# Nea1's Code Library

Nea1

ORZ

He is Nea1

# Contents

# Intro

## Main template

```cpp
#include <bits/stdc++.h>
using namespace std;

#define FOR(x,n) for(int x=0;x<n;x++)
#define forn(i, n) for (int i = 0; i < int(n); i++)
#define all(v) v.begin(),v.end()
using ll = long long;
using ld = long double;
using pii = pair<int, int>;
const char nl = '\n';

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cout << fixed << setprecision(20);
  // mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
}
```

## Fast IO

```cpp
namespace io {
constexpr int SIZE = 1 << 16;
char buf[SIZE], *head, *tail;
char get_char() {
  if (head == tail) tail = (head = buf) + fread(buf, 1, SIZE, stdin);
  return *head++;
}
ll read() {
  ll x = 0, f = 1;
  char c = get_char();
  for (; !isdigit(c); c = get_char()) (c == '-') && (f = -1);
  for (; isdigit(c); c = get_char()) x = x * 10 + c - '0';
  return x * f;
}
string read_s() {
  string str;
  char c = get_char();
  while (c == ' ' || c == '\n' || c == '\r') c = get_char();
  while (c != ' ' && c != '\n' && c != '\r') str += c, c = get_char();
  return str;
}
void print(int x) {
  if (x > 9) print(x / 10);
  putchar(x % 10 | '0');
}
void println(int x) { print(x), putchar('\n'); }
struct Read {
  Read& operator>>(ll& x) { return x = read(), *this; }
  Read& operator>>(long double& x) { return x = stold(read_s()), *this; }
} in;
}  // namespace io
```

## Pragmas (lol)

```cpp
#pragma GCC optimize(2)
#pragma GCC optimize(3)
#pragma GCC optimize("Ofast")
#pragma GCC optimize("inline")
#pragma GCC optimize("-fgcse")
#pragma GCC optimize("-fgcse-lm")
#pragma GCC optimize("-fipa-sra")
#pragma GCC optimize("-ftree-pre")
#pragma GCC optimize("-ftree-vrp")
#pragma GCC optimize("-fpeephole2")
#pragma GCC optimize("-ffast-math")
#pragma GCC optimize("-fsched-spec")
#pragma GCC optimize("unroll-loops")
```

```cpp
#pragma GCC optimize("-falign-jumps")
#pragma GCC optimize("-falign-loops")
#pragma GCC optimize("-falign-labels")
#pragma GCC optimize("-fdevirtualize")
#pragma GCC optimize("-fcaller-saves")
#pragma GCC optimize("-fcrossjumping")
#pragma GCC optimize("-fthread-jumps")
#pragma GCC optimize("-funroll-loops")
#pragma GCC optimize("-fwhole-program")
#pragma GCC optimize("-freorder-blocks")
#pragma GCC optimize("-fschedule-insns")
#pragma GCC optimize("inline-functions")
#pragma GCC optimize("-ftree-tail-merge")
#pragma GCC optimize("-fschedule-insns2")
#pragma GCC optimize("-fstrict-aliasing")
#pragma GCC optimize("-fstrict-overflow")
#pragma GCC optimize("-falign-functions")
#pragma GCC optimize("-fcse-skip-blocks")
#pragma GCC optimize("-fcse-follow-jumps")
#pragma GCC optimize("-fsched-interblock")
#pragma GCC optimize("-fpartial-inlining")
#pragma GCC optimize("no-stack-protector")
#pragma GCC optimize("-freorder-functions")
#pragma GCC optimize("-findirect-inlining")
#pragma GCC optimize("-fhoist-adjacent-loads")
#pragma GCC optimize("-frerun-cse-after-loop")
#pragma GCC optimize("inline-small-functions")
#pragma GCC optimize("-finline-small-functions")
#pragma GCC optimize("-ftree-switch-conversion")
#pragma GCC optimize("-foptimize-sibling-calls")
#pragma GCC optimize("-fexpensive-optimizations")
#pragma GCC optimize("-funsafe-loop-optimizations")
#pragma GCC optimize("inline-functions-called-once")
#pragma GCC optimize("-fdelete-null-pointer-checks")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4.1,sse4.2,avx,avx2,popcnt,tune=native")
```

# Data Structures

## Segment Tree

### Recursive

- Implicit segment tree, range query + point update

```cpp
struct Node {
  int lc, rc, p;
};

struct SegTree {
  vector<Node> t = {{}};
  SegTree(int n) { t.reserve(n * 40); }
  int modify(int p, int l, int r, int x, int v) {
    int u = p;
    if (p == 0) {
      t.push_back(t[p]);
      u = (int)t.size() - 1;
    }
    if (r - l == 1) {
      t[u].p = t[p].p + v;
    } else {
      int m = (l + r) / 2;
      if (x < m) {
        t[u].lc = modify(t[p].lc, l, m, x, v); // ub before c++17
      } else {
        t[u].rc = modify(t[p].rc, m, r, x, v);
      }
      t[u].p = t[t[u].lc].p + t[t[u].rc].p;
    }
    return u;
  }
```

```
  int query(int p, int l, int r, int x, int y) {
    if (x <= l && r <= y) return t[p].p;
    int m = (l + r) / 2, res = 0;
    if (x < m) res += query(t[p].lc, l, m, x, y);
    if (y > m) res += query(t[p].rc, m, r, x, y);
    return res;
  }
};
```

- Persistent implicit, range query + point update

```
struct Node {
  int lc = 0, rc = 0, p = 0;
};

struct SegTree {
  vector<Node> t = {{}};  // init all
  SegTree() = default;
  SegTree(int n) { t.reserve(n * 20); }
  int modify(int p, int l, int r, int x, int v) {
    // p: original node, update a[x] -> v
    t.push_back(t[p]);
    int u = (int)t.size() - 1;
    if (r - l == 1) {
      t[u].p = v;
    } else {
      int m = (l + r) / 2;
      if (x < m) {
        t[u].lc = modify(t[p].lc, l, m, x, v);
        t[u].rc = t[p].rc;
      } else {
        t[u].lc = t[p].lc;
        t[u].rc = modify(t[p].rc, m, r, x, v);
      }
      t[u].p = t[t[u].lc].p + t[t[u].rc].p;
    }
    return u;
  }
  int query(int p, int l, int r, int x, int y) {
    // query sum a[x]...a[y-1] rooted at p
    // t[p] holds the info of [l, r)
    if (x <= l && r <= y) return t[p].p;
    int m = (l + r) / 2, res = 0;
    if (x < m) res += query(t[p].lc, l, m, x, y);
    if (y > m) res += query(t[p].rc, m, r, x, y);
    return res;
  }
};
```

