Contents

Think twice, code once

Template

tem.cpp

```
#pragma GCC optimize("Ofast,unroll-loops,no-stack-protector
    ")
#include <bits/stdc++.h>
using namespace std;

#define fore(i, l, r) \
  for (auto i = (l) - ((l) > (r)); i != (r) - ((l) > (r));
      i += 1 - 2 * ((l) > (r)))
#define sz(x) int(x.size())
#define all(x) begin(x), end(x)
#define f first
#define s second
#define pb push_back

#ifdef LOCAL
#include "debug.h"
#else
#define debug(...)
#endif

using ld = long double;
using lli = long long;
using ii = pair<int, int>;
using vi = vector<int>;

int main() {
  cin.tie(0)->sync_with_stdio(0), cout.tie(0);
  // solve the problem here D:
  return 0;
}
```

debug.h

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

template <class A, class B, class C>
basic_ostream<A, B>& operator<<(basic_ostream<A, B>& os,
    const C& c) {
  os << "[";
  for (const auto& x : c)
    os << ", " + 2 * (&x == &*begin(c)) << x;
  return os << "]";
}

void print(string s) {
  cout << endl;
}

template <class H, class... T>
void print(string s, const H& h, const T&... t) {
  const static string reset = "\033[0m", blue = "\033[1;34m
      ", purple = "\033[3;95m";
  bool ok = 1;
  do {
    if (s[0] == '\"')
      ok = 0;
    else
      cout << blue << s[0] << reset;
    s = s.substr(1);
  } while (s.size() && s[0] != ',');
```

Randoms

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
    count());

template <class T>
T uid(T l, T r) {
  return uniform_int_distribution<T>(l, r)(rng);
}
```

Compilation (gedit ~/.zshenv)

```
touch a_in{1..9} // make files a_in1, a_in2,..., a_in9
tee {a..m}.cpp < tem.cpp // "" with tem.cpp like base
cat > a_in1 // write on file a_in1
gedit a_in1 // open file a_in1
rm -r a.cpp // deletes file a.cpp :'(

red='\x1B[0;31m'
green='\x1B[0;32m'
noColor='\x1B[0m'
alias flags='-Wall -Wextra -Wshadow -D_GLIBCXX_ASSERTIONS -
    fmax-errors=3 -O2 -w'
go() { g++ --std=c++11 $2 ${flags} $1.cpp && ./a.out }
debug() { go $1 -DLOCAL < $2 }
run() { go $1 "" < $2 }

random() { // Make small test cases!!!
 g++ --std=c++11 $1.cpp -o prog
 g++ --std=c++11 gen.cpp -o gen
 g++ --std=c++11 brute.cpp -o brute
 for ((i = 1; i <= 200; i++)); do
  printf "Test case #$i"
  ./gen > in
  diff -uwi <(./prog < in) <(./brute < in) > $1_diff
  if [[ ! $? -eq 0 ]]; then
   printf "${red} Wrong answer ${noColor}\n"
   break
  else
   printf "${green} Accepted ${noColor}\n"
  fi
 done
}
```

Bump allocator

```
static char buf[450 << 20];
void* operator new(size_t s) {
  static size_t i = sizeof buf;
```

```
  assert(s < i);
  return (void*)&buf[i -= s];
}
void operator delete(void*) {}
```

1 Data structures

1.1 DSU with rollback

```
struct Dsu {
  vector<int> par, tot;
  stack<ii> mem;

  Dsu(int n = 1) : par(n + 1), tot(n + 1, 1) {
    iota(all(par), 0);
  }

  int find(int u) {
    return par[u] == u ? u : find(par[u]);
  }

  void unite(int u, int v) {
    u = find(u), v = find(v);
    if (u != v) {
      if (tot[u] < tot[v])
        swap(u, v);
      mem.emplace(u, v);
      tot[u] += tot[v];
      par[v] = u;
    } else {
      mem.emplace(-1, -1);
    }
  }

  void rollback() {
    auto [u, v] = mem.top();
    mem.pop();
    if (u != -1) {
      tot[u] -= tot[v];
      par[v] = v;
    }
  }
};
```

1.2 Monotone queue

```
template <class T, class F = less<T>>
struct MonotoneQueue {
  deque<pair<T, int>> pref;
  F f;

  void add(int pos, T val) {
    while (pref.size() && !f(pref.back().f, val))
      pref.pop_back();
    pref.emplace_back(val, pos);
  }

  void keep(int pos) { // >= pos
    while (pref.size() && pref.front().s < pos)
      pref.pop_front();
  }

  T query() {
    return pref.empty() ? T() : pref.front().f;
  }
};
```

1.3 Stack queue

```
template <class T, class F = function<T(const T&, const T&)
    >>
struct Stack : vector<T> {
  vector<T> s;
  F f;

  Stack(const F& f) : f(f) {}

  void push(T x) {
    this->pb(x);
    s.pb(s.empty() ? x : f(s.back(), x));
  }

  T pop() {
    T x = this->back();
    this->pop_back();
    s.pop_back();
    return x;
  }

  T query() {
    return s.back();
  }
};

template <class T, class F = function<T(const T&, const T&)
    >>
struct Queue {
  Stack<T> a, b;
  F f;

  Queue(const F& f) : a(f), b(f), f(f) {}

  void push(T x) {
    b.push(x);
  }

  T pop() {
    if (a.empty())
      while (!b.empty())
        a.push(b.pop());
    return a.pop();
  }

  T query() {
    if (a.empty())
      return b.query();
    if (b.empty())
      return a.query();
    return f(a.query(), b.query());
  }
};
```

1.4 Mo's algorithm $\mathcal{O}((N+Q)\cdot\sqrt{N}\cdot F)$

```
const int BLOCK = sqrt(N);
sort(all(queries), [&](Query& a, Query& b) {
  const int ga = a.l / BLOCK, gb = b.l / BLOCK;
  if (ga == gb)
    return a.r < b.r;
  return ga < gb;
});

int l = queries[0].l, r = l - 1;
for (Query& q : queries) {
  while (r < q.r)
    add(++r);
  while (r > q.r)
    rem(r--);
  while (l < q.l)
```

To make it faster, change the order to $hilbert(l, r)$

```
lli hilbert(int x, int y, int pw = 21, int rot = 0) {
  if (pw == 0)
```

```
    return 0;
  int hpw = 1 << (pw - 1);
  int k = ((x < hpw ? y < hpw ? 0 : 3 : y < hpw ? 1 : 2) +
      rot) & 3;
  const int d[4] = {3, 0, 0, 1};
  lli a = 1LL << ((pw << 1) - 2);
  lli b = hilbert(x & (x ^ hpw), y & (y ^ hpw), pw - 1, (
      rot + d[k]) & 3);
  return k * a + (d[k] ? a - b - 1 : b);
}
```

1.5 Static to dynamic $\mathcal{O}(N \cdot F \cdot logN)$

```
template <class Black, class T>
struct StaticDynamic {
  Black box[25];
  vector<T> st[25];

  void insert(T& x) {
    int p = 0;
    while (p < 25 && !st[p].empty())
      p++;
    st[p].pb(x);
    fore (i, 0, p) {
      st[p].insert(st[p].end(), all(st[i]));
      box[i].clear(), st[i].clear();
    }
    for (auto y : st[p])
      box[p].insert(y);
    box[p].init();
  }
};
```

1.6 Disjoint intervals

insert, erase: $\mathcal{O}(logN)$

```
template <class T>
struct DisjointIntervals {
  set<pair<T, T>> st;

  void insert(T l, T r) {
    auto it = st.lower_bound({l, -1});
    if (it != st.begin() && l <= prev(it)->s)
      l = (--it)->f;
    for (; it != st.end() && it->f <= r; st.erase(it++))
      r = max(r, it->s);
    st.insert({l, r});
  }

  void erase(T l, T r) {
    auto it = st.lower_bound({l, -1});
    if (it != st.begin() && l <= prev(it)->s)
      --it;
    T mn = l, mx = r;
    for (; it != st.end() && it->f <= r; st.erase(it++))
      mn = min(mn, it->f), mx = max(mx, it->s);
    if (mn < l)
      st.insert({mn, l - 1});
    if (r < mx)
      st.insert({r + 1, mx});
  }
};
```

2 Static range queries

2.1 Sparse table

build: $\mathcal{O}(N \cdot logN)$, query idempotent: $\mathcal{O}(1)$, normal: $\mathcal{O}(logN)$

```
template <class T, class F = function<T(const T&, const T&)
    >>
struct Sparse {
  vector<T> sp[25];
  F f;
  int n;

  Sparse(T* begin, T* end, const F& f) : Sparse(vector<T>(
      begin, end), f) {}

  Sparse(const vector<T>& a, const F& f) : f(f), n(sz(a)) {
    sp[0] = a;
    for (int k = 1; (1 << k) <= n; k++) {
      sp[k].resize(n - (1 << k) + 1);
      fore (l, 0, sz(sp[k])) {
        int r = l + (1 << (k - 1));
        sp[k][l] = f(sp[k - 1][l], sp[k - 1][r]);
      }
    }
  }

  T query(int l, int r) {
#warning Can give TLE D:, change it to a log table
    int k = __lg(r - l + 1);
    return f(sp[k][l], sp[k][r - (1 << k) + 1]);
  }
};
```

2.2 Squirtle decomposition

build $\mathcal{O}(N \cdot \sqrt{N})$, update, query: $\mathcal{O}(\sqrt{N})$
The perfect block size is *squirtle* of N

```
const int BLOCK = sqrt(N);
int blo[N]; // blo[i] = i / BLOCK

void update(int i) {}

int query(int l, int r) {
  while (l <= r)
    if (l % BLOCK == 0 && l + BLOCK - 1 <= r) {
      // solve for block
      l += BLOCK;
    } else {
      // solve for individual element
      l++;
    }
}
```

2.3 Parallel binary search $\mathcal{O}((N+Q) \cdot logN \cdot F)$

```
int lo[Q], hi[Q];
queue<int> solve[N];
vector<Query> queries;

fore (it, 0, 1 + __lg(N)) {
  fore (i, 0, sz(queries))
    if (lo[i] != hi[i]) {
      int mid = (lo[i] + hi[i]) / 2;
      solve[mid].emplace(i);
    }
  fore (x, 0, n) { // 0th-indexed
    // simulate
    while (!solve[x].empty()) {
      int i = solve[x].front();
      solve[x].pop();
      if (can(queries[i]))
        hi[i] = x;
      else
        lo[i] = x + 1;
    }
  }
}
```

3 Dynamic range queries

3.1 Fenwick tree

```cpp
template <class T>
struct Fenwick {
  vector<T> fenw;

  Fenwick(int n) : fenw(n, T()) {} // 0-indexed

  void update(int i, T v) {
    for (; i < sz(fenw); i |= i + 1)
      fenw[i] += v;
  }

  T query(int i) {
    T v = T();
    for (; i >= 0; i &= i + 1, --i)
      v += fenw[i];
    return v;
  }

  int lower_bound(T v) {
    int pos = 0;
    fore (k, 1 + __lg(sz(fenw)), 0)
      if (pos + (1 << k) <= sz(fenw) && fenw[pos + (1 << k)
           - 1] < v) {
        pos += (1 << k);
        v -= fenw[pos - 1];
      }
    return pos + (v == 0);
  }
};
```

