Team notebook

July 6, 2022

Contents

1 algorithms

1.1 Mo's Algorithm

```
#include "bits/stdc++.h"
using namespace std;
#define for_(i, s, e) for (int i = s; i < (int) e; i++)
#define for__(i, s, e) for (ll i = s; i < e; i++)
typedef long long ll;
typedef vector<int> vi;
typedef array<int, 2> ii;
#define endl '\n'

int n, m;
int block_size = 450;
vi freq(1e6+1);
ll val = 0;
vi nums;

struct Query {
	int l, r, idx;
	bool operator<(Query other) const {
		int b1 = l/block_size, b2 = other.l/block_size;
		if (b1 != b2) return b1 < b2;
		else if (b1 % 2 == 0) return r < other.r;
		else return r > other.r;
	}
};

void add(ll v) {
	freq[v] += 1;
}

void remove(ll v) {
	freq[v] -= 1;
}

int main() {
	#ifdef mlocal
	freopen("test.in", "r", stdin);
	#endif

	ios_base::sync_with_stdio(false);
	cin.tie(0);

	cin >> n >> m;
	nums.resize(n);
	for_(i, 0, n) cin >> nums[i];

	vector<Query> queries(m);
	vector<ll> ans(m);
	for_(i, 0, m) {
		cin >> queries[i].l >> queries[i].r;
		queries[i].r -= 1; queries[i].l -= 1;
		queries[i].idx = i;
	}

	sort(queries.begin(), queries.end());

	int cur_l = 0, cur_r = -1;
	for (auto q: queries) {
		while (cur_l > q.l) {
			cur_l--;
			add(nums[cur_l]);
		}
		while (cur_r < q.r) {
			cur_r++;
			add(nums[cur_r]);
		}
		while (cur_l < q.l) {
			remove(nums[cur_l]);
			cur_l++;
		}
		while (cur_r > q.r) {
			remove(nums[cur_r]);
			cur_r--;
		}

		ans[q.idx] = val;
	}
}
```

1.2 OrderStatisticTree

```
/**
 * Description: A set (not multiset!) with support for finding the n'th
 * element, and finding the index of an element.
 * To get a map, change \texttt{null\_type}.
 * Time: O(\log N)
 */
#pragma once

#include <bits/extc++.h> /** keep-include */
using namespace __gnu_pbds;

template<class T>
using Tree = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

void example() {
	Tree<int> t, t2; t.insert(8);
	auto it = t.insert(10).first;
	assert(it == t.lower_bound(9));
	assert(t.order_of_key(10) == 1);
	assert(t.order_of_key(11) == 2);
	assert(*t.find_by_order(0) == 8);
	t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}
```

1.3 TernarySearch

```
/**
 * Description:
 * Find the smallest i in $[a,b]$ that maximizes $f(i)$, assuming that
     $f(a) < \dots < f(i) \ge \dots \ge f(b)$.
 * To reverse which of the sides allows non-strict inequalities, change
     the < marked with (A) to <=, and reverse the loop at (B).
 * To minimize $f$, change it to >, also at (B).
 * Usage:
	int ind = ternSearch(0,n-1,[\&](int i){return a[i];});
 * Time: O(\log(b-a))
 */
#pragma once

template<class F>
int ternSearch(int a, int b, F f) {
	assert(a <= b);
	while (b - a >= 5) {
		int mid = (a + b) / 2;
		if (f(mid) < f(mid+1)) a = mid; // (A)
		else b = mid+1;
	}
	rep(i,a+1,b+1) if (f(a) < f(i)) a = i; // (B)
	return a;
}
```

1.4 centroid-decomp

```
int n;
vector<vector<array<ll, 2>>> adjList;
vi removed, subtreeSize, centroid;
vector<ll> whiteDist;
vector<vector<ll>> centroidDist;
const ll INF = 1e15;

void dfs(int p, int parent) {
	for (auto i: adjList[p]) if (i[0] != parent) {
			dfs(i[0], p);
			subtreeSize[p] += subtreeSize[i[0]];
		}

	subtreeSize[p] += 1;
}

void noteCentroidDist(int p, int parent) {
	ll d = centroidDist[p].back();
	for (auto i: adjList[p]) if (!removed[i[0]] and i[0] != parent) {
			centroidDist[i[0]].push_back(d+i[1]);
			noteCentroidDist(i[0], p);
		}
}

void decompose(int p, int c) {
	int invalidChild = -1, sizeLimit = (subtreeSize[p] >> 1);
	for (auto i: adjList[p]) if (!removed[i[0]] and subtreeSize[i[0]]
	    > sizeLimit) {
			invalidChild = i[0];
			break;
```

```
		}

	if (invalidChild != -1) {
		subtreeSize[p] -= subtreeSize[invalidChild];
		subtreeSize[invalidChild] += subtreeSize[p];
		return decompose(invalidChild, c);
	}

	removed[p] = true;
	centroid[p] = c;

	centroidDist[p].push_back(0);
	noteCentroidDist(p, p);

	for (auto i: adjList[p]) if (!removed[i[0]]) {
			centroid[i[0]] = p;
			decompose(i[0], p);
		}
}

vi updated;
int pt = 0;

void update(int p) {
	int v = p, cpt = (int) centroidDist[p].size() - 1;
	while (v != -1) {
		ll d = centroidDist[p][cpt--];
		whiteDist[v] = min(whiteDist[v], d);
		updated[pt++] = v;
		v = centroid[v];
	}
}

ll ans(int p) {
	ll val = INF;
	int v = p, cpt = (int) centroidDist[p].size() - 1;
	while (v != -1) {
		val = min(val, whiteDist[v] + centroidDist[p][cpt--]);
		v = centroid[v];
	}
	return (val == 1e9 ? -1 : val);
}

void Init(int N, int A[], int B[], int D[]) {
	n = N;
	adjList.resize(n); subtreeSize.resize(n); removed.resize(n);
	    centroid.resize(n, -1); whiteDist.resize(n, INF);
	    updated.resize(10*n); centroidDist.resize(n);
	for_(i, 0, n-1) {
		int a = A[i], b = B[i]; ll w = D[i];
		adjList[a].push_back({b, w});
		adjList[b].push_back({a, w});
	}

	dfs(0, 0);
	decompose(0, -1);
}

ll Query(int S, int X[], int T, int Y[]) {
	pt = 0;

	for_(i, 0, S) update(X[i]);
	ll val = INF;
	for_(i, 0, T) val = min(val, ans(Y[i]));
	for_(i, 0, pt) whiteDist[updated[i]] = INF;

	return val;
}

int main() {
	freopen("test.in", "r", stdin);

	ios_base::sync_with_stdio(false);
	cin.tie(0);

	int N, q;
	cin >> N >> q;
	int A[100], B[100], D[100];
	for_(i, 0, N - 1) {
		int a, b, w;
		cin >> a >> b >> w;
		A[i] = a;
		B[i] = b;
		D[i] = w;
	}

	Init(N, A, B, D);

	int X[100], Y[100];
	while (q--) {
```

```
        int S, T;
        cin >> S >> T;
        for_(i, 0, S) cin >> X[i];
        for_(i, 0, T) cin >> Y[i];
        cout << Query(S, X, T, Y) << endl;
    }

    return 0;
}
```

