_flow(s, t);
    for (int i = 1; i < s; i++)
      e[s][i] = e[i][s] = min(tmp, e[t][i]);
    for (int i = s + 1; i <= n; i++)
      if (p[i] == t && F.side[i]) p[i] = s;
  }
}
```

### 3.4.4. Global Minimum Cut

```
// weights is an adjacency matrix, undirected
pair<int, vi> getMinCut(vector<vi> &weights) {
  int N = sz(weights);
  vi used(N), cut, best_cut;
  int best_weight = -1;

  for (int phase = N - 1; phase >= 0; phase--) {
    vi w = weights[0], added = used;
    int prev, k = 0;
    rep(i, 0, phase) {
      prev = k;
      k = -1;
      rep(j, 1, N) if (!added[j] &&
                      (k == -1 || w[j] > w[k])) k = j;
      if (i == phase - 1) {
        rep(j, 0, N) weights[prev][j] += weights[k][j];
        rep(j, 0, N) weights[j][prev] = weights[prev][j];
        used[k] = true;
        cut.push_back(k);
        if (best_weight == -1 || w[k] < best_weight) {
          best_cut = cut;
          best_weight = w[k];
        }
      } else {
        rep(j, 0, N) w[j] += weights[k][j];
        added[k] = true;
      }
    }
  }
  return {best_weight, best_cut};
}
```

### 3.4.5. Bipartite Minimum Cover

Requires: Dinic's Algorithm

```
// maximum independent set = all vertices not covered
// x : [0, n), y : [0, m]
struct Bipartite_vertex_cover {
  Dinic D;
  int n, m, s, t, x[maxn], y[maxn];
  void make_edge(int x, int y) { D.make_edge(x, y + n, 1); }
  int matching() {
    int re = D.max_flow(s, t);
    for (int i = 0; i < n; i++)
      for (Dinic::edge &e : D.v[i])
        if (e.to != s && e.flow == 1) {
          x[i] = e.to - n, y[e.to - n] = i;
          break;
        }
    return re;
  }
  // init() and matching() before use
  void solve(vector<int> &vx, vector<int> &vy) {
    bitset<maxn * 2 + 10> vis;
    queue<int> q;
    for (int i = 0; i < n; i++)
      if (x[i] == -1) q.push(i), vis[i] = 1;
    while (!q.empty()) {
      int now = q.front();
      q.pop();
      if (now < n) {
        for (Dinic::edge &e : D.v[now])
          if (e.to != s && e.to - n != x[now] && !vis[e.to])
            vis[e.to] = 1, q.push(e.to);
      } else {
        if (!vis[y[now - n]])
          vis[y[now - n]] = 1, q.push(y[now - n]);
      }
    }
    for (int i = 0; i < n; i++)
      if (!vis[i]) vx.pb(i);
    for (int i = 0; i < m; i++)
      if (vis[i + n]) vy.pb(i);
  }
  void init(int _n, int _m) {
    n = _n, m = _m, s = n + m, t = s + 1;
    for (int i = 0; i < n; i++)
      x[i] = -1, D.make_edge(s, i, 1);
    for (int i = 0; i < m; i++)
      y[i] = -1, D.make_edge(i + n, t, 1);
  }
};
```

### 3.4.6. Edmonds' Algorithm

```
struct Edmonds {
  int n, T;
  vector<vector<int>> g;
  vector<int> pa, p, used, base;
  Edmonds(int n)
      : n(n), T(0), g(n), pa(n, -1), p(n), used(n),
        base(n) {}
  void add(int a, int b) {
    g[a].push_back(b);
    g[b].push_back(a);
  }
  int getBase(int i) {
    while (i != base[i])
      base[i] = base[base[i]], i = base[i];
    return i;
  }
  vector<int> toJoin;
  void mark_path(int v, int x, int b, vector<int> &path) {
    for (; getBase(v) != b; v = p[x]) {
      p[v] = x, x = pa[v];
```

```cpp
      toJoin.push_back(v);
      toJoin.push_back(x);
      if (!used[x]) used[x] = ++T, path.push_back(x);
    }
  }
  bool go(int v) {
    for (int x : g[v]) {
      int b, bv = getBase(v), bx = getBase(x);
      if (bv == bx) {
        continue;
      } else if (used[x]) {
        vector<int> path;
        toJoin.clear();
        if (used[bx] < used[bv])
          mark_path(v, x, b = bx, path);
        else
          mark_path(x, v, b = bv, path);
        for (int z : toJoin) base[getBase(z)] = b;
        for (int z : path)
          if (go(z)) return 1;
      } else if (p[x] == -1) {
        p[x] = v;
        if (pa[x] == -1) {
          for (int y; x != -1; x = v)
            y = p[x], v = pa[y], pa[x] = y, pa[y] = x;
          return 1;
        }
        if (!used[pa[x]]) {
          used[pa[x]] = ++T;
          if (go(pa[x])) return 1;
        }
      }
    }
    return 0;
  }
  void init_dfs() {
    for (int i = 0; i < n; i++)
      used[i] = 0, p[i] = -1, base[i] = i;
  }
  bool dfs(int root) {
    used[root] = ++T;
    return go(root);
  }
  void match() {
    int ans = 0;
    for (int v = 0; v < n; v++)
      for (int x : g[v])
        if (pa[v] == -1 && pa[x] == -1) {
          pa[v] = x, pa[x] = v, ans++;
          break;
        }
    init_dfs();
    for (int i = 0; i < n; i++)
      if (pa[i] == -1 && dfs(i)) ans++, init_dfs();
    cout << ans * 2 << "\n";
    for (int i = 0; i < n; i++)
      if (pa[i] > i)
        cout << i + 1 << " " << pa[i] + 1 << "\n";
  }
};
```

### 3.4.7. Minimum Weight Matching

```cpp
struct Graph {
  static const int MAXN = 105;
  int n, e[MAXN][MAXN];
  int match[MAXN], d[MAXN], onstk[MAXN];
  vector<int> stk;
  void init(int _n) {
    n = _n;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        // change to appropriate infinity
        // if not complete graph
        e[i][j] = 0;
  }
  void add_edge(int u, int v, int w) {
    e[u][v] = e[v][u] = w;
  }
  bool SPFA(int u) {
    if (onstk[u]) return true;
    stk.push_back(u);
    onstk[u] = 1;
    for (int v = 0; v < n; v++) {
      if (u != v && match[u] != v && !onstk[v]) {
        int m = match[v];
        if (d[m] > d[u] - e[v][m] + e[u][v]) {
          d[m] = d[u] - e[v][m] + e[u][v];
          onstk[v] = 1;
          stk.push_back(v);
          if (SPFA(m)) return true;
          stk.pop_back();
          onstk[v] = 0;
        }
      }
    }
    onstk[u] = 0;
    stk.pop_back();
    return false;
  }
  int solve() {
    for (int i = 0; i < n; i += 2) {
      match[i] = i + 1;
      match[i + 1] = i;
    }
    while (true) {
      int found = 0;
      for (int i = 0; i < n; i++) onstk[i] = d[i] = 0;
      for (int i = 0; i < n; i++) {
        stk.clear();
        if (!onstk[i] && SPFA(i)) {
          found = 1;
          while (stk.size() >= 2) {
            int u = stk.back();
            stk.pop_back();
            int v = stk.back();
            stk.pop_back();
            match[u] = v;
            match[v] = u;
          }
        }
      }
      if (!found) break;
    }
    int ret = 0;
    for (int i = 0; i < n; i++) ret += e[i][match[i]];
    ret /= 2;
    return ret;
  }
} graph;
```

### 3.4.8. Stable Marriage

```cpp
// normal stable marriage problem
/* input:
3
Albert Laura Nancy Marcy
Brad Marcy Nancy Laura
Chuck Laura Marcy Nancy
Laura Chuck Albert Brad
Marcy Albert Chuck Brad
Nancy Brad Albert Chuck
*/


using namespace std;
const int MAXN = 505;

int n;
int favor[MAXN][MAXN]; // favor[boy_id][rank] = girl_id;
int order[MAXN][MAXN]; // order[girl_id][boy_id] = rank;
int current[MAXN];     // current[boy_id] = rank;
// boy_id will pursue current[boy_id] girl.
int girl_current[MAXN]; // girl[girl_id] = boy_id;

void initialize() {
  for (int i = 0; i < n; i++) {
    current[i] = 0;
    girl_current[i] = n;
    order[i][n] = n;
  }
}

map<string, int> male, female;
string bname[MAXN], gname[MAXN];
int fit = 0;

void stable_marriage() {

  queue<int> que;
  for (int i = 0; i < n; i++) que.push(i);
  while (!que.empty()) {
    int boy_id = que.front();
    que.pop();

    int girl_id = favor[boy_id][current[boy_id]];
    current[boy_id]++;

    if (order[girl_id][boy_id] <
        order[girl_id][girl_current[girl_id]]) {
      if (girl_current[girl_id] < n)
```

```
        que.push(girl_current[girl_id]);
      girl_current[girl_id] = boy_id;
    } else {
      que.push(boy_id);
    }
  }
}

int main() {
  cin >> n;

  for (int i = 0; i < n; i++) {
    string p, t;
    cin >> p;
    male[p] = i;
    bname[i] = p;
    for (int j = 0; j < n; j++) {
      cin >> t;
      if (!female.count(t)) {
        gname[fit] = t;
        female[t] = fit++;
      }
      favor[i][j] = female[t];
    }
  }

  for (int i = 0; i < n; i++) {
    string p, t;
    cin >> p;
    for (int j = 0; j < n; j++) {
      cin >> t;
      order[female[p]][male[t]] = j;
    }
  }

  initialize();
  stable_marriage();

  for (int i = 0; i < n; i++) {
    cout << bname[i] << " "
         << gname[favor[i][current[i] - 1]] << endl;
  }
}
```

### 3.4.9. Kuhn-Munkres algorithm

```
// Maximum Weight Perfect Bipartite Matching
// Detect non-perfect-matching:
// 1. set all edge[i][j] as INF
// 2. if solve() >= INF, it is not perfect matching.

typedef long long ll;
struct KM {
  static const int MAXN = 1050;
  static const ll INF = 1LL << 60;
  int n, match[MAXN], vx[MAXN], vy[MAXN];
  ll edge[MAXN][MAXN], lx[MAXN], ly[MAXN], slack[MAXN];
  void init(int _n) {
    n = _n;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) edge[i][j] = 0;
  }
  void add_edge(int x, int y, ll w) { edge[x][y] = w; }
  bool DFS(int x) {
    vx[x] = 1;
    for (int y = 0; y < n; y++) {
      if (vy[y]) continue;
      if (lx[x] + ly[y] > edge[x][y]) {
        slack[y] =
        min(slack[y], lx[x] + ly[y] - edge[x][y]);
      } else {
        vy[y] = 1;
        if (match[y] == -1 || DFS(match[y])) {
          match[y] = x;
          return true;
        }
      }
    }
    return false;
  }
  ll solve() {
    fill(match, match + n, -1);
    fill(lx, lx + n, -INF);
    fill(ly, ly + n, 0);
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        lx[i] = max(lx[i], edge[i][j]);
    for (int i = 0; i < n; i++) {
      fill(slack, slack + n, INF);
      while (true) {
        fill(vx, vx + n, 0);
        fill(vy, vy + n, 0);
        if (DFS(i)) break;
        ll d = INF;
        for (int j = 0; j < n; j++)
          if (!vy[j]) d = min(d, slack[j]);
        for (int j = 0; j < n; j++) {
          if (vx[j]) lx[j] -= d;
          if (vy[j])
            ly[j] += d;
          else
            slack[j] -= d;
        }
      }
    }
    ll res = 0;
    for (int i = 0; i < n; i++) {
      res += edge[match[i]][i];
    }
    return res;
  }
} graph;
```

## 3.5. Shortest Path Faster Algorithm

```
struct SPFA {
  static const int maxn = 1010, INF = 1e9;
  int dis[maxn];
  bitset<maxn> inq, inneg;
  queue<int> q, tq;
  vector<pii> v[maxn];
  void make_edge(int s, int t, int w) {
    v[s].emplace_back(t, w);
  }
  void dfs(int a) {
    inneg[a] = 1;
    for (pii i : v[a])
      if (!inneg[i.F]) dfs(i.F);
  }
  bool solve(int n, int s) { // true if have neg-cycle
    for (int i = 0; i <= n; i++) dis[i] = INF;
    dis[s] = 0, q.push(s);
    for (int i = 0; i < n; i++) {
      inq.reset();
      int now;
      while (!q.empty()) {
        now = q.front(), q.pop();
        for (pii &i : v[now]) {
          if (dis[i.F] > dis[now] + i.S) {
            dis[i.F] = dis[now] + i.S;
            if (!inq[i.F]) tq.push(i.F), inq[i.F] = 1;
          }
        }
      }
      q.swap(tq);
    }
    bool re = !q.empty();
    inneg.reset();
    while (!q.empty()) {
      if (!inneg[q.front()]) dfs(q.front());
      q.pop();
    }
    return re;
  }
  void reset(int n) {
    for (int i = 0; i <= n; i++) v[i].clear();
  }
};
```

## 3.6. Strongly Connected Components