3.2 Dynamic segment tree

```cpp
template <class T>
struct Dyn {
  int l, r;
  Dyn *left, *right;
  T val;

  Dyn(int l, int r) : l(l), r(r), left(0), right(0) {}

  void pull() {
    val = (left ? left->val : T()) + (right ? right->val :
         T());
  }

  template <class... Args>
  void update(int p, const Args&... args) {
    if (l == r) {
      val = T(args...);
      return;
    }
    int m = (l + r) >> 1;
    if (p <= m) {
      if (!left)
        left = new Dyn(l, m);
      left->update(p, args...);
    } else {
      if (!right)
        right = new Dyn(m + 1, r);
      right->update(p, args...);
    }
    pull();
  }

  T query(int ll, int rr) {
    if (rr < l || r < ll || r < l)
      return T();
    if (ll <= l && r <= rr)
      return val;
    int m = (l + r) >> 1;
    return (left ? left->query(ll, rr) : T()) + (right ?
         right->query(ll, rr) : T());
  }
};
```

3.3 Persistent segment tree

```cpp
template <class T>
struct Per {
  int l, r;
  Per *left, *right;
  T val;

  Per(int l, int r) : l(l), r(r), left(0), right(0) {}

  Per* pull() {
    val = left->val + right->val;
    return this;
  }

  void build() {
    if (l == r)
      return;
    int m = (l + r) >> 1;
    (left = new Per(l, m))->build();
    (right = new Per(m + 1, r))->build();
    pull();
  }

  template <class... Args>
  Per* update(int p, const Args&... args) {
    if (p < l || r < p)
      return this;
    Per* tmp = new Per(l, r);
    if (l == r) {
      tmp->val = T(args...);
      return tmp;
    }
    tmp->left = left->update(p, args...);
    tmp->right = right->update(p, args...);
    return tmp->pull();
  }

  T query(int ll, int rr) {
    if (r < ll || rr < l)
      return T();
    if (ll <= l && r <= rr)
      return val;
    return left->query(ll, rr) + right->query(ll, rr);
  }
};
```

3.4 Wavelet tree

```cpp
struct Wav {
  int lo, hi;
  Wav *left, *right;
  vector<int> amt;

  template <class Iter>
  Wav(int lo, int hi, Iter b, Iter e) : lo(lo), hi(hi) { //
       array 1-indexed
    if (lo == hi || b == e)
      return;
    amt.reserve(e - b + 1);
    amt.pb(0);
    int mid = (lo + hi) >> 1;
    auto leq = [mid](auto x) {
      return x <= mid;
    };
    for (auto it = b; it != e; it++)
      amt.pb(amt.back() + leq(*it));
    auto p = stable_partition(b, e, leq);
    left = new Wav(lo, mid, b, p);
```

```
    right = new Wav(mid + 1, hi, p, e);
  }

  int kth(int l, int r, int k) {
    if (r < l)
      return 0;
    if (lo == hi)
      return lo;
    if (k <= amt[r] - amt[l - 1])
      return left->kth(amt[l - 1] + 1, amt[r], k);
    return right->kth(l - amt[l - 1], r - amt[r], k - amt[r
        ] + amt[l - 1]);
  }

  int count(int l, int r, int x, int y) {
    if (r < l || y < x || y < lo || hi < x)
      return 0;
    if (x <= lo && hi <= y)
      return r - l + 1;
    return left->count(amt[l - 1] + 1, amt[r], x, y) +
           right->count(l - amt[l - 1], r - amt[r], x, y);
  }
};
```

3.5 Li Chao tree

```
struct Fun {
  lli m = 0, c = INF;
  lli operator()(lli x) const {
    return m * x + c;
  }
};

struct LiChao {
  lli l, r;
  LiChao *left, *right;
  Fun f;

  LiChao(lli l, lli r, Fun f) : l(l), r(r), f(f) left(0),
      right(0) {}

  void add(Fun& g) {
    if (f(l) <= g(l) && f(r) <= g(r))
      return;
    if (g(l) < f(l) && g(r) < f(r)) {
      f = g;
      return;
    }
    lli m = (l + r) >> 1;
    if (g(m) < f(m))
      swap(f, g);
    if (g(l) <= f(l))
      left = left ? (left->add(g), left) : new LiChao(l, m,
          g);
    else
      right = right ? (right->add(g), right) : new LiChao(m
          + 1, r, g);
  }

  lli query(lli x) {
    if (l == r)
      return f(x);
    lli m = (l + r) >> 1;
    if (x <= m)
      return min(f(x), left ? left->query(x) : INF);
    return min(f(x), right ? right->query(x) : INF);
  }
};
```

4 Binary trees

4.1 Ordered tree

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

template <class K, class V = null_type>
using OrderedTree = tree<K, V, less<K>, rb_tree_tag,
    tree_order_statistics_node_update>;
#define rank order_of_key
#define kth find_by_order
```

4.2 Treap

```
struct Treap {
  static Treap* null;
  Treap *left, *right;
  unsigned pri = rng(), sz = 0;
  int val = 0;

  void push() {
    // propagate like segtree, key-values aren't modified!!
  }

  Treap* pull() {
    sz = left->sz + right->sz + (this != null);
    // merge(left, this), merge(this, right)
    return this;
  }

  Treap() {
    left = right = null;
  }

  Treap(int val) : val(val) {
    left = right = null;
    pull();
  }

  template <class F>
  pair<Treap*, Treap*> split(const F& leq) { // {<= val, >
      val}
    if (this == null)
      return {null, null};
    push();
    if (leq(this)) {
      auto p = right->split(leq);
      right = p.f;
      return {pull(), p.s};
    } else {
      auto p = left->split(leq);
      left = p.s;
      return {p.f, pull()};
    }
  }

  Treap* merge(Treap* other) {
    if (this == null)
      return other;
    if (other == null)
```

4.3 Implicit treap (Rope)

```
      return right = right->merge(other), pull();
    } else {
      return other->left = merge(other->left), other->pull
          ();
    }
  }

  pair<Treap*, Treap*> leftmost(int k) {
    return split([&](Treap* n) {
      int sz = n->left->sz + 1;
      if (k >= sz) {
        k -= sz;
```

5 Graphs

5.1 Topological sort $\mathcal{O}(V + E)$

```
vector<int> order;
int indeg[N];

void topologicalSort() { // first fill the indeg[]
  queue<int> qu;
  fore (u, 1, n + 1)
    if (indeg[u] == 0)
      qu.push(u);
  while (!qu.empty()) {
    int u = qu.front();
    qu.pop();
    order.pb(u);
    for (auto& v : graph[u])
      if (--indeg[v] == 0)
        qu.push(v);
  }
}
```

5.2 Tarjan algorithm (SCC) $\mathcal{O}(V + E)$

```
int tin[N], fup[N];
bitset<N> still;
stack<int> stk;
int timer = 0;

void tarjan(int u) {
  tin[u] = fup[u] = ++timer;
  still[u] = true;
  stk.push(u);
  for (auto& v : graph[u]) {
    if (!tin[v])
      tarjan(v);
    if (still[v])
      fup[u] = min(fup[u], fup[v]);
  }
  if (fup[u] == tin[u]) {
    int v;
    do {
      v = stk.top();
      stk.pop();
      still[v] = false;
      // u and v are in the same scc
    } while (v != u);
  }
}
```

5.3 Kosaraju algorithm (SCC) $\mathcal{O}(V + E)$

```
int scc[N], k = 0;
char vis[N];
vector<int> order;

void dfs1(int u) {
  vis[u] = 1;
  for (int v : graph[u])
    if (vis[v] != 1)
      dfs1(v);
  order.pb(u);
}

void dfs2(int u, int k) {
  vis[u] = 2, scc[u] = k;
  for (int v : rgraph[u]) // reverse graph
    if (vis[v] != 2)
      dfs2(v, k);
}

void kosaraju() {
  fore (u, 1, n + 1)
    if (vis[u] != 1)
      dfs1(u);
  reverse(all(order));
  for (int u : order)
    if (vis[u] != 2)
      dfs2(u, ++k);
}
```

5.4 Cutpoints and Bridges $\mathcal{O}(V + E)$

```
int tin[N], fup[N], timer = 0;

void weakness(int u, int p = -1) {
  tin[u] = fup[u] = ++timer;
  int children = 0;
  for (int v : graph[u])
    if (v != p) {
      if (!tin[v]) {
        ++children;
        weakness(v, u);
        fup[u] = min(fup[u], fup[v]);
        if (fup[v] >= tin[u] && !(p == -1 && children < 2))
            // u is a cutpoint
          if (fup[v] > tin[u]) // bridge u -> v
      }
      fup[u] = min(fup[u], tin[v]);
    }
}
```

5.5 Two Sat $\mathcal{O}(V + E)$

```
// 1-indexed
struct TwoSat {
  int n;
  vector<vector<int>> imp;

  TwoSat(int k) : n(k + 1), imp(2 * n) {}

  // a || b
  void either(int a, int b) {
    a = max(2 * a, -1 - 2 * a);
    b = max(2 * b, -1 - 2 * b);
    imp[a ^ 1].pb(b);
    imp[b ^ 1].pb(a);
  }

  // if a then b
  // a  b   a => b
  // F  F     T
  // T  T     T
  // F  T     T
  // T  F     F
  void implies(int a, int b) {
    either(~a, b);
  }

  // setVal(a): set a = true
  // setVal(~a): set a = false
  void setVal(int a) {
    either(a, a);
  }

  optional<vector<int>> solve() {
    int k = sz(imp);
    vector<int> s, b, id(sz(imp));
    function<void(int)> dfs = [&](int u) {
      b.pb(id[u] = sz(s)), s.pb(u);
      for (int v : imp[u]) {
        if (!id[v])
          dfs(v);
        else
          while (id[v] < b.back())
            b.pop_back();
      }
```

```
      if (id[u] == b.back())
        for (b.pop_back(), ++k; id[u] < sz(s); s.pop_back()
            )
          id[s.back()] = k;
    };
    vector<int> val(n);
    fore (u, 0, sz(imp))
      if (!id[u])
        dfs(u);
    fore (u, 0, n) {
      int x = 2 * u;
      if (id[x] == id[x ^ 1])
        return nullopt;
      val[u] = id[x] < id[x ^ 1];
    }
    return optional(val);
  }
};
```

5.6 Detect a cycle $\mathcal{O}(V + E)$

```
bool cycle(int u) {
  vis[u] = 1;
  for (int v : graph[u]) {
    if (vis[v] == 1)
      return true;
    if (!vis[v] && cycle(v))
      return true;
  }
  vis[u] = 2;
  return false;
}
```

5.7 Isomorphism $\mathcal{O}(V + E)$

```
  // K * n <= 9e18
  static uniform_int_distribution<lli> uid(1, K);
  if (!mp.count(x))
    mp[x] = uid(rng);
  return mp[x];
}

lli hsh(int u, int p = -1) {
  dp[u] = h[u] = 0;
  for (auto& v : graph[u]) {
    if (v == p)
      continue;
    dp[u] += hsh(v, u);
  }
  return h[u] = f(dp[u]);
}
```

5.8 Dynamic connectivity $\mathcal{O}((N + Q) \cdot logQ)$