1.5 gphashtable

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

const int RANDOM =
    chrono::high_resolution_clock::now().time_since_epoch().count();
struct chash {
    int operator()(int x) const { return x ^ RANDOM; }
};
gp_hash_table<key, int, chash> table;
```

2 data structures

2.1 2D Prefix Sum

```
ll sum(vector<vector<ll>> &mat, int x1, int y1, int x2, int y2) {
    ll ans = mat[x2][y2];
    if (x1 > 0) ans -= mat[x1-1][y2];
    if (y1 > 0) ans -= mat[x2][y1-1];
    if (x1 > 0 and y1 > 0) ans += mat[x1-1][y1-1];
    return ans;
}

void make(vector<vector<ll>> &mat) {
    for_(i, 0, n) for_(j, 0, m) {
        if (i > 0) mat[i][j] += mat[i-1][j];
        if (j > 0) mat[i][j] += mat[i][j-1];
        if (i > 0 and j > 0) mat[i][j] -= mat[i-1][j-1];
    }
}
```

2.2 Dynamic CHT LineContainer

```
struct Line {
    mutable ll k, m, p;

    bool operator<(const Line &o) const { return k < o.k; }

    bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
    // (for doubles, use inf = 1/.0, div(a,b) = a/b)
    static const ll inf = LLONG_MAX;

    ll div(ll a, ll b) { // floored division
        return a / b - ((a ^ b) < 0 && a % b);
    }

    bool isect(iterator x, iterator y) {
        if (y == end()) return x->p = inf, 0;
        if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
        else x->p = div(y->m - x->m, x->k - y->k);
        return x->p >= y->p;
    }

    void add(ll k, ll m) {
        auto z = insert({k, m, 0}), y = z++, x = y;
        while (isect(y, z)) z = erase(z);
        if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
        while ((y = x) != begin() && (--x)->p >= y->p)
            isect(x, erase(y));
    }

    ll query(ll x) {
        assert(!empty());
        auto l = *lower_bound(x);
        return l.k * x + l.m;
    }
};
```

2.3 Fenwick Tree - 2D

```
// Ask for sum 1 -> n for full (one based indexing)
class BIT {
	private: vector<vector<ll>> tree; int n, m;
	int LSOne(int x) {
		return x&(-x);
	}

	public:
		BIT(int r, int c) {
			n = r, m = c;
			tree.resize(n+1, vector<ll>(m+1));
		}
		ll sum(int x, int y) {
			ll sum = 0;
			for (; x > 0; x -= LSOne(x)) for (int yy = y; yy >
			    0; yy -= LSOne(yy)) sum += tree[x][yy];
			return sum;
		}
		ll sum(int x1, int y1, int x2, int y2) {
			ll a = sum(x2, y2);
			if (x1 > 1) a -= sum(x1-1, y2);
			if (y1 > 1) a -= sum(x2, y1-1);
			if (x1 > 1 and y1 > 1) a += sum(x1-1, y1-1);
			return a;
		}
		void update(int x, int y, ll v) {
			for (; x < n+1; x += LSOne(x)) for (int yy = y; yy
			    < m+1; yy += LSOne(yy)) tree[x][yy] += v;
		}
};
```

2.4 Fenwick Tree

```
// Ask for sum 1 -> n for full (one based indexing)
class BIT {
	private: vector<ll> tree; int n;
	int LSOne(int x) {
		return x&(-x);
	}

	public:
		BIT(int x) {
			n = x;
			tree.resize(n+1);
		}
		ll sum(int a) {
			ll sum = 0;
			for (; a > 0; a -= LSOne(a)) sum += tree[a];
			return sum;
		}
		ll sum(int a, int b) {
			return sum(b) - (a == 1 ? 0 : sum(a-1));
		}
		void update(int p, ll v) {
			for (; p < n+1; p += LSOne(p)) tree[p] += v;
		}
};
```

2.5 Max Segment Tree Lazy Prop

```
const ll INF = 1e16;

class MaxSegmentTree {
private:
	struct node {
		ll val;
		ll lazy;
	};

	vector<node> tree; int n;

	void push(int p, int c1, int c2) {
		ll toadd = tree[p].lazy;
		tree[c1].lazy += toadd; tree[c2].lazy += toadd;
		tree[c1].val += toadd; tree[c2].val += toadd;
		tree[p].lazy = 0;
	}

	void update(int i, ll val, int l, int r, int p) {
		if (l > i or r < i) return;
		if (l == r) {
			tree[p].val = val;
			return;
		}
```

```
        int mid = (l + r) / 2;
        int c1 = 2*p+1, c2 = 2*p+2;
        if (tree[p].lazy) push(p, c1, c2);

        update(i, val, l, mid, c1); update(i, val, mid+1, r, c2);
        tree[p].val = max(tree[c1].val, tree[c2].val);
    }

    void add(int i, int j, ll val, int l, int r, int p) {
        if (l > j or r < i) return;
        if (l >= i and r <= j) {
            tree[p].val += val, tree[p].lazy += val;
            return;
        }

        int mid = (l + r) / 2;
        int c1 = 2*p+1, c2 = 2*p+2;
        if (tree[p].lazy) push(p, c1, c2);

        add(i, j, val, l, mid, c1); add(i, j, val, mid+1, r, c2);
        tree[p].val = max(tree[c1].val, tree[c2].val);
    }

    ll mx(int i, int j, int l, int r, int p) {
        if (l > j or r < i) return -INF;
        if (l >= i and r <= j) return tree[p].val;

        int mid = (l + r) / 2;
        int c1 = 2*p+1, c2 = 2*p+2;
        if (tree[p].lazy) push(p, c1, c2);

        return max(mx(i, j, l, mid, c1), mx(i, j, mid+1, r, c2));
    }

public:
    MaxSegmentTree(int _n) {
        n = _n;
        tree.assign(4*n, {-INF, 0});
    }

    ll mx(int i, int j) {
        return mx(i, j, 0, n-1, 0);
    }

    void update(int i, ll val) {
        update(i, val, 0, n-1, 0);
    }

    void add(int i, int j, ll val) {
        add(i, j, val, 0, n-1, 0);
    }
};
```

2.6 Segment Tree (range max subarray sum)