### Iterating

- Iterating, range query + point update

```
struct Node {
  ll v = 0, init = 0;
};

Node pull(const Node &a, const Node &b) {
  if (!a.init) return b;
  if (!b.init) return a;
  Node c;
  return c;
}

struct SegTree {
  ll n;
  vector<Node> t;
  SegTree(ll _n) : n(_n), t(2 * n){};
  void modify(ll p, const Node &v) {
    t[p += n] = v;
    for (p /= 2; p; p /= 2) t[p] = pull(t[p * 2], t[p * 2 + 1]);
```

```
  }
  Node query(ll l, ll r) {
    Node left, right;
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
      if (l & 1) left = pull(left, t[l++]);
      if (r & 1) right = pull(t[--r], right);
    }
    return pull(left, right);
  }
};
```

- Iterating, range query + range update

```
struct SegTree {
  ll n, h = 0;
  vector<Node> t;
  SegTree(ll _n) : n(_n), h((ll)log2(n)), t(n * 2) {}
  void apply(ll x, ll v) {
    if (v == 0) {
      t[x].one = 0;
    } else {
      t[x].one = t[x].total;
    }
    t[x].lazy = v;
  }
  void build(ll l) {
    for (l = (l + n) / 2; l > 0; l /= 2) {
      if (t[l].lazy == -1) {
        t[l] = pull(t[l * 2], t[l * 2 + 1]);
      }
    }
  }
  void push(ll l) {
    l += n;
    for (ll s = h; s > 0; s--) {
      ll i = l >> s;
      if (t[i].lazy != -1) {
        apply(2 * i, t[i].lazy);
        apply(2 * i + 1, t[i].lazy);
      }
      t[i].lazy = -1;
    }
  }
  void modify(ll l, ll r, int v) {
    push(l), push(r - 1);
    ll l0 = l, r0 = r;
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
      if (l & 1) apply(l++, v);
      if (r & 1) apply(--r, v);
    }
    build(l0), build(r0 - 1);
  }
  Node query(ll l, ll r) {
    push(l), push(r - 1);
    Node left, right;
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
      if (l & 1) left = pull(left, t[l++]);
      if (r & 1) right = pull(t[--r], right);
    }
    return pull(left, right);
  }
};
```

- AtCoder Segment Tree (recursive structure but iterative)

```
template <class T> struct PointSegmentTree {
  int size = 1;
  vector<T> tree;
  PointSegmentTree(int n) : PointSegmentTree(vector<T>(n)) {}
  PointSegmentTree(vector<T>& arr) {
    while(size < (int)arr.size())
      size <<= 1;
```

```
    tree = vector<T>(size << 1);
    for(int i = size + arr.size() - 1; i >= 1; i--)
      if(i >= size) tree[i] = arr[i - size];
      else consume(i);
  }
  void set(int i, T val) {
    tree[i += size] = val;
    for(i >>= 1; i >= 1; i >>= 1)
      consume(i);
  }
  T get(int i) { return tree[i + size]; }
  T query(int l, int r) {
    T resl, resr;
    for(l += size, r += size + 1; l < r; l >>= 1, r >>= 1) {
      if(l & 1) resl = resl * tree[l++];
      if(r & 1) resr = tree[--r] * resr;
    }
    return resl * resr;
  }
  T query_all() { return tree[1]; }
  void consume(int i) { tree[i] = tree[i << 1] * tree[i << 1 | 1]; }
};


struct SegInfo {
  ll v;
  SegInfo() : SegInfo(0) {}
  SegInfo(ll val) : v(val) {}
  SegInfo operator*(SegInfo b) {
    return SegInfo(v + b.v);
  }
};
```

## Union Find

```
vector<int> p(n);
iota(p.begin(), p.end(), 0);
function<int(int)> find = [&](int x) { return p[x] == x ? x : (p[x] = find(p[x])); };
auto merge = [&](int x, int y) { p[find(x)] = find(y); };
```

- Persistent version

```
struct Node {
  int lc, rc, p;
};

struct SegTree {
  vector<Node> t = {{0, 0, -1}};  // init all
  SegTree() = default;
  SegTree(int n) { t.reserve(n * 20); }
  int modify(int p, int l, int r, int x, int v) {
    // p: original node, update a[x] -> v
    t.push_back(t[p]);
    int u = (int)t.size() - 1;
    if (r - l == 1) {
      t[u].p = v;
    } else {
      int m = (l + r) / 2;
      if (x < m) {
        t[u].lc = modify(t[p].lc, l, m, x, v);
        t[u].rc = t[p].rc;
      } else {
        t[u].lc = t[p].lc;
        t[u].rc = modify(t[p].rc, m, r, x, v);
      }
      t[u].p = t[t[u].lc].p + t[t[u].rc].p;
    }
    return u;
  }
  int query(int p, int l, int r, int x, int y) {
    // query sum a[x]...a[y-1] rooted at p
```

```
    // t[p] holds the info of [l, r)
    if (x <= l && r <= y) return t[p].p;
    int m = (l + r) / 2, res = 0;
    if (x < m) res += query(t[p].lc, l, m, x, y);
    if (y > m) res += query(t[p].rc, m, r, x, y);
    return res;
  }
};

struct DSU {
  int n;
  SegTree seg;
  DSU(int _n) : n(_n), seg(n) {}
  int get(int p, int x) { return seg.query(p, 0, n, x, x + 1); }
  int set(int p, int x, int v) { return seg.modify(p, 0, n, x, v); }
  int find(int p, int x) {
    int parent = get(p, x);
    if (parent < 0) return x;
    return find(p, parent);
  }
  int is_same(int p, int x, int y) { return find(p, x) == find(p, y); }
  int merge(int p, int x, int y) {
    int rx = find(p, x), ry = find(p, y);
    if (rx == ry) return -1;
    int rank_x = -get(p, rx), rank_y = -get(p, ry);
    if (rank_x < rank_y) {
      p = set(p, rx, ry);
    } else if (rank_x > rank_y) {
      p = set(p, ry, rx);
    } else {
      p = set(p, ry, rx);
      p = set(p, rx, -rx - 1);
    }
    return p;
  }
};
```