```
struct DynamicConnectivity {
  struct Query {
    int op, u, v, at;
  };

  Dsu dsu; // with rollback
  vector<Query> queries;
  map<ii, int> mp;
  int timer = -1;

  DynamicConnectivity(int n = 0) : dsu(n) {}

  void add(int u, int v) {
    mp[minmax(u, v)] = ++timer;
    queries.pb({'+', u, v, INT_MAX});
  }

  void rem(int u, int v) {
    int in = mp[minmax(u, v)];
    queries.pb({'-', u, v, in});
    queries[in].at = ++timer;
    mp.erase(minmax(u, v));
  }

  void query() {
    queries.push_back({'?', -1, -1, ++timer});
  }

  void solve(int l, int r) {
    if (l == r) {
      if (queries[l].op == '?') // solve the query here
        return;
    }
    int m = (l + r) >> 1;
    int before = sz(dsu.mem);
    for (int i = m + 1; i <= r; i++) {
      Query& q = queries[i];
      if (q.op == '-' && q.at < l)
        dsu.unite(q.u, q.v);
    }
    solve(l, m);
    while (sz(dsu.mem) > before)
      dsu.rollback();
    for (int i = l; i <= m; i++) {
      Query& q = queries[i];
      if (q.op == '+' && q.at > r)
        dsu.unite(q.u, q.v);
    }
    solve(m + 1, r);
    while (sz(dsu.mem) > before)
      dsu.rollback();
  }
};
```

6 Tree queries

6.1 Euler tour for Mo's in a tree $\mathcal{O}((V + E) \cdot \sqrt{V})$

Mo's in a tree, extended euler tour tin[u] = ++timer, tout[u] = ++timer

- u = lca(u, v), query(tin[u], tin[v])
- u $\neq$ lca(u, v), query(tout[u], tin[v]) + query(tin[lca], tin[lca])

6.2 Lowest common ancestor (LCA)

build: $\mathcal{O}(N \cdot logN)$, query: $\mathcal{O}(logN)$

```
const int LogN = 1 + __lg(N);
int par[LogN][N], depth[N];

void dfs(int u, int par[]) {
  for (auto& v : graph[u])
    if (v != par[u]) {
      par[v] = u;
      depth[v] = depth[u] + 1;
      dfs(v, par);
    }
}

int lca(int u, int v) {
  if (depth[u] > depth[v])
    swap(u, v);
  fore (k, LogN, 0)
    if (dep[v] - dep[u] >= (1 << k))
      v = par[k][v];
  if (u == v)
    return u;
  fore (k, LogN, 0)
    if (par[k][v] != par[k][u])
      u = par[k][u], v = par[k][v];
  return par[0][u];
```

```cpp
}

int dist(int u, int v) {
  return depth[u] + depth[v] - 2 * depth[lca(u, v)];
}

void init(int r) {
  dfs(r, par[0]);
  fore (k, 1, LogN)
    fore (u, 1, n + 1)
      par[k][u] = par[k - 1][par[k - 1][u]];
}
```

6.3 Virtual tree

build: $\mathcal{O}(|ver| \cdot logN)$

```cpp
vector<int> virt[N];

int virtualTree(vector<int>& ver) {
  auto byDfs = [&](int u, int v) {
    return tin[u] < tin[v];
  };
  sort(all(ver), byDfs);
  fore (i, sz(ver), 1)
    ver.pb(lca(ver[i - 1], ver[i]));
  sort(all(ver), byDfs);
  ver.erase(unique(all(ver)), ver.end());
  for (int u : ver)
    virt[u].clear();
  fore (i, 1, sz(ver))
    virt[lca(ver[i - 1], ver[i])].pb(ver[i]);
  return ver[0];
}
```

6.4 Guni

Solve subtrees problems $\mathcal{O}(N \cdot logN \cdot F)$

```cpp
int cnt[C], color[N];
int sz[N];

int guni(int u, int p = -1) {
  sz[u] = 1;
  for (int& v : graph[u])
    if (v != p) {
      sz[u] += guni(v, u);
      if (sz[v] > sz[graph[u][0]] || p == graph[u][0])
        swap(v, graph[u][0]);
    }
  return sz[u];
}

void update(int u, int p, int add, bool skip) {
  cnt[color[u]] += add;
  for (int i = skip; i < sz(graph[u]); i++) // don't use
      fore !!!
    if (graph[u][i] != p)
      update(graph[u][i], u, add, 0);
}

void solve(int u, int p = -1, bool keep = 0) {
  fore (i, sz(graph[u]), 0)
    if (graph[u][i] != p)
      solve(graph[u][i], u, !i);
  update(u, p, +1, 1); // add
  // now cnt[i] has how many times the color i appears in
      the subtree of u
  if (!keep)
    update(u, p, -1, 0); // remove
}
```

6.5 Centroid decomposition

Solves "all pairs of nodes" problems $\mathcal{O}(N \cdot logN \cdot F)$

```cpp
int cdp[N], sz[N];
bitset<N> rem;

int dfsz(int u, int p = -1) {
  sz[u] = 1;
  for (int v : graph[u])
    if (v != p && !rem[v])
      sz[u] += dfsz(v, u);
  return sz[u];
}

int centroid(int u, int size, int p = -1) {
  for (int v : graph[u])
    if (v != p && !rem[v] && 2 * sz[v] > size)
      return centroid(v, size, u);
  return u;
}

void solve(int u, int p = -1) {
  cdp[u = centroid(u, dfsz(u))] = p;
  rem[u] = true;
  for (int v : graph[u])
    if (!rem[v])
      solve(v, u);
}
```

6.6 Heavy-light decomposition and Euler tour

Solves subtrees and paths problems $\mathcal{O}(N \cdot logN \cdot F)$

```cpp
int par[N], nxt[N], depth[N], sz[N];
int tin[N], tout[N], who[N], timer = 0;

int dfs(int u) {
  sz[u] = 1;
  for (auto& v : graph[u])
    if (v != par[u]) {
      par[v] = u;
      depth[v] = depth[u] + 1;
      sz[u] += dfs(v);
      if (graph[u][0] == par[u] || sz[v] > sz[graph[u][0]])
        swap(v, graph[u][0]);
    }
  return sz[u];
}

void hld(int u) {
  tin[u] = ++timer, who[timer] = u;
  for (auto& v : graph[u])
    if (v != par[u]) {
      nxt[v] = (v == graph[u][0] ? nxt[u] : v);
      hld(v);
    }
  tout[u] = timer;
}

template <bool OverEdges = 0, class F>
void processPath(int u, int v, F f) {
  for (; nxt[u] != nxt[v]; u = par[nxt[u]]) {
    if (depth[nxt[u]] < depth[nxt[v]])
      swap(u, v);
    f(tin[nxt[u]], tin[u]);
  }
  if (depth[u] < depth[v])
    swap(u, v);
  f(tin[v] + OverEdges, tin[u]);
}
```

```cpp
void updatePath(int u, int v, lli z) {
  processPath(u, v, [&](int l, int r) {
    tree->update(l, r, z);
  });
}

void updateSubtree(int u, lli z) {
  tree->update(tin[u], tout[u], z);
}

lli queryPath(int u, int v) {
  lli sum = 0;
  processPath(u, v, [&](int l, int r) {
    sum += tree->query(l, r);
  });
  return sum;
}

lli querySubtree(int u) {
  return tree->query(tin[u], tout[u]);
}

int lca(int u, int v) {
  int last = -1;
  processPath(u, v, [&](int l, int r) {
    last = who[l];
  });
  return last;
}
```

6.7 Link-Cut tree

Solves dynamic trees problems, can handle subtrees and paths maybe with a high constant $\mathcal{O}(N \cdot logN \cdot F)$

```cpp
struct LinkCut {
  struct Node {
    Node *left{0}, *right{0}, *par{0};
    bool rev = 0;
    int sz = 1;
    int sub = 0, vsub = 0; // subtree
    lli path = 0; // path
    lli self = 0; // node info

    void push() {
      if (rev) {
        swap(left, right);
        if (left)
          left->rev ^= 1;
        if (right)
          right->rev ^= 1;
        rev = 0;
      }
    }

    void pull() {
      sz = 1;
      sub = vsub + self;
      path = self;
      if (left) {
        sz += left->sz;
        sub += left->sub;
        path += left->path;
      }
      if (right) {
        sz += right->sz;
        sub += right->sub;
        path += right->path;
      }
    }

    void addVsub(Node* v, lli add) {
      if (v)
        vsub += 1LL * add * v->sub;
    }
  };

  vector<Node> a;

  LinkCut(int n = 1) : a(n) {}

  void splay(Node* u) {
    auto assign = [&](Node* u, Node* v, int d) {
      if (v)
        v->par = u;
      if (d >= 0)
        (d == 0 ? u->left : u->right) = v;
    };
    auto dir = [&](Node* u) {
      if (!u->par)
        return -1;
      return u->par->left == u ? 0 : (u->par->right == u ?
          1 : -1);
    };
    auto rotate = [&](Node* u) {
      Node *p = u->par, *g = p->par;
      int d = dir(u);
      assign(p, d ? u->left : u->right, d);
      assign(g, u, dir(p));
      assign(u, p, !d);
      p->pull(), u->pull();
    };
    while (~dir(u)) {
      Node *p = u->par, *g = p->par;
      if (~dir(p))
        g->push();
      p->push(), u->push();
      if (~dir(p))
        rotate(dir(p) == dir(u) ? p : u);
      rotate(u);
    }
    u->push(), u->pull();
  }

  void access(int u) {
    Node* last = NULL;
    for (Node* x = &a[u]; x; last = x, x = x->par) {
      splay(x);
      x->addVsub(x->right, +1);
      x->right = last;
      x->addVsub(x->right, -1);
      x->pull();
    }
    splay(&a[u]);
  }

  void reroot(int u) {
    access(u);
    a[u].rev ^= 1;
  }

  void link(int u, int v) {
    reroot(v), access(u);
    a[u].addVsub(v, +1);
    a[v].par = &a[u];
    a[u].pull();
  }

  void cut(int u, int v) {
    reroot(v), access(u);
    a[u].left = a[v].par = NULL;
```

```
    a[u].pull();
  }

  int lca(int u, int v) {
    if (u == v)
      return u;
    access(u), access(v);
    if (!a[u].par)
      return -1;
    return splay(&a[u]), a[u].par ? -1 : u;
  }

  int depth(int u) {
    access(u);
    return a[u].left ? a[u].left->sz : 0;
  }

  // get k-th parent on path to root
  int ancestor(int u, int k) {
    k = depth(u) - k;
    assert(k >= 0);
    for (;; a[u].push()) {
      int sz = a[u].left->sz;
      if (sz == k)
        return access(u), u;
      if (sz < k)
        k -= sz + 1, u = u->ch[1];
      else
        u = u->ch[0];
    }
    assert(0);
  }

  lli queryPath(int u, int v) {
    reroot(u), access(v);
    return a[v].path;
  }

  lli querySubtree(int u, int x) {
    // query subtree of u, x is outside
    reroot(x), access(u);
    return a[u].vsub + a[u].self;
  }

  void update(int u, lli val) {
    access(u);
    a[u].self = val;
    a[u].pull();
  }

  Node& operator[](int u) {
    return a[u];
  }
};
```

7 Flows

7.1 Dinic $\mathcal{O}(min(E \cdot flow, V^2E))$

If the network is massive, try to compress it by looking for patterns.