```
struct Node {
    ll pref, suf, mx, sum;
};

class SegmentTree {
    private:
        vector<Node> tree; vector<ll> raw; int n;
        ll INF = 1e15;
        Node bound = {0, 0, -INF, 0};

        Node merge(Node a, Node b) {
            if (a.mx == -INF) return b;
            if (b.mx == -INF) return a;

            Node ans;
            ans.sum = a.sum + b.sum;
            ans.mx = max({a.mx, b.mx, b.pref + a.suf});
            ans.suf = max(b.suf, b.sum + a.suf);
            ans.pref = max(a.pref, a.sum + b.pref);
            return ans;
        }

        void buildTree(int l, int r, int p) {
            if (l == r) {
                tree[p] = {raw[l], raw[l], raw[l], raw[l]};
                return;
            }

            int mid = (l + r) / 2;
            int c1 = 2*p+1, c2 = 2*p+2;
            buildTree(l, mid, c1); buildTree(mid+1, r, c2);
            tree[p] = merge(tree[c1], tree[c2]);
```

```
	}

	void update(int i, ll val, int l, int r, int p) {
		if (l > i or r < i) return;
		if (l == r) {
			tree[p] = {val, val, val, val};
			return;
		}

		int mid = (l + r) / 2;
		int c1 = 2*p+1, c2 = 2*p+2;
		update(i, val, l, mid, c1); update(i, val, mid+1,
		    r, c2);
		tree[p] = merge(tree[c1], tree[c2]);
	}

	Node mx(int i, int j, int l, int r, int p) {
		if (l > j or r < i) return bound;
		if (l >= i and r <= j) return tree[p];

		int mid = (l + r) / 2;
		int c1 = 2*p+1, c2 = 2*p+2;
		return merge(mx(i, j, l, mid, c1), mx(i, j, mid+1,
		    r, c2));
	}

public:
	SegmentTree(vector<ll> input) {
		raw = input;
		n = raw.size();
		tree.resize(4*n);
		buildTree(0, n-1, 0);
	}

	ll mx(int i, int j) {
		return mx(i, j, 0, n-1, 0).mx;
	}

	void update(int i, ll val) {
		update(i, val, 0, n-1, 0);
	}
};
```

2.7 Sparse Table (pair range min for LCA)

```
class SparseTable {
	vector<vector<ii>> st;
	int n;

	void buildTable(vector<ii> &raw) {
		int k = __lg(n)+1;
		st.resize(k, vector<ii>(n));
		for_(i, 0, n) st[0][i] = raw[i];
		for_(p, 1, k+1) for_(i, 0, n - (1<<p) + 1) st[p][i] =
		    min(st[p-1][i], st[p-1][i + (1 << (p-1))]);
	}

public:
	void build(vector<ii> &nums) {
		n = nums.size();
		buildTable(nums);
	}

	int mn(int l, int r) {
		if (l > r) swap(l, r);
		int p = __lg(r-l+1);
		return min(st[p][l], st[p][r - (1<<p) + 1])[1];
	}
};

vector<ii> tour;

void dfs(int p, int parent, int d) {
	tin[p] = tour.size();
	tour.push_back({d, p});

	for (auto i: adj[p]) if (i != parent) {
		dfs(i, p, d+1);
		tour.push_back({d, p});
	}
}

function<int(int, int)> nodeDist = [&] (int a, int b) {
	int lca = st.mn(min(tin[a], tin[b]), max(tin[a], tin[b]));
	return depth[a] + depth[b] - 2 * depth[lca];
};
```

2.8 Trie

```
class Trie {
	static const int MXN = 1e6, MXV = 26; // max node count (worst
	    case: max sequence length * count), max sequence value
	int node = 1, child[MXN+1][MXV+1], ct[MXN+1];

	public:
	void insert(vi s) {
		int p = 0;
		for (int i: s) {
			if (!child[p][i]) child[p][i] = node++;
			p = child[p][i];
			ct[p] += 1;
		}
	}

	int countPref(vi s) {
		int ans = 0, p = 0;
		for (int i: s) {
			if (!child[p][i]) return 0;
			p = child[p][i];
			ans = ct[p];
		}

		return ans;
	}
} trie;

// convert string to sequence of integers
vi stov(string s) {
	vi ans(s.size());
	for_(i, 0, s.size()) ans[i] = s[i]-'a';
	return ans;
}
```

2.9 suffix-arrays

```
/**
 * Description: Builds suffix array for a string.
 * \texttt{sa[i]} is the starting index of the suffix which
 * is $i$'th in the sorted suffix array.
 * The returned vector is of size $n+1$, and \texttt{sa[0] = n}.
 * The \texttt{lcp} array contains longest common prefixes for
 * neighbouring strings in the suffix array:
 * \texttt{lcp[i] = lcp(sa[i], sa[i-1])}, \texttt{lcp[0] = 0}.
 * The input string must not contain any zero bytes.
 * Time: O(n \log n)
 * Status: stress-tested
 */
#pragma once

struct SuffixArray {
	vi sa, lcp;
	SuffixArray(string& s, int lim=256) { // or basic_string<int>
		int n = sz(s) + 1, k = 0, a, b;
		vi x(all(s)+1), y(n), ws(max(n, lim)), rank(n);
		sa = lcp = y, iota(all(sa), 0);
		for (int j = 0, p = 0; p < n; j = max(1, j * 2), lim = p) {
			p = j, iota(all(y), n - j);
			rep(i,0,n) if (sa[i] >= j) y[p++] = sa[i] - j;
			fill(all(ws), 0);
			rep(i,0,n) ws[x[i]]++;
			rep(i,1,lim) ws[i] += ws[i - 1];
			for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
			swap(x, y), p = 1, x[sa[0]] = 0;
			rep(i,1,n) a = sa[i - 1], b = sa[i], x[b] =
				(y[a] == y[b] && y[a + j] == y[b + j]) ? p -
				    1 : p++;
		}
		rep(i,1,n) rank[sa[i]] = i;
		for (int i = 0, j; i < n - 1; lcp[rank[i++]] = k)
			for (k && k--, j = sa[rank[i] - 1];
					s[i + k] == s[j + k]; k++);
	}
};
```

2.10 treap