```
struct strongly_connected_components {
  int n, component_count;
  vector<vector<int>> edges, reverse_edges;
  vector<int> component, component_size;
  strongly_connected_components(int n)
      : n(n), edges(n + 1), reverse_edges(n + 1),
        component(n + 1), component_size(n + 1),
        component_count(0) {}
  void add_edge(int x, int y) {
    edges[x].push_back(y);
    reverse_edges[y].push_back(x);
  }
  void dfs(int c, vector<int> &out_order,
               vector<bool> &done) {
    done[c] = true;
    for (int i : edges[c])
      if (!done[i]) dfs(i, out_order, done);
    out_order.push_back(c);
  }
  void dfs_reverse(int c, vector<bool> &done) {
```

```
    done[c] = true;
    component[c] = component_count;
    ++component_size[component_count];
    for (int i : reverse_edges[c])
      if (!done[i]) dfs_reverse(i, done);
  }
  void build_scc() {
    vector<int> out_order;
    vector<bool> done(n + 1, false),
    done_reverse(n + 1, false);
    for (int i = 1; i <= n; ++i)
      if (!done[i]) dfs(i, out_order, done);
    component_count = 0;
    for (int i = out_order.size() - 1; i >= 0; --i)
      if (!done_reverse[out_order[i]]) {
        ++component_count;
        component_size[component_count] = 0;
        dfs_reverse(out_order[i], done_reverse);
      }
  }
};
```

### 3.6.1. 2-Satisfiability

Requires: Strongly Connected Components

```
// 1 based, vertex in SCC = MAXN * 2
// (not i) is i + n
struct two_SAT {
  int n, ans[MAXN];
  SCC S;
  void imply(int a, int b) { S.make_edge(a, b); }
  bool solve(int _n) {
    n = _n;
    S.solve(n * 2);
    for (int i = 1; i <= n; i++) {
      if (S.scc[i] == S.scc[i + n]) return false;
      ans[i] = (S.scc[i] < S.scc[i + n]);
    }
    return true;
  }
  void init(int _n) {
    n = _n;
    fill_n(ans, n + 1, 0);
    S.init(n * 2);
  }
} SAT;
```

## 3.7. Biconnected Components

### 3.7.1. Articulation Points

```
void dfs(int v, int p = -1) {
  visited[v] = true;
  tin[v] = low[v] = timer++;
  int children = 0;
  for (int to : adj[v]) {
    if (to == p) continue;
    if (visited[to]) {
      low[v] = min(low[v], tin[to]);
    } else {
      dfs(to, v);
      low[v] = min(low[v], low[to]);
      if (low[to] >= tin[v] && p != -1) IS_CUTPOINT(v);
      ++children;
    }
  }
  if (p == -1 && children > 1) IS_CUTPOINT(v);
}
```

### 3.7.2. Bridges

```
// if there are multi-edges, then they are not bridges
void dfs(int v, int p = -1) {
  visited[v] = true;
  tin[v] = low[v] = timer++;
  for (int to : adj[v]) {
    if (to == p) continue;
    if (visited[to]) {
      low[v] = min(low[v], tin[to]);
    } else {
      dfs(to, v);
      low[v] = min(low[v], low[to]);
      if (low[to] > tin[v]) IS_BRIDGE(v, to);
    }
  }
}
```

### 3.7.3. Block Cut Tree

```
struct block_cut_tree {
  int n, edge_id, bct_node_cnt;
  vector<vector<array<int, 2>>> edges;
  vector<vector<int>> edges_bct;
  vector<bool> is_split;
  block_cut_tree(int n)
      : n(n), edges(n + 1), edges_bct(n * 3 + 1),
        edge_id(0) {}
  void add_edge(int x, int y) {
    edges[x].push_back({y, edge_id});
    edges[y].push_back({x, edge_id});
    ++edge_id;
    is_split.push_back(false);
  }
  void dfs_split(int c, int pid, int &timer,
                 vector<int> &tin, vector<int> &low,
                 vector<bool> &done) {
    done[c] = true;
    tin[c] = low[c] = ++timer;
    for (auto i : edges[c])
      if (i[1] != pid) {
        if (done[i[0]])
          low[c] = min(low[c], tin[i[0]]);
        else {
          dfs_split(i[0], i[1], timer, tin, low, done);
          low[c] = min(low[c], low[i[0]]);
          is_split[i[1]] = tin[c] <= low[i[0]];
        }
      }
  }
  void dfs_build_bct(int c, int g, int &timer,
                     vector<bool> &done) {
    done[c] = true;
    edges_bct[c].push_back(g + n);
    edges_bct[g + n].push_back(c);
    for (auto i : edges[c])
      if (!done[i[0]]) {
        if (is_split[i[1]]) {
          ++timer;
          edges_bct[c].push_back(timer + n);
          edges_bct[timer + n].push_back(c);
          dfs_build_bct(i[0], timer, timer, done);
        } else
          dfs_build_bct(i[0], g, timer, done);
      }
  }
  void build_bct() {
    int timer = 0;
    vector<int> tin(n + 1), low(n + 1);
    vector<bool> done_dfs(n + 1), done_build(n + 1);
    bct_node_cnt = 0;
    for (int i = 1; i <= n; ++i)
      if (!done_dfs[i]) {
        dfs_split(i, -1, timer, tin, low, done_dfs);
        ++bct_node_cnt;
        dfs_build_bct(i, bct_node_cnt, bct_node_cnt,
                      done_build);
      }
  }
};
```

## 3.8. Triconnected Components

```
// requires a union-find data structure
struct ThreeEdgeCC {
  int V, ind;
  vector<int> id, pre, post, low, deg, path;
  vector<vector<int>> components;
  UnionFind uf;
  template <class Graph>
  void dfs(const Graph &G, int v, int prev) {
    pre[v] = ++ind;
    for (int w : G[v])
      if (w != v) {
        if (w == prev) {
          prev = -1;
          continue;
        }
        if (pre[w] != -1) {
          if (pre[w] < pre[v]) {
            deg[v]++;
            low[v] = min(low[v], pre[w]);
          } else {
            deg[v]--;
            int &u = path[v];
            for (; u != -1 && pre[u] <= pre[w] &&
                   pre[w] <= post[u];) {
              uf.join(v, u);
              deg[v] += deg[u];
```

```
              u = path[u];
            }
          }
          continue;
        }
        dfs(G, w, v);
        if (path[w] == -1 && deg[w] <= 1) {
          deg[v] += deg[w];
          low[v] = min(low[v], low[w]);
          continue;
        }
        if (deg[w] == 0) w = path[w];
        if (low[v] > low[w]) {
          low[v] = min(low[v], low[w]);
          swap(w, path[v]);
        }
        for (; w != -1; w = path[w]) {
          uf.join(v, w);
          deg[v] += deg[w];
        }
      }
    post[v] = ind;
  }
  template <class Graph>
  ThreeEdgeCC(const Graph &G)
      : V(G.size()), ind(-1), id(V, -1), pre(V, -1),
        post(V), low(V, INT_MAX), deg(V, 0), path(V, -1),
        uf(V) {
    for (int v = 0; v < V; v++)
      if (pre[v] == -1) dfs(G, v, -1);
    components.reserve(uf.cnt);
    for (int v = 0; v < V; v++)
      if (uf.find(v) == v) {
        id[v] = components.size();
        components.emplace_back(1, v);
        components.back().reserve(uf.getSize(v));
      }
    for (int v = 0; v < V; v++)
      if (id[v] == -1)
        components[id[v] = id[uf.find(v)]].push_back(v);
  }
};
```

### 3.9. Centroid Decomposition

```
void get_center(int now) {
  v[now] = true;
  vtx.push_back(now);
  sz[now] = 1;
  mx[now] = 0;
  for (int u : G[now])
    if (!v[u]) {
      get_center(u);
      mx[now] = max(mx[now], sz[u]);
      sz[now] += sz[u];
    }
}
void get_dis(int now, int d, int len) {
  dis[d][now] = cnt;
  v[now] = true;
  for (auto u : G[now])
    if (!v[u.first]) { get_dis(u, d, len + u.second); }
}
void dfs(int now, int fa, int d) {
  get_center(now);
  int c = -1;
  for (int i : vtx) {
    if (max(mx[i], (int)vtx.size() - sz[i]) <=
        (int)vtx.size() / 2)
      c = i;
    v[i] = false;
  }
  get_dis(c, d, 0);
  for (int i : vtx) v[i] = false;
  v[c] = true;
  vtx.clear();
  dep[c] = d;
  p[c] = fa;
  for (auto u : G[c])
    if (u.first != fa && !v[u.first]) {
      dfs(u.first, c, d + 1);
    }
}
```

### 3.10. Minimum Mean Cycle

```
// d[i][j] == 0 if {i,j} !in E
long long d[1003][1003], dp[1003][1003];

pair<long long, long long> MMWC() {
  memset(dp, 0x3f, sizeof(dp));
  for (int i = 1; i <= n; ++i) dp[0][i] = 0;
  for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
      for (int k = 1; k <= n; ++k) {
        dp[i][k] = min(dp[i - 1][j] + d[j][k], dp[i][k]);
      }
    }
  }
  long long au = 1ll << 31, ad = 1;
  for (int i = 1; i <= n; ++i) {
    if (dp[n][i] == 0x3f3f3f3f3f3f3f3f) continue;
    long long u = 0, d = 1;
    for (int j = n - 1; j >= 0; --j) {
      if ((dp[n][i] - dp[j][i]) * d > u * (n - j)) {
        u = dp[n][i] - dp[j][i];
        d = n - j;
      }
    }
    if (u * ad < au * d) au = u, ad = d;
  }
  long long g = __gcd(au, ad);
  return make_pair(au / g, ad / g);
}
```

### 3.11. Directed MST

```
template <typename T> struct DMST {
  T g[maxn][maxn], fw[maxn];
  int n, fr[maxn];
  bool vis[maxn], inc[maxn];
  void clear() {
    for (int i = 0; i < maxn; ++i) {
      for (int j = 0; j < maxn; ++j) g[i][j] = inf;
      vis[i] = inc[i] = false;
    }
  }
  void addedge(int u, int v, T w) {
    g[u][v] = min(g[u][v], w);
  }
  T operator()(int root, int _n) {
    n = _n;
    if (dfs(root) != n) return -1;
    T ans = 0;
    while (true) {
      for (int i = 1; i <= n; ++i) fw[i] = inf, fr[i] = i;
      for (int i = 1; i <= n; ++i)
        if (!inc[i]) {
          for (int j = 1; j <= n; ++j) {
            if (!inc[j] && i != j && g[j][i] < fw[i]) {
              fw[i] = g[j][i];
              fr[i] = j;
            }
          }
        }
      int x = -1;
      for (int i = 1; i <= n; ++i)
        if (i != root && !inc[i]) {
          int j = i, c = 0;
          while (j != root && fr[j] != i && c <= n)
            ++c, j = fr[j];
          if (j == root || c > n)
            continue;
          else {
            x = i;
            break;
          }
        }
      if (!~x) {
        for (int i = 1; i <= n; ++i)
          if (i != root && !inc[i]) ans += fw[i];
        return ans;
      }
      int y = x;
      for (int i = 1; i <= n; ++i) vis[i] = false;
      do {
        ans += fw[y];
        y = fr[y];
        vis[y] = inc[y] = true;
      } while (y != x);
      inc[x] = false;
      for (int k = 1; k <= n; ++k)
        if (vis[k]) {
          for (int j = 1; j <= n; ++j)
            if (!vis[j]) {
              if (g[x][j] > g[k][j]) g[x][j] = g[k][j];
              if (g[j][k] < inf &&
                  g[j][k] - fw[k] < g[j][x])
                g[j][x] = g[j][k] - fw[k];
            }
```

```
      }
    }
    return ans;
  }
  int dfs(int now) {
    int r = 1;
    vis[now] = true;
    for (int i = 1; i <= n; ++i)
      if (g[now][i] < inf && !vis[i]) r += dfs(i);
    return r;
  }
};
```

## 3.12. Maximum Clique