```
template <class F>
struct Dinic {
  struct Edge {
    int v, inv;
    F cap, flow;
    Edge(int v, F cap, int inv) : v(v), cap(cap), flow(0),
        inv(inv) {}
  };

  F EPS = (F)1e-9;
  int s, t, n;
  vector<vector<Edge>> graph;
  vector<int> dist, ptr;

  Dinic(int n) : n(n), graph(n), dist(n), ptr(n), s(n - 2),
      t(n - 1) {}

  void add(int u, int v, F cap) {
    graph[u].pb(Edge(v, cap, sz(graph[v])));
    graph[v].pb(Edge(u, 0, sz(graph[u]) - 1));
  }

  bool bfs() {
    fill(all(dist), -1);
    queue<int> qu({s});
    dist[s] = 0;
    while (sz(qu) && dist[t] == -1) {
      int u = qu.front();
      qu.pop();
      for (Edge& e : graph[u])
        if (dist[e.v] == -1)
          if (e.cap - e.flow > EPS) {
            dist[e.v] = dist[u] + 1;
            qu.push(e.v);
          }
    }
    return dist[t] != -1;
  }

  F dfs(int u, F flow = numeric_limits<F>::max()) {
    if (flow <= EPS || u == t)
      return max<F>(0, flow);
    for (int& i = ptr[u]; i < sz(graph[u]); i++) {
      Edge& e = graph[u][i];
      if (e.cap - e.flow > EPS && dist[u] + 1 == dist[e.v])
          {
        F pushed = dfs(e.v, min<F>(flow, e.cap - e.flow));
        if (pushed > EPS) {
          e.flow += pushed;
          graph[e.v][e.inv].flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }

  F maxFlow() {
    F flow = 0;
    while (bfs()) {
      fill(all(ptr), 0);
      while (F pushed = dfs(s))
        flow += pushed;
    }
    return flow;
  }

  bool leftSide(int u) {
    // left side comes from sink
    return dist[u] != -1;
  }
};
```

7.2 Min cost flow $\mathcal{O}(min(E \cdot flow, V^2E))$

If the network is massive, try to compress it by looking for patterns.

```
template <class C, class F>
struct Mcmf {
  struct Edge {
    int u, v, inv;
    F cap, flow;
    C cost;
```

```
  Edge(int u, int v, C cost, F cap, int inv)
      : u(u), v(v), cost(cost), cap(cap), flow(0), inv(
          inv) {}
};

F EPS = (F)1e-9;
int s, t, n;
vector<vector<Edge>> graph;
vector<Edge*> prev;
vector<C> cost;
vector<int> state;

Mcmf(int n) : n(n), graph(n), cost(n), state(n), prev(n),
      s(n - 2), t(n - 1) {}

void add(int u, int v, C cost, F cap) {
  graph[u].pb(Edge(u, v, cost, cap, sz(graph[v])));
  graph[v].pb(Edge(v, u, -cost, 0, sz(graph[u]) - 1));
}

bool bfs() {
  fill(all(state), 0);
  fill(all(cost), numeric_limits<C>::max());
  deque<int> qu;
  qu.push_back(s);
  state[s] = 1, cost[s] = 0;
  while (sz(qu)) {
    int u = qu.front();
    qu.pop_front();
    state[u] = 2;
    for (Edge& e : graph[u])
      if (e.cap - e.flow > EPS)
        if (cost[u] + e.cost < cost[e.v]) {
          cost[e.v] = cost[u] + e.cost;
          prev[e.v] = &e;
          if (state[e.v] == 2 || (sz(qu) && cost[qu.front
              ()] > cost[e.v]))
            qu.push_front(e.v);
          else if (state[e.v] == 0)
            qu.push_back(e.v);
          state[e.v] = 1;
        }
  }
  return cost[t] != numeric_limits<C>::max();
}

pair<C, F> minCostFlow() {
  C cost = 0;
  F flow = 0;
  while (bfs()) {
    F pushed = numeric_limits<F>::max();
    for (Edge* e = prev[t]; e != nullptr; e = prev[e->u])
      pushed = min(pushed, e->cap - e->flow);
    for (Edge* e = prev[t]; e != nullptr; e = prev[e->u])
      {
      e->flow += pushed;
      graph[e->v][e->inv].flow -= pushed;
      cost += e->cost * pushed;
    }
    flow += pushed;
  }
  return make_pair(cost, flow);
}
};
```

7.3 Hopcroft-Karp $\mathcal{O}(E\sqrt{V})$

```
struct HopcroftKarp {
  int n, m;
  vector<vector<int>> graph;
  vector<int> dist, match;

  HopcroftKarp(int k) : n(k + 1), graph(n), dist(n), match(
      n, 0) {} // 1-indexed!!

  void add(int u, int v) {
    graph[u].pb(v), graph[v].pb(u);
  }

  bool bfs() {
    queue<int> qu;
    fill(all(dist), -1);
    fore (u, 1, n)
      if (!match[u])
        dist[u] = 0, qu.push(u);
    while (!qu.empty()) {
      int u = qu.front();
      qu.pop();
      for (int v : graph[u])
        if (dist[match[v]] == -1) {
          dist[match[v]] = dist[u] + 1;
          if (match[v])
            qu.push(match[v]);
        }
    }
    return dist[0] != -1;
  }

  bool dfs(int u) {
    for (int v : graph[u])
      if (!match[v] || (dist[u] + 1 == dist[match[v]] &&
          dfs(match[v]))) {
        match[u] = v, match[v] = u;
        return 1;
      }
    dist[u] = 1 << 30;
    return 0;
  }

  int maxMatching() {
    int tot = 0;
    while (bfs())
      fore (u, 1, n)
        tot += match[u] ? 0 : dfs(u);
    return tot;
  }
};
```

7.4 Hungarian $\mathcal{O}(N^3)$

n jobs, m people

```
template <class C>
pair<C, vector<int>> Hungarian(vector<vector<C>>& a) { //
    max assignment
  int n = sz(a), m = sz(a[0]), p, q, j, k; // n <= m
  vector<C> fx(n, numeric_limits<C>::min()), fy(m, 0);
  vector<int> x(n, -1), y(m, -1);
  fore (i, 0, n)
    fore (j, 0, m)
      fx[i] = max(fx[i], a[i][j]);
  fore (i, 0, n) {
    vector<int> t(m, -1), s(n + 1, i);
    for (p = q = 0; p <= q && x[i] < 0; p++)
      for (k = s[p], j = 0; j < m && x[i] < 0; j++)
        if (abs(fx[k] + fy[j] - a[k][j]) < EPS && t[j] < 0)
            {
          s[++q] = y[j], t[j] = k;
          if (s[q] < 0)
            for (p = j; p >= 0; j = p)
              y[j] = k = t[j], p = x[k], x[k] = j;
        }
    if (x[i] < 0) {
      C d = numeric_limits<C>::max();
```

```
      fore (k, 0, q + 1)
        fore (j, 0, m)
          if (t[j] < 0)
            d = min(d, fx[s[k]] + fy[j] - a[s[k]][j]);
      fore (j, 0, m)
        fy[j] += (t[j] < 0 ? 0 : d);
      fore (k, 0, q + 1)
        fx[s[k]] -= d;
      i--;
    }
  }
  C cost = 0;
  fore (i, 0, n)
    cost += a[i][x[i]];
  return make_pair(cost, x);
}
```

8 Strings

8.1 Hash $\mathcal{O}(N)$

```
using Hash = int; // maybe an arrray<int, 2>
Hash pw[N], ipw[N];

struct Hashing {
  static constexpr int P = 10166249, M = 1070777777;
  vector<Hash> h;

  static void init() {
    const int Q = inv(P, M);
    pw[0] = ipw[0] = 1;
    fore (i, 1, N) {
      pw[i] = 1LL * pw[i - 1] * P % M;
      ipw[i] = 1LL * ipw[i - 1] * Q % M;
    }
  }

  Hashing(string& s) : h(sz(s) + 1, 0) {
    fore (i, 0, sz(s)) {
      lli x = s[i] - 'a' + 1;
      h[i + 1] = (h[i] + x * pw[i]) % M;
    }
  }

  Hash query(int l, int r) {
    return 1LL * (h[r + 1] - h[l] + M) * ipw[l] % M;
  }

  friend pair<Hash, int> merge(vector<pair<Hash, int>>&
       cuts) {
    pair<Hash, int> ans = {0, 0};
    fore (i, sz(cuts), 0) {
      ans.f = (cuts[i].f + 1LL * ans.f * pw[cuts[i].s] % M)
           % M;
      ans.s += cuts[i].s;
    }
    return ans;
  }
};
```

8.2 KMP $\mathcal{O}(N)$

period $= n - p[n-1]$, period$(abcabc) = 3$, $n \mod period \equiv 0$

```
template <class T>
vector<int> lps(T s) {
  vector<int> p(sz(s), 0);
  for (int j = 0, i = 1; i < sz(s); i++) {
    while (j && s[i] != s[j])
      j = p[j - 1];
    if (s[i] == s[j])
      j++;
    p[i] = j;
  }
  return p;
}

// positions where t is on s
template <class T>
vector<int> kmp(T& s, T& t) {
  vector<int> p = lps(t), pos;
  for (int j = 0, i = 0; i < sz(s); i++) {
    while (j && s[i] != t[j])
      j = p[j - 1];
    if (s[i] == t[j])
      j++;
    if (j == sz(t))
      pos.pb(i - sz(t) + 1);
  }
  return pos;
}
```

8.3 KMP automaton $\mathcal{O}(Alphabet * N)$

```
template <class T, int ALPHA = 26>
struct KmpAutomaton : vector<vector<int>> {
  KmpAutomaton() {}
  KmpAutomaton(T s) : vector<vector<int>>(sz(s) + 1, vector
       <int>(ALPHA)) {
    s.pb(0);
    vector<int> p = lps(s);
    auto& nxt = *this;
    nxt[0][s[0] - 'a'] = 1;
    fore (i, 1, sz(s))
      fore (c, 0, ALPHA)
        nxt[i][c] = (s[i] - 'a' == c ? i + 1 : nxt[p[i - 1
             ]][c]);
  }
};
```

8.4 Z algorithm $\mathcal{O}(N)$

```
template <class T>
vector<int> getZ(T& s) {
  vector<int> z(sz(s), 0);
  for (int i = 1, l = 0, r = 0; i < sz(s); i++) {
    if (i <= r)
      z[i] = min(r - i + 1, z[i - l]);
    while (i + z[i] < sz(s) && s[i + z[i]] == s[z[i]])
      ++z[i];
    if (i + z[i] - 1 > r)
      l = i, r = i + z[i] - 1;
  }
  return z;
}
```

8.5 Manacher algorithm $\mathcal{O}(N)$

```
template <class T>
vector<vector<int>> manacher(T& s) {
  vector<vector<int>> pal(2, vector<int>(sz(s), 0));
  fore (k, 0, 2) {
    int l = 0, r = 0;
    fore (i, 0, sz(s)) {
      int t = r - i + !k;
      if (i < r)
        pal[k][i] = min(t, pal[k][l + t]);
      int p = i - pal[k][i], q = i + pal[k][i] - !k;
      while (p >= 1 && q + 1 < sz(s) && s[p - 1] == s[q + 1
           ])
        ++pal[k][i], --p, ++q;
      if (q > r)
        l = p, r = q;
    }
  }
  return pal;
}
```

8.6 Suffix array $\mathcal{O}(N * logN)$

- Duplicates $\sum_{i=1}^{n} lcp[i]$
- Longest Common Substring of various strings
  Add $notUsed$ characters between strings, i.e. $a+\$+b+\#+c$
  Use two-pointers to find a range $[l,r]$ such that all $notUsed$ characters are present, then $query(lcp[l+1],..,lcp[r])$ for that window is the common length.