```
struct item {
   int key, prior;
   item *l, *r;
   item () { }
   item (int key) : key(key), prior(rand()), l(NULL), r(NULL) { }
   item (int key, int prior) : key(key), prior(prior), l(NULL), r(NULL)
       { }
```

```
};
typedef item* pitem;

void split (pitem t, int key, pitem & l, pitem & r) {
    if (!t)
        l = r = NULL;
    else if (t->key <= key)
        split (t->r, key, t->r, r), l = t;
    else
        split (t->l, key, l, t->l), r = t;
}

// pitem l = nullptr, r = nullptr;
// split(t, 5, l, r);
// if (l) cout << "Left subtree size: " << (l->size) << endl;
// if (r) cout << "Right subtree size: " << (r->size) << endl;

void insert (pitem & t, pitem it) {
    if (!t)
        t = it;
    else if (it->prior > t->prior)
        split (t, it->key, it->l, it->r), t = it;
    else
        insert (t->key <= it->key ? t->r : t->l, it);
}

void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r), t = l;
    else
        merge (r->l, l, r->l), t = r;
}

void erase (pitem & t, int key) {
    if (t->key == key) {
        pitem th = t;
        merge (t, t->l, t->r);
        delete th;
    }
    else
        erase (key < t->key ? t->l : t->r, key);
}
pitem unite (pitem l, pitem r) {
    if (!l || !r) return l ? l : r;
    if (l->prior < r->prior) swap (l, r);
    pitem lt, rt;
    split (r, l->key, lt, rt);
    l->l = unite (l->l, lt);
    l->r = unite (l->r, rt);
    return l;
}

int cnt (pitem t) {
    return t ? t->cnt : 0;
}

void upd_cnt (pitem t) {
    if (t)
        t->cnt = 1 + cnt(t->l) + cnt (t->r);
}
```

3 graph

3.1 DFSMatching

```
/**
 * Description: Simple bipartite matching algorithm. Graph $g$ should be
     a list
 * of neighbors of the left partition, and $btoa$ should be a vector full
     of
 * -1's of the same size as the right partition. Returns the size of
 * the matching. $btoa[i]$ will be the match for vertex $i$ on the right
     side,
 * or $-1$ if it's not matched.
 * Time: O(VE)
 * Usage: vi btoa(m, -1); dfsMatching(g, btoa);
 */
#pragma once

bool find(int j, vector<vi>& g, vi& btoa, vi& vis) {
        if (btoa[j] == -1) return 1;
        vis[j] = 1; int di = btoa[j];
        for (int e : g[di])
                if (!vis[e] && find(e, g, btoa, vis)) {
```

```
				btoa[e] = di;
				return 1;
			}
		return 0;
}
int dfsMatching(vector<vi>& g, vi& btoa) {
	vi vis;
	rep(i,0,sz(g)) {
		vis.assign(sz(btoa), 0);
		for (int j : g[i])
			if (find(j, g, btoa, vis)) {
				btoa[j] = i;
				break;
			}
	}
	return sz(btoa) - (int)count(all(btoa), -1);
}
```

3.2 Dinic

```
/**
 * Description: Flow algorithm with complexity $O(VE\log U)$ where $U =
     \max |\text{cap}|$.
 * $O(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$; $O(\sqrt{V}E)$ for bipartite
     matching.
 * Status: Tested on SPOJ FASTFLOW and SPOJ MATCHING, stress-tested
 */
#pragma once

struct Dinic {
	struct Edge {
		int to, rev;
		ll c, oc;
		ll flow() { return max(oc - c, 0LL); } // if you need flows
	};
	vi lvl, ptr, q;
	vector<vector<Edge>> adj;
	Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {}
	void addEdge(int a, int b, ll c, ll rcap = 0) {
		adj[a].push_back({b, sz(adj[b]), c, c});
		adj[b].push_back({a, sz(adj[a]) - 1, rcap, rcap});
	}
	ll dfs(int v, int t, ll f) {
		if (v == t || !f) return f;
		for (int& i = ptr[v]; i < sz(adj[v]); i++) {
			Edge& e = adj[v][i];
			if (lvl[e.to] == lvl[v] + 1)
				if (ll p = dfs(e.to, t, min(f, e.c))) {
					e.c -= p, adj[e.to][e.rev].c += p;
					return p;
				}
		}
		return 0;
	}
	ll calc(int s, int t) {
		ll flow = 0; q[0] = s;
		rep(L,0,31) do { // 'int L=30' maybe faster for random data
			lvl = ptr = vi(sz(q));
			int qi = 0, qe = lvl[s] = 1;
			while (qi < qe && !lvl[t]) {
				int v = q[qi++];
				for (Edge e : adj[v])
					if (!lvl[e.to] && e.c >> (30 - L))
						q[qe++] = e.to, lvl[e.to] =
						   lvl[v] + 1;
			}
			while (ll p = dfs(s, t, LLONG_MAX)) flow += p;
		} while (lvl[t]);
		return flow;
	}
	bool leftOfMinCut(int a) { return lvl[a] != 0; }
};
```

3.3 EulerWalk

```
/**
 * Description: Eulerian undirected/directed path/cycle algorithm.
 * Input should be a vector of (dest, global edge index), where
 * for undirected graphs, forward/backward edges have the same index.
 * Returns a list of nodes in the Eulerian path/cycle with src at both
     start and end, or
 * empty list if no cycle/path exists.
 * To get edge indices back, add .second to s and ret.
 * Time: O(V + E)
 */
#pragma once
```