## Fenwick Tree

- askd version

```
template <typename T> struct FenwickTree {
  int size = 1, high_bit = 1;
  vector<T> tree;
  FenwickTree(int _size) : size(_size) {
    tree.resize(size + 1);
    while((high_bit << 1) <= size) high_bit <<= 1;
  }
  FenwickTree(vector<T>& arr) : FenwickTree(arr.size()) {
    for(int i = 0; i < size; i++) update(i, arr[i]);
  }
  int lower_bound(T x) {
    int res = 0; T cur = 0;
    for(int bit = high_bit; bit > 0; bit >>= 1) {
      if((res|bit) <= size && cur + tree[res|bit] < x) {
        res |= bit; cur += tree[res];
      }
    }
    return res;
  }
  T prefix_sum(int i) {
    T ret = 0;
    for(i++; i > 0; i -= (i & -i)) ret += tree[i];
    return ret;
  }
  T range_sum(int l, int r) { return (l > r) ? 0 : prefix_sum(r) - prefix_sum(l - 1); }
  void update(int i, T delta) { for(i++; i <= size; i += (i & -i)) tree[i] += delta; }
};
```

- Nea1 version

```cpp
template <typename T>
struct Fenwick {
  const int n;
  vector<T> a;
  Fenwick(int n) : n(n), a(n) {}
  void add(int x, T v) {
    for (int i = x + 1; i <= n; i += i & -i) {
      a[i - 1] += v;
    }
  }
  T sum(int x) {
    T ans = 0;
    for (int i = x; i > 0; i -= i & -i) {
      ans += a[i - 1];
    }
    return ans;
  }
  T rangeSum(int l, int r) { return sum(r) - sum(l); }
};
```

## PBDS

```cpp
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
using namespace std;
using namespace __gnu_pbds;
template<typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
template<typename T, typename X>
using ordered_map = tree<T, X, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
template<typename T, typename X>
using fast_map = cc_hash_table<T, X>;
template<typename T, typename X>
using ht = gp_hash_table<T, X>;
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

struct splitmix64 {
    size_t operator()(size_t x) const {
        static const size_t fixed = chrono::steady_clock::now().time_since_epoch().count();
        x += 0x9e3779b97f4a7c15 + fixed;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
};
```

## Treap

- (No rotation version)

```cpp
struct Node {
  Node *l, *r;
  int s, sz;
  // int t = 0, a = 0, g = 0; // for lazy propagation
  ll w;

  Node(int _s) : l(nullptr), r(nullptr), s(_s), sz(1), w(rng()) {}
  void apply(int vt, int vg) {
    // for lazy propagation
    // s -= vt;
    // t += vt, a += vg, g += vg;
  }
  void push() {
    // for lazy propagation
    // if (l != nullptr) l->apply(t, g);
    // if (r != nullptr) r->apply(t, g);
    // t = g = 0;
  }
  void pull() { sz = 1 + (l ? l->sz : 0) + (r ? r->sz : 0); }
};
```

```cpp

std::pair<Node *, Node *> split(Node *t, int v) {
  if (t == nullptr) return {nullptr, nullptr};
  t->push();
  if (t->s < v) {
    auto [x, y] = split(t->r, v);
    t->r = x;
    t->pull();
    return {t, y};
  } else {
    auto [x, y] = split(t->l, v);
    t->l = y;
    t->pull();
    return {x, t};
  }
}

Node *merge(Node *p, Node *q) {
  if (p == nullptr) return q;
  if (q == nullptr) return p;
  if (p->w < q->w) swap(p, q);
  auto [x, y] = split(q, p->s + rng() % 2);
  p->push();
  p->l = merge(p->l, x);
  p->r = merge(p->r, y);
  p->pull();
  return p;
}

Node *insert(Node *t, int v) {
  auto [x, y] = split(t, v);
  return merge(merge(x, new Node(v)), y);
}

Node *erase(Node *t, int v) {
  auto [x, y] = split(t, v);
  auto [p, q] = split(y, v + 1);
  return merge(merge(x, merge(p->l, p->r)), q);
}

int get_rank(Node *&t, int v) {
  auto [x, y] = split(t, v);
  int res = (x ? x->sz : 0) + 1;
  t = merge(x, y);
  return res;
}

Node *kth(Node *t, int k) {
  k--;
  while (true) {
    int left_sz = t->l ? t->l->sz : 0;
    if (k < left_sz) {
      t = t->l;
    } else if (k == left_sz) {
      return t;
    } else {
      k -= left_sz + 1, t = t->r;
    }
  }
}

Node *get_prev(Node *&t, int v) {
  auto [x, y] = split(t, v);
  Node *res = kth(x, x->sz);
  t = merge(x, y);
  return res;
}

Node *get_next(Node *&t, int v) {
  auto [x, y] = split(t, v + 1);
  Node *res = kth(y, 1);
```

```
  t = merge(x, y);
  return res;
}
```

- USAGE

```
int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int n;
  cin >> n;
  Node *t = nullptr;
  for (int op, x; n--;) {
    cin >> op >> x;
    if (op == 1) {
      t = insert(t, x);
    } else if (op == 2) {
      t = erase(t, x);
    } else if (op == 3) {
      cout << get_rank(t, x) << "\n";
    } else if (op == 4) {
      cout << kth(t, x)->s << "\n";
    } else if (op == 5) {
      cout << get_prev(t, x)->s << "\n";
    } else {
      cout << get_next(t, x)->s << "\n";
    }
  }
}
```