```
// source: KACTL

typedef vector<bitset<200>> vb;
struct Maxclique {
  double limit = 0.025, pk = 0;
  struct Vertex {
    int i, d = 0;
  };
  typedef vector<Vertex> vv;
  vb e;
  vv V;
  vector<vi> C;
  vi qmax, q, S, old;
  void init(vv &r) {
    for (auto &v : r) v.d = 0;
    for (auto &v : r)
      for (auto j : r) v.d += e[v.i][j.i];
    sort(all(r), [](auto a, auto b) { return a.d > b.d; });
    int mxD = r[0].d;
    rep(i, 0, sz(r)) r[i].d = min(i, mxD) + 1;
  }
  void expand(vv &R, int lev = 1) {
    S[lev] += S[lev - 1] - old[lev];
    old[lev] = S[lev - 1];
    while (sz(R)) {
      if (sz(q) + R.back().d <= sz(qmax)) return;
      q.push_back(R.back().i);
      vv T;
      for (auto v : R)
        if (e[R.back().i][v.i]) T.push_back({v.i});
      if (sz(T)) {
        if (S[lev]++ / ++pk < limit) init(T);
        int j = 0, mxk = 1,
            mnk = max(sz(qmax) - sz(q) + 1, 1);
        C[1].clear(), C[2].clear();
        for (auto v : T) {
          int k = 1;
          auto f = [&](int i) { return e[v.i][i]; };
          while (any_of(all(C[k]), f)) k++;
          if (k > mxk) mxk = k, C[mxk + 1].clear();
          if (k < mnk) T[j++].i = v.i;
          C[k].push_back(v.i);
        }
        if (j > 0) T[j - 1].d = 0;
        rep(k, mnk, mxk + 1) for (int i : C[k]) T[j].i = i,
                                                T[j++].d =
                                                k;
        expand(T, lev + 1);
      } else if (sz(q) > sz(qmax))
        qmax = q;
      q.pop_back(), R.pop_back();
    }
  }
  vi maxClique() {
    init(V), expand(V);
    return qmax;
  }
  Maxclique(vb conn)
      : e(conn), C(sz(e) + 1), S(sz(C)), old(S) {
    rep(i, 0, sz(e)) V.push_back({i});
  }
};
```

## 3.13. Dominator Tree

```
// idom[n] is the unique node that strictly dominates n but
// does not strictly dominate any other node that strictly
// dominates n. idom[n] = 0 if n is entry or the entry
// cannot reach n.
struct DominatorTree {
  static const int MAXN = 200010;
  int n, s;
  vector<int> g[MAXN], pred[MAXN];
  vector<int> cov[MAXN];
  int dfn[MAXN], nfd[MAXN], ts;
  int par[MAXN];
  int sdom[MAXN], idom[MAXN];
  int mom[MAXN], mn[MAXN];

  inline bool cmp(int u, int v) { return dfn[u] < dfn[v]; }

  int eval(int u) {
    if (mom[u] == u) return u;
    int res = eval(mom[u]);
    if (cmp(sdom[mn[mom[u]]], sdom[mn[u]]))
      mn[u] = mn[mom[u]];
    return mom[u] = res;
  }

  void init(int _n, int _s) {
    n = _n;
    s = _s;
    REP1(i, 1, n) {
      g[i].clear();
      pred[i].clear();
      idom[i] = 0;
    }
  }
  void add_edge(int u, int v) {
    g[u].push_back(v);
    pred[v].push_back(u);
  }
  void DFS(int u) {
    ts++;
    dfn[u] = ts;
    nfd[ts] = u;
    for (int v : g[u])
      if (dfn[v] == 0) {
        par[v] = u;
        DFS(v);
      }
  }
  void build() {
    ts = 0;
    REP1(i, 1, n) {
      dfn[i] = nfd[i] = 0;
      cov[i].clear();
      mom[i] = mn[i] = sdom[i] = i;
    }
    DFS(s);
    for (int i = ts; i >= 2; i--) {
      int u = nfd[i];
      if (u == 0) continue;
      for (int v : pred[u])
        if (dfn[v]) {
          eval(v);
          if (cmp(sdom[mn[v]], sdom[u]))
            sdom[u] = sdom[mn[v]];
        }
      cov[sdom[u]].push_back(u);
      mom[u] = par[u];
      for (int w : cov[par[u]]) {
        eval(w);
        if (cmp(sdom[mn[w]], par[u]))
          idom[w] = mn[w];
        else
          idom[w] = par[u];
      }
      cov[par[u]].clear();
    }
    REP1(i, 2, ts) {
      int u = nfd[i];
      if (u == 0) continue;
      if (idom[u] != sdom[u]) idom[u] = idom[idom[u]];
    }
  }
} dom;
```

## 3.14. Manhattan Distance MST

```
// returns [(dist, from, to), ...]
// then do normal mst afterwards
typedef Point<int> P;
vector<array<int, 3>> manhattanMST(vector<P> ps) {
  vi id(sz(ps));
  iota(all(id), 0);
  vector<array<int, 3>> edges;
  rep(k, 0, 4) {
    sort(all(id), [&](int i, int j) {
      return (ps[i] - ps[j]).x < (ps[j] - ps[i]).y;
    });
    map<int, int> sweep;
    for (int i : id) {
      for (auto it = sweep.lower_bound(-ps[i].y);
           it != sweep.end(); sweep.erase(it++)) {
```

```
        int j = it->second;
        P d = ps[i] - ps[j];
        if (d.y > d.x) break;
        edges.push_back({d.y + d.x, i, j});
      }
      sweep[-ps[i].y] = i;
    }
    for (P &p : ps)
      if (k & 1)
        p.x = -p.x;
      else
        swap(p.x, p.y);
  }
  return edges;
}
```

# 4. Math

## 4.1. Number Theory

### 4.1.1. Mod Struct

A list of safe primes: 26003, 27767, 28319, 28979, 29243, 29759, 30467
910927547, 919012223, 947326223, 990669467, 1007939579, 1019126699
929760389146037459, 975500632317046523, 989312547895528379

| NTT prime $p$ | $p-1$ | primitive root |
|---|---|---|
| 65537 | $1 \ll 16$ | 3 |
| 998244353 | $119 \ll 23$ | 3 |
| 2748779069441 | $5 \ll 39$ | 3 |
| 1945555039024054273 | $27 \ll 56$ | 5 |

Requires: Extended GCD

```
template <typename T> struct M {
  static T MOD; // change to constexpr if already known
  T v;
  M(T x = 0) {
    v = (-MOD <= x && x < MOD) ? x : x % MOD;
    if (v < 0) v += MOD;
  }
  explicit operator T() const { return v; }
  bool operator==(const M &b) const { return v == b.v; }
  bool operator!=(const M &b) const { return v != b.v; }
  M operator-() { return M(-v); }
  M operator+(M b) { return M(v + b.v); }
  M operator-(M b) { return M(v - b.v); }
  M operator*(M b) { return M((__int128)v * b.v % MOD); }
  M operator/(M b) { return *this * (b ^ (MOD - 2)); }
  // change above implementation to this if MOD is not prime
  M inv() {
    auto [p, _, g] = extgcd(v, MOD);
    return assert(g == 1), p;
  }
  friend M operator^(M a, ll b) {
    M ans(1);
    for (; b; b >>= 1, a *= a)
      if (b & 1) ans *= a;
    return ans;
  }
  friend M &operator+=(M &a, M b) { return a = a + b; }
  friend M &operator-=(M &a, M b) { return a = a - b; }
  friend M &operator*=(M &a, M b) { return a = a * b; }
  friend M &operator/=(M &a, M b) { return a = a / b; }
};
using Mod = M<int>;
template <> int Mod::MOD = 1'000'000'007;
int &MOD = Mod::MOD;
```

### 4.1.2. Miller-Rabin

Requires: Mod Struct

```
// checks if Mod::MOD is prime
bool is_prime() {
  if (MOD < 2 || MOD % 2 == 0) return MOD == 2;
  Mod A[] = {2, 7, 61}; // for int values (< 2^31)
  // ll: 2, 325, 9375, 28178, 450775, 9780504, 1795265022
  int s = __builtin_ctzll(MOD - 1), i;
  for (Mod a : A) {
    Mod x = a ^ (MOD >> s);
    for (i = 0; i < s && (x + 1).v > 2; i++) x *= x;
    if (i && x != -1) return 0;
  }
  return 1;
}
```

### 4.1.3. Linear Sieve

```
constexpr ll MAXN = 1000000;
bitset<MAXN> is_prime;
vector<ll> primes;
ll mpf[MAXN], phi[MAXN], mu[MAXN];

void sieve() {
  is_prime.set();
  is_prime[1] = 0;
  mu[1] = phi[1] = 1;
  for (ll i = 2; i < MAXN; i++) {
    if (is_prime[i]) {
      mpf[i] = i;
      primes.push_back(i);
      phi[i] = i - 1;
      mu[i] = -1;
    }
    for (ll p : primes) {
      if (p > mpf[i] || i * p >= MAXN) break;
      is_prime[i * p] = 0;
      mpf[i * p] = p;
      mu[i * p] = -mu[i];
      if (i % p == 0)
        phi[i * p] = phi[i] * p, mu[i * p] = 0;
      else
        phi[i * p] = phi[i] * (p - 1);
    }
  }
}
```

### 4.1.4. Get Factors

Requires: Linear Sieve

```
vector<ll> all_factors(ll n) {
  vector<ll> fac = {1};
  while (n > 1) {
    const ll p = mpf[n];
    vector<ll> cur = {1};
    while (n % p == 0) {
      n /= p;
      cur.push_back(cur.back() * p);
    }
    vector<ll> tmp;
    for (auto x : fac)
      for (auto y : cur) tmp.push_back(x * y);
    tmp.swap(fac);
  }
  return fac;
}
```

### 4.1.5. Binary GCD

```
// returns the gcd of non-negative a, b
ull bin_gcd(ull a, ull b) {
  if (!a || !b) return a + b;
  int s = __builtin_ctzll(a | b);
  a >>= __builtin_ctzll(a);
  while (b) {
    if ((b >>= __builtin_ctzll(b)) < a) swap(a, b);
    b -= a;
  }
  return a << s;
}
```

### 4.1.6. Extended GCD

```
// returns (p, q, g): p * a + q * b == g == gcd(a, b)
// g is not guaranteed to be positive when a < 0 or b < 0
tuple<ll, ll, ll> extgcd(ll a, ll b) {
  ll s = 1, t = 0, u = 0, v = 1;
  while (b) {
    ll q = a / b;
    swap(a -= q * b, b);
    swap(s -= q * t, t);
    swap(u -= q * v, v);
  }
  return {s, u, a};
}
```

### 4.1.7. Chinese Remainder Theorem

Requires: Extended GCD

```
// for 0 <= a < m, 0 <= b < n, returns the smallest x >= 0
// such that x % m == a and x % n == b
ll crt(ll a, ll m, ll b, ll n) {
  if (n > m) swap(a, b), swap(m, n);
  auto [x, y, g] = extgcd(m, n);
  assert((a - b) % g == 0); // no solution
  x = ((b - a) / g * x) % (n / g) * m + a;
  return x < 0 ? x + m / g * n : x;
}
```

### 4.1.8. Baby-Step Giant-Step

Requires: Mod Struct

```
// returns x such that a ^ x = b where x \in [l, r)
ll bsgs(Mod a, Mod b, ll l = 0, ll r = MOD - 1) {
  int m = sqrt(r - l) + 1, i;
  unordered_map<ll, ll> tb;
  Mod d = (a ^ l) / b;
  for (i = 0, d = (a ^ l) / b; i < m; i++, d *= a)
    if (d == 1)
      return l + i;
    else
      tb[(ll)d] = l + i;
  Mod c = Mod(1) / (a ^ m);
  for (i = 0, d = 1; i < m; i++, d *= c)
    if (auto j = tb.find((ll)d); j != tb.end())
      return j->second + i * m;
  return assert(0), -1; // no solution
}
```

### 4.1.9. Pollard's Rho

```
ll f(ll x, ll mod) { return (x * x + 1) % mod; }
// n should be composite
ll pollard_rho(ll n) {
  if (!(n & 1)) return 2;
  while (1) {
    ll y = 2, x = RNG() % (n - 1) + 1, res = 1;
    for (int sz = 2; res == 1; sz *= 2) {
      for (int i = 0; i < sz && res <= 1; i++) {
        x = f(x, n);
        res = __gcd(abs(x - y), n);
      }
      y = x;
    }
    if (res != 0 && res != n) return res;
  }
}
```

### 4.1.10. Tonelli-Shanks Algorithm

Requires: Mod Struct