```
template <class T>
struct SuffixArray {
  int n;
  T s;
  vector<int> sa, pos, dp[25];

  SuffixArray(const T& x) : n(sz(x) + 1), s(x), sa(n), pos(
      n) {
    s.pb(0);
    fore (i, 0, n)
      sa[i] = i, pos[i] = s[i];
    vector<int> nsa(sa), npos(n), cnt(max(260, n), 0);
    for (int k = 0; k < n; k ? k *= 2 : k++) {
      fill(all(cnt), 0);
      fore (i, 0, n)
        nsa[i] = (sa[i] - k + n) % n, cnt[pos[i]]++;
      partial_sum(all(cnt), cnt.begin());
      fore (i, n, 0)
        sa[--cnt[pos[nsa[i]]]] = nsa[i];
      for (int i = 1, cur = 0; i < n; i++) {
        cur += (pos[sa[i]] != pos[sa[i - 1]] || pos[(sa[i]
            + k) % n] != pos[(sa[i - 1] + k) % n]);
        npos[sa[i]] = cur;
      }
      pos = npos;
      if (pos[sa[n - 1]] >= n - 1)
        break;
    }
    dp[0].assign(n, 0);
    for (int i = 0, j = pos[0], k = 0; i < n - 1; ++i, ++k)
        {
      while (k >= 0 && s[i] != s[sa[j - 1] + k])
        dp[0][j] = k--, j = pos[sa[j] + 1];
    }
    for (int k = 1, pw = 1; pw < n; k++, pw <<= 1) {
      dp[k].assign(n, 0);
      for (int l = 0; l + pw < n; l++)
        dp[k][l] = min(dp[k - 1][l], dp[k - 1][l + pw]);
    }
  }

  int lcp(int l, int r) {
    if (l == r)
      return n - l;
    tie(l, r) = minmax(pos[l], pos[r]);
    int k = __lg(r - l);
    return min(dp[k][l + 1], dp[k][r - (1 << k) + 1]);
  }

  auto at(int i, int j) {
    return sa[i] + j < n ? s[sa[i] + j] : 'z' + 1;
  }

  int count(T& t) {
    int l = 0, r = n - 1;
    fore (i, 0, sz(t)) {
      int p = l, q = r;
      for (int k = n; k > 0; k >>= 1) {
        while (p + k < r && at(p + k, i) < t[i])
          p += k;
        while (q - k > l && t[i] < at(q - k, i))
          q -= k;
      }
      l = (at(p, i) == t[i] ? p : p + 1);
      r = (at(q, i) == t[i] ? q : q - 1);
      if (at(l, i) != t[i] && at(r, i) != t[i] || l > r)
        return 0;
    }
    return r - l + 1;
  }

  bool compare(ii a, ii b) {
    // s[a.f ... a.s] < s[b.f ... b.s]
    int common = lcp(a.f, b.f);
    int szA = a.s - a.f + 1, szB = b.s - b.f + 1;
    if (common >= min(szA, szB))
      return tie(szA, a) < tie(szB, b);
    return s[a.f + common] < s[b.f + common];
  }
};
```

8.7 Suffix automaton $\mathcal{O}(\sum s_i)$

- $sam[u].len - sam[sam[u].link].len$ = distinct strings
- Number of different substrings (dp)

$$diff(u) = 1 + \sum_{v \in trie[u]} diff(v)$$

- Total length of all different substrings (2 x dp)

$$totLen(u) = \sum_{v \in trie[u]} diff(v) + totLen(v)$$

- Leftmost occurrence $trie[u].pos = trie[u].len - 1$
  if it is **clone** then $trie[clone].pos = trie[q].pos$
- All occurrence positions
- Smallest cyclic shift Construct sam of $s+s$, find the lexicographically smallest path of $sz(s)$
- Shortest non-appearing string

$$nonAppearing(u) = \min_{v \in trie[u]} nonAppearing(v) + 1$$

```
struct SuffixAutomaton {
  struct Node : map<char, int> {
    int link = -1, len = 0;
  };

  vector<Node> trie;
  int last;

  SuffixAutomaton(int n = 1) {
    trie.reserve(2 * n), last = newNode();
  }

  int newNode() {
    trie.pb({});
    return sz(trie) - 1;
  }

  void extend(char c) {
    int u = newNode();
    trie[u].len = trie[last].len + 1;
    int p = last;
    while (p != -1 && !trie[p].count(c)) {
      trie[p][c] = u;
      p = trie[p].link;
    }
    if (p == -1)
      trie[u].link = 0;
    else {
      int q = trie[p][c];
      if (trie[p].len + 1 == trie[q].len)
        trie[u].link = q;
      else {
```

```
      int clone = newNode();
      trie[clone] = trie[q];
      trie[clone].len = trie[p].len + 1;
      while (p != -1 && trie[p][c] == q) {
        trie[p][c] = clone;
        p = trie[p].link;
      }
      trie[q].link = trie[u].link = clone;
    }
  }
  last = u;
}

string kthSubstring(lli kth, int u = 0) {
  // number of different substrings (dp)
  string s = "";
  while (kth > 0)
    for (auto& [c, v] : trie[u]) {
      if (kth <= diff(v)) {
        s.pb(c), kth--, u = v;
        break;
      }
      kth -= diff(v);
    }
  return s;
}

void substringOccurrences() {
  // trie[u].occ = 1, trie[clone].occ = 0
  vi who(sz(trie) - 1);
  iota(all(who), 1);
  sort(all(who), [&](int u, int v) {
    return trie[u].len > trie[v].len;
  });
  for (int u : who) {
    int l = trie[u].link;
    trie[l].occ += trie[u].occ;
  }
}

lli occurences(string& s, int u = 0) {
  for (char c : s) {
    if (!trie[u].count(c))
      return 0;
    u = trie[u][c];
  }
  return trie[u].occ;
}

int longestCommonSubstring(string& s, int u = 0) {
  int mx = 0, clen = 0;
  for (char c : s) {
    while (u && !trie[u].count(c)) {
      u = trie[u].link;
      clen = trie[u].len;
    }
    if (trie[u].count(c))
      u = trie[u][c], clen++;
    mx = max(mx, clen);
  }
  return mx;
}

string smallestCyclicShift(int n, int u = 0) {
  string s = "";
  fore (i, 0, n) {
    char c = trie[u].begin()->f;
    s += c;
    u = trie[u][c];
  }
    return s;
  }

  int leftmost(string& s, int u = 0) {
    for (char c : s) {
      if (!trie[u].count(c))
        return -1;
      u = trie[u][c];
    }
    return trie[u].pos - sz(s) + 1;
  }

  Node& operator[](int u) {
    return trie[u];
  }
};
```

8.8 Aho corasick $\mathcal{O}(\sum s_i)$

```
struct AhoCorasick {
  struct Node : map<char, int> {
    int link = 0, up = 0;
    int cnt = 0, isw = 0;
  };

  vector<Node> trie;

  AhoCorasick(int n = 1) {
    trie.reserve(n), newNode();
  }

  int newNode() {
    trie.pb({});
    return sz(trie) - 1;
  }

  void insert(string& s, int u = 0) {
    for (char c : s) {
      if (!trie[u][c])
        trie[u][c] = newNode();
      u = trie[u][c];
    }
    trie[u].cnt++, trie[u].isw = 1;
  }

  int next(int u, char c) {
    while (u && !trie[u].count(c))
      u = trie[u].link;
    return trie[u][c];
  }

  void pushLinks() {
    queue<int> qu;
    qu.push(0);
    while (!qu.empty()) {
      int u = qu.front();
      qu.pop();
      for (auto& [c, v] : trie[u]) {
        int l = (trie[v].link = u ? next(trie[u].link, c) :
             0);
        trie[v].cnt += trie[l].cnt;
        trie[v].up = trie[l].isw ? l : trie[l].up;
        qu.push(v);
      }
    }
  }

  template <class F>
  void goUp(int u, F f) {
    for (; u != 0; u = trie[u].up)
      f(u);
  }
```

```
  int match(string& s, int u = 0) {
    int ans = 0;
    for (char c : s) {
      u = next(u, c);
      ans += trie[u].cnt;
    }
    return ans;
  }

  Node& operator[](int u) {
    return trie[u];
  }
};
```

8.9 Eertree $\mathcal{O}(\sum s_i)$

```
struct Eertree {
  struct Node : map<char, int> {
    int link = 0, len = 0;
  };

  vector<Node> trie;
  string s = "$";
  int last;

  Eertree(int n = 1) {
    trie.reserve(n), last = newNode(), newNode();
    trie[0].link = 1, trie[1].len = -1;
  }

  int newNode() {
    trie.pb({});
    return sz(trie) - 1;
  }

  int next(int u) {
    while (s[sz(s) - trie[u].len - 2] != s.back())
      u = trie[u].link;
    return u;
  }

  void extend(char c) {
    s.push_back(c);
    last = next(last);
    if (!trie[last][c]) {
      int v = newNode();
      trie[v].len = trie[last].len + 2;
      trie[v].link = trie[next(trie[last].link)][c];
      trie[last][c] = v;
    }
    last = trie[last][c];
  }

  Node& operator[](int u) {
    return trie[u];
  }

  void substringOccurrences() {
    fore (u, sz(s), 0) {
      trie[trie[u].link].occ += trie[u].occ;
    }
  }

  lli occurences(string& s, int u = 0) {
    for (char c : s) {
      if (!trie[u].count(c))
        return 0;
      u = trie[u][c];
    }
    return trie[u].occ;
  }
};
```

9 Dynamic Programming

9.1 All submasks of a mask

```
for (int B = A; B > 0; B = (B - 1) & A)
```

9.2 Matrix Chain Multiplication

```
int dp(int l, int r) {
  if (l > r)
    return 0;
  int& ans = mem[l][r];
  if (!done[l][r]) {
    done[l][r] = true, ans = inf;
    fore (k, l, r + 1) // split in [l, k] [k + 1, r]
      ans = min(ans, dp(l, k) + dp(k + 1, r) + add);
  }
  return ans;
}
```

9.3 Digit DP

Counts the amount of numbers in $[l, r]$ such are divisible by k. (flag *nonzero* is for different lengths)
It can be reduced to $dp(i, x, small)$, and has to be solve like $f(r) - f(l - 1)$