## Implicit treap

- Split by size

```
struct Node {
  Node *l, *r;
  int s, sz;
  // int lazy = 0;
  ll w;

  Node(int _s) : l(nullptr), r(nullptr), s(_s), sz(1), w(rnd()) {}
  void apply() {
    // for lazy propagation
    // lazy ^= 1;
  }
  void push() {
    // for lazy propagation
    // if (lazy) {
    //   swap(l, r);
    //   if (l != nullptr) l->apply();
    //   if (r != nullptr) r->apply();
    //   lazy = 0;
    // }
  }
  void pull() { sz = 1 + (l ? l->sz : 0) + (r ? r->sz : 0); }
};

std::pair<Node *, Node *> split(Node *t, int v) {
  // first->sz == v
  if (t == nullptr) return {nullptr, nullptr};
  t->push();
  int left_sz = t->l ? t->l->sz : 0;
  if (left_sz < v) {
    auto [x, y] = split(t->r, v - left_sz - 1);
    t->r = x;
    t->pull();
    return {t, y};
  } else {
    auto [x, y] = split(t->l, v);
    t->l = y;
    t->pull();
    return {x, t};
```

```
  }
}

Node *merge(Node *p, Node *q) {
  if (p == nullptr) return q;
  if (q == nullptr) return p;
  if (p->w < q->w) {
    p->push();
    p->r = merge(p->r, q);
    p->pull();
    return p;
  } else {
    q->push();
    q->l = merge(p, q->l);
    q->pull();
    return q;
  }
}
```

## Persistent implicit treap

```
pair<Node *, Node *> split(Node *t, int v) {
  // first->sz == v
  if (t == nullptr) return {nullptr, nullptr};
  t->push();
  int left_sz = t->l ? t->l->sz : 0;
  t = new Node(*t);
  if (left_sz < v) {
    auto [x, y] = split(t->r, v - left_sz - 1);
    t->r = x;
    t->pull();
    return {t, y};
  } else {
    auto [x, y] = split(t->l, v);
    t->l = y;
    t->pull();
    return {x, t};
  }
}

Node *merge(Node *p, Node *q) {
  if (p == nullptr) return new Node(*q);
  if (q == nullptr) return new Node(*p);
  if (p->w < q->w) {
    p = new Node(*p);
    p->push();
    p->r = merge(p->r, q);
    p->pull();
    return p;
  } else {
    q = new Node(*q);
    q->push();
    q->l = merge(p, q->l);
    q->pull();
    return q;
  }
}
```

## 2D Sparse Table

- Sorry that this sucks - askd

```
template <class T, class Compare = less<T>>
struct SparseTable2d {
  int n = 0, m = 0;
  T**** table;
  int* log;
  inline T choose(T x, T y) {
    return Compare()(x, y) ? x : y;
  }
```

```
  SparseTable2d(vector<vector<T>>& grid) {
    if(grid.empty() || grid[0].empty()) return;
    n = grid.size(); m = grid[0].size();
    log = new int[max(n, m) + 1];
    log[1] = 0;
    for(int i = 2; i <= max(n, m); i++)
      log[i] = log[i - 1] + ((i ^ (i - 1)) > i);
    table = new T***[n];
    for(int i = n - 1; i >= 0; i--) {
      table[i] = new T**[m];
      for(int j = m - 1; j >= 0; j--) {
        table[i][j] = new T*[log[n - i] + 1];
        for(int k = 0; k <= log[n - i]; k++) {
          table[i][j][k] = new T[log[m - j] + 1];
          if(!k) table[i][j][k][0] = grid[i][j];
          else table[i][j][k][0] = choose(table[i][j][k-1][0], table[i+(1<<(k-1))][j][k-1][0]);
          for(int l = 1; l <= log[m - j]; l++)
            table[i][j][k][l] = choose(table[i][j][k][l-1], table[i][j+(1<<(l-1))][k][l-1]);
        }
      }
    }
  }
  T query(int r1, int r2, int c1, int c2) {
    assert(r1 >= 0 && r2 < n && r1 <= r2);
    assert(c1 >= 0 && c2 < m && c1 <= c2);
    int rl = log[r2 - r1 + 1], cl = log[c2 - c1 + 1];
    T ca1 = choose(table[r1][c1][rl][cl], table[r2-(1<<rl)+1][c1][rl][cl]);
    T ca2 = choose(table[r1][c2-(1<<cl)+1][rl][cl], table[r2-(1<<rl)+1][c2-(1<<cl)+1][rl][cl]);
    return choose(ca1, ca2);
  }
};
```

- USAGE

```
vector<vector<int>> test =  {
  {1, 2, 3, 4}, {2, 3, 4, 5}, {9, 9, 9, 9}, {-1, -1, -1, -1}
};

SparseTable2d<int> st(test);                // Range min query
SparseTable2d<int,greater<int>> st2(test);  // Range max query
```

# Geometry

## Basic stuff

```
using ll = long long;
using ld = long double;

constexpr auto eps = 1e-8;
const auto PI = acos(-1);
int sgn(ld x) { return (abs(x) <= eps) ? 0 : (x < 0 ? -1 : 1); }

struct Point {
  ld x = 0, y = 0;
  Point() = default;
  Point(ld _x, ld _y) : x(_x), y(_y) {}
  bool operator<(const Point &p) const { return !sgn(p.x - x) ? sgn(y - p.y) < 0 : x < p.x; }
  bool operator==(const Point &p) const { return !sgn(p.x - x) && !sgn(p.y - y); }
  Point operator+(const Point &p) const { return {x + p.x, y + p.y}; }
  Point operator-(const Point &p) const { return {x - p.x, y - p.y}; }
  Point operator*(ld a) const { return {x * a, y * a}; }
  Point operator/(ld a) const { return {x / a, y / a}; }
  auto operator*(const Point &p) const { return x * p.x + y * p.y; }  // dot
  auto operator^(const Point &p) const { return x * p.y - y * p.x; }  // cross
  friend auto &operator>>(istream &i, Point &p) { return i >> p.x >> p.y; }
  friend auto &operator<<(ostream &o, Point p) { return o << p.x << ' ' << p.y; }
};

struct Line {
  Point s = {0, 0}, e = {0, 0};
```

```
  Line() = default;
  Line(Point _s, Point _e) : s(_s), e(_e) {}
  friend auto &operator>>(istream &i, Line &l) { return i >> l.s >> l.e; }  // ((x1, y1), (x2, y2)
};

struct Segment : Line {
  using Line::Line;
};

struct Circle {
  Point o = {0, 0};
  ld r = 0;
  Circle() = default;
  Circle(Point _o, ld _r) : o(_o), r(_r) {}
};
```

```
auto dist2(const Point &a) { return a * a; }
auto dist2(const Point &a, const Point &b) { return dist2(a - b); }
auto dist(const Point &a) { return sqrt(dist2(a)); }
auto dist(const Point &a, const Point &b) { return sqrt(dist2(a - b)); }
auto dist(const Point &a, const Line &l) { return abs((a - l.s) ^ (l.e - l.s)) / dist(l.s, l.e); }
auto dist(const Point &p, const Segment &l) {
  if (l.s == l.e) return dist(p, l.s);
  auto d = dist2(l.s, l.e), t = min(d, max((ld)0, (p - l.s) * (l.e - l.s)));
  return dist((p - l.s) * d, (l.e - l.s) * t) / d;
}
/* Needs is_intersect
auto dist(const Segment &l1, const Segment &l2) {
  if (is_intersect(l1, l2)) return (ld)0;
  return min({dist(l1.s, l2), dist(l1.e, l2), dist(l2.s, l1), dist(l2.e, l1)});
} */

Point perp(const Point &p) { return Point(-p.y, p.x); }

auto rad(const Point &p) { return atan2(p.y, p.x); }
```