```
int legendre(Mod a) {
  if (a == 0) return 0;
  return (a ^ ((MOD - 1) / 2)) == 1 ? 1 : -1;
}
Mod sqrt(Mod a) {
  assert(legendre(a) != -1); // no solution
  ll p = MOD, s = p - 1;
  if (a == 0) return 0;
  if (p == 2) return 1;
  if (p % 4 == 3) return a ^ ((p + 1) / 4);
  int r, m;
  for (r = 0; !(s & 1); r++) s >>= 1;
  Mod n = 2;
  while (legendre(n) != -1) n += 1;
  Mod x = a ^ ((s + 1) / 2), b = a ^ s, g = n ^ s;
  while (b != 1) {
    Mod t = b;
    for (m = 0; t != 1; m++) t *= t;
    Mod gs = g ^ (1LL << (r - m - 1));
    g = gs * gs, x *= gs, b *= g, r = m;
  }
  return x;
}
// to get sqrt(X) modulo p^k, where p is an odd prime:
// c = x^2 (mod p), c = X^2 (mod p^k), q = p^(k-1)
// X = x^q * c^((p^k-2q+1)/2) (mod p^k)
```

### 4.1.11. Chinese Sieve

```
const ll N = 1000000;
// f, g, h multiplicative, h = f (dirichlet convolution) g
ll pre_g(ll n);
ll pre_h(ll n);
// preprocessed prefix sum of f
ll pre_f[N];
// prefix sum of multiplicative function f
ll solve_f(ll n) {
  static unordered_map<ll, ll> m;
  if (n < N) return pre_f[n];
  if (m.count(n)) return m[n];
  ll ans = pre_h(n);
  for (ll l = 2, r; l <= n; l = r + 1) {
    r = n / (n / l);
    ans -= (pre_g(r) - pre_g(l - 1)) * djs_f(n / l);
  }
  return m[n] = ans;
}
```

### 4.1.12. Rational Number Binary Search

```
struct QQ {
  ll p, q;
  QQ go(QQ b, ll d) { return {p + b.p * d, q + b.q * d}; }
};
bool pred(QQ);
// returns smallest p/q in [lo, hi] such that
// pred(p/q) is true, and 0 <= p,q <= N
QQ frac_bs(ll N) {
  QQ lo{0, 1}, hi{1, 0};
  if (pred(lo)) return lo;
  assert(pred(hi));
  bool dir = 1, L = 1, H = 1;
  for (; L || H; dir = !dir) {
    ll len = 0, step = 1;
    for (int t = 0; t < 2 && (t ? step /= 2 : step *= 2);)
      if (QQ mid = hi.go(lo, len + step);
        mid.p > N || mid.q > N || dir ^ pred(mid))
        t++;
      else
        len += step;
    swap(lo, hi = hi.go(lo, len));
    (dir ? L : H) = !!len;
  }
  return dir ? hi : lo;
}
```

### 4.1.13. Farey Sequence

```
// returns (e/f), where (a/b, c/d, e/f) are
// three consecutive terms in the order n farey sequence
// to start, call next_farey(n, 0, 1, 1, n)
pll next_farey(ll n, ll a, ll b, ll c, ll d) {
  ll p = (n + b) / d;
  return pll(p * c - a, p * d - b);
}
```

## 4.2. Combinatorics

### 4.2.1. Matroid Intersection

This template assumes 2 weighted matroids of the same type, and that removing an element is much more expensive than checking if one can be added. **Remember to change the implementation details.**

The ground set is $0, 1, \ldots, n-1$, where element $i$ has weight $w[i]$. For the unweighted version, remove weights and change BF/SPFA to BFS.

```
constexpr int N = 100;
constexpr int INF = 1e9;

struct Matroid {       // represents an independent set
  Matroid(bitset<N>);  // initialize from an independent set
  bool can_add(int);   // if adding will break independence
  Matroid remove(int); // removing from the set
};

auto matroid_intersection(int n, const vector<int> &w) {
  bitset<N> S;
  for (int sz = 1; sz <= n; sz++) {
    Matroid M1(S), M2(S);

    vector<vector<pii>> e(n + 2);
    for (int j = 0; j < n; j++)
      if (!S[j]) {
        if (M1.can_add(j)) e[n].emplace_back(j, -w[j]);
        if (M2.can_add(j)) e[j].emplace_back(n + 1, 0);
      }
    for (int i = 0; i < n; i++)
      if (S[i]) {
        Matroid T1 = M1.remove(i), T2 = M2.remove(i);
        for (int j = 0; j < n; j++)
          if (!S[j]) {
            if (T1.can_add(j)) e[i].emplace_back(j, -w[j]);
            if (T2.can_add(j)) e[j].emplace_back(i, w[i]);
          }
      }

    vector<pii> dis(n + 2, {INF, 0});
    vector<int> prev(n + 2, -1);
    dis[n] = {0, 0};
    // change to SPFA for more speed, if necessary
    bool upd = 1;
    while (upd) {
      upd = 0;
      for (int u = 0; u < n + 2; u++)
        for (auto [v, c] : e[u]) {
          pii x(dis[u].first + c, dis[u].second + 1);
          if (x < dis[v]) dis[v] = x, prev[v] = u, upd = 1;
```

```cpp
        }
    }

    if (dis[n + 1].first < INF)
      for (int x = prev[n + 1]; x != n; x = prev[x])
        S.flip(x);
    else
      break;

    // S is the max-weighted independent set with size sz
  }
  return S;
}
```

### 4.2.2. De Brujin Sequence

```cpp
int res[kN], aux[kN], a[kN], sz;
void Rec(int t, int p, int n, int k) {
  if (t > n) {
    if (n % p == 0)
      for (int i = 1; i <= p; ++i) res[sz++] = aux[i];
  } else {
    aux[t] = aux[t - p];
    Rec(t + 1, p, n, k);
    for (aux[t] = aux[t - p] + 1; aux[t] < k; ++aux[t])
      Rec(t + 1, t, n, k);
  }
}
int DeBruijn(int k, int n) {
  // return cyclic string of length k^n such that every
  // string of length n using k character appears as a
  // substring.
  if (k == 1) return res[0] = 0, 1;
  fill(aux, aux + k * n, 0);
  return sz = 0, Rec(1, 1, n, k), sz;
}
```

### 4.2.3. Multinomial

```cpp
// ways to permute v[i]
ll multinomial(vi &v) {
  ll c = 1, m = v.empty() ? 1 : v[0];
  for (int i = 1; i < v.size(); i++)
    for (int j = 0; i < v[i]; j++) c = c * ++m / (j + 1);
  return c;
}
```

## 4.3. Algebra

### 4.3.1. Formal Power Series

```cpp
template <typename mint>
struct FormalPowerSeries : vector<mint> {
  using vector<mint>::vector;
  using FPS = FormalPowerSeries;

  FPS &operator+=(const FPS &r) {
    if (r.size() > this->size()) this->resize(r.size());
    for (int i = 0; i < (int)r.size(); i++)
      (*this)[i] += r[i];
    return *this;
  }

  FPS &operator+=(const mint &r) {
    if (this->empty()) this->resize(1);
    (*this)[0] += r;
    return *this;
  }

  FPS &operator-=(const FPS &r) {
    if (r.size() > this->size()) this->resize(r.size());
    for (int i = 0; i < (int)r.size(); i++)
      (*this)[i] -= r[i];
    return *this;
  }

  FPS &operator-=(const mint &r) {
    if (this->empty()) this->resize(1);
    (*this)[0] -= r;
    return *this;
  }

  FPS &operator*=(const mint &v) {
    for (int k = 0; k < (int)this->size(); k++)
      (*this)[k] *= v;
    return *this;
  }

  FPS &operator/=(const FPS &r) {
    if (this->size() < r.size()) {
      this->clear();
      return *this;
    }
    int n = this->size() - r.size() + 1;
    if ((int)r.size() <= 64) {
      FPS f(*this), g(r);
      g.shrink();
      mint coeff = g.back().inverse();
      for (auto &x : g) x *= coeff;
      int deg = (int)f.size() - (int)g.size() + 1;
      int gs = g.size();
      FPS quo(deg);
      for (int i = deg - 1; i >= 0; i--) {
        quo[i] = f[i + gs - 1];
        for (int j = 0; j < gs; j++)
          f[i + j] -= quo[i] * g[j];
      }
      *this = quo * coeff;
      this->resize(n, mint(0));
      return *this;
    }
    return *this = ((*this).rev().pre(n) * r.rev().inv(n))
                       .pre(n)
                       .rev();
  }

  FPS &operator%=(const FPS &r) {
    *this -= *this / r * r;
    shrink();
    return *this;
  }

  FPS operator+(const FPS &r) const {
    return FPS(*this) += r;
  }
  FPS operator+(const mint &v) const {
    return FPS(*this) += v;
  }
  FPS operator-(const FPS &r) const {
    return FPS(*this) -= r;
  }
  FPS operator-(const mint &v) const {
    return FPS(*this) -= v;
  }
  FPS operator*(const FPS &r) const {
    return FPS(*this) *= r;
  }
  FPS operator*(const mint &v) const {
    return FPS(*this) *= v;
  }
  FPS operator/(const FPS &r) const {
    return FPS(*this) /= r;
  }
  FPS operator%(const FPS &r) const {
    return FPS(*this) %= r;
  }
  FPS operator-() const {
    FPS ret(this->size());
    for (int i = 0; i < (int)this->size(); i++)
      ret[i] = -(*this)[i];
    return ret;
  }

  void shrink() {
    while (this->size() && this->back() == mint(0))
      this->pop_back();
  }

  FPS rev() const {
    FPS ret(*this);
    reverse(begin(ret), end(ret));
    return ret;
  }

  FPS dot(FPS r) const {
    FPS ret(min(this->size(), r.size()));
    for (int i = 0; i < (int)ret.size(); i++)
      ret[i] = (*this)[i] * r[i];
    return ret;
  }

  FPS pre(int sz) const {
    return FPS(begin(*this),
               begin(*this) + min((int)this->size(), sz));
  }

  FPS operator>>(int sz) const {
    if ((int)this->size() <= sz) return {};
    FPS ret(*this);
    ret.erase(ret.begin(), ret.begin() + sz);
```

```cpp
    return ret;
  }

  FPS operator<<(int sz) const {
    FPS ret(*this);
    ret.insert(ret.begin(), sz, mint(0));
    return ret;
  }

  FPS diff() const {
    const int n = (int)this->size();
    FPS ret(max(0, n - 1));
    mint one(1), coeff(1);
    for (int i = 1; i < n; i++) {
      ret[i - 1] = (*this)[i] * coeff;
      coeff += one;
    }
    return ret;
  }

  FPS integral() const {
    const int n = (int)this->size();
    FPS ret(n + 1);
    ret[0] = mint(0);
    if (n > 0) ret[1] = mint(1);
    auto mod = mint::get_mod();
    for (int i = 2; i <= n; i++)
      ret[i] = (-ret[mod % i]) * (mod / i);
    for (int i = 0; i < n; i++) ret[i + 1] *= (*this)[i];
    return ret;
  }

  mint eval(mint x) const {
    mint r = 0, w = 1;
    for (auto &v : *this) r += w * v, w *= x;
    return r;
  }

  FPS log(int deg = -1) const {
    assert((*this)[0] == mint(1));
    if (deg == -1) deg = (int)this->size();
    return (this->diff() * this->inv(deg))
    .pre(deg - 1)
    .integral();
  }