```cpp
vi eulerWalk(vector<vector<pii>>& gr, int nedges, int src=0) {
	int n = sz(gr);
	vi D(n), its(n), eu(nedges), ret, s = {src};
	D[src]++; // to allow Euler paths, not just cycles
	while (!s.empty()) {
		int x = s.back(), y, e, &it = its[x], end = sz(gr[x]);
		if (it == end){ ret.push_back(x); s.pop_back(); continue; }
		tie(y, e) = gr[x][it++];
		if (!eu[e]) {
			D[x]--, D[y]++;
			eu[e] = 1; s.push_back(y);
		}}
	for (int x : D) if (x < 0 || sz(ret) != nedges+1) return {};
	return {ret.rbegin(), ret.rend()};
}
```

3.4 HLD

```cpp
/**
 * TODO
 * 1. Stress-test
 * NOTE
 * 1. When handling edge weights, REMEMBER to exclude LCA.
 *     - in query(a, b), change the last st query to st[a]+1, st[b]
*/

struct HLD{
    const int identity = 0;
    int merge(int a, int b) { return a+b; }

    int n, root, timer = 0, t;
    vi size, heavy, top, st, en, tree, dep, par;

    void st_update(int v, int val){
        for(tree[v+=t] = val; v > 1; v >>= 1)
            tree[v>>1] = merge(tree[v], tree[v^1]);
    }

    int st_query(int l, int r){
        int res = identity;
        for(l += t, r += t; l <= r; l>>=1, r>>=1){
            if(l == r) return merge(res, tree[l]);
            if(l&1) res = merge(res, tree[l++]);
            if(!(r&1)) res = merge(res, tree[r--]);
        }
        return res;
    }

    int query(int a, int b){
        int ans = identity;
        for(; top[a] != top[b]; b = par[top[b]]){
            if(dep[top[a]] > dep[top[b]]) swap(a, b);
            ans = merge(ans, st_query(st[top[b]], st[b]));
        }
        if(dep[a] > dep[b]) swap(a, b);
        ans = merge(ans, st_query(st[a], st[b]));
        return ans;
    }

    void update(int node, int val){
        st_update(st[node], val);
        st_update(en[node], -val);
    }

    void build(){
        t = sz(tree);
        tree.resize(2*t);
        for(int i=0; i<t; i++) tree[t+i] = tree[i];
        for(int i=t-1; i >= 1; i--)
            tree[i] = merge(tree[2*i], tree[2*i+1]);
    }

    void dfs_hvy(int v, vvi &adj, int p){
        top[v] = v;
        for(auto &to:adj[v]){
            if(to == p) continue;
            dep[to] = dep[v] + 1;
            par[to] = v;
            dfs_hvy(to, adj, v);
            size[v] += size[to];
            if(heavy[v] == -1 or size[to] > size[heavy[v]]) heavy[v] = to;
        }
    }

    void dfs_hld(int v, vvi &adj, int p, vi &arr){
        st[v] = timer++; tree.pb(arr[v]);
```

```
        if(heavy[v] != -1){
            top[heavy[v]] = top[v];
            dfs_hld(heavy[v], adj, v, arr);
        }

        for(auto &to:adj[v]){
            if(to == p or to == heavy[v]) continue;
            dfs_hld(to, adj, v, arr);
        }

        en[v] = timer++; tree.pb(-arr[v]);
    }

    HLD(vvi &adj, int r, vi &arr) : n(sz(adj)), heavy(n, -1), top(n),
        st(n), en(n), dep(n), par(n) {
        size.assign(n, 1);
        tree.reserve(2*n);
        root = r;
        dfs_hvy(root, adj, -1);
        dfs_hld(root, adj, -1, arr);
        build();
    }
};
```

3.5 MaximumIndependentSet

```
/**
 * Author: chilli
 * Date: 2019-05-17
 * Source: Wikipedia
 * Description: To obtain a maximum independent set of a graph, find a max
 * clique of the complement. If the graph is bipartite, see
     MinimumVertexCover.
 */
```

3.6 MinCostMaxFlow

```
/**
 * Description: Min-cost max-flow. cap[i][j] != cap[j][i] is allowed;
     double edges are not.
 * If costs can be negative, call setpi before maxflow, but note that
     negative cost cycles are not supported.
 * To obtain the actual flow, look at positive values only.
 * Time: Approximately O(E^2)
 */
#pragma once

// #include <bits/extc++.h> /// include-line, keep-include

const ll INF = numeric_limits<ll>::max() / 4;
typedef vector<ll> VL;

struct MCMF {
        int N;
        vector<vi> ed, red;
        vector<VL> cap, flow, cost;
        vi seen;
        VL dist, pi;
        vector<pii> par;

        MCMF(int N) :
                N(N), ed(N), red(N), cap(N, VL(N)), flow(cap), cost(cap),
                seen(N), dist(N), pi(N), par(N) {}

        void addEdge(int from, int to, ll cap, ll cost) {
                this->cap[from][to] = cap;
                this->cost[from][to] = cost;
                ed[from].push_back(to);
                red[to].push_back(from);
        }

        void path(int s) {
                fill(all(seen), 0);
                fill(all(dist), INF);
                dist[s] = 0; ll di;

                __gnu_pbds::priority_queue<pair<ll, int>> q;
                vector<decltype(q)::point_iterator> its(N);
                q.push({0, s});

                auto relax = [&](int i, ll cap, ll cost, int dir) {
                        ll val = di - pi[i] + cost;
                        if (cap && val < dist[i]) {
                                dist[i] = val;
                                par[i] = {s, dir};
```

```
                if (its[i] == q.end()) its[i] =
                    q.push({-dist[i], i});
                else q.modify(its[i], {-dist[i], i});
            }
        };

        while (!q.empty()) {
            s = q.top().second; q.pop();
            seen[s] = 1; di = dist[s] + pi[s];
            for (int i : ed[s]) if (!seen[i])
                relax(i, cap[s][i] - flow[s][i], cost[s][i],
                    1);
            for (int i : red[s]) if (!seen[i])
                relax(i, flow[i][s], -cost[i][s], 0);
        }
        rep(i,0,N) pi[i] = min(pi[i] + dist[i], INF);
    }

    pair<ll, ll> maxflow(int s, int t) {
        ll totflow = 0, totcost = 0;
        while (path(s), seen[t]) {
            ll fl = INF;
            for (int p,r,x = t; tie(p,r) = par[x], x != s; x =
                p)
                fl = min(fl, r ? cap[p][x] - flow[p][x] :
                    flow[x][p]);
            totflow += fl;
            for (int p,r,x = t; tie(p,r) = par[x], x != s; x =
                p)
                if (r) flow[p][x] += fl;
                else flow[x][p] -= fl;
        }
        rep(i,0,N) rep(j,0,N) totcost += cost[i][j] * flow[i][j];
        return {totflow, totcost};
    }