## Transformation

```
Point project(const Point &p, const Line &l) {
  return l.s + ((l.e - l.s) * ((l.e - l.s) * (p - l.s))) / dist2(l.e - l.s);
}

Point reflect(const Point &p, const Line &l) {
  return project(p, l) * 2 - p;
}

Point dilate(const Point &p, ld scale_x = 1, ld scale_y = 1) { return Point(p.x * scale_x, p.y * scale_y); }
Line dilate(const Line &l, ld scale_x = 1, ld scale_y = 1) { return Line(dilate(l.s, scale_x, scale_y), dilate(l.e,
 ↪  scale_x, scale_y)); }
Segment dilate(const Segment &l, ld scale_x = 1, ld scale_y = 1) { return Segment(dilate(l.s, scale_x, scale_y),
 ↪  dilate(l.e, scale_x, scale_y)); }
vector<Point> dilate(const vector<Point> &p, ld scale_x = 1, ld scale_y = 1) {
  int n = p.size();
  vector<Point> res(n);
  for (int i = 0; i < n; i++)
    res[i] = dilate(p[i], scale_x, scale_y);
  return res;
}

Point rotate(const Point &p, ld a) { return Point(p.x * cos(a) - p.y * sin(a), p.x * sin(a) + p.y * cos(a)); }
Line rotate(const Line &l, ld a) { return Line(rotate(l.s, a), rotate(l.e, a)); }
Segment rotate(const Segment &l, ld a) { return Segment(rotate(l.s, a), rotate(l.e, a)); }
Circle rotate(const Circle &c, ld a) { return Circle(rotate(c.o, a), c.r); }
vector<Point> rotate(const vector<Point> &p, ld a) {
  int n = p.size();
  vector<Point> res(n);
  for (int i = 0; i < n; i++)
    res[i] = rotate(p[i], a);
  return res;
}

```

```cpp
Point translate(const Point &p, ld dx = 0, ld dy = 0) { return Point(p.x + dx, p.y + dy); }
Line translate(const Line &l, ld dx = 0, ld dy = 0) { return Line(translate(l.s, dx, dy), translate(l.e, dx, dy)); }
Segment translate(const Segment &l, ld dx = 0, ld dy = 0) { return Segment(translate(l.s, dx, dy), translate(l.e, dx, dy)); }
Circle translate(const Circle &c, ld dx = 0, ld dy = 0) { return Circle(translate(c.o, dx, dy), c.r); }
vector<Point> translate(const vector<Point> &p, ld dx = 0, ld dy = 0) {
  int n = p.size();
  vector<Point> res(n);
  for (int i = 0; i < n; i++)
    res[i] = translate(p[i], dx, dy);
  return res;
}
```