```
#define state [i][x][small][big][nonzero]
int dp(int i, int x, bool small, bool big, bool nonzero) {
  if (i == sz(r))
    return x % k == 0 && nonzero;
  int& ans = mem state;
  if (done state != timer) {
    done state = timer;
    ans = 0;
    int lo = small ? 0 : l[i] - '0';
    int hi = big ? 9 : r[i] - '0';
    fore (y, lo, max(lo, hi) + 1) {
      bool small2 = small | (y > lo);
      bool big2 = big | (y < hi);
      bool nonzero2 = nonzero | (x > 0);
      ans += dp(i + 1, (x * 10 + y) % k, small2, big2,
          nonzero2);
    }
  }
  return ans;
}
```

9.4 Knapsack 0/1

```
for (auto& cur : items)
  fore (w, W + 1, cur.w) // [cur.w, W]
    umax(dp[w], dp[w - cur.w] + cur.cost);
```

9.5 Convex Hull Trick $\mathcal{O}(n^2) \Rightarrow \mathcal{O}(n)$

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

```
// for doubles, use INF = 1/.0, div(a,b) = a / b
struct Line {
  mutable lli m, c, p;
  bool operator<(const Line& l) const {
    return m < l.m;
  }
  bool operator<(lli x) const {
    return p < x;
  }
  lli operator()(lli x) const {
    return m * x + c;
  }
};
```

```
template <bool MAX>
struct DynamicHull : multiset<Line, less<>> {
  lli div(lli a, lli b) {
    return a / b - ((a ^ b) < 0 && a % b);
  }

  bool isect(iterator i, iterator j) {
    if (j == end())
      return i->p = INF, 0;
    if (i->m == j->m)
      i->p = i->c > j->c ? INF : -INF;
    else
      i->p = div(i->c - j->c, j->m - i->m);
    return i->p >= j->p;
  }

  void add(lli m, lli c) {
    if (!MAX)
      m = -m, c = -c;
    auto k = insert({m, c, 0}), j = k++, i = j;
    while (isect(j, k))
      k = erase(k);
    if (i != begin() && isect(--i, j))
      isect(i, j = erase(j));
    while ((j = i) != begin() && (--i)->p >= j->p)
      isect(i, erase(j));
  }
```

9.6 Divide and conquer $\mathcal{O}(kn^2) \Rightarrow \mathcal{O}(k \cdot nlogn)$

Split the array of size n into k continuous groups. $k \leq n$
$cost(a,c) + cost(b,d) \leq cost(a,d) + cost(b,c)$ with $a \leq b \leq c \leq d$

```
void solve(int cut, int l, int r, int optl, int optr) {
  if (r < l)
    return;
  int mid = (l + r) / 2;
  pair<lli, int> best = {INF, -1};
  fore (p, optl, min(mid, optr) + 1)
    best = min(best, {dp[~cut & 1][p - 1] + cost(p, mid), p
        });
  dp[cut & 1][mid] = best.f;
  solve(cut, l, mid - 1, optl, best.s);
  solve(cut, mid + 1, r, best.s, optr);
}

fore (i, 1, n + 1)
  dp[1][i] = cost(1, i);
fore (cut, 2, k + 1)
  solve(cut, cut, n, cut, n);
```

9.7 Knuth optimization $\mathcal{O}(n^3) \Rightarrow \mathcal{O}(n^2)$

$dp[l][r] = \min_{l \leq k \leq r}\{dp[l][k] + dp[k][r]\} + cost(l,r)$

```
fore (len, 1, n + 1)
  fore (l, 0, n) {
    int r = l + len - 1;
    if (r > n - 1)
      break;
    if (len <= 2) {
      dp[l][r] = 0;
      opt[l][r] = l;
      continue;
    }
    dp[l][r] = INF;
    fore (k, opt[l][r - 1], opt[l + 1][r] + 1) {
      lli cur = dp[l][k] + dp[k][r] + cost(l, r);
      if (cur < dp[l][r]) {
        dp[l][r] = cur;
        opt[l][r] = k;
      }
    }
  }
```

10 Game Theory

10.1 Grundy Numbers

If the moves are consecutive $S = \{1, 2, 3, ..., x\}$ the game can be solved like $stackSize \pmod{x+1} \neq 0$

```
int mem[N];

int mex(set<int>& st) {
  int x = 0;
  while (st.count(x))
    x++;
  return x;
}

int grundy(int n) {
  if (n < 0)
    return INF;
  if (n == 0)
    return 0;
  int& g = mem[n];
  if (g == -1) {
    set<int> st;
    for (int x : {a, b})
      st.insert(grundy(n - x));
    g = mex(st);
  }
  return g;
}
```

11 Math

Math table		
Number	Factorial	Catalan
0	1	1
1	1	1
2	2	2
3	6	5
4	24	14
5	120	42
6	720	132
7	5,040	429
8	40,320	1,430
9	362,880	4,862
10	3,628,800	16,796
11	39,916,800	58,786
12	479,001,600	208,012
13	6,227,020,800	742,900

11.1 Factorial

```
fac[0] = 1LL;
fore (i, 1, N)
  fac[i] = lli(i) * fac[i - 1] % mod;
ifac[n - 1] = fpow(fac[n - 1], mod - 2, mod);
for (int i = N - 1; i >= 0; i--)
  ifac[i] = lli(i + 1) * ifac[i + 1] % mod;
```

11.2 Factorial mod $smallPrime$

```
lli facMod(lli n, int p) {
  lli r = 1LL;
  for (; n > 1; n /= p) {
    r = (r * ((n / p) % 2 ? p - 1 : 1)) % p;
    fore (i, 2, n % p + 1)
```

```
    r = r * i % p;
  }
  return r % p;
}
```

11.3 Lucas theorem

Changes $\binom{n}{k}$ mod p, with $n \geq 2e6, k \geq 2e6$ and $p \leq 1e7$

$$\binom{n}{k} \equiv \prod_{i=0}^{n} \binom{n_i}{k_i} \mod p$$

```
lli lucas(lli n, lli k) {
  if (k == 0)
    return 1LL;
  return lucas(n / mod, k / mod) * choose(n % mod, k % mod)
      % mod;
}
```

11.4 Stars and bars

Enclosing n objects in k boxes

$$\binom{n+k-1}{k-1} = \binom{n+k-1}{n}$$

11.5 N choose K

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\binom{n}{k_1, k_2, ..., k_m} = \frac{n!}{k_1! * k_2! * ... * k_m!}$$

```
lli choose(int n, int k) {
  if (n < 0 || k < 0 || n < k)
    return 0LL;
  return fac[n] * ifac[k] % mod * ifac[n - k] % mod;
}

lli choose(int n, int k) {
  lli r = 1;
  int to = min(k, n - k);
  if (to < 0)
    return 0;
  fore (i, 0, to)
    r = r * (n - i) / (i + 1);
  return r;
}
```

11.6 Catalan

```
catalan[0] = 1LL;
fore (i, 0, N) {
  catalan[i + 1] = catalan[i] * lli(4 * i + 2) % mod * fpow
      (i + 2, mod - 2) % mod;
}
```

11.7 Burnside's lemma

$$|classes| = \frac{1}{|G|} \cdot \sum_{x \in G} f(x)$$

11.8 Prime factors of N!

```
vector<ii> factorialFactors(lli n) {
  vector<ii> fac;
  for (auto p : primes) {
    if (n < p)
      break;
    lli mul = 1LL, k = 0;
    while (mul <= n / p) {
      mul *= p;
      k += n / mul;
    }
    fac.emplace_back(p, k);
  }
  return fac;
}
```

12 Number Theory

12.1 Goldbach conjecture

- All number ≥ 6 can be written as sum of 3 *primes*
- All even number > 2 can be written as sum of 2 *primes*

12.2 Prime numbers distribution

Amount of primes approximately $\frac{n}{\ln(n)}$

12.3 Sieve of Eratosthenes $\mathcal{O}(N \cdot log(logN))$

To factorize divide x by $factor[x]$ until is equal to 1

```
void factorizeSieve() {
  iota(factor, factor + N, 0);
  for (int i = 2; i * i < N; i++)
    if (factor[i] == i)
      for (int j = i * i; j < N; j += i)
        factor[j] = i;
}

map<int, int> factorize(int n) {
  map<int, int> cnt;
  while (n > 1) {
    cnt[factor[n]]++;
    n /= factor[n];
  }
  return cnt;
}
```

Use it if you need a huge amount of $phi[x]$ up to some N

```
void phiSieve() {
  isPrime.set();
  iota(phi, phi + N, 0);
  fore (i, 2, N)
    if (isPrime[i])
      for (int j = i; j < N; j += i) {
        isPrime[j] = (i == j);
        phi[j] = phi[j] / i * (i - 1);
      }
}
```

12.4 Phi of euler $\mathcal{O}(\sqrt{N})$

```
lli phi(lli n) {
  if (n == 1)
    return 0;
  lli r = n;
  for (lli i = 2; i * i <= n; i++)
    if (n % i == 0) {
      while (n % i == 0)
        n /= i;
      r -= r / i;
    }
  if (n > 1)
    r -= r / n;
  return r;
}
```

12.5 Miller-Rabin $\mathcal{O}(Witnesses \cdot (logN)^3)$

```
ull mul(ull x, ull y, ull mod) {
  lli ans = x * y - mod * ull(1.L / mod * x * y);
  return ans + mod * (ans < 0) - mod * (ans >= lli(mod));
}

// use mul(x, y, mod) inside fpow
bool miller(ull n) {
```

```
  if (n < 2 || n % 6 % 4 != 1)
    return (n | 1) == 3;
  ull k = __builtin_ctzll(n - 1), d = n >> k;
  for (ull p : {2, 325, 9375, 28178, 450775, 9780504, 17952
       65022}) {
    ull x = fpow(p % n, d, n), i = k;
    while (x != 1 && x != n - 1 && p % n && i--)
      x = mul(x, x, n);
    if (x != n - 1 && i != k)
      return 0;
  }
  return 1;
}
```

12.6 Pollard-Rho $\mathcal{O}(N^{1/4})$

```
ull rho(ull n) {
  auto f = [n](ull x) {
    return mul(x, x, n) + 1;
  };
  ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
  while (t++ % 40 || __gcd(prd, n) == 1) {
    if (x == y)
      x = ++i, y = f(x);
    if (q = mul(prd, max(x, y) - min(x, y), n))
      prd = q;
    x = f(x), y = f(f(y));
  }
  return __gcd(prd, n);
}

// if used multiple times, try memorization!!
// try factoring small numbers with sieve
void pollard(ull n, map<ull, int>& fac) {
  if (n == 1)
    return;
  if (miller(n)) {
    fac[n]++;
  } else {
    ull x = rho(n);
    pollard(x, fac);
    pollard(n / x, fac);
  }
}
```

12.7 Amount of divisors $\mathcal{O}(N^{1/3})$

```
ull amountOfDivisors(ull n) {
  ull cnt = 1;
  for (auto p : primes) {
    if (1LL * p * p * p > n)
      break;
    if (n % p == 0) {
      ull k = 0;
      while (n > 1 && n % p == 0)
        n /= p, ++k;
      cnt *= (k + 1);
    }
  }
  ull sq = mysqrt(n); // the last x * x <= n
  if (miller(n))
    cnt *= 2;
  else if (sq * sq == n && miller(sq))
    cnt *= 3;
  else if (n > 1)
    cnt *= 4;
  return cnt;
}
```

12.8 Bézout's identity

$a_1 * x_1 + a_2 * x_2 + ... + a_n * x_n = g$
$g = \gcd(a_1, a_2, ..., a_n)$

12.9 GCD

$a \le b; \gcd(a + k, b + k) = \gcd(b - a, a + k)$

12.10 LCM

$x = p * lcm(a_1, a_2, ..., a_k) + q, 0 \le q \le lcm(a_1, a_2, ..., a_k)$
$x \pmod{a_i} \equiv q \pmod{a_i}$ as $a_i \mid lcm(a_1, a_2, ..., a_k)$

12.11 Euclid $\mathcal{O}(log(a \cdot b))$

