Contents

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. 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.

```
#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
```

1.2.2. Fast I/O

```
struct scanner {
  static constexpr size_t LEN = 32 << 20;
  char *buf, *buf_ptr, *buf_end;
  scanner()
      : buf(new char[LEN]), buf_ptr(buf + LEN),
        buf_end(buf + LEN) {}
  ~scanner() { delete[] buf; }
  char getc() {
    if (buf_ptr == buf_end) [[unlikely]]
      buf_end = buf + fread_unlocked(buf, 1, LEN, stdin),
      buf_ptr = buf;
    return *(buf_ptr++);
  }
  char seek(char del) {
    char c;
    while ((c = getc()) < del) {}
    return c;
  }
  void read(int &t) {
    bool neg = false;
    char c = seek('-');
    if (c == '-') neg = true, t = 0;
    else t = c ^ '0';
    while ((c = getc()) >= '0') t = t * 10 + (c ^ '0');
    if (neg) t = -t;
  }
};
struct printer {
  static constexpr size_t CPI = 21, LEN = 32 << 20;
  char *buf, *buf_ptr, *buf_end, *tbuf;
  char *int_buf, *int_buf_end;
  printer()
      : buf(new char[LEN]), buf_ptr(buf),
        buf_end(buf + LEN), int_buf(new char[CPI + 1]()),
        int_buf_end(int_buf + CPI - 1) {}
  ~printer() {
    flush();
    delete[] buf, delete[] int_buf;
  }
  void flush() {
    fwrite_unlocked(buf, 1, buf_ptr - buf, stdout);
    buf_ptr = buf;
  }
  void write_(const char &c) {
    *buf_ptr = c;
    if (++buf_ptr == buf_end) [[unlikely]]
      flush();
  }
  void write_(const char *s) {
    for (; *s != '\0'; ++s) write_(*s);
  }
  void write(int x) {
    if (x < 0) write_('-'), x = -x;
    if (x == 0) [[unlikely]]
      return write_('0');
    for (tbuf = int_buf_end; x != 0; --tbuf, x /= 10)
      *tbuf = '0' + char(x % 10);
    write_(++tbuf);
  }
};
```

Kotlin

```kotlin
import java.io.*
import java.util.*

@JvmField val cin = System.`in`.bufferedReader()
@JvmField val cout = PrintWriter(System.out, false)
@JvmField var tokenizer: StringTokenizer = StringTokenizer("")
fun nextLine() = cin.readLine()!!
fun read(): String {
  while(!tokenizer.hasMoreTokens())
    tokenizer = StringTokenizer(nextLine())
  return tokenizer.nextToken()
}

// example
fun main() {
  val n = read().toInt()
  val a = DoubleArray(n) { read().toDouble() }
  cout.println("omg hi")
  cout.flush()
}
```

1.2.3. constexpr

Some default limits in gcc (7.x - trunk):

- constexpr recursion depth: 512
- constexpr loop iteration per function: 262 144
- constexpr operation count per function: 33 554 432
- template recursion depth: 900 (gcc *might* segfault first)

```
constexpr array<int, 10> fibonacci{[] {
  array<int, 10> a{};
  a[0] = a[1] = 1;
  for (int i = 2; i < 10; i++) a[i] = a[i - 1] + a[i - 2];
  return a;
}()};
static_assert(fibonacci[9] == 55, "CE");

template <typename F, typename INT, INT... S>
constexpr void for_constexpr(integer_sequence<INT, S...>,
                             F &&func) {
  int _[] = {(func(integral_constant<INT, S>{}), 0)...};
}
// example
template <typename... T> void print_tuple(tuple<T...> t) {
  for_constexpr(make_index_sequence<sizeof...(T)>{},
              [&](auto i) { cout << get<i>(t) << '\n'; });
}
```

1.2.4. Bump Allocator

```cpp



// global bump allocator
char mem[256 << 20]; // 256 MB
size_t rsp = sizeof mem;
void *operator new(size_t s) {
  assert(s < rsp); // MLE
  return (void *)&mem[rsp -= s];
}
void operator delete(void *) {}

// bump allocator for STL / pbds containers
char mem[256 << 20];
size_t rsp = sizeof mem;
template <typename T> struct bump {
  typedef T value_type;
  bump() {}
  template <typename U> bump(U, ...) {}
  T *allocate(size_t n) {
    rsp -= n * sizeof(T);
    rsp &= 0 - alignof(T);
    return (T *)(mem + rsp);
  }
  void deallocate(T *, size_t n) {}
};
```