    // If some costs can be negative, call this before maxflow:
    void setpi(int s) { // (otherwise, leave this out)
        fill(all(pi), INF); pi[s] = 0;
        int it = N, ch = 1; ll v;
        while (ch-- && it--)
            rep(i,0,N) if (pi[i] != INF)
                for (int to : ed[i]) if (cap[i][to])
                    if ((v = pi[i] + cost[i][to]) <
                        pi[to])
                        pi[to] = v, ch = 1;
        assert(it >= 0); // negative cost cycle
    }
};
```

3.7 MinCut

```
/**
 * Description: After running max-flow, the left side of a min-cut from
     $s$ to $t$ is given by all vertices reachable from $s$, only
     traversing edges with positive residual capacity.
 */
```

3.8 MinimumVertexCover

```
/**
 * Description: Finds a minimum vertex cover in a bipartite graph.
 *  The size is the same as the size of a maximum matching, and
 *  the complement is a maximum independent set.
 */
#pragma once

#include "DFSMatching.h"

vi cover(vector<vi>& g, int n, int m) {
    vi match(m, -1);
    int res = dfsMatching(g, match);
    vector<bool> lfound(n, true), seen(m);
    for (int it : match) if (it != -1) lfound[it] = false;
    vi q, cover;
    rep(i,0,n) if (lfound[i]) q.push_back(i);
    while (!q.empty()) {
        int i = q.back(); q.pop_back();
        lfound[i] = 1;
        for (int e : g[i]) if (!seen[e] && match[e] != -1) {
            seen[e] = true;
            q.push_back(match[e]);
        }
    }
    rep(i,0,n) if (!lfound[i]) cover.push_back(i);
    rep(i,0,m) if (seen[i]) cover.push_back(n+i);
```

```
    assert(sz(cover) == res);
    return cover;
}
```

3.9 bellman-ford

```
/*
Do N iterations of Bellman-Ford algorithm. If there were no changes on
    the last iteration, there is no cycle of negative weight in the
    graph. Otherwise take a vertex the distance to which has changed, and
    go from it via its ancestors until a cycle is found. This cycle will
    be the desired cycle of negative weight.
*/
struct edge
{
    int a, b, cost;
};

int n, m, v;
vector<edge> e;
const int INF = 1000000000;

void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    for (int i=0; i<n-1; ++i)
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                d[e[j].b] = min (d[e[j].b], d[e[j].a] + e[j].cost);
}
```

3.10 block-cut-tree

```
/**
 * TODO:
 * 1. Remove set<int> in tarjans, get complexity back to O(n). I think
     reverse dfs order works?
 * 2. Cleanup code, less hacky way to identify cut vertices on tree
*/
void tarjans_c(int node, vvi &adj, int p, vi &disc, vi &low, vb &iscut,
     vvi &components, vi &stack){
    static int timer = 1;
    if(disc[node]) return;
    disc[node] = low[node] = timer++;
    stack.pb(node);

    set<int> special;
    for(auto &to:adj[node]){
        if(to == p) continue;
        if(disc[to]) low[node] = min(low[node], disc[to]);
        else{
            tarjans_c(to, adj, node, disc, low, iscut, components, stack);
            low[node] = min(low[node], low[to]);
            if(p == -1) special.insert(to);
            if(low[to] >= disc[node] and p != -1) {
                components.pb({node});
                while(components.back().back() != to)
                    components.back().pb(stack.back()), stack.pop_back();
                iscut[node] = true;
            }
        }
    }
    if(p == -1 and sz(special) > 1) {
        for(auto &to:special){
            components.pb({node});
            while(special.find(components.back().back()) == special.end())
                components.back().pb(stack.back()), stack.pop_back();
        }
        iscut[node] = true;
    }
}

// Returns number of cut vertices. v < num_cut_vertices => v is a cut
     vertex in the tree
int blockCutTree(vvi &adj, vvi &tree, vi &m, vb &iscut){
    int n = sz(adj);
    vi disc(n), low(n), stack;
    m.assign(n, -1);
    tree.assign(2*n, vi());
    iscut.assign(n, false);
    vb vis(n);
    vvi components;
    tarjans_c(0, adj, -1, disc, low, iscut, components, stack);
```

```
    if(sz(stack) > 1) components.pb(stack); // If parent is not a cut
         vertex

    int id = 0;
    for(int v = 0; v<n; v++) if(iscut[v]) m[v] = id++;
    int ans = id;
    for(auto &comp:components){
        bool special = true;
        for(auto &v:comp)
            if(!iscut[v]) special = false;

        // If the size of a component is 2 and both vertices are cut
             vertices then
        // the tree has an edge between these two cut vertices.
        if(special && sz(comp) == 2){
            tree[m[comp[0]]].pb(m[comp[1]]);
            tree[m[comp[1]]].pb(m[comp[0]]);
        }
        else{
            int comp_id = id++;
            for(auto &v:comp){
                if(!iscut[v]) m[v] = comp_id;
                else {
                    tree[m[v]].pb(comp_id);
                    tree[comp_id].pb(m[v]);
                }
            }
        }
    }
    return ans;
}
```

3.11 bridge-cut-tree

```
void tarjans_b(int node, vvi &adj, vi &disc, vi &low, int p, DSU &dsu){
    static int timer = 1;
    if(disc[node]) return;
    low[node] = disc[node] = timer++;

    for(auto &x:adj[node]){
        if(x==p) continue;
        tarjans_b(x, adj, disc, low, node, dsu);
        low[node] = min(low[node], low[x]);
        if(low[x] <= disc[node])
            dsu.unify(node, x);
    }
}

void bridgeCutTree(vvi &adj, vvi &tree, DSU &dsu){
    int n = (int) adj.size();
    vi disc(n);
    vi low(n);
    dsu.make(n);
    tree.resize(n);

    tarjans_b(0, adj, disc, low, -1, dsu);
    for(int i=0; i<n; i++){
        for(auto j:adj[i]){
            int ip = dsu[i];
            int jp = dsu[j];
            if(ip==jp) continue;
            tree[ip].pb(jp);
        }
    }
}
```

4 math

4.1 fractions

