ICPC Notebook

MWNWMWNNWMWNWN

Contents

1 Geometry

1.1 Circle

```
/* Basic structure of circle and operations related with it. This template works
* only with double numbers since most of the operations of a circle can't be
* done with only integers. Therefore, this template depends on point_double.cpp.
*
* All operations' time complexity are O(1)
*/

const ld PI = acos(-1);

struct circle {
    point o; ld r;
    circle() {}
    circle(point o, ld r) : o(o), r(r) {}
    bool has(point p) {
        return (o - p).norm2() < r*r + EPS;
    }
    vector<point> operator/(circle c) { // Intersection of circles.
        vector<point> inter;                       // The points in the output are in ccw order.
        ld d = (o - c.o).norm();
        if(r + c.r < d - EPS || d + min(r, c.r) < max(r, c.r) - EPS)
            return {};
        ld x = (r*r - c.r*c.r + d*d) / (2*d);
        ld y = sqrt(r*r - x*x);
        point v = (c.o - o) / d;
        inter.pb(o + v*x + v.rotate(cw90)*y);
        if(y > EPS) inter.pb(o + v*x + v.rotate(ccw90)*y);
        return inter;
    }
```

```
    vector<point> tang(point p){
        ld d = sqrt((p - o).norm2() - r*r);
        return *this / circle(p, d);
    }
    bool in(circle c){ // non strictly inside
        ld d = (o - c.o).norm();
        return d + r < c.r + EPS;
    }
};
```

1.2 Halfplane Intersection

```
/* Half-plane intersection algorithm. The result of intersecting half-planes is either
 * empty or a convex polygon (maybe degenerated). This template depends on point_double.cpp
 * and line_double.cpp.
 *
 * h - (input) set of half-planes to be intersected. Each half-plane is described as a pair
 * of points such that the half-plane is at the left of them.
 * pol - the intersection of the half-planes as a vector of points. If not empty, these
 * points describe the vertices of the resulting polygon in clock-wise order.
 * WARNING: Some points of the polygon might be repeated. This may be undesirable in some
 * cases but it's useful to distinguish between empty intersections and degenerated
 * polygons (such as a point, line, segment or half-line).
 *
 * Time complexity: O(n logn)
 */

struct halfplane: public line {
        ld ang;
        halfplane() {}
        halfplane(point _p, point _q) {
                p = _p; q = _q;
                point vec(q - p);
                ang = atan2(vec.y, vec.x);
        }
        bool operator <(const halfplane& other) const {
                if (fabsl(ang - other.ang) < EPS) return right(p, q, other.p);
                return ang < other.ang;
        }
        bool operator ==(const halfplane& other) const {
                return fabsl(ang - other.ang) < EPS;
        }
        bool out(point r) {
                return right(p, q, r);
        }
};

vector<point> hp_intersect(vector<halfplane> h) {
        point box[4] = {{-INF, -INF}, {INF, -INF}, {INF, INF}, {-INF, INF}};
        for(int i = 0; i < 4; i++)
                h.pb(halfplane(box[i], box[(i+1) % 4]));
        sort(h.begin(), h.end());
        h.resize(unique(h.begin(), h.end()) - h.begin());
        deque<halfplane> dq;
        for(auto hp: h) {
                while(sz(dq) > 1 && hp.out(intersect(dq.back(), dq[sz(dq) - 2])))
                        dq.pop_back();
                while(sz(dq) > 1 && hp.out(intersect(dq[0], dq[1])))
                        dq.pop_front();
                dq.pb(hp);
        }
        while(sz(dq) > 2 && dq[0].out(intersect(dq.back(), dq[sz(dq) - 2])))
                dq.pop_back();
```

```
		while(sz(dq) > 2 && dq.back().out(intersect(dq[0], dq[1])))
			dq.pop_front();
		if(sz(dq) < 3) return {};
		vector<point> pol(sz(dq));
		for(int i = 0; i < sz(dq); i++) {
			pol[i] = intersect(dq[i], dq[(i+1) % sz(dq)]);
		}
		return pol;
}
```