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. 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;
}
```

1.4.3. 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;
}
```

1.4.4. 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. 2D Partial Sums

```cpp
using vvi = vector<vector<int>>;
using vvll = vector<vector<ll>>;
using vll = vector<ll>;

struct PrefixSum2D {
  vvll pref;                  // 0-based 2-D prefix sum
  void build(const vvll &v) { // creates a copy
    int n = v.size(), m = v[0].size();
    pref.assign(n, vll(m, 0));
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        pref[i][j] = v[i][j] + (i ? pref[i - 1][j] : 0) +
                     (j ? pref[i][j - 1] : 0) -
                     (i && j ? pref[i - 1][j - 1] : 0);
      }
    }
  }
  ll query(int ulx, int uly, int brx, int bry) const {
    ll ans = pref[brx][bry];
    if (ulx) ans -= pref[ulx - 1][bry];
    if (uly) ans -= pref[brx][uly - 1];
    if (ulx && uly) ans += pref[ulx - 1][uly - 1];
    return ans;
  }
  ll query(int ulx, int uly, int size) const {
    return query(ulx, uly, ulx + size - 1, uly + size - 1);
  }
};
struct PartialSum2D : PrefixSum2D {
  vvll diff; // 0 based
  int n, m;
  PartialSum2D(int _n, int _m) : n(_n), m(_m) {
    diff.assign(n + 1, vll(m + 1, 0));
  }
  // add c from [ulx,uly] to [brx,bry]
  void update(int ulx, int uly, int brx, int bry, ll c) {
    diff[ulx][uly] += c;
    diff[ulx][bry + 1] -= c;
    diff[brx + 1][uly] -= c;
    diff[brx + 1][bry + 1] += c;
  }
  void update(int ulx, int uly, int size, ll c) {
    int brx = ulx + size - 1;
    int bry = uly + size - 1;
    update(ulx, uly, brx, bry, c);
  }
  // process the grid using prefix sum
  void process() { this->build(diff); }
};
// usage
PrefixSum2D pref;
pref.build(v); // takes 2d 0-based vector as input
pref.query(x1, y1, x2, y2); // sum of region

PartialSum2D part(n, m);         // dimension of grid 0 based
part.update(x1, y1, x2, y2, 1); // add 1 in region
// must run after all updates
part.process(); // prefix sum on diff array
// only exists after processing
vvll &grid = part.pref;     // processed diff array
part.query(x1, y1, x2, y2); // gives sum of region
```

2.3. Heavy-Light Decomposition

```cpp

template <bool VALS_EDGES> struct HLD {
  int N, tim = 0;
  vector<vi> adj;
  vi par, siz, depth, rt, pos;
  Node *tree;
  HLD(vector<vi> adj_)
      : N(sz(adj_)), adj(adj_), par(N, -1), siz(N, 1),
        depth(N), rt(N), pos(N), tree(new Node(0, N)) {
    dfsSz(0);
    dfsHld(0);
  }
  void dfsSz(int v) {
    if (par[v] != -1)
      adj[v].erase(find(all(adj[v]), par[v]));
    for (int &u : adj[v]) {
      par[u] = v, depth[u] = depth[v] + 1;
      dfsSz(u);
      siz[v] += siz[u];
      if (siz[u] > siz[adj[v][0]]) swap(u, adj[v][0]);
    }
  }
  void dfsHld(int v) {
    pos[v] = tim++;
    for (int u : adj[v]) {
      rt[u] = (u == adj[v][0] ? rt[v] : u);
      dfsHld(u);
    }
  }
  template <class B> void process(int u, int v, B op) {
    for (; rt[u] != rt[v]; v = par[rt[v]]) {
      if (depth[rt[u]] > depth[rt[v]]) swap(u, v);
      op(pos[rt[v]], pos[v] + 1);
    }
    if (depth[u] > depth[v]) swap(u, v);
    op(pos[u] + VALS_EDGES, pos[v] + 1);
  }
  void modifyPath(int u, int v, int val) {
    process(u, v,
            [&](int l, int r) { tree->add(l, r, val); });
  }
  int queryPath(int u,
                int v) { // Modify depending on problem
    int res = -1e9;
    process(u, v, [&](int l, int r) {
      res = max(res, tree->query(l, r));
    });
    return res;
  }
  int querySubtree(int v) { // modifySubtree is similar
    return tree->query(pos[v] + VALS_EDGES,
                       pos[v] + siz[v]);
  }
};
```

3. Graph

3.1. 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 ∞ (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 ∞ 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 $c_{xyx'y'}$.