```
// is a <= b?
bool comp(ii &a, ii &b) {
        if (a[0] >= 0 and b[0] < 0) return false;
        else if (a[0] < 0 and b[0] >= 0) return true;


        if (a == b or b[1] == 0) return true;
        else if (a[1] == 0) return false;

        bool flip = (a[1] < 0) ^ (b[1] < 0);
        bool ans = flip ^ (a[0] * b[1] < b[0] * a[1]);
        return flip ^ (a[0] * b[1] < b[0] * a[1]);
}

void normalise(ii &f) {
```

```
    ll g = __gcd(f[0], f[1]);
    if (g) f[0] /= g, f[1] /= g;
    if (f[1] < 0) f[0] = - f[0], f[1] = -f[1];
}

// x-coordinate of intersection of lines {m1, c1} {m2, c2} ->
    (c2-c1)/(m1-m2).
```

4.2 gauss

```
const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to be infinity or a big
    number

int gauss (vector < vector<double> > a, vector<double> & ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs (a[sel][col]))
                sel = i;
        if (abs (a[sel][col]) < EPS)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] * c;
            }
        ++row;
    }

    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)
            ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)
            sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > EPS)
            return 0;
    }

    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return INF;
    return 1;
}
```

4.3 matrix expo + multip

```
typedef vector<vector<ll>> Matrix;

const ll mod = 1e9+7;

Matrix matMul(Matrix a, Matrix b) {
        int sa = a.size(), sb = b[0].size();
        Matrix ans(sa, vector<ll> (sb));
        for_(r, 0, sa) for_(c, 0, sb) for_(i, 0, b.size()) {
                ans[r][c] = (ans[r][c] + (a[r][i] * b[i][c]) % mod);
                if (ans[r][c] >= mod) ans[r][c] -= mod;
        }
        return ans;
}

Matrix matPow(Matrix a, ll p) {
        int s = a.size();
        Matrix ans(s, vector<ll> (s));
        for_(i, 0, s) ans[i][i] = 1;

        while (p) {
                if (p & 1) ans = matMul(ans, a);
                a = matMul(a, a);
                p >>= 1;
        }
        return ans;
}
```

4.4 modInverse

```
ll power(ll x, ll y, ll m) {
	if (y == 0) return 1;
	ll p = power(x, y/2, m) % m;
	p = (p * p) % m;

	return (y%2 == 0)? p : (x * p) % m;
}

ll modInverse(ll a, ll mod) {
	return power(a, mod-2, mod);
}
```

4.5 modpow

```
ll modpow(ll a, ll b, ll mod) {
	ll res = 1;
	while (b) {
		if (b & 1) res = (res * a) % mod;
		a = (a * a) % mod;
		b >>= 1;
	}
	return res;
}
```

4.6 nCr mod P O(1)

```
const ll MAXN = 1000001, mod = 1e9+7;
ll numInv[MAXN+10], facInv[MAXN+10], fac[MAXN+10];

void pre() {
	numInv[0] = numInv[1] = 1;
	for (int i = 2; i <= MAXN; i++) numInv[i] = numInv[mod % i] * (mod
	    - mod / i) % mod;

	facInv[0] = facInv[1] = 1;
	for (int i = 2; i <= MAXN; i++) facInv[i] = (numInv[i] * facInv[i
	    - 1]) % mod;

	fac[0] = 1;
	for (int i = 1; i <= MAXN; i++) fac[i] = (fac[i - 1] * i) % mod;
}

ll nCr(ll n, ll r) {
	return ((fac[n] * facInv[r]) % mod * facInv[n - r]) % mod;
}
```

4.7 nCr mod P

```
ll mod = 1e9+7;
const ll MAXR = 1e6;

ll s, inv[MAXR+10];
ll nCr(ll n, ll r) {
	if (r > n) return 0;
	if (n - r < r) r = n - r;
	n %= mod;
	ll ans = 1;
	for_(i, 0, r) {
			ans = (ans * (n - i)) % mod;
			ans = (ans * inv[i + 1]) % mod;
	}
	return ans;
}

ll modpow(ll a, ll b, ll mod) {
	ll res = 1;
	while (b) {
		if (b & 1) res = (res * a) % mod;
		a = (a * a) % mod;
		b >>= 1;
	}
	return res;
}

void pre() {
	for (int i = 1; i <= MAXR; i++) {
		inv[i] = modpow(i, mod - 2, mod);
	}
}
```

4.8 totient-func

```
// Computes for n in sqrt(n)
int phi(int n) {
    int result = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }
    if (n > 1)
        result -= result / n;
    return result;
}

// Computes from 1 to n on O(n log log n)
void phi_1_to_n(int n) {
    vi phi(n + 1);
    for (int i = 0; i <= n; i++)
        phi[i] = i;

    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
        }
    }
}
```

5 number-theory

5.1 Factor

```
/**
 * Description: Pollard-rho randomized factorization algorithm. Returns
    prime
 * factors of a number, in arbitrary order (e.g. 2299 -> \{11, 19, 11\}).
 * Time: $O(n^{1/4})$, less for numbers with small factors.
 * Status: stress-tested
 *
 */
#pragma once

#include "ModMulLL.h"
#include "MillerRabin.h"

ull pollard(ull n) {
        auto f = [n](ull x) { return modmul(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 = modmul(prd, max(x,y) - min(x,y), n))) prd = q;
                x = f(x), y = f(f(y));
        }
        return __gcd(prd, n);
}
vector<ull> factor(ull n) {
        if (n == 1) return {};
        if (isPrime(n)) return {n};
        ull x = pollard(n);
        auto l = factor(x), r = factor(n / x);
        l.insert(l.end(), all(r));
        return l;
}
```

5.2 FastEratosthenes

```
/**
 * Description: Prime sieve for generating all primes smaller than LIM.
 * Time: LIM=1e9 $\approx$ 1.5s
 * Status: Stress-tested
 * Details: Despite its n log log n complexity, segmented sieve is still
    faster
 * than other options, including bitset sieves and linear sieves. This is
 * primarily due to its low memory usage, which reduces cache misses. This
 * implementation skips even numbers.
 */
#pragma once

const int LIM = 1e6;
bitset<LIM> isPrime;
vi eratosthenes() {
        const int S = (int)round(sqrt(LIM)), R = LIM / 2;
```