  FPS pow(int64_t k, int deg = -1) const {
    const int n = (int)this->size();
    if (deg == -1) deg = n;
    for (int i = 0; i < n; i++) {
      if ((*this)[i] != mint(0)) {
        if (i * k > deg) return FPS(deg, mint(0));
        mint rev = mint(1) / (*this)[i];
        FPS ret =
        (((*this * rev) >> i).log(deg) * k).exp(deg) *
        ((*this)[i].pow(k));
        ret = (ret << (i * k)).pre(deg);
        if ((int)ret.size() < deg) ret.resize(deg, mint(0));
        return ret;
      }
    }
    return FPS(deg, mint(0));
  }

  static void *ntt_ptr;
  static void set_fft();
  FPS &operator*=(const FPS &r);
  void ntt();
  void intt();
  void ntt_doubling();
  static int ntt_pr();
  FPS inv(int deg = -1) const;
  FPS exp(int deg = -1) const;
};
template <typename mint>
void *FormalPowerSeries<mint>::ntt_ptr = nullptr;
```

### 4.4. Theorems

#### 4.4.1. Kirchhoff's Theorem

Denote $L$ be a $n \times n$ matrix as the Laplacian matrix of graph $G$, where $L_{ii} = d(i)$, $L_{ij} = -c$ where $c$ is the number of edge $(i, j)$ in $G$.

- The number of undirected spanning in $G$ is $|\det(\tilde{L}_{11})|$.
- The number of directed spanning tree rooted at $r$ in $G$ is $|\det(\tilde{L}_{rr})|$.

#### 4.4.2. Tutte's Matrix

Let $D$ be a $n \times n$ matrix, where $d_{ij} = x_{ij}$ ($x_{ij}$ is chosen uniformly at random) if $i < j$ and $(i, j) \in E$, otherwise $d_{ij} = -d_{ji}$. $\frac{rank(D)}{2}$ is the maximum matching on $G$.

#### 4.4.3. Cayley's Formula

- Given a degree sequence $d_1, d_2, \ldots, d_n$ for each *labeled* vertices, there are
$$\frac{(n-2)!}{(d_1 - 1)!(d_2 - 1)! \cdots (d_n - 1)!}$$
spanning trees.
- Let $T_{n,k}$ be the number of *labeled* forests on $n$ vertices with $k$ components, such that vertex $1, 2, \ldots, k$ belong to different components. Then $T_{n,k} = kn^{n-k-1}$.

#### 4.4.4. Erdős–Gallai Theorem

A sequence of non-negative integers $d_1 \geq d_2 \geq \ldots \geq d_n$ can be represented as the degree sequence of a finite simple graph on $n$ vertices if and only if $d_1 + d_2 + \ldots + d_n$ is even and

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k)$$

holds for all $1 \leq k \leq n$.

#### 4.4.5. Burnside's Lemma

Let $X$ be a set and $G$ be a group that acts on $X$. For $g \in G$, denote by $X^g$ the elements fixed by $g$:

$$X^g = \{x \in X \mid gx \in X\}$$

Then

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|.$$

## 5. Randomisation

### 5.1. Simulated Annealing

```cpp
double get_rand_double() {
  return double(
         uniform_int_distribution<int>(0, INF)(rng)) /
         INF;
}

int main() {
  const int iterations = 10000000;
  double mpl = pow((1e-5) / temp, (double)1 / iterations);
  for (int i = 0; i < iterations; ++i) {
    t *= mpl;
    if (new_score >= score ||
        get_rand_double() <
        exp((new_score - score) / temp)) {
      score = new_score;
      return;
    }
  }
}
```

## 6. Numeric

### 6.1. Barrett Reduction

```cpp
using ull = unsigned long long;
using uL = __uint128_t;
// very fast calculation of a % m
struct reduction {
  const ull m, d;
  explicit reduction(ull m) : m(m), d(((uL)1 << 64) / m) {}
  inline ull operator()(ull a) const {
    ull q = (ull)(((uL)d * a) >> 64);
    return (a -= q * m) >= m ? a - m : a;
  }
};
```

### 6.2. Long Long Multiplication

```cpp
using ull = unsigned long long;
using ll = long long;
using ld = long double;
// returns a * b % M where a, b < M < 2**63
ull mult(ull a, ull b, ull M) {
  ll ret = a * b - M * ull(ld(a) * ld(b) / ld(M));
  return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
```

## 6.3. Fast Fourier Transform

```
template <typename T>
void fft_(int n, vector<T> &a, vector<T> &rt, bool inv) {
  vector<int> br(n);
  for (int i = 1; i < n; i++) {
    br[i] = (i & 1) ? br[i - 1] + n / 2 : br[i / 2] / 2;
    if (br[i] > i) swap(a[i], a[br[i]]);
  }
  for (int len = 2; len <= n; len *= 2)
    for (int i = 0; i < n; i += len)
      for (int j = 0; j < len / 2; j++) {
        int pos = n / len * (inv ? len - j : j);
        T u = a[i + j], v = a[i + j + len / 2] * rt[pos];
        a[i + j] = u + v, a[i + j + len / 2] = u - v;
      }
  if (T minv = T(1) / T(n); inv)
    for (T &x : a) x *= minv;
}
```

Requires: Mod Struct

```
void ntt(vector<Mod> &a, bool inv, Mod primitive_root) {
  int n = a.size();
  Mod root = primitive_root ^ (MOD - 1) / n;
  vector<Mod> rt(n + 1, 1);
  for (int i = 0; i < n; i++) rt[i + 1] = rt[i] * root;
  fft_(n, a, rt, inv);
}
void fft(vector<complex<double>> &a, bool inv) {
  int n = a.size();
  vector<complex<double>> rt(n + 1);
  double arg = acos(-1) * 2 / n;
  for (int i = 0; i <= n; i++)
    rt[i] = {cos(arg * i), sin(arg * i)};
  fft_(n, a, rt, inv);
}
```

## 6.4. Fast Walsh-Hadamard Transform

Requires: Mod Struct

```
void fwht(vector<Mod> &a, bool inv) {
  int n = a.size();
  for (int d = 1; d < n; d <<= 1)
    for (int m = 0; m < n; m++)
      if (!(m & d)) {
        inv ? a[m] -= a[m | d] : a[m] += a[m | d]; // AND
        inv ? a[m | d] -= a[m] : a[m | d] += a[m]; // OR
        Mod x = a[m], y = a[m | d];                // XOR
        a[m] = x + y, a[m | d] = x - y;            // XOR
      }
  if (Mod iv = Mod(1) / n; inv) // XOR
    for (Mod &i : a) i *= iv;   // XOR
}
```

## 6.5. Subset Convolution

Requires: Mod Struct

```
#pragma GCC target("popcnt")
#include <immintrin.h>

void fwht(int n, vector<vector<Mod>> &a, bool inv) {
  for (int h = 0; h < n; h++)
    for (int i = 0; i < (1 << n); i++)
      if (!(i & (1 << h)))
        for (int k = 0; k <= n; k++)
          inv ? a[i | (1 << h)][k] -= a[i][k]
              : a[i | (1 << h)][k] += a[i][k];
}
// c[k] = sum(popcnt(i & j) == sz && i | j == k) a[i] * b[j]
vector<Mod> subset_convolution(int n, int sz,
                               const vector<Mod> &a_,
                               const vector<Mod> &b_) {
  int len = n + sz + 1, N = 1 << n;
  vector<vector<Mod>> a(1 << n, vector<Mod>(len, 0)), b = a;
  for (int i = 0; i < N; i++)
    a[i][_mm_popcnt_u64(i)] = a_[i],
    b[i][_mm_popcnt_u64(i)] = b_[i];
  fwht(n, a, 0), fwht(n, b, 0);
  for (int i = 0; i < N; i++) {
    vector<Mod> tmp(len);
    for (int j = 0; j < len; j++)
      for (int k = 0; k <= j; k++)
        tmp[j] += a[i][k] * b[i][j - k];
    a[i] = tmp;
  }
  fwht(n, a, 1);
  vector<Mod> c(N);
  for (int i = 0; i < N; i++)
    c[i] = a[i][_mm_popcnt_u64(i) + sz];
  return c;
}
```

## 6.6. Linear Recurrences

### 6.6.1. Berlekamp-Massey Algorithm

```
template <typename T>
vector<T> berlekamp_massey(const vector<T> &s) {
  int n = s.size(), l = 0, m = 1;
  vector<T> r(n), p(n);
  r[0] = p[0] = 1;
  T b = 1, d = 0;
  for (int i = 0; i < n; i++, m++, d = 0) {
    for (int j = 0; j <= l; j++) d += r[j] * s[i - j];
    if ((d /= b) == 0) continue; // change if T is float
    auto t = r;
    for (int j = m; j < n; j++) r[j] -= d * p[j - m];
    if (l * 2 <= i) l = i + 1 - l, b *= d, m = 0, p = t;
  }
  return r.resize(l + 1), reverse(r.begin(), r.end()), r;
}
```

### 6.6.2. Linear Recurrence Calculation

```
template <typename T> struct lin_rec {
  using poly = vector<T>;
  poly mul(poly a, poly b, poly m) {
    int n = m.size();
    poly r(n);
    for (int i = n - 1; i >= 0; i--) {
      r.insert(r.begin(), 0), r.pop_back();
      T c = r[n - 1] + a[n - 1] * b[i];
      // c /= m[n - 1]; if m is not monic
      for (int j = 0; j < n; j++)
        r[j] += a[j] * b[i] - c * m[j];
    }
    return r;
  }
  poly pow(poly p, ll k, poly m) {
    poly r(m.size());
    r[0] = 1;
    for (; k; k >>= 1, p = mul(p, p, m))
      if (k & 1) r = mul(r, p, m);
    return r;
  }
  T calc(poly t, poly r, ll k) {
    int n = r.size();
    poly p(n);
    p[1] = 1;
    poly q = pow(p, k, r);
    T ans = 0;
    for (int i = 0; i < n; i++) ans += t[i] * q[i];
    return ans;
  }
};
```

## 6.7. Matrices

### 6.7.1. Determinant

Requires: Mod Struct

```
Mod det(vector<vector<Mod>> a) {
  int n = a.size();
  Mod ans = 1;
  for (int i = 0; i < n; i++) {
    int b = i;
    for (int j = i + 1; j < n; j++)
      if (a[j][i] != 0) {
        b = j;
        break;
      }
    if (i != b) swap(a[i], a[b]), ans = -ans;
    ans *= a[i][i];
    if (ans == 0) return 0;
    for (int j = i + 1; j < n; j++) {
      Mod v = a[j][i] / a[i][i];
      if (v != 0)
        for (int k = i + 1; k < n; k++)
          a[j][k] -= v * a[i][k];
    }
  }
  return ans;
}
```

```
double det(vector<vector<double>> a) {
  int n = a.size();
  double ans = 1;
  for (int i = 0; i < n; i++) {
    int b = i;
    for (int j = i + 1; j < n; j++)
      if (fabs(a[j][i]) > fabs(a[b][i])) b = j;
```

```cpp
    if (i != b) swap(a[i], a[b]), ans = -ans;
    ans *= a[i][i];
    if (ans == 0) return 0;
    for (int j = i + 1; j < n; j++) {
      double v = a[j][i] / a[i][i];
      if (v != 0)
        for (int k = i + 1; k < n; k++)
          a[j][k] -= v * a[i][k];
    }
  }
  return ans;
}
```

### 6.7.2. Inverse

```cpp
// Returns rank.
// Result is stored in A unless singular (rank < n).
// For prime powers, repeatedly set
// A^{-1} = A^{-1} (2I - A*A^{-1}) (mod p^k)
// where A^{-1} starts as the inverse of A mod p,
// and k is doubled in each step.

int matInv(vector<vector<double>> &A) {
  int n = sz(A);
  vi col(n);
  vector<vector<double>> tmp(n, vector<double>(n));
  rep(i, 0, n) tmp[i][i] = 1, col[i] = i;

  rep(i, 0, n) {
    int r = i, c = i;
    rep(j, i, n)
    rep(k, i, n) if (fabs(A[j][k]) > fabs(A[r][c])) r = j,
                                                    c = k;
    if (fabs(A[r][c]) < 1e-12) return i;
    A[i].swap(A[r]);
    tmp[i].swap(tmp[r]);
    rep(j, 0, n) swap(A[j][i], A[j][c]),
    swap(tmp[j][i], tmp[j][c]);
    swap(col[i], col[c]);
    double v = A[i][i];
    rep(j, i + 1, n) {
      double f = A[j][i] / v;
      A[j][i] = 0;
      rep(k, i + 1, n) A[j][k] -= f * A[i][k];
      rep(k, 0, n) tmp[j][k] -= f * tmp[i][k];
    }
    rep(j, i + 1, n) A[i][j] /= v;
    rep(j, 0, n) tmp[i][j] /= v;
    A[i][i] = 1;
  }


  for (int i = n - 1; i > 0; --i) rep(j, 0, i) {
      double v = A[j][i];
      rep(k, 0, n) tmp[j][k] -= v * tmp[i][k];
    }

  rep(i, 0, n) rep(j, 0, n) A[col[i]][col[j]] = tmp[i][j];
  return n;
}

int matInv_mod(vector<vector<ll>> &A) {
  int n = sz(A);
  vi col(n);
  vector<vector<ll>> tmp(n, vector<ll>(n));
  rep(i, 0, n) tmp[i][i] = 1, col[i] = i;

  rep(i, 0, n) {
    int r = i, c = i;
    rep(j, i, n) rep(k, i, n) if (A[j][k]) {
      r = j;
      c = k;
      goto found;
    }
    return i;
  found:
    A[i].swap(A[r]);
    tmp[i].swap(tmp[r]);
    rep(j, 0, n) swap(A[j][i], A[j][c]),
    swap(tmp[j][i], tmp[j][c]);
    swap(col[i], col[c]);
    ll v = modpow(A[i][i], mod - 2);
    rep(j, i + 1, n) {
      ll f = A[j][i] * v % mod;
      A[j][i] = 0;
      rep(k, i + 1, n) A[j][k] =
      (A[j][k] - f * A[i][k]) % mod;
      rep(k, 0, n) tmp[j][k] =
      (tmp[j][k] - f * tmp[i][k]) % mod;
    }
    rep(j, i + 1, n) A[i][j] = A[i][j] * v % mod;
    rep(j, 0, n) tmp[i][j] = tmp[i][j] * v % mod;
    A[i][i] = 1;
  }

  for (int i = n - 1; i > 0; --i) rep(j, 0, i) {
      ll v = A[j][i];
      rep(k, 0, n) tmp[j][k] =
      (tmp[j][k] - v * tmp[i][k]) % mod;
    }

  rep(i, 0, n) rep(j, 0, n) A[col[i]][col[j]] =
  tmp[i][j] % mod + (tmp[i][j] < 0 ? mod : 0);
  return n;
}
```

### 6.7.3. Characteristic Polynomial