3.2. Matching/Flows

3.2.1. Kuhn-Munkres algorithm

```cpp
// 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;
```

3.3. 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();
  }
};
```

3.4. Strongly Connected Components

```
struct TarjanScc {
  int n, step;
  vector<int> time, low, instk, stk;
  vector<vector<int>> e, scc;
  TarjanScc(int n_)
      : n(n_), step(0), time(n), low(n), instk(n), e(n) {}
  void add_edge(int u, int v) { e[u].push_back(v); }
  void dfs(int x) {
    time[x] = low[x] = ++step;
    stk.push_back(x);
    instk[x] = 1;
    for (int y : e[x])
      if (!time[y]) {
        dfs(y);
        low[x] = min(low[x], low[y]);
      } else if (instk[y]) {
        low[x] = min(low[x], time[y]);
      }
    if (time[x] == low[x]) {
      scc.emplace_back();
      for (int y = -1; y != x;) {
        y = stk.back();
        stk.pop_back();
        instk[y] = 0;
        scc.back().push_back(y);
      }
    }
  }
  void solve() {
    for (int i = 0; i < n; i++)
      if (!time[i]) dfs(i);
    reverse(scc.begin(), scc.end());
    // scc in topological order
  }
};
```

3.5. Biconnected Components

3.5.1. Articulation Points

```
void dfs(int x, int p) {
  tin[x] = low[x] = ++t;
  int ch = 0;
  for (auto u : g[x])
    if (u.first != p) {
      if (!ins[u.second])
        st.push(u.second), ins[u.second] = true;
      if (tin[u.first]) {
        low[x] = min(low[x], tin[u.first]);
        continue;
      }
      ++ch;
      dfs(u.first, x);
      low[x] = min(low[x], low[u.first]);
      if (low[u.first] >= tin[x]) {
        cut[x] = true;
        ++sz;
        while (true) {
          int e = st.top();
          st.pop();
          bcc[e] = sz;
          if (e == u.second) break;
        }
      }
    }
  if (ch == 1 && p == -1) cut[x] = false;
}
```

3.5.2. Bridges

```
// if there are multi-edges, then they are not bridges
void dfs(int x, int p) {
  tin[x] = low[x] = ++t;
  st.push(x);
  for (auto u : g[x])
    if (u.first != p) {
      if (tin[u.first]) {
        low[x] = min(low[x], tin[u.first]);
        continue;
      }
      dfs(u.first, x);
      low[x] = min(low[x], low[u.first]);
      if (low[u.first] == tin[u.first]) br[u.second] = true;
    }
  if (tin[x] == low[x]) {
    ++sz;
    while (st.size()) {
      int u = st.top();
      st.pop();
      bcc[u] = sz;
      if (u == x) break;
    }
  }
}
```

3.6. 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);
    }
}
```

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

```cpp


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

```cpp


// 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

```cpp
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

```cpp

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

```cpp
// 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

```cpp
// 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. 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;
  }
}
```

4.1.9. 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;
}
```

4.2. Combinatorics

4.2.1. De Brujin Sequence

```
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;
}
```

5. Numeric

5.1. Long Long Multiplication

```
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. Geometry

6.1. Point

```cpp
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>;
```

6.1.1. Quarternion

```cpp
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();
  }
};
```

6.1.2. Spherical Coordinates

```cpp
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};
}
```

6.2. Segments

```cpp
// 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));
  }
}
```

6.3. Convex Hull

```cpp
// 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;
}
```

6.3.1. 3D Hull

```cpp


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;
};
```

6.4. Angular Sort

```cpp
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);
}
```

6.5. Convex Polygon Minkowski Sum

```cpp
// 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;
}
```

6.6. Point In Polygon

```cpp
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;
}
```

6.6.1. Convex Version

```cpp
// 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) {
```

```cpp
  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);
}
```

6.6.2. Offline Multiple Points Version

Requires: GNU PBDS, Point

```cpp





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);
    }
  }
}
```

6.7. Closest Pair

```cpp
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;
}
```

6.8. 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. Strings

7.1. 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;
}
```

7.2. 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;
  }
}
```

7.3. 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;
  }
}
```

7.4. 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;
}
```

7.5. 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. 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?
```