```
pair<lli, lli> euclid(lli a, lli b) {
  if (b == 0)
    return {1, 0};
  auto p = euclid(b, a % b);
  return {p.s, p.f - a / b * p.s};
}
```

12.12 Chinese remainder theorem

```
pair<lli, lli> crt(pair<lli, lli> a, pair<lli, lli> b) {
  if (a.s < b.s)
    swap(a, b);
  auto p = euclid(a.s, b.s);
  lli g = a.s * p.f + b.s * p.s, l = a.s / g * b.s;
  if ((b.f - a.f) % g != 0)
    return {-1, -1}; // no solution
  p.f = a.f + (b.f - a.f) % b.s * p.f % b.s / g * a.s;
  return {p.f + (p.f < 0) * l, l};
}
```

13 Math

13.1 Progressions

Arithmetic progressions

$a_n = a_1 + (n - 1) * diff$
$\sum_{i=1}^{n} a_i = n * \frac{a_1 + a_n}{2}$

Geometric progressions

$a_n = a_1 * r^{n-1}$
$\sum_{k=1}^{n} a_1 * r^k = a_1 * \left(\frac{r^{n+1} - 1}{r - 1}\right) : r \neq 1$

13.2 Fpow

```
template <class T>
T fpow(T x, lli n) {
  T r(1);
  for (; n > 0; n >>= 1) {
    if (n & 1)
      r = r * x;
    x = x * x;
  }
  return r;
}
```

13.3 Fibonacci

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} fib_{n+1} & fib_n \\ fib_n & fib_{n-1} \end{bmatrix}$$

14 Probability

14.1 Conditional

The event A happens and the event B has already happened

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

If **independent** events

$$P(A|B) = P(A), P(B|A) = P(B)$$

14.2 Bayes theorem

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

14.3 Binomial

$$B = \binom{n}{x} \cdot p^x \cdot (1-p)^{n-x}$$

n = number of trials
x = number of **success** from n trials
p = probability of **success** on a single trial

14.4 Geometric

Probability of success at the nth-event after failing the others

$$G = (1-p)^{n-1} \cdot p$$

n = number of trials
p = probability of *success* on a single trial

14.5 Poisson

$$Po = \frac{\lambda^k \cdot e^{-\lambda}}{k!}$$

λ = number of times an event is expected (occurs / time)
k = number of occurring events in the limited period of time

Example: The event happens 4 times per minute and we want k events to happen in 10 minutes, then $\lambda = 4 \cdot 10 = 40$

14.6 Expected value

$$E_x = \sum_{\forall x} x \cdot p(x)$$

15 Bit tricks

15.1 Xor Basis

Keeps the set of all xors among all possible subsets

```
template <int D>
struct XorBasis {
  array<int, D> basis;
  int n = 0;

  XorBasis() {
    basis.fill(0);
  }

  bool insert(int x) {
    fore (i, D, 0)
      if ((x >> i) & 1) {
        if (!basis[i]) {
          basis[i] = x, n++;
          return 1;
        }
        x ^= basis[i];
      }
    return 0;
  }

  optional<int> find(int x) {
    // which number is needed to generate x
    // num ^ (num ^ x) = x
    int num = 0;
    fore (i, D, 0)
      if ((x >> i) & 1) {
        if (!basis[i])
          return nullopt;
        x ^= basis[i];
        num |= (1 << i);
      }
    return optional(num);
  }

  optional<int> operator[](lli k) {
    lli tot = (1LL << n);
    if (k > tot)
      return nullopt;
    int num = 0;
    fore (i, D, 0)
      if (basis[i]) {
        lli low = tot / 2;
        if ((low < k && ((num >> i) & 1) == 0) || (low >= k
            && ((num >> i) & 1)))
          num ^= basis[i];
        if (low < k)
          k -= low;
        tot /= 2;
      }
    return optional(num);
  }
};
```

Bits++	
Operations on *int*	Function
`x & -x`	Least significant bit in x
`__lg(x)`	Most significant bit in x
`c = x&-x, r = x+c; (((r^x) » 2)/c) | r`	Next number after x with same number of bits set
`__builtin_`	Function
popcount(x)	Amount of 1's in x
clz(x)	0's to the **left** of biggest bit
ctz(x)	0's to the **right** of smallest bit

15.2 Bitset

Bitset<Size>	
Operation	Function
_Find_first()	Least significant bit
_Find_next(idx)	First set bit after index *idx*
any(), none(), all()	Just what the expression says
set(), reset(), flip()	Just what the expression says x2
to_string('.', 'A')	Print 011010 like .AA.A.

16 Geometry

```
const ld EPS = 1e-20;
const ld INF = 1e18;
const ld PI = acos(-1.0);
enum { ON = -1, OUT, IN, OVERLAP };

#define eq(a, b) (abs((a) - (b)) <= +EPS)
#define neq(a, b) (!eq(a, b))
#define geq(a, b) ((a) - (b) >= -EPS)
#define leq(a, b) ((a) - (b) <= +EPS)
#define ge(a, b) ((a) - (b) > +EPS)
#define le(a, b) ((a) - (b) < -EPS)

int sgn(ld a) {
  return (a > EPS) - (a < -EPS);
}
```

17 Points

17.1 Points

```
struct Pt {
  ld x, y;
  explicit Pt(ld x = 0, ld y = 0) : x(x), y(y) {}

  Pt operator+(Pt p) const {
    return Pt(x + p.x, y + p.y);
  }

  Pt operator-(Pt p) const {
    return Pt(x - p.x, y - p.y);
  }
```

```
  Pt operator*(ld k) const {
    return Pt(x * k, y * k);
  }

  Pt operator/(ld k) const {
    return Pt(x / k, y / k);
  }

  ld dot(Pt p) const {
    // 0 if vectors are orthogonal
    // - if vectors are pointing in opposite directions
    // + if vectors are pointing in the same direction
    return x * p.x + y * p.y;
  }

  ld cross(Pt p) const {
    // 0 if collinear
    // - if b is to the right of a
    // + if b is to the left of a
    // gives you 2 * area
    return x * p.y - y * p.x;
  }

  ld norm() const {
    return x * x + y * y;
  }

  ld length() const {
    return sqrtl(norm());
  }

  Pt unit() const {
    return (*this) / length();
  }

  ld angle() const {
    ld ang = atan2(y, x);
    return ang + (ang < 0 ? 2 * acos(-1) : 0);
  }
```

17.2 Angle between vectors

```
double angleBetween(Pt a, Pt b) {
  double x = a.dot(b) / a.length() / b.length();
  return acosl(max(-1.0, min(1.0, x)));
}
```

17.3 Closest pair of points $\mathcal{O}(N \cdot logN)$

```
pair<Pt, Pt> closestPairOfPoints(vector<Pt>& pts) {
  sort(all(pts), [&](Pt a, Pt b) {
    return le(a.y, b.y);
  });
  set<Pt> st;
  ld ans = INF;
  Pt p, q;
  int pos = 0;
  fore (i, 0, sz(pts)) {
    while (pos < i && geq(pts[i].y - pts[pos].y, ans))
      st.erase(pts[pos++]);
    auto lo = st.lower_bound(Pt(pts[i].x - ans - eps, -INF)
        );
    auto hi = st.upper_bound(Pt(pts[i].x + ans + eps, -INF)
        );
    for (auto it = lo; it != hi; ++it) {
      ld d = (pts[i] - *it).length();
      if (le(d, ans))
        ans = d, p = pts[i], q = *it;
    }
    st.insert(pts[i]);
  }
  return {p, q};
}
```

17.4 Projection

```
ld proj(Pt a, Pt b) {
  return a.dot(b) / b.length();
}
```

17.5 KD-Tree

build: $\mathcal{O}(N \cdot logN)$, nearest: $\mathcal{O}(logN)$

```
struct KDTree {
// p.pos(0) = x, p.pos(1) = y, p.pos(2) = z
#define iter Pt* // vector<Pt>::iterator
  KDTree *left, *right;
  Pt p;
  ld val;
  int k;

  KDTree(iter b, iter e, int k = 0) : k(k), left(0), right(
      0) {
    int n = e - b;
    if (n == 1) {
      p = *b;
      return;
    }
    nth_element(b, b + n / 2, e, [&](Pt a, Pt b) {
      return a.pos(k) < b.pos(k);
    });
    val = (b + n / 2)->pos(k);
    left = new KDTree(b, b + n / 2, (k + 1) % 2);
    right = new KDTree(b + n / 2, e, (k + 1) % 2);
  }

  pair<ld, Pt> nearest(Pt q) {
    if (!left && !right) // take care if is needed a
        different one
      return make_pair((p - q).norm(), p);
    pair<ld, Pt> best;
    if (q.pos(k) <= val) {
      best = left->nearest(q);
      if (geq(q.pos(k) + sqrt(best.f), val))
        best = min(best, right->nearest(q));
    } else {
      best = right->nearest(q);
      if (leq(q.pos(k) - sqrt(best.f), val))
        best = min(best, left->nearest(q));
    }
    return best;
  }
};
```

18 Lines and segments

18.1 Line

```
struct Line {
  Pt a, b, v;

  Line() {}
  Line(Pt a, Pt b) : a(a), b(b), v((b - a).unit()) {}

  bool contains(Pt p) {
    return eq((p - a).cross(b - a), 0);
  }

  int intersects(Line l) {
    if (eq(v.cross(l.v), 0))
      return eq((l.a - a).cross(v), 0) ? INF : 0;
    return 1;
  }

  int intersects(Seg s) {
```

```
    if (eq(v.cross(s.v), 0))
      return eq((s.a - a).cross(v), 0) ? INF : 0;
    return sgn(v.cross(s.a - a)) != sgn(v.cross(s.b - a));
  }

  template <class Line>
  Pt intersection(Line l) { // can be a segment too
    return a + v * ((l.a - a).cross(l.v) / v.cross(l.v));
  }

  Pt projection(Pt p) {
    return a + v * proj(p - a, v);
  }

  Pt reflection(Pt p) {
    return a * 2 - p + v * 2 * proj(p - a, v);
  }
};
```

18.2 Segment

```
struct Seg {
  Pt a, b, v;

  Seg() {}
  Seg(Pt a, Pt b) : a(a), b(b), v(b - a) {}

  bool contains(Pt p) {
    return eq(v.cross(p - a), 0) && leq((a - p).dot(b - p),
         0);
  }

  int intersects(Seg s) {
    int t1 = sgn(v.cross(s.a - a)), t2 = sgn(v.cross(s.b -
        a));
    if (t1 == t2)
      return t1 == 0 && (contains(s.a) || contains(s.b) ||
          s.contains(a) || s.contains(b)) ? INF





    return sgn(s.v.cross(a - s.a)) != sgn(s.v.cross(b - s.a
        ));
  }

  template <class Seg>
  Pt intersection(Seg s) { // can be a line too
    return a + v * ((s.a - a).cross(s.v) / v.cross(s.v));
  }
};
```

18.3 Distance point-line