```cpp
// calculate det(a - xI)
template <typename T>
vector<T> CharacteristicPolynomial(vector<vector<T>> a) {
  int N = a.size();

  for (int j = 0; j < N - 2; j++) {
    for (int i = j + 1; i < N; i++) {
      if (a[i][j] != 0) {
        swap(a[j + 1], a[i]);
        for (int k = 0; k < N; k++)
          swap(a[k][j + 1], a[k][i]);
        break;
      }
    }
    if (a[j + 1][j] != 0) {
      T inv = T(1) / a[j + 1][j];
      for (int i = j + 2; i < N; i++) {
        if (a[i][j] == 0) continue;
        T coe = inv * a[i][j];
        for (int l = j; l < N; l++)
          a[i][l] -= coe * a[j + 1][l];
        for (int k = 0; k < N; k++)
          a[k][j + 1] += coe * a[k][i];
      }
    }
  }

  vector<vector<T>> p(N + 1);
  p[0] = {T(1)};
  for (int i = 1; i <= N; i++) {
    p[i].resize(i + 1);
    for (int j = 0; j < i; j++) {
      p[i][j + 1] -= p[i - 1][j];
      p[i][j] += p[i - 1][j] * a[i - 1][i - 1];
    }
    T x = 1;
    for (int m = 1; m < i; m++) {
      x *= -a[i - m][i - m - 1];
      T coe = x * a[i - m - 1][i - 1];
      for (int j = 0; j < i - m; j++)
        p[i][j] += coe * p[i - m - 1][j];
    }
  }
  return p[N];
}
```

### 6.7.4. Solve Linear Equation

```cpp
typedef vector<double> vd;
const double eps = 1e-12;

// solves for x: A * x = b
int solveLinear(vector<vd> &A, vd &b, vd &x) {
  int n = sz(A), m = sz(x), rank = 0, br, bc;
  if (n) assert(sz(A[0]) == m);
  vi col(m);
  iota(all(col), 0);

  rep(i, 0, n) {
    double v, bv = 0;
    rep(r, i, n) rep(c, i, m) if ((v = fabs(A[r][c])) > bv)
    br = r,
    bc = c, bv = v;
    if (bv <= eps) {
      rep(j, i, n) if (fabs(b[j]) > eps) return -1;
      break;
    }
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    rep(j, 0, n) swap(A[j][i], A[j][bc]);
    bv = 1 / A[i][i];
```

```
    rep(j, i + 1, n) {
      double fac = A[j][i] * bv;
      b[j] -= fac * b[i];
      rep(k, i + 1, m) A[j][k] -= fac * A[i][k];
    }
    rank++;
  }

  x.assign(m, 0);
  for (int i = rank; i--;) {
    b[i] /= A[i][i];
    x[col[i]] = b[i];
    rep(j, 0, i) b[j] -= A[j][i] * b[i];
  }
  return rank; // (multiple solutions if rank < m)
}
```

## 6.8. Polynomial Interpolation

```
// returns a, such that a[0]x^0 + a[1]x^1 + a[2]x^2 + ...
// passes through the given points
typedef vector<double> vd;
vd interpolate(vd x, vd y, int n) {
  vd res(n), temp(n);
  rep(k, 0, n - 1) rep(i, k + 1, n) y[i] =
  (y[i] - y[k]) / (x[i] - x[k]);
  double last = 0;
  temp[0] = 1;
  rep(k, 0, n) rep(i, 0, n) {
    res[i] += y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] -= last * x[k];
  }
  return res;
}
```

## 6.9. 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 long double ld;
typedef vector<ld> vd;
typedef vector<vd> vvd;
typedef vector<int> vi;

const ld 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);
    }
  }

  ld 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<ld>::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<ld>::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];
  }
};

int main() {

  const int m = 4;
  const int n = 3;
  ld _A[m][n] = {
  {6, -1, 0}, {-1, -5, 0}, {1, 5, 1}, {-1, -5, -1}};
  ld _b[m] = {10, -4, 5, -5};
  ld _c[n] = {1, -1, 0};

  vvd A(m);
  vd b(_b, _b + m);
  vd c(_c, _c + n);
  for (int i = 0; i < m; i++) A[i] = vd(_A[i], _A[i] + n);

  LPSolver solver(A, b, c);
  vd x;
  ld value = solver.Solve(x);

  cerr << "VALUE: " << value << endl; // VALUE: 1.29032
  cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
  cerr << endl;
  return 0;
}
```

# 7. Geometry

## 7.1. Point

```
template <typename T> struct P {
  T x, y;
  P(T x = 0, T y = 0) : x(x), y(y) {}
  bool operator<(const P &p) const {
    return tie(x, y) < tie(p.x, p.y);
  }
  bool operator==(const P &p) const {
    return tie(x, y) == tie(p.x, p.y);
  }
  P operator-() const { return {-x, -y}; }
  P operator+(P p) const { return {x + p.x, y + p.y}; }
  P operator-(P p) const { return {x - p.x, y - p.y}; }
  P operator*(T d) const { return {x * d, y * d}; }
  P operator/(T d) const { return {x / d, y / d}; }
  T dist2() const { return x * x + y * y; }
  double len() const { return sqrt(dist2()); }
  P unit() const { return *this / len(); }
  friend T dot(P a, P b) { return a.x * b.x + a.y * b.y; }
  friend T cross(P a, P b) { return a.x * b.y - a.y * b.x; }
  friend T cross(P a, P b, P o) {
    return cross(a - o, b - o);
  }
};
using pt = P<ll>;
```

### 7.1.1. Quarternion

```
constexpr double PI = 3.141592653589793;
constexpr double EPS = 1e-7;
struct Q {
  using T = double;
  T x, y, z, r;
  Q(T r = 0) : x(0), y(0), z(0), r(r) {}
  Q(T x, T y, T z, T r = 0) : x(x), y(y), z(z), r(r) {}
  friend bool operator==(const Q &a, const Q &b) {
    return (a - b).abs2() <= EPS;
  }
  friend bool operator!=(const Q &a, const Q &b) {
    return !(a == b);
  }
  Q operator-() { return Q(-x, -y, -z, -r); }
  Q operator+(const Q &b) const {
    return Q(x + b.x, y + b.y, z + b.z, r + b.r);
  }
  Q operator-(const Q &b) const {
    return Q(x - b.x, y - b.y, z - b.z, r - b.r);
  }
  Q operator*(const T &t) const {
    return Q(x * t, y * t, z * t, r * t);
  }
  Q operator*(const Q &b) const {
    return Q(r * b.x + x * b.r + y * b.z - z * b.y,
             r * b.y - x * b.z + y * b.r + z * b.x,
             r * b.z + x * b.y - y * b.x + z * b.r,
             r * b.r - x * b.x - y * b.y - z * b.z);
  }
  Q operator/(const Q &b) const { return *this * b.inv(); }
  T abs2() const { return r * r + x * x + y * y + z * z; }
  T len() const { return sqrt(abs2()); }
  Q conj() const { return Q(-x, -y, -z, r); }
  Q unit() const { return *this * (1.0 / len()); }
  Q inv() const { return conj() * (1.0 / abs2()); }
  friend T dot(Q a, Q b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
  }
  friend Q cross(Q a, Q b) {
    return Q(a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z,
             a.x * b.y - a.y * b.x);
  }
  friend Q rotation_around(Q axis, T angle) {
    return axis.unit() * sin(angle / 2) + cos(angle / 2);
  }
  Q rotated_around(Q axis, T angle) {
    Q u = rotation_around(axis, angle);
    return u * *this / u;
  }
  friend Q rotation_between(Q a, Q b) {
    a = a.unit(), b = b.unit();
    if (a == -b) {
      // degenerate case
      Q ortho = abs(a.y) > EPS ? cross(a, Q(1, 0, 0))
                               : cross(a, Q(0, 1, 0));
      return rotation_around(ortho, PI);
    }
    return (a * (a + b)).conj();
  }
};
```

### 7.1.2. Spherical Coordinates

```
struct car_p {
  double x, y, z;
};
struct sph_p {
  double r, theta, phi;
};

sph_p conv(car_p p) {
  double r = sqrt(p.x * p.x + p.y * p.y + p.z * p.z);
  double theta = asin(p.y / r);
  double phi = atan2(p.y, p.x);
  return {r, theta, phi};
}
car_p conv(sph_p p) {
  double x = p.r * cos(p.theta) * sin(p.phi);
  double y = p.r * cos(p.theta) * cos(p.phi);
  double z = p.r * sin(p.theta);
  return {x, y, z};
}
```

## 7.2. Segments

```
// for non-collinear ABCD, if segments AB and CD intersect
bool intersects(pt a, pt b, pt c, pt d) {
  if (cross(b, c, a) * cross(b, d, a) > 0) return false;
  if (cross(d, a, c) * cross(d, b, c) > 0) return false;
  return true;
}
// the intersection point of lines AB and CD
pt intersect(pt a, pt b, pt c, pt d) {
  auto x = cross(b, c, a), y = cross(b, d, a);
  if (x == y) {
    // if(abs(x, y) < 1e-8) {
    // is parallel
  } else {
    return d * (x / (x - y)) - c * (y / (x - y));
  }
}
```

## 7.3. Convex Hull

```
// returns a convex hull in counterclockwise order
// for a non-strict one, change cross >= to >
vector<pt> convex_hull(vector<pt> p) {
  sort(ALL(p));
  if (p[0] == p.back()) return {p[0]};
  int n = p.size(), t = 0;
  vector<pt> h(n + 1);
  for (int _ = 2, s = 0; _--; s = --t, reverse(ALL(p)))
    for (pt i : p) {
      while (t > s + 1 && cross(i, h[t - 1], h[t - 2]) >= 0)
        t--;
      h[t++] = i;
    }
  return h.resize(t), h;
}
```

### 7.3.1. 3D Hull

```
typedef Point3D<double> P3;

struct PR {
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};

struct F {
  P3 q;
  int a, b, c;
};

vector<F> hull3d(const vector<P3> &A) {
  assert(sz(A) >= 4);
  vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
#define E(x, y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
    if (q.dot(A[l]) > q.dot(A[i])) q = q * -1;
    F f{q, i, j, k};
    E(a, b).ins(k);
    E(a, c).ins(j);
    E(b, c).ins(i);
    FS.push_back(f);
  };
  rep(i, 0, 4) rep(j, i + 1, 4) rep(k, j + 1, 4)
```

```
  mf(i, j, k, 6 - i - j - k);

  rep(i, 4, sz(A)) {
    rep(j, 0, sz(FS)) {
      F f = FS[j];
      if (f.q.dot(A[i]) > f.q.dot(A[f.a])) {
        E(a, b).rem(f.c);
        E(a, c).rem(f.b);
        E(b, c).rem(f.a);
        swap(FS[j--], FS.back());
        FS.pop_back();
      }
    }
    int nw = sz(FS);
    rep(j, 0, nw) {
      F f = FS[j];
#define C(a, b, c)                                      \
  if (E(a, b).cnt() != 2) mf(f.a, f.b, i, f.c);
      C(a, b, c);
      C(a, c, b);
      C(b, c, a);
    }
  }
  for (F &it : FS)
    if ((A[it.b] - A[it.a])
          .cross(A[it.c] - A[it.a])
          .dot(it.q) <= 0)
      swap(it.c, it.b);
  return FS;
};
```

## 7.4. Angular Sort

```
auto angle_cmp = [](const pt &a, const pt &b) {
  auto btm = [](const pt &a) {
    return a.y < 0 || (a.y == 0 && a.x < 0);
  };
  return make_tuple(btm(a), a.y * b.x, abs2(a)) <
         make_tuple(btm(b), a.x * b.y, abs2(b));
};
void angular_sort(vector<pt> &p) {
  sort(p.begin(), p.end(), angle_cmp);
}
```