2 Graphs

2.1 Block Cut Tree

```
// Builds forest of block cut trees for an UNDIRECTED graph
// Constructor: SCC(|V|, |E|, [[v, e]; |V|])
// Complexity: O(N+M)
// be9e10
struct BlockCutTree {
	int ncomp; // number of components
	vector<int> comp; // comp[e]: component of edge e
	vector<vector<int>> gart; // gart[v]: list of components an articulation point v is adjacent to
	                          // if v is NOT an articulation point, then gart[v] is empty

	template <typename E> // assumes auto [neighbor_vertex, edge_id] = g[current_vertex][i]
	BlockCutTree(int n, int m, vector<E> g[]): ncomp(0), comp(m), gart(n) {
		vector<bool> vis(n), vise(m);
		vector<int> low(n), prof(n);
		stack<pair<int,int>> st;

		auto dfs = [&](auto& self, int v, bool root = 0) -> void {
			vis[v] = 1;
			int arb = 0; // arborescences
			for(auto [p, e]: g[v]) if(!vise[e]) {
				vise[e] = 1;
				int in = st.size();
				st.emplace(e, vis[p] ? -1 : p);
				if(!vis[p]) {
					arb++;
					low[p] = prof[p] = prof[v] + 1;
					self(self, p);
					low[v] = min(low[v], low[p]);
				} else low[v] = min(low[v], prof[p]);
				if(low[p] >= prof[v]) {
					gart[v].push_back(ncomp);
					while(st.size() > in) {
						auto [es, ps] = st.top();
						comp[es] = ncomp;
						if(ps != -1 && !gart[ps].empty())
							gart[ps].push_back(ncomp);
						st.pop();
					}
					ncomp++;
				}
			}
			if(root && arb <= 1) gart[v].clear();
		};
		for(int v=0;v<n;v++) if(!vis[v]) dfs(dfs, v, 1);
	}
};
```

2.2 Bridges

```
// Builds forest of strongly connected components for an UNDIRECTED graph
// Constructor: SCC(|V|, |E|, [[v, e]; |V|])
//
// Complexity: O(N+M)
// 3abbaa
struct SCC {
	vector<bool> bridge; // bridge[e]: true if edge e is a bridge
	vector<int> comp; // comp[v]: component of vertex v

	int ncomp; // number of components
	vector<int> sz; // sz[c]: size of component i (number of vertexes)
	vector<vector<pair<int, int>>> gc; // gc[i]: list of adjacent components

	SCC(int n, int m, vector<pair<int, int>> g[]): bridge(m), comp(n, -1), ncomp(0) {
		vector<bool> vis(n);
		vector<int> low(n), prof(n);

		auto dfs = [&](auto& self, int v, int dad = -1) -> void {
			vis[v] = 1;
			for(auto [p, e]: g[v]) if(p != dad) {
				if(!vis[p]) {
					low[p] = prof[p] = prof[v] + 1;
					self(self, p, v);
					low[v] = min(low[v], low[p]);
				} else low[v] = min(low[v], prof[p]);
			}
			if(low[v] == prof[v]) ncomp++;
		};
		for(int i=0;i<n;i++) if(!vis[i]) dfs(dfs, i);

		sz.resize(ncomp); gc.resize(ncomp);

		int cnt = 0;
		auto build = [&](auto& self, int v, int c = -1) -> void {
			if(low[v] == prof[v]) c = cnt++;
			comp[v] = c;
			sz[c]++;
			for(auto [p, e]: g[v]) if(comp[p] == -1) {
				self(self, p, c);
				int pc = comp[p];
				if(c != pc) {
					bridge[e] = true;
					gc[c].emplace_back(pc, e);
					gc[pc].emplace_back(c, e);
				}
			}
		};
		for(int i=0;i<n;i++) if(comp[i] == -1) build(build, i);
	}
};
```