## Relation

```cpp
enum class Relation { SEPARATE, EX_TOUCH, OVERLAP, IN_TOUCH, INSIDE };
Relation get_relation(const Circle &a, const Circle &b) {
  auto c1c2 = dist(a.o, b.o);
  auto r1r2 = a.r + b.r, diff = abs(a.r - b.r);
  if (sgn(c1c2 - r1r2) > 0) return Relation::SEPARATE;
  if (sgn(c1c2 - r1r2) == 0) return Relation::EX_TOUCH;
  if (sgn(c1c2 - diff) > 0) return Relation::OVERLAP;
  if (sgn(c1c2 - diff) == 0) return Relation::IN_TOUCH;
  return Relation::INSIDE;
}

auto get_cos_from_triangle(ld a, ld b, ld c) { return (a * a + b * b - c * c) / (2.0 * a * b); }

bool on_line(const Line &l, const Point &p) { return !sgn((l.s - p) ^ (l.e - p)); }

bool on_segment(const Segment &l, const Point &p) {
  return !sgn((l.s - p) ^ (l.e - p)) && sgn((l.s - p) * (l.e - p)) <= 0;
}

bool on_segment2(const Segment &l, const Point &p) { // assume p on Line l
  if (l.s == p || l.e == p) return true;
  if (min(l.s, l.e) < p && p < max(l.s, l.e)) return true;
  return false;
}

bool is_parallel(const Line &a, const Line &b) { return !sgn((a.s - a.e) ^ (b.s - b.e)); }
bool is_orthogonal(const Line &a, const Line &b) { return !sgn((a.s - a.e) * (b.s - b.e)); }

int is_intersect(const Segment &a, const Segment &b) {
  auto d1 = sgn((a.e - a.s) ^ (b.s - a.s)), d2 = sgn((a.e - a.s) ^ (b.e - a.s));
  auto d3 = sgn((b.e - b.s) ^ (a.s - b.s)), d4 = sgn((b.e - b.s) ^ (a.e - b.s));
  if (d1 * d2 < 0 && d3 * d4 < 0) return 2;  // intersect at non-end point
  return (d1 == 0 && sgn((b.s - a.s) * (b.s - a.e)) <= 0) ||
         (d2 == 0 && sgn((b.e - a.s) * (b.e - a.e)) <= 0) ||
         (d3 == 0 && sgn((a.s - b.s) * (a.s - b.e)) <= 0) ||
         (d4 == 0 && sgn((a.e - b.s) * (a.e - b.e)) <= 0);
}

int is_intersect(const Line &a, const Segment &b) {
  auto d1 = sgn((a.e - a.s) ^ (b.s - a.s)), d2 = sgn((a.e - a.s) ^ (b.e - a.s));
  if (d1 * d2 < 0) return 2;  // intersect at non-end point
  return d1 == 0 || d2 == 0;
}

Point intersect(const Line &a, const Line &b) {
  auto u = a.e - a.s, v = b.e - b.s;
  auto t = ((b.s - a.s) ^ v) / (u ^ v);
  return a.s + u * t;
}

int is_intersect(const Circle &c, const Line &l) {
  auto d = dist(c.o, l);
  return sgn(d - c.r) < 0 ? 2 : !sgn(d - c.r);
}

vector<Point> intersect(const Circle &a, const Circle &b) {
```

```
  auto relation = get_relation(a, b);
  if (relation == Relation::INSIDE || relation == Relation::SEPARATE) return {};
  auto vec = b.o - a.o;
  auto d2 = dist2(vec);
  auto p = (d2 + a.r * a.r - b.r * b.r) / ((long double)2 * d2), h2 = a.r * a.r - p * p * d2;
  auto mid = a.o + vec * p, per = perp(vec) * sqrt(max((long double)0, h2) / d2);
  if (relation == Relation::OVERLAP)
    return {mid + per, mid - per};
  else
    return {mid};
}

vector<Point> intersect(const Circle &c, const Line &l) {
  if (!is_intersect(c, l)) return {};
  auto v = l.e - l.s, t = v / dist(v);
  Point a = l.s + t * ((c.o - l.s) * t);
  auto d = sqrt(max((ld)0, c.r * c.r - dist2(c.o, a)));
  if (!sgn(d)) return {a};
  return {a - t * d, a + t * d};
}

int in_poly(const vector<Point> &p, const Point &a) {
  int cnt = 0, n = (int)p.size();
  for (int i = 0; i < n; i++) {
    auto q = p[(i + 1) % n];
    if (on_segment(Segment(p[i], q), a)) return 1;  // on the edge of the polygon
    cnt ^= ((a.y < p[i].y) - (a.y < q.y)) * ((p[i] - a) ^ (q - a)) > 0;
  }
  return cnt ? 2 : 0;
}

int is_intersect(const vector<Point> &p, const Line &a) {
  // 1: touching, >=2: intersect count
  int cnt = 0, edge_cnt = 0, n = (int)p.size();
  for (int i = 0; i < n; i++) {
    auto q = p[(i + 1) % n];
    if (on_line(a, p[i]) && on_line(a, q)) return -1;  // infinity
    auto t = is_intersect(a, Segment(p[i], q));
    (t == 1) && edge_cnt++, (t == 2) && cnt++;
  }
  return cnt + edge_cnt / 2;
}

vector<Point> tangent(const Circle &c, const Point &p) {
  auto d = dist(c.o, p), l = c.r * c.r / d, h = sqrt(c.r * c.r - l * l);
  auto v = (p - c.o) / d;
  return {c.o + v * l + perp(v) * h, c.o + v * l - perp(v) * h};
}

Circle get_circumscribed(const Point &a, const Point &b, const Point &c) {
  Line u((a + b) / 2, ((a + b) / 2) + perp(b - a));
  Line v((b + c) / 2, ((b + c) / 2) + perp(c - b));
  auto o = intersect(u, v);
  return Circle(o, dist(o, a));
}

Circle get_inscribed(const Point &a, const Point &b, const Point &c) {
  auto l1 = dist(b - c), l2 = dist(c - a), l3 = dist(a - b);
  Point o = (a * l1 + b * l2 + c * l3) / (l1 + l2 + l3);
  return Circle(o, dist(o, Line(a, b)));
}

pair<ld, ld> get_centroid(const vector<Point> &p) {
  int n = (int)p.size();
  ld x = 0, y = 0, sum = 0;
  auto a = p[0], b = p[1];
  for (int i = 2; i < n; i++) {
    auto c = p[i];
    auto s = area({a, b, c});
    sum += s;
    x += s * (a.x + b.x + c.x);
```

```
    y += s * (a.y + b.y + c.y);
    swap(b, c);
  }
  return {x / (3 * sum), y / (3 * sum)};
}
```