## 7.5. Convex Polygon Minkowski Sum

```
// O(n) convex polygon minkowski sum
// must be sorted and counterclockwise
vector<pt> minkowski_sum(vector<pt> p, vector<pt> q) {
  auto diff = [](vector<pt> &c) {
    auto rcmp = [](pt a, pt b) {
      return pt{a.y, a.x} < pt{b.y, b.x};
    };
    rotate(c.begin(), min_element(ALL(c), rcmp), c.end());
    c.push_back(c[0]);
    vector<pt> ret;
    for (int i = 1; i < c.size(); i++)
      ret.push_back(c[i] - c[i - 1]);
    return ret;
  };
  auto dp = diff(p), dq = diff(q);
  pt cur = p[0] + q[0];
  vector<pt> d(dp.size() + dq.size()), ret = {cur};
  // include angle_cmp from angular-sort.cpp
  merge(ALL(dp), ALL(dq), d.begin(), angle_cmp);
  // optional: make ret strictly convex (UB if degenerate)
  int now = 0;
  for (int i = 1; i < d.size(); i++) {
    if (cross(d[i], d[now]) == 0)
      d[now] = d[now] + d[i];
    else
      d[++now] = d[i];
  }
  d.resize(now + 1);
  // end optional part
  for (pt v : d) ret.push_back(cur = cur + v);
  return ret.pop_back(), ret;
}
```

## 7.6. Point In Polygon

```
bool on_segment(pt a, pt b, pt p) {
  return cross(a, b, p) == 0 && dot((p - a), (p - b)) <= 0;
}
// p can be any polygon, but this is O(n)
bool inside(const vector<pt> &p, pt a) {
  int cnt = 0, n = p.size();
  for (int i = 0; i < n; i++) {
    pt l = p[i], r = p[(i + 1) % n];
    // change to return 0; for strict version
    if (on_segment(l, r, a)) return 1;
    cnt ^= ((a.y < l.y) - (a.y < r.y)) * cross(l, r, a) > 0;
  }
  return cnt;
}
```

### 7.6.1. Convex Version

```
// no preprocessing version
// p must be a strict convex hull, counterclockwise
// if point is inside or on border
bool is_inside(const vector<pt> &c, pt p) {
  int n = c.size(), l = 1, r = n - 1;
  if (cross(c[0], c[1], p) < 0) return false;
  if (cross(c[n - 1], c[0], p) < 0) return false;
  while (l < r - 1) {
    int m = (l + r) / 2;
    T a = cross(c[0], c[m], p);
    if (a > 0)
      l = m;
    else if (a < 0)
      r = m;
    else
      return dot(c[0] - p, c[m] - p) <= 0;
  }
  if (l == r)
    return dot(c[0] - p, c[l] - p) <= 0;
  else
    return cross(c[l], c[r], p) >= 0;
}

// with preprocessing version
vector<pt> vecs;
pt center;
// p must be a strict convex hull, counterclockwise
// BEWARE OF OVERFLOWS!!
void preprocess(vector<pt> p) {
  for (auto &v : p) v = v * 3;
  center = p[0] + p[1] + p[2];
  center.x /= 3, center.y /= 3;
  for (auto &v : p) v = v - center;
  vecs = (angular_sort(p), p);
}
bool intersect_strict(pt a, pt b, pt c, pt d) {
  if (cross(b, c, a) * cross(b, d, a) > 0) return false;
  if (cross(d, a, c) * cross(d, b, c) >= 0) return false;
  return true;
}
// if point is inside or on border
bool query(pt p) {
  p = p * 3 - center;
  auto pr = upper_bound(ALL(vecs), p, angle_cmp);
  if (pr == vecs.end()) pr = vecs.begin();
  auto pl = (pr == vecs.begin()) ? vecs.back() : *(pr - 1);
  return !intersect_strict({0, 0}, p, pl, *pr);
}
```

### 7.6.2. Offline Multiple Points Version

Requires: Point, GNU PBDS

```
using Double = __float128;
using Point = pt<Double, Double>;

int n, m;
vector<Point> poly;
vector<Point> query;
vector<int> ans;

struct Segment {
  Point a, b;
  int id;
};
vector&lt;Segment&gt; segs;

Double Xnow;
inline Double get_y(const Segment &u, Double xnow = Xnow) {
  const Point &a = u.a;
  const Point &b = u.b;
  return (a.y * (b.x - xnow) + b.y * (xnow - a.x)) /
         (b.x - a.x);
}
bool operator<(Segment u, Segment v) {
  Double yu = get_y(u);
  Double yv = get_y(v);
  if (yu != yv) return yu < yv;
  return u.id < v.id;
}
ordered_map&lt;Segment&gt; st;

struct Event {
```

```
  int type; // +1 insert seg, -1 remove seg, 0 query
  Double x, y;
  int id;
};
bool operator<(Event a, Event b) {
  if (a.x != b.x) return a.x < b.x;
  if (a.type != b.type) return a.type < b.type;
  return a.y < b.y;
}
vector<Event> events;

void solve() {
  set<Double> xs;
  set<Point> ps;
  for (int i = 0; i < n; i++) {
    xs.insert(poly[i].x);
    ps.insert(poly[i]);
  }
  for (int i = 0; i < n; i++) {
    Segment s{poly[i], poly[(i + 1) % n], i};
    if (s.a.x > s.b.x ||
        (s.a.x == s.b.x && s.a.y > s.b.y)) {
      swap(s.a, s.b);
    }
    segs.push_back(s);

    if (s.a.x != s.b.x) {
      events.push_back({+1, s.a.x + 0.2, s.a.y, i});
      events.push_back({-1, s.b.x - 0.2, s.b.y, i});
    }
  }
  for (int i = 0; i < m; i++) {
    events.push_back({0, query[i].x, query[i].y, i});
  }
  sort(events.begin(), events.end());
  int cnt = 0;
  for (Event e : events) {
    int i = e.id;
    Xnow = e.x;
    if (e.type == 0) {
      Double x = e.x;
      Double y = e.y;
      Segment tmp = {{x - 1, y}, {x + 1, y}, -1};
      auto it = st.lower_bound(tmp);

      if (ps.count(query[i]) > 0) {
        ans[i] = 0;
      } else if (xs.count(x) > 0) {
        ans[i] = -2;
      } else if (it != st.end() &&
                 get_y(*it) == get_y(tmp)) {
        ans[i] = 0;
      } else if (it != st.begin() &&
                 get_y(*prev(it)) == get_y(tmp)) {
        ans[i] = 0;
      } else {
        int rk = st.order_of_key(tmp);
        if (rk % 2 == 1) {
          ans[i] = 1;
        } else {
          ans[i] = -1;
        }
      }
    } else if (e.type == 1) {
      st.insert(segs[i]);
      assert((int)st.size() == ++cnt);
    } else if (e.type == -1) {
      st.erase(segs[i]);
      assert((int)st.size() == --cnt);
    }
  }
}
```

## 7.7. Closest Pair

```
vector<pll> p; // sort by x first!
bool cmpy(const pll &a, const pll &b) const {
  return a.y < b.y;
}
ll sq(ll x) { return x * x; }
// returns (minimum dist)^2 in [l, r)
ll solve(int l, int r) {
  if (r - l <= 1) return 1e18;
  int m = (l + r) / 2;
  ll mid = p[m].x, d = min(solve(l, m), solve(m, r));
  auto pb = p.begin();
  inplace_merge(pb + l, pb + m, pb + r, cmpy);
  vector<pll> s;
  for (int i = l; i < r; i++)
    if (sq(p[i].x - mid) < d) s.push_back(p[i]);
  for (int i = 0; i < s.size(); i++)
    for (int j = i + 1;
         j < s.size() && sq(s[j].y - s[i].y) < d; j++)
      d = min(d, dis(s[i], s[j]));
  return d;
}
```

## 7.8. Minimum Enclosing Circle

```
typedef Point<double> P;
double ccRadius(const P &A, const P &B, const P &C) {
  return (B - A).dist() * (C - B).dist() * (A - C).dist() /
         abs((B - A).cross(C - A)) / 2;
}
P ccCenter(const P &A, const P &B, const P &C) {
  P b = C - A, c = B - A;
  return A + (b * c.dist2() - c * b.dist2()).perp() /
             b.cross(c) / 2;
}
pair<P, double> mec(vector<P> ps) {
  shuffle(all(ps), mt19937(time(0)));
  P o = ps[0];
  double r = 0, EPS = 1 + 1e-8;
  rep(i, 0, sz(ps)) if ((o - ps[i]).dist() > r * EPS) {
    o = ps[i], r = 0;
    rep(j, 0, i) if ((o - ps[j]).dist() > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      rep(k, 0, j) if ((o - ps[k]).dist() > r * EPS) {
        o = ccCenter(ps[i], ps[j], ps[k]);
        r = (o - ps[i]).dist();
      }
    }
  }
  return {o, r};
}
```

## 7.9. Delaunay Triangulation

```
typedef Point<ll> P;
typedef struct Quad *Q;
typedef __int128_t lll; // (can be ll if coords are < 2e4)
P arb(LLONG_MAX, LLONG_MAX); // not equal to any other point

struct Quad {
  bool mark;
  Q o, rot;
  P p;
  P F() { return r()->p; }
  Q r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
};

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
  lll p2 = p.dist2(), A = a.dist2() - p2,
      B = b.dist2() - p2, C = c.dist2() - p2;
  return p.cross(a, b) * C + p.cross(b, c) * A +
         p.cross(c, a) * B >
         0;
}
Q makeEdge(P orig, P dest) {
  Q q[] = {new Quad{0, 0, 0, orig}, new Quad{0, 0, 0, arb},
           new Quad{0, 0, 0, dest}, new Quad{0, 0, 0, arb}};
  rep(i, 0, 4) q[i]->o = q[-i & 3],
               q[i]->rot = q[(i + 1) & 3];
  return *q;
}
void splice(Q a, Q b) {
  swap(a->o->rot->o, b->o->rot->o);
  swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}

pair<Q, Q> rec(const vector<P> &s) {
  if (sz(s) <= 3) {
    Q a = makeEdge(s[0], s[1]),
      b = makeEdge(s[1], s.back());
    if (sz(s) == 2) return {a, a->r()};
    splice(a->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a, side < 0 ? c : b->r()};
  }
```

```
#define H(e)     e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = sz(s) / 2;
  tie(ra, A) = rec({all(s) - half});
  tie(B, rb) = rec({sz(s) - half + all(s)});
  while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
         (A->p.cross(H(B)) > 0 && (B = B->r()->o)))
    ;
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;

#define DEL(e, init, dir)                              \
  Q e = init->dir;                                     \
  if (valid(e))                                        \
    while (circ(e->dir->F(), H(base), e->F())) {       \
      Q t = e->dir;                                    \
      splice(e, e->prev());                            \
      splice(e->r(), e->r()->prev());                  \
      e = t;                                           \
    }
  for (;;) {
    DEL(LC, base->r(), o);
    DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return {ra, rb};
}

// returns [A_0, B_0, C_0, A_1, B_1, ...]
// where A_i, B_i, C_i are counter-clockwise triangles
vector<P> triangulate(vector<P> pts) {
  sort(all(pts));
  assert(unique(all(pts)) == pts.end());
  if (sz(pts) < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD                                            \
  {                                                    \
    Q c = e;                                           \
    do {                                               \
      c->mark = 1;                                     \
      pts.push_back(c->p);                             \
      q.push_back(c->r());                             \
      c = c->next();                                   \
    } while (c != e);                                  \
  }
  ADD;
  pts.clear();
  while (qi < sz(q))
    if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

### 7.9.1. Slower Version

```
template <class P, class F>
void delaunay(vector<P> &ps, F trifun) {
  if (sz(ps) == 3) {
    int d = (ps[0].cross(ps[1], ps[2]) < 0);
    trifun(0, 1 + d, 2 - d);
  }
  vector<P3> p3;
  for (P p : ps) p3.emplace_back(p.x, p.y, p.dist2());
  if (sz(ps) > 3)
    for (auto t : hull3d(p3))
      if ((p3[t.b] - p3[t.a])
              .cross(p3[t.c] - p3[t.a])
              .dot(P3(0, 0, 1)) < 0)
        trifun(t.a, t.c, t.b);
}
```

## 7.10. Half Plane Intersection