3 Math

3.1 FWHT

```
/*
	Title: Fast Walsh-Hadamard trasform
	Description: Multiply two polynomials such that x^a * x^b = x^(op(a, b))
		- op(a, b) = a "xor" b, a "or" b, a "and" b
	Complexity: O(n log n)
	Credits: https://github.com/mochow13/competitive-programming-library/tree/master/Math
*/
```

```cpp
const ll N = 1<<20;

template <typename T>
struct FWHT {
	void fwht(T io[], ll n) {
		for (ll d = 1; d < n; d <<= 1) {
			for (ll i = 0, m = d<<1; i < n; i += m) {
				for (ll j = 0; j < d; j++) { /// Don't forget modulo if required
					T x = io[i+j], y = io[i+j+d];
					io[i+j] = (x+y), io[i+j+d] = (x-y);	// xor
					// io[i+j] = x+y; // and
					// io[i+j+d] = x+y; // or
				}
			}
		}
	}
	void ufwht(T io[], ll n) {
		for (ll d = 1; d < n; d <<= 1) {
			for (ll i = 0, m = d<<1; i < n; i += m) {
				for (ll j = 0; j < d; j++) { /// Don't forget modulo if required
					T x = io[i+j], y = io[i+j+d];
					 /// Modular inverse if required here
					io[i+j] = (x+y)>>1, io[i+j+d] = (x-y)>>1; // xor
					// io[i+j] = x-y; // and
					// io[i+j+d] = y-x; // or
				}
			}
		}
	}
	// a, b are two polynomials and n is size which is power of two
	void convolution(T a[], T b[], ll n) {
		fwht(a, n), fwht(b, n);
		for (ll i = 0; i < n; i++)
			a[i] = a[i]*b[i];
		ufwht(a, n);
	}
	// for a*a
	void self_convolution(T a[], ll n) {
		fwht(a, n);
		for (ll i = 0; i < n; i++)
			a[i] = a[i]*a[i];
		ufwht(a, n);
	}
};
FWHT<ll> fwht;
```