```
ld distance(Pt p, Line l) {
  Pt q = l.projection(p);
  return (p - q).length();
}
```

18.4 Distance point-segment

```
ld distance(Pt p, Seg s) {
  if (le((p - s.a).dot(s.b - s.a), 0))
    return (p - s.a).length();
  if (le((p - s.b).dot(s.a - s.b), 0))
    return (p - s.b).length();
  return abs((s.a - p).cross(s.b - p) / (s.b - s.a).length
      ());
}
```

18.5 Distance segment-segment

```
ld distance(Seg a, Seg b) {
  if (a.intersects(b))
    return 0.L;
  return min({distance(a.a, b), distance(a.b, b), distance(
      b.a, a), distance(b.b, a)});
}
```

19 Circles

19.1 Circle

```
struct Cir {
  Pt o;
  ld r;
  Cir() {}
  Cir(ld x, ld y, ld r) : o(x, y), r(r) {}
  Cir(Pt o, ld r) : o(o), r(r) {}

  int inside(Cir c) {
    ld l = c.r - r - (o - c.o).length();
    return ge(l, 0) ? IN : eq(l, 0) ? ON : OVERLAP;
  }

  int outside(Cir c) {
    ld l = (o - c.o).length() - r - c.r;
    return ge(l, 0) ? OUT : eq(l, 0) ? ON : OVERLAP;
  }

  int contains(Pt p) {
    ld l = (p - o).length() - r;
    return le(l, 0) ? IN : eq(l, 0) ? ON : OUT;
  }

  Pt projection(Pt p) {
    return o + (p - o).unit() * r;
  }

  vector<Pt> tangency(Pt p) {
    // point outside the circle
    Pt v = (p - o).unit() * r;
    ld d2 = (p - o).norm(), d = sqrt(d2);
    if (leq(d, 0))
      return {}; // on circle, no tangent
    Pt v1 = v * (r / d), v2 = v.perp() * (sqrt(d2 - r * r)
         / d);
    return {o + v1 - v2, o + v1 + v2};
  }

  vector<Pt> intersection(Cir c) {
    ld d = (c.o - o).length();
    if (eq(d, 0) || ge(d, r + c.r) || le(d, abs(r - c.r)))
      return {}; // circles don't intersect
    Pt v = (c.o - o).unit();
    ld a = (r * r + d * d - c.r * c.r) / (2 * d);
    Pt p = o + v * a;
    if (eq(d, r + c.r) || eq(d, abs(r - c.r)))
      return {p}; // circles touch at one point
    ld h = sqrt(r * r - a * a);
    Pt q = v.perp() * h;
    return {p - q, p + q}; // circles intersects twice
  }

  template <class Line>
  vector<Pt> intersection(Line l) {
    // for a segment you need to check that the point lies
         on the segment
    ld h2 = r * r - l.v.cross(o - l.a) * l.v.cross(o - l.a)
          / l.v.norm();
    Pt p = l.a + l.v * l.v.dot(o - l.a) / l.v.norm();
    if (eq(h2, 0))
      return {p}; // line tangent to circle
    if (le(h2, 0))
      return {}; // no intersection
    Pt q = l.v.unit() * sqrt(h2);
```

```
    return {p - q, p + q}; // two points of intersection (
        chord)
  }

  Cir(Pt a, Pt b, Pt c) {
    // find circle that passes through points a, b, c
    Pt mab = (a + b) / 2, mcb = (b + c) / 2;
    Seg ab(mab, mab + (b - a).perp());
    Seg cb(mcb, mcb + (b - c).perp());
    o = ab.intersection(cb);
    r = (o - a).length();
  }

  ld commonArea(Cir c) {
    if (le(r, c.r))
      return c.commonArea(*this);
    ld d = (o - c.o).length();
    if (leq(d + c.r, r))
      return c.r * c.r * PI;
    if (geq(d, r + c.r))
      return 0.0;
    auto angle = [&](ld a, ld b, ld c) {
      return acos((a * a + b * b - c * c) / (2 * a * b));
    };
    auto cut = [&](ld a, ld r) {
      return (a - sin(a)) * r * r / 2;
    };
    ld a1 = angle(d, r, c.r), a2 = angle(d, c.r, r);
    return cut(a1 * 2, r) + cut(a2 * 2, c.r);
  }
};
```

19.2 Distance point-circle

```
ld distance(Pt p, Cir c) {
  return max(0.L, (p - c.o).length() - c.r);
}
```

19.3 Minimum enclosing circle $\mathcal{O}(N)$ wow!!

```
Cir minEnclosing(vector<Pt>& pts) { // a bunch of points
  shuffle(all(pts), rng);
  Cir c(0, 0, 0);
  fore (i, 0, sz(pts))
    if (!c.contains(pts[i])) {
      c = Cir(pts[i], 0);
      fore (j, 0, i)
        if (!c.contains(pts[j])) {
          c = Cir((pts[i] + pts[j]) / 2, (pts[i] - pts[j]).
              length() / 2);
          fore (k, 0, j)
            if (!c.contains(pts[k]))
              c = Cir(pts[i], pts[j], pts[k]);
        }
    }
  return c;
}
```

19.4 Common area circle-polygon $\mathcal{O}(N)$

```
ld commonArea(const Cir& c, const vector<Pt>& poly) {
  auto arg = [&](Pt p, Pt q) {
    return atan2(p.cross(q), p.dot(q));
  };
  auto tri = [&](Pt p, Pt q) {
    Pt d = q - p;
    ld a = d.dot(p) / d.norm(), b = (p.norm() - c.r * c.r)
        / d.norm();
    ld det = a * a - b;
    if (leq(det, 0))
      return arg(p, q) * c.r * c.r;
    ld s = max(0.L, -a - sqrt(det)), t = min(1.L, -a + sqrt
        (det));
    if (t < 0 || 1 <= s)
      return arg(p, q) * c.r * c.r;
    Pt u = p + d * s, v = p + d * t;
    return u.cross(v) + (arg(p, u) + arg(v, q)) * c.r * c.r
        ;
  };
  ld sum = 0;
  fore (i, 0, sz(poly))
    sum += tri(poly[i] - c.o, poly[(i + 1) % sz(poly)] - c.
        o);
  return abs(sum / 2);
}
```

20 Polygons

20.1 Area of polygon $\mathcal{O}(N)$

```
ld area(const vector<Pt>& pts) {
  ld sum = 0;
  fore (i, 0, sz(pts))
    sum += pts[i].cross(pts[(i + 1) % sz(pts)]);
  return abs(sum / 2);
}
```

20.2 Convex-Hull $\mathcal{O}(N \cdot logN)$

```
vector<Pt> convexHull(vector<Pt> pts) {
  vector<Pt> low, up;
  sort(all(pts), [&](Pt a, Pt b) {
    return a.x == b.x ? a.y < b.y : a.x < b.x;
  });
  pts.erase(unique(all(pts)), pts.end());
  if (sz(pts) <= 2)
    return pts;
  fore (i, 0, sz(pts)) {
    while (sz(low) >= 2 && (low.end()[-1] - low.end()[-2]).
        cross(pts[i] - low.end()[-1]) <= 0)
      low.pop_back();
    low.pb(pts[i]);
  }
  fore (i, sz(pts), 0) {
    while (sz(up) >= 2 && (up.end()[-1] - up.end()[-2]).
        cross(pts[i] - up.end()[-1]) <= 0)
      up.pop_back();
    up.pb(pts[i]);
  }
  low.pop_back(), up.pop_back();
  low.insert(low.end(), all(up));
  return low;
}
```

20.3 Cut polygon by a line $\mathcal{O}(N)$

```
vector<Pt> cut(const vector<Pt>& pts, Line l) {
  vector<Pt> ans;
  int n = sz(pts);
  fore (i, 0, n) {
    int j = (i + 1) % n;
    if (geq(l.v.cross(pts[i] - l.a), 0)) // left
      ans.pb(pts[i]);
    Seg s(pts[i], pts[j]);
    if (l.intersects(s) == 1) {
      Pt p = l.intersection(s);
      if (p != pts[i] && p != pts[j])
        ans.pb(p);
    }
  }
  return ans;
}
```

20.4 Perimeter $\mathcal{O}(N)$

```
ld perimeter(const vector<Pt>& pts) {
  ld sum = 0;
  fore (i, 0, sz(pts))
    sum += (pts[(i + 1) % sz(pts)] - pts[i]).length();
  return sum;
```

```
}
```

20.5 Point in polygon $\mathcal{O}(N)$

```
int contains(const vector<Pt>& pts, Pt p) {
  int rays = 0, n = sz(pts);
  fore (i, 0, n) {
    Pt a = pts[i], b = pts[(i + 1) % n];
    if (ge(a.y, b.y))
      swap(a, b);
    if (Seg(a, b).contains(p))
      return ON;
    rays ^= (leq(a.y, p.y) && le(p.y, b.y) && ge((a - p).
        cross(b - p), 0));
  }
  return rays & 1 ? IN : OUT;
}
```

20.6 Point in convex-polygon $\mathcal{O}(logN)$

```
bool contains(const vector<Pt>& a, Pt p) {
  int lo = 1, hi = sz(a) - 1;
  if (a[0].dir(a[lo], a[hi]) > 0)
    swap(lo, hi);
  if (p.dir(a[0], a[lo]) >= 0 || p.dir(a[0], a[hi]) <= 0)
    return false;
  while (abs(lo - hi) > 1) {
    int mid = (lo + hi) >> 1;
    (p.dir(a[0], a[mid]) > 0 ? hi : lo) = mid;
  }
  return p.dir(a[lo], a[hi]) < 0;
}
```

20.7 Is convex $\mathcal{O}(N)$

```
bool isConvex(const vector<Pt>& pts) {
  int n = sz(pts);
  bool pos = 0, neg = 0;
  fore (i, 0, n) {
    Pt a = pts[(i + 1) % n] - pts[i];
    Pt b = pts[(i + 2) % n] - pts[(i + 1) % n];
    int dir = sgn(a.cross(b));
    if (dir > 0)
      pos = 1;
    if (dir < 0)
      neg = 1;
  }
  return !(pos && neg);
}
```

21 Geometry misc

21.1 Radial order

```
struct Radial {
  Pt c;
  Radial(Pt c) : c(c) {}

  int cuad(Pt p) const {
    if (p.x > 0 && p.y >= 0)
      return 0;
    if (p.x <= 0 && p.y > 0)
      return 1;
    if (p.x < 0 && p.y <= 0)
      return 2;
    if (p.x >= 0 && p.y < 0)
      return 3;
    return -1;
  }

  bool operator()(Pt a, Pt b) const {
    Pt p = a - c, q = b - c;
    if (cuad(p) == cuad(q))
      return p.y * q.x < p.x * q.y;
    return cuad(p) < cuad(q);
  }
};
```

21.2 Sort along a line $\mathcal{O}(N \cdot logN)$

```
void sortAlongLine(vector<Pt>& pts, Line l) {
  sort(all(pts), [&](Pt a, Pt b) {
    return a.dot(l.v) < b.dot(l.v);
  });
}
```