## Area

```
auto area(const vector<Point> &p) {
  int n = (int)p.size();
  long double area = 0;
  for (int i = 0; i < n; i++) area += p[i] ^ p[(i + 1) % n];
  return area / 2.0;
}

auto area(const Point &a, const Point &b, const Point &c) {
  return ((long double)((b - a) ^ (c - a))) / 2.0;
}

auto area2(const Point &a, const Point &b, const Point &c) { return (b - a) ^ (c - a); }

auto area_intersect(const Circle &c, const vector<Point> &ps) {
  int n = (int)ps.size();
  auto arg = [&](const Point &p, const Point &q) { return atan2(p ^ q, p * q); };
  auto tri = [&](const Point &p, const Point &q) {
    auto r2 = c.r * c.r / (long double)2;
    auto d = q - p;
    auto a = d * p / dist2(d), b = (dist2(p) - c.r * c.r) / dist2(d);
    long double det = a * a - b;
    if (sgn(det) <= 0) return arg(p, q) * r2;
    auto s = max((long double)0, -a - sqrt(det)), t = min((long double)1, -a + sqrt(det));
    if (sgn(t) < 0 || sgn(1 - s) <= 0) return arg(p, q) * r2;
    auto u = p + d * s, v = p + d * t;
    return arg(p, u) * r2 + (u ^ v) / 2 + arg(v, q) * r2;
  };
  long double sum = 0;
  for (int i = 0; i < n; i++) sum += tri(ps[i] - c.o, ps[(i + 1) % n] - c.o);
  return sum;
}

auto adaptive_simpson(ld _l, ld _r, function<ld(ld)> f) {
  auto simpson = [&](ld l, ld r) { return (r - l) * (f(l) + 4 * f((l + r) / 2) + f(r)) / 6; };
  function<ld(ld, ld, ld)> asr = [&](ld l, ld r, ld s) {
    auto mid = (l + r) / 2;
    auto left = simpson(l, mid), right = simpson(mid, r);
    if (!sgn(left + right - s)) return left + right;
    return asr(l, mid, left) + asr(mid, r, right);
  };
  return asr(_l, _r, simpson(_l, _r));
}

vector<Point> half_plane_intersect(vector<Line> &L) {
  int n = (int)L.size(), l = 0, r = 0;  // [left, right]
  sort(L.begin(), L.end(),
       [](const Line &a, const Line &b) { return rad(a.s - a.e) < rad(b.s - b.e); });
  vector<Point> p(n), res;
  vector<Line> q(n);
  q[0] = L[0];
  for (int i = 1; i < n; i++) {
    while (l < r && sgn((L[i].e - L[i].s) ^ (p[r - 1] - L[i].s)) <= 0) r--;
    while (l < r && sgn((L[i].e - L[i].s) ^ (p[l] - L[i].s)) <= 0) l++;
    q[++r] = L[i];
    if (sgn((q[r].e - q[r].s) ^ (q[r - 1].e - q[r - 1].s)) == 0) {
      r--;
      if (sgn((q[r].e - q[r].s) ^ (L[i].s - q[r].s)) > 0) q[r] = L[i];
    }
    if (l < r) p[r - 1] = intersect(q[r - 1], q[r]);
  }
  while (l < r && sgn((q[l].e - q[l].s) ^ (p[r - 1] - q[l].s)) <= 0) r--;
  if (r - l <= 1) return {};
  p[r] = intersect(q[r], q[l]);
```

```
  return vector<Point>(p.begin() + l, p.begin() + r + 1);
}
```

## Convex

```
vector<Point> get_convex(vector<Point> &points, bool allow_collinear = false) {
  // strict, no repeat, two pass
  sort(points.begin(), points.end());
  points.erase(unique(points.begin(), points.end()), points.end());
  vector<Point> L, U;
  for (auto &t : points) {
    for (ll sz = L.size(); sz > 1 && (sgn((t - L[sz - 2]) ^ (L[sz - 1] - L[sz - 2])) >= 0);
         L.pop_back(), sz = L.size()) {
    }
    L.push_back(t);
  }
  for (auto &t : points) {
    for (ll sz = U.size(); sz > 1 && (sgn((t - U[sz - 2]) ^ (U[sz - 1] - U[sz - 2])) <= 0);
         U.pop_back(), sz = U.size()) {
    }
    U.push_back(t);
  }
  // contain repeats if all collinear, use a set to remove repeats
  if (allow_collinear) {
    for (int i = (int)U.size() - 2; i >= 1; i--) L.push_back(U[i]);
  } else {
    set<Point> st(L.begin(), L.end());
    for (int i = (int)U.size() - 2; i >= 1; i--) {
      if (st.count(U[i]) == 0) L.push_back(U[i]), st.insert(U[i]);
    }
  }
  return L;
}

vector<Point> get_convex2(vector<Point> &points, bool allow_collinear = false) {  // strict, no repeat, one pass
  nth_element(points.begin(), points.begin(), points.end());
  sort(points.begin() + 1, points.end(), [&](const Point &a, const Point &b) {
    int rad_diff = sgn((a - points[0]) ^ (b - points[0]));
    return !rad_diff ? (dist2(a - points[0]) < dist2(b - points[0])) : (rad_diff > 0);
  });
  if (allow_collinear) {
    int i = (int)points.size() - 1;
    while (i >= 0 && !sgn((points[i] - points[0]) ^ (points[i] - points.back()))) i--;
    reverse(points.begin() + i + 1, points.end());
  }
  vector<Point> hull;
  for (auto &t : points) {
    for (ll sz = hull.size();
         sz > 1 && (sgn((t - hull[sz - 2]) ^ (hull[sz - 1] - hull[sz - 2])) >= allow_collinear);
         hull.pop_back(), sz = hull.size()) {
    }
    hull.push_back(t);
  }
  return hull;
}

vector<Point> get_convex_safe(vector<Point> points, bool allow_collinear = false) {
  return get_convex(points, allow_collinear);
}

vector<Point> get_convex2_safe(vector<Point> points, bool allow_collinear = false) {
  return get_convex2(points, allow_collinear);
}

bool is_convex(const vector<Point> &p, bool allow_collinear = false) {
  int n = p.size();
  int lo = 1, hi = -1;
  for (int i = 0; i < n; i++) {
    int cur = sgn((p[(i + 2) % n] - p[(i + 1) % n]) ^ (p[(i + 1) % n] - p[i]));
    lo = min(lo, cur); hi = max(hi, cur);
  }
```

```
  return allow_collinear ? (hi - lo) < 2 : (lo == hi && lo);
}

auto rotating_calipers(const vector<Point> &hull) {
  // use get_convex2
  int n = (int)hull.size();  // return the square of longest dist
  assert(n > 1);
  if (n <= 2) return dist2(hull[0], hull[1]);
  ld res = 0;
  for (int i = 0, j = 2; i < n; i++) {
    auto d = hull[i], e = hull[(i + 1) % n];
    while (area2(d, e, hull[j]) < area2(d, e, hull[(j + 1) % n])) j = (j + 1) % n;
    res = max(res, max(dist2(d, hull[j]), dist2(e, hull[j])));
  }
  return res;
}

// Find polygon cut to the left of l
vector<Point> convex_cut(const vector<Point> &p, const Line &l) {
  int n = p.size();
  vector<Point> cut;
  for (int i = 0; i < n; i++) {
    auto a = p[i], b = p[(i + 1) % n];
    if (sgn((l.e - l.s) ^ (a - l.s)) >= 0)
      cut.push_back(a);
    if (sgn((l.e - l.s) ^ (a - l.s)) * sgn((l.e - l.s) ^ (b - l.s)) == -1)
      cut.push_back(intersect(Line(a, b), l));
  }
  return cut;
}

// Sort by angle in range [0, 2pi)
template <class RandomIt>
void polar_sort(RandomIt first, RandomIt last, Point origin = Point(0, 0)) {
  auto get_quad = [&](const Point& p) {
    Point diff = p - origin;
    if (diff.x > 0 && diff.y >= 0) return 1;
    if (diff.x <= 0 && diff.y > 0) return 2;
    if (diff.x < 0 && diff.y <= 0) return 3;
    return 4;
  };
  auto polar_cmp = [&](const Point& p1, const Point& p2) {
    int q1 = get_quad(p1), q2 = get_quad(p2);
    if (q1 != q2) return q1 < q2;
    return ((p1 - origin) ^ (p2 - origin)) > 0;
  };
  sort(first, last, polar_cmp);
}
```