4 Data Structures

4.1 Implicit Lazy Treap

```cpp
// All operations are O(log N)
// If changes need to be made in lazy propagation,
// see Treap::reverse() and change Treap::no::prop()
//
// Important functions:
// Treap::insert(T val, int idx)
// Treap::erase(int idx)
// Treap::reverse(int l, int r)
// Treap::operator[](int idx)

mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

// HASH FROM HERE:
// c018fe
```

```cpp
template <typename T>
struct Treap {
	struct no {
		array<no*, 2> c;
		T dat;
		int cnt, h;

		// Example: reverse interval
		bool rev;

		no(T dat=T()): c({0, 0}), dat(dat), cnt(1), h(rng()), rev(0) {}

		// propagate
		void prop() {
			if(rev) {
				swap(c[0], c[1]);
				for(no* x: c) if(x) x->rev ^= !x->rev;
				rev = 0;
			}
		}

		// refresh
		no* ref() {
			cnt = 1;
			for(no* x: c) if(x) {
				x->prop();
				cnt += x->cnt;
			}
			return this;
		}

		// left child size
		int l() {
			return c[0] ? c[0]->cnt : 0;
		}
	};

	int sz;
	no *root;
	unique_ptr<no[]> arena;

	// prealloc: number of new_no() calls that will be made in total
	Treap(int prealloc): sz(0), root(0), arena(new no[prealloc]) {}

	no* new_no(T dat) {
		arena[sz] = no(dat);
		return &arena[sz++];
	}

	int cnt(no* x) { return x ? x->cnt : 0; }

	void merge(array<no*, 2> c, no*& res) {
		if(!c[0] || !c[1]) {
			res = c[0] ? c[0] : c[1];
			return;
		}
		for(no* x: c) x->prop();
		int i = c[0]->h < c[1]->h;
		no *l = c[i]->c[!i], *r = c[!i];
		if(i) swap(l, r);
		merge({l, r}, c[i]->c[!i]);
		res = c[i]->ref();
	}
```

```
// left treap has size pos
void split(no* x, int pos, array<no*, 2>& res, int ra = 0) {
	if(!x) {
		res.fill(0);
		return;
	}
	x->prop();
	ra += x->l();
	int i = pos > ra;
	split(x->c[i], pos, res, ra+(i?1:-x->l()));
	x->c[i] = res[!i];
	res[!i] = x->ref();
}

// Merges all s and makes them root
template <int SZ>
void merge(array<no*, SZ> s) {
	root = s[0];
	for(int i=1;i<SZ;i++)
		merge({root, s[i]}, root);
}

// Splits root into SZ EXCLUSIVE intervals
// [0..s[0]), [s[0]..s[1]), [s[1]..s[2])... [s[SZ-1]..end)
// Example: split<2>({l, r}) gets the exclusive interval [l, r)
template <int SZ>
array<no*, SZ> split(array<int, SZ-1> s) {
	array<no*, SZ> res;
	array<no*, 2> aux;
	split(root, s[0], aux);
	res[0] = aux[0]; res[1] = aux[1];
	for(int i=1;i<SZ-1;i++) {
		split(res[i], s[i]-s[i-1], aux);
		res[i] = aux[0]; res[i+1] = aux[1];
	}
	root = nullptr;
	return res;
}

void insert(T val, int idx) {
	auto s = split<2>({idx});
	merge<3>({s[0], new_no(val), s[1]});
}

void erase(int idx) {
	auto s = split<3>({idx, idx+1});
	merge<2>({s[0], s[2]});
}

// Inclusive
void reverse(int l, int r) {
	auto s = split<3>({l, r+1});
	s[1]->rev = !s[1]->rev;
	merge<3>(s);
}

T operator[](int idx) {
	no* x = root;
	//assert(0 <= idx && idx < x->cnt);
	x->prop();
	for(int ra = x->l(); ra != idx; ra += x->l()) {
		if(ra < idx) ra++, x = x->c[1];
		else ra -= x->l(), x = x->c[0];
		x->prop();
```

```
			}
			return x->dat;
		}
};
```

4.2 Seg Persistente

```
/* Persistent segment tree. This example is for queries of sum in range
 * and updates of sum in position, but any query or update can be achieved
 * changing the NEUT value, and functions updNode and merge.
 * The version 0 of the persistent segTree has implicitly an array of
 * length n full of NEUT values.
 *
 * It's recommend to set n as the actual length of the array.
 * ```
 * int n; cin >> n;
 * segTree::n = n;
 * ```
 *
 * Complexity: O(logn) memory and time per query/update
 * 50e1d1
 */

const int NEUT = 0;

struct segTree {
	vector<int> t = vector<int>(1, NEUT);
	vector<int> left = vector<int>(1, 0), right = vector<int>(1, 0);
	static int n;
	int newNode(int v, int l=0, int r=0) {
		t.pb(v), left.pb(l), right.pb(r);
		return sz(t) - 1;
	}
	int merge(int a, int b) {
		return a + b;
	}
	// Initializes a segTree with the values of the array A of length n
	int init(int* A, int L=0, int R=n) {
		if(L + 1 == R) return newNode(A[L]);
		int M = (L + R) >> 1;
		int l = init(A, L, M), r = init(A, M, R);
		return newNode(merge(t[l], t[r]), l , r);
	}
	int updNode(int cur_value, int upd_value) {
		return cur_value + upd_value;
	}
	// updates the position pos of version k with the value v
	int upd(int k, int pos, int v, int L=0, int R=n) {
		int nxt = newNode(t[k], left[k], right[k]);
		if(L + 1 == R) t[nxt] = updNode(t[nxt], v);
		else {
			int M = (L + R) >> 1;
			int temp;
			if(pos < M) temp = upd(left[nxt], pos, v, L, M), left[nxt] = temp;
			else temp = upd(right[nxt], pos, v, M, R), right[nxt] = temp;
			t[nxt] = merge(t[left[nxt]], t[right[nxt]]);
		}
		return nxt;
	}
	// query in the range [l, r) of version k
	int que(int k, int l, int r, int L=0, int R=n){
		if(r <= L || R <= l) return NEUT;
		if(l <= L && R <= r) return t[k];
		int M = (L + R) >> 1;
```

```
            return merge(que(left[k], l, r, L, M), que(right[k], l, r, M, R));
        }
};
int segTree::n = N;
```