```
struct Line {
  Point P;
  Vector v;
  bool operator<(const Line &b) const {
    return atan2(v.y, v.x) < atan2(b.v.y, b.v.x);
  }
};
bool OnLeft(const Line &L, const Point &p) {
  return Cross(L.v, p - L.P) > 0;
}
Point GetIntersection(Line a, Line b) {
  Vector u = a.P - b.P;
  Double t = Cross(b.v, u) / Cross(a.v, b.v);
  return a.P + a.v * t;
}
int HalfplaneIntersection(Line *L, int n, Point *poly) {
  sort(L, L + n);

  int first, last;
  Point *p = new Point[n];
  Line *q = new Line[n];
  q[first = last = 0] = L[0];
  for (int i = 1; i < n; i++) {
    while (first < last && !OnLeft(L[i], p[last - 1]))
      last--;
    while (first < last && !OnLeft(L[i], p[first])) first++;
    q[++last] = L[i];
    if (fabs(Cross(q[last].v, q[last - 1].v)) < EPS) {
      last--;
      if (OnLeft(q[last], L[i].P)) q[last] = L[i];
    }
    if (first < last)
      p[last - 1] = GetIntersection(q[last - 1], q[last]);
  }
  while (first < last && !OnLeft(q[first], p[last - 1]))
    last--;
  if (last - first <= 1) return 0;
  p[last] = GetIntersection(q[last], q[first]);

  int m = 0;
  for (int i = first; i <= last; i++) poly[m++] = p[i];
  return m;
}
```

# 8. Strings

## 8.1. Knuth-Morris-Pratt Algorithm

```
vector<int> pi(const string &s) {
  vector<int> p(s.size());
  for (int i = 1; i < s.size(); i++) {
    int g = p[i - 1];
    while (g && s[i] != s[g]) g = p[g - 1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}
vector<int> match(const string &s, const string &pat) {
  vector<int> p = pi(pat + '\0' + s), res;
  for (int i = p.size() - s.size(); i < p.size(); i++)
    if (p[i] == pat.size())
      res.push_back(i - 2 * pat.size());
  return res;
}
```

## 8.2. Aho-Corasick Automaton

```
struct Aho_Corasick {
  static const int maxc = 26, maxn = 4e5;
  struct NODES {
    int Next[maxc], fail, ans;
  };
  NODES T[maxn];
  int top, qtop, q[maxn];
  int get_node(const int &fail) {
    fill_n(T[top].Next, maxc, 0);
    T[top].fail = fail;
    T[top].ans = 0;
    return top++;
  }
  int insert(const string &s) {
    int ptr = 1;
    for (char c : s) { // change char id
      c -= 'a';
      if (!T[ptr].Next[c]) T[ptr].Next[c] = get_node(ptr);
      ptr = T[ptr].Next[c];
    }
    return ptr;
  } // return ans_last_place
  void build_fail(int ptr) {
    int tmp;
    for (int i = 0; i < maxc; i++)
      if (T[ptr].Next[i]) {
        tmp = T[ptr].fail;
        while (tmp != 1 && !T[tmp].Next[i])
          tmp = T[tmp].fail;
        if (T[tmp].Next[i] != T[ptr].Next[i])
          if (T[tmp].Next[i]) tmp = T[tmp].Next[i];
```

```
        T[T[ptr].Next[i]].fail = tmp;
        q[qtop++] = T[ptr].Next[i];
      }
  }
  void AC_auto(const string &s) {
    int ptr = 1;
    for (char c : s) {
      while (ptr != 1 && !T[ptr].Next[c]) ptr = T[ptr].fail;
      if (T[ptr].Next[c]) {
        ptr = T[ptr].Next[c];
        T[ptr].ans++;
      }
    }
  }
  void Solve(string &s) {
    for (char &c : s) // change char id
      c -= 'a';
    for (int i = 0; i < qtop; i++) build_fail(q[i]);
    AC_auto(s);
    for (int i = qtop - 1; i > -1; i--)
      T[T[q[i]].fail].ans += T[q[i]].ans;
  }
  void reset() {
    qtop = top = q[0] = 1;
    get_node(1);
  }
} AC;
// usage example
string s, S;
int n, t, ans_place[50000];
int main() {
  Tie cin >> t;
  while (t--) {
    AC.reset();
    cin >> S >> n;
    for (int i = 0; i < n; i++) {
      cin >> s;
      ans_place[i] = AC.insert(s);
    }
    AC.Solve(S);
    for (int i = 0; i < n; i++)
      cout << AC.T[ans_place[i]].ans << '\n';
  }
}
```

## 8.3. Suffix Array

```
// sa[i]: starting index of suffix at rank i
//        0-indexed, sa[0] = n (empty string)
// lcp[i]: lcp of sa[i] and sa[i - 1], lcp[0] = 0
struct SuffixArray {
  vector<int> sa, lcp;
  SuffixArray(string &s,
              int lim = 256) { // or basic_string<int>
    int n = sz(s) + 1, k = 0, a, b;
    vector<int> x(all(s) + 1), y(n), ws(max(n, lim)),
    rank(n);
    sa = lcp = y, iota(all(sa), 0);
    for (int j = 0, p = 0; p < n;
         j = max(1, j * 2), lim = p) {
      p = j, iota(all(y), n - j);
      for (int i = 0; i < n; i++)
        if (sa[i] >= j) y[p++] = sa[i] - j;
      fill(all(ws), 0);
      for (int i = 0; i < n; i++) ws[x[i]]++;
      for (int i = 1; i < lim; i++) ws[i] += ws[i - 1];
      for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
      swap(x, y), p = 1, x[sa[0]] = 0;
      for (int i = 1; i < n; i++)
        a = sa[i - 1], b = sa[i],

        x[b] = (y[a] == y[b] && y[a + j] == y[b + j])
               ? p - 1 : p++;

    }
    for (int i = 1; i < n; i++) rank[sa[i]] = i;
    for (int i = 0, j; i < n - 1; lcp[rank[i++]] = k)
      for (k &&k--, j = sa[rank[i] - 1];
           s[i + k] == s[j + k]; k++)
        ;
  }
};
```

## 8.4. Suffix Tree

```
struct SAM {
  static const int maxc = 26;    // char range
  static const int maxn = 10010; // string len
  struct Node {
    Node *green, *edge[maxc];
    int max_len, in, times;
  } * root, *last, reg[maxn * 2];
  int top;
  Node *get_node(int _max) {
    Node *re = &reg[top++];
    re->in = 0, re->times = 1;
    re->max_len = _max, re->green = 0;
    for (int i = 0; i < maxc; i++) re->edge[i] = 0;
    return re;
  }
  void insert(const char c) { // c in range [0, maxc)
    Node *p = last;
    last = get_node(p->max_len + 1);
    while (p && !p->edge[c])
      p->edge[c] = last, p = p->green;
    if (!p)
      last->green = root;
    else {
      Node *pot_green = p->edge[c];
      if ((pot_green->max_len) == (p->max_len + 1))
        last->green = pot_green;
      else {
        Node *wish = get_node(p->max_len + 1);
        wish->times = 0;
        while (p && p->edge[c] == pot_green)
          p->edge[c] = wish, p = p->green;
        for (int i = 0; i < maxc; i++)
          wish->edge[i] = pot_green->edge[i];
        wish->green = pot_green->green;
        pot_green->green = wish;
        last->green = wish;
      }
    }
  }
  Node *q[maxn * 2];
  int ql, qr;
  void get_times(Node *p) {
    ql = 0, qr = -1, reg[0].in = 1;
    for (int i = 1; i < top; i++) reg[i].green->in++;
    for (int i = 0; i < top; i++)
      if (!reg[i].in) q[++qr] = &reg[i];
    while (ql <= qr) {
      q[ql]->green->times += q[ql]->times;
      if (!(--q[ql]->green->in)) q[++qr] = q[ql]->green;
      ql++;
    }
  }
  void build(const string &s) {
    top = 0;
    root = last = get_node(0);
    for (char c : s) insert(c - 'a'); // change char id
    get_times(root);
  }
  // call build before solve
  int solve(const string &s) {
    Node *p = root;
    for (char c : s)
      if (!(p = p->edge[c - 'a'])) // change char id
        return 0;
    return p->times;
  }
};
```

## 8.5. Z Value

```
int z[n];
void zval(string s) {
  // z[i] => longest common prefix of s and s[i:], i > 0
  int n = s.size();
  z[0] = 0;
  for (int b = 0, i = 1; i < n; i++) {
    if (z[b] + b <= i)
      z[i] = 0;
    else
      z[i] = min(z[i - b], z[b] + b - i);
    while (s[i + z[i]] == s[z[i]]) z[i]++;
    if (i + z[i] > b + z[b]) b = i;
  }
}
```

## 8.6. Manacher's Algorithm

```
int z[n];
void manacher(string s) {
  // z[i] => longest odd palindrome centered at i is
  //          s[i - z[i] ... i + z[i]]
  // to get all palindromes (including even length),
  // insert a '#' between each s[i] and s[i + 1]
  int n = s.size();
  z[0] = 0;
```

```
  for (int b = 0, i = 1; i < n; i++) {
    if (z[b] + b >= i)
      z[i] = min(z[2 * b - i], b + z[b] - i);
    else
      z[i] = 0;
    while (i + z[i] + 1 < n && i - z[i] - 1 >= 0 &&
           s[i + z[i] + 1] == s[i - z[i] - 1])
      z[i]++;
    if (z[i] + i > z[b] + b) b = i;
  }
}
```

### 8.7. Minimum Rotation

```
int min_rotation(string s) {
  int a = 0, n = s.size();
  s += s;
  for (int b = 0; b < n; b++) {
    for (int k = 0; k < n; k++) {
      if (a + k == b || s[a + k] < s[b + k]) {
        b += max(0, k - 1);
        break;
      }
      if (s[a + k] > s[b + k]) {
        a = b;
        break;
      }
    }
  }
  return a;
}
```

### 8.8. Palindromic Tree

```
struct palindromic_tree {
  struct node {
    int next[26], fail, len;
    int cnt,
    num; // cnt: appear times, num: number of pal. suf.
    node(int l = 0) : fail(0), len(l), cnt(0), num(0) {
      for (int i = 0; i < 26; ++i) next[i] = 0;
    }
  };
  vector<node> St;
  vector<char> s;
  int last, n;
  palindromic_tree() : St(2), last(1), n(0) {
    St[0].fail = 1, St[1].len = -1, s.pb(-1);
  }
  inline void clear() {
    St.clear(), s.clear(), last = 1, n = 0;
    St.pb(0), St.pb(-1);
    St[0].fail = 1, s.pb(-1);
  }
  inline int get_fail(int x) {
    while (s[n - St[x].len - 1] != s[n]) x = St[x].fail;
    return x;
  }
  inline void add(int c) {
    s.push_back(c -= 'a'), ++n;
    int cur = get_fail(last);
    if (!St[cur].next[c]) {
      int now = SZ(St);
      St.pb(St[cur].len + 2);
      St[now].fail = St[get_fail(St[cur].fail)].next[c];
      St[cur].next[c] = now;
      St[now].num = St[St[now].fail].num + 1;
    }
    last = St[cur].next[c], ++St[last].cnt;
  }
  inline void count() { // counting cnt
    auto i = St.rbegin();
    for (; i != St.rend(); ++i) {
      St[i->fail].cnt += i->cnt;
    }
  }
  inline int size() { // The number of diff. pal.
    return SZ(St) - 2;
  }
};
```

## 9. Debug List

```
- Pre-submit:
  - Did you make a typo when copying a template?
  - Test more cases if unsure.
    - Write a naive solution and check small cases.
  - Submit the correct file.

- General Debugging:
  - Read the whole problem again.
  - Have a teammate read the problem.
  - Have a teammate read your code.
    - Explain you solution to them (or a rubber duck).
  - Print the code and its output / debug output.
  - Go to the toilet.

- Wrong Answer:
  - Any possible overflows?
    - > `__int128` ?
    - Try `-ftrapv` or `#pragma GCC optimize("trapv")`
  - Floating point errors?
    - > `long double` ?
    - turn off math optimizations
    - check for `==`, `>=`, `acos(1.000000001)`, etc.
  - Did you forget to sort or unique?
  - Generate large and worst "corner" cases.
  - Check your `m` / `n`, `i` / `j` and `x` / `y`.
  - Are everything initialized or reset properly?
  - Are you sure about the STL thing you are using?
    - Read cppreference (should be available).
  - Print everything and run it on pen and paper.

- Time Limit Exceeded:
  - Calculate your time complexity again.
  - Does the program actually end?
    - Check for `while(q.size())` etc.
  - Test the largest cases locally.
  - Did you do unnecessary stuff?
    - e.g. pass vectors by value
    - e.g. `memset` for every test case
  - Is your constant factor reasonable?

- Runtime Error:
  - Check memory usage.
    - Forget to clear or destroy stuff?
    - > `vector::shrink_to_fit()`
  - Stack overflow?
  - Bad pointer / array access?
    - Try `-fsanitize=address`
  - Division by zero? NaN's?
```