## Basic 3D

```
using ll = long long;
using ld = long double;

constexpr auto eps = 1e-8;
const auto PI = acos(-1);
int sgn(ld x) { return (abs(x) <= eps) ? 0 : (x < 0 ? -1 : 1); }

struct Point3D {
  ld x = 0, y = 0, z = 0;
  Point3D() = default;
  Point3D(ld _x, ld _y, ld _z) : x(_x), y(_y), z(_z) {}
  bool operator<(const Point3D &p) const { return !sgn(p.x - x) ? (!sgn(p.y - y) ? sgn(p.z - z) < 0 : y < p.y) : x <
   ↪  p.x; }
  bool operator==(const Point3D &p) const { return !sgn(p.x - x) && !sgn(p.y - y) && !sgn(p.z - z); }
  Point3D operator+(const Point3D &p) const { return {x + p.x, y + p.y, z + p.z}; }
  Point3D operator-(const Point3D &p) const { return {x - p.x, y - p.y, z - p.z}; }
  Point3D operator*(ld a) const { return {x * a, y * a, z * a}; }
  Point3D operator/(ld a) const { return {x / a, y / a, z / a}; }
  auto operator*(const Point3D &p) const { return x * p.x + y * p.y + z * p.z; }  // dot
```

```
  Point3D operator^(const Point3D &p) const { return {y * p.z - z * p.y, z * p.x - x * p.z, x * p.y - y * p.x}; } // cross
  friend auto &operator>>(istream &i, Point3D &p) { return i >> p.x >> p.y >> p.z; }
};

struct Line3D {
  Point3D s = {0, 0, 0}, e = {0, 0, 0};
  Line3D() = default;
  Line3D(Point3D _s, Point3D _e) : s(_s), e(_e) {}
};

struct Segment3D : Line3D {
  using Line3D::Line3D;
};

auto dist2(const Point3D &a) { return a * a; }
auto dist2(const Point3D &a, const Point3D &b) { return dist2(a - b); }
auto dist(const Point3D &a) { return sqrt(dist2(a)); }
auto dist(const Point3D &a, const Point3D &b) { return sqrt(dist2(a - b)); }
auto dist(const Point3D &a, const Line3D &l) { return dist((a - l.s) ^ (l.e - l.s)) / dist(l.s, l.e); }
auto dist(const Point3D &p, const Segment3D &l) {
  if (l.s == l.e) return dist(p, l.s);
  auto d = dist2(l.s, l.e), t = min(d, max((ld)0, (p - l.s) * (l.e - l.s)));
  return dist((p - l.s) * d, (l.e - l.s) * t) / d;
}
```

## Miscellaneous

```
tuple<int,int,ld> closest_pair(vector<Point> &p) {
  using Pt = pair<Point,int>;
  int n = p.size();
  assert(n > 1);
  vector<Pt> pts(n), buf;
  for (int i = 0; i < n; i++) pts[i] = {p[i], i};
  sort(pts.begin(), pts.end());
  buf.reserve(n);
  auto cmp_y = [](const Pt& p1, const Pt& p2) { return p1.first.y < p2.first.y; };
  function<tuple<int,int,ld>(int, int)> recurse = [&](int l, int r) -> tuple<int,int,ld> {
    int i = pts[l].second, j = pts[l + 1].second;
    ld d = dist(pts[l].first, pts[l + 1].first);
    if (r - l < 5) {
      for (int a = l; a < r; a++) for (int b = a + 1; b < r; b++) {
        ld cur = dist(pts[a].first, pts[b].first);
        if (cur < d) { i = pts[a].second; j = pts[b].second; d = cur; }
      }
      sort(pts.begin() + l, pts.begin() + r, cmp_y);
    }
    else {
      int mid = (l + r)/2;
      ld x = pts[mid].first.x;
      auto [li, lj, ldist] = recurse(l, mid);
      auto [ri, rj, rdist] = recurse(mid, r);
      if (ldist < rdist) { i = li; j = lj; d = ldist; }
      else { i = ri; j = rj; d = rdist; }
      inplace_merge(pts.begin() + l, pts.begin() + mid, pts.begin() + r, cmp_y);
      buf.clear();
      for (int a = l; a < r; a++) {
        if (abs(x - pts[a].first.x) >= d) continue;
        for (int b = buf.size() - 1; b >= 0; b--) {
          if (pts[a].first.y - buf[b].first.y >= d) break;
          ld cur = dist(pts[a].first, buf[b].first);
          if (cur < d) { i = pts[a].second; j = buf[b].second; d = cur; }
        }
        buf.push_back(pts[a]);
      }
    }
    return {i, j, d};
  };
  return recurse(0, n);
}

```

```
Line abc_to_line(ld a, ld b, ld c) {
  assert(!sgn(a) || !sgn(b));
  if(a == 0) return Line(Point(0, -c/b), Point(1, -c/b));
  if(b == 0) return Line(Point(-c/a, 0), Point(-c/a, 1));
  Point s(0, -c/b), e(1, (-c - a)/b), diff = e - s;
  return Line(s, s + diff/dist(diff));
}

tuple<ld,ld,ld> line_to_abc(const Line& l) {
  Point diff = l.e - l.s;
  return {-diff.y, diff.x, -(diff ^ l.s)};
}
```