5 Extra

5.1 hash

```
/* Persistent segment tree. This example is for queries of sum in range
 * and updates of sum in position, but any query or update can be achieved
 * changing the NEUT value, and functions updNode and merge.
 * The version 0 of the persistent segTree has implicitly an array of
 * length n full of NEUT values.
 *
 * It's recommend to set n as the actual length of the array.
 * ```
 * int n; cin >> n;
 * segTree::n = n;
 * ```
 *
 * Complexity: O(logn) memory and time per query/update
 * 50e1d1
 */

const int NEUT = 0;

struct segTree {
        vector<int> t = vector<int>(1, NEUT);
        vector<int> left = vector<int>(1, 0), right = vector<int>(1, 0);
        static int n;
        int newNode(int v, int l=0, int r=0) {
                t.pb(v), left.pb(l), right.pb(r);
                return sz(t) - 1;
        }
        int merge(int a, int b) {
                return a + b;
        }
        // Initializes a segTree with the values of the array A of length n
        int init(int* A, int L=0, int R=n) {
                if(L + 1 == R) return newNode(A[L]);
                int M = (L + R) >> 1;
                int l = init(A, L, M), r = init(A, M, R);
                return newNode(merge(t[l], t[r]), l , r);
        }
        int updNode(int cur_value, int upd_value) {
                return cur_value + upd_value;
        }
        // updates the position pos of version k with the value v
        int upd(int k, int pos, int v, int L=0, int R=n) {
                int nxt = newNode(t[k], left[k], right[k]);
                if(L + 1 == R) t[nxt] = updNode(t[nxt], v);
                else {
                        int M = (L + R) >> 1;
                        int temp;
                        if(pos < M) temp = upd(left[nxt], pos, v, L, M), left[nxt] = temp;
                        else temp = upd(right[nxt], pos, v, M, R), right[nxt] = temp;
                        t[nxt] = merge(t[left[nxt]], t[right[nxt]]);
                }
                return nxt;
        }
        // query in the range [l, r) of version k
        int que(int k, int l, int r, int L=0, int R=n){
                if(r <= L || R <= l) return NEUT;
```

```
            if(l <= L && R <= r) return t[k];
            int M = (L + R) >> 1;
            return merge(que(left[k], l, r, L, M), que(right[k], l, r, M, R));
        }
};
int segTree::n = N;
```

5.2 PBDS

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
// iterator find_by_order(size_t index), size_t order_of_key(T key)
template <typename T>
using ordered_set=__gnu_pbds::tree<T, __gnu_pbds::null_type,std::less<T>, __gnu_pbds::rb_tree_tag,__gnu_pbd
```

5.3 Random

```
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
shuffle(permutation.begin(), permutation.end(), rng);
uniform_int_distribution<int>(a,b)(rng);
```

5.4 Templat

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

#define all(x) x.begin(), x.end()
#define IOS ios::sync_with_stdio(0);cin.tie(0)

using ll = long long;

void dbg_out() { cerr << endl; }
template<typename Head, typename... Tail> void dbg_out(Head H, Tail... T){
  cerr << ' ' << H;
  dbg_out(T...);
}
#define dbg(...) cerr<<"(" << #__VA_ARGS__<<"):" , dbg_out(__VA_ARGS__) , cerr << endl

void solve(){

}

signed main(){
    IOS;
    solve();
}
```