```
	vi pr = {2}, sieve(S+1); pr.reserve(int(LIM/log(LIM)*1.1));
	vector<pii> cp;
	for (int i = 3; i <= S; i += 2) if (!sieve[i]) {
		cp.push_back({i, i * i / 2});
		for (int j = i * i; j <= S; j += 2 * i) sieve[j] = 1;
	}
	for (int L = 1; L <= R; L += S) {
		array<bool, S> block{};
		for (auto &[p, idx] : cp)
			for (int i=idx; i < S+L; idx = (i+=p)) block[i-L] =
			    1;
		rep(i,0,min(S, R - L))
			if (!block[i]) pr.push_back((L + i) * 2 + 1);
	}
	for (int i : pr) isPrime[i] = 1;
	return pr;
}
```

5.3 MillerRabin

```
/**
 * Description: Deterministic Miller-Rabin primality test.
 * Guaranteed to work for numbers up to $7 \cdot 10^{18}$; for larger
    numbers, use Python and extend A randomly.
 * Time: 7 times the complexity of $a^b \mod c$.
 */
#pragma once

#include "ModMulLL.h"

bool isPrime(ull n) {
	if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
	ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
	    s = __builtin_ctzll(n-1), d = n >> s;
	for (ull a : A) { // ^ count trailing zeroes
		ull p = modpow(a%n, d, n), i = s;
		while (p != 1 && p != n - 1 && a % n && i--)
			p = modmul(p, p, n);
		if (p != n-1 && i != s) return 0;
	}
	return 1;
}
```

5.4 ModInverse

```
/**
 * Description: Pre-computation of modular inverses. Assumes LIM $\le$
    mod and that mod is a prime.
 */
#pragma once

// const ll mod = 1000000007, LIM = 200000; ///include-line
ll* inv = new ll[LIM] - 1; inv[1] = 1;
rep(i,2,LIM) inv[i] = mod - (mod / i) * inv[mod % i] % mod;
```

5.5 ModLog

```
/**
 * Description: Returns the smallest $x > 0$ s.t. $a^x = b \pmod m$, or
 * $-1$ if no such $x$ exists. modLog(a,1,m) can be used to
 * calculate the order of $a$.
 * Time: $O(\sqrt m)$
 */
#pragma once

ll modLog(ll a, ll b, ll m) {
	ll n = (ll) sqrt(m) + 1, e = 1, f = 1, j = 1;
	unordered_map<ll, ll> A;
	while (j <= n && (e = f = e * a % m) != b % m)
		A[e * b % m] = j++;
	if (e == b % m) return j;
	if (__gcd(m, e) == __gcd(m, b))
		rep(i,2,n+2) if (A.count(e = e * f % m))
			return n * i - A[e];
	return -1;
}
```

5.6 ModMulLL

```
/**
 * Description: Calculate $a\cdot b\bmod c$ (or $a^b \bmod c$) for $0 \le
    a, b \le c \le 7.2\cdot 10^{18}$.
 * Time: O(1) for \texttt{modmul}, O(\log b) for \texttt{modpow}
 */
```

```
#pragma once

typedef unsigned long long ull;
ull modmul(ull a, ull b, ull M) {
	ll ret = a * b - M * ull(1.L / M * a * b);
	return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
ull modpow(ull b, ull e, ull mod) {
	ull ans = 1;
	for (; e; b = modmul(b, b, mod), e /= 2)
		if (e & 1) ans = modmul(ans, b, mod);
	return ans;
}
```

5.7 ModPow

```
#pragma once

const ll mod = 1000000007; // faster if const

ll modpow(ll b, ll e) {
	ll ans = 1;
	for (; e; b = b * b % mod, e /= 2)
		if (e & 1) ans = ans * b % mod;
	return ans;
}
```

5.8 ModSqrt

```
/**
 * Description: Tonelli-Shanks algorithm for modular square roots. Finds
     $x$ s.t. $x^2 = a \pmod p$ ($-x$ gives the other solution).
 * Time: O(\log^2 p) worst case, O(\log p) for most $p$
 */
#pragma once

#include "ModPow.h"

ll sqrt(ll a, ll p) {
	a %= p; if (a < 0) a += p;
	if (a == 0) return 0;
	assert(modpow(a, (p-1)/2, p) == 1); // else no solution
	if (p % 4 == 3) return modpow(a, (p+1)/4, p);
	// a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
	ll s = p - 1, n = 2;
	int r = 0, m;
	while (s % 2 == 0)
		++r, s /= 2;
	/// find a non-square mod p
	while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
	ll x = modpow(a, (s + 1) / 2, p);
	ll b = modpow(a, s, p), g = modpow(n, s, p);
	for (;; r = m) {
		ll t = b;
		for (m = 0; m < r && t != 1; ++m)
			t = t * t % p;
		if (m == 0) return x;
		ll gs = modpow(g, 1LL << (r - m - 1), p);
		g = gs * gs % p;
		x = x * gs % p;
		b = b * g % p;
	}
}
```

5.9 ModSum

```
/**
 * Description: Sums of mod'ed arithmetic progressions.
 *
 * \texttt{modsum(to, c, k, m)} = $\sum_{i=0}^{\mathrm{to}-1}{(ki+c) \%
     m}$.
 * \texttt{divsum} is similar but for floored division.
 * Time: $\log(m)$, with a large constant.
 */
#pragma once

typedef unsigned long long ull;
ull sumsq(ull to) { return to / 2 * ((to-1) | 1); }
/// ^ written in a weird way to deal with overflows correctly

ull divsum(ull to, ull c, ull k, ull m) {
	ull res = k / m * sumsq(to) + c / m * to;
	k %= m; c %= m;
```

```
	if (!k) return res;
	ull to2 = (to * k + c) / m;
	return res + (to - 1) * to2 - divsum(to2, m-1 - c, m, k);
}

ll modsum(ull to, ll c, ll k, ll m) {
	c = ((c % m) + m) % m;
	k = ((k % m) + m) % m;
	return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
}
```

5.10 euclid

```
/**
 * Description: Finds two integers $x$ and $y$, such that
     $ax+by=\gcd(a,b)$. If
 * you just need gcd, use the built in \texttt{\_\_gcd} instead.
 * If $a$ and $b$ are coprime, then $x$ is the inverse of $a \pmod{b}$.
 */
#pragma once

ll euclid(ll a, ll b, ll &x, ll &y) {
	if (!b) return x = 1, y = 0, a;
	ll d = euclid(b, a % b, y, x);
	return y -= a/b * x, d;
}
```

5.11 phiFunction

```
/**
 * Description: \emph{Euler's $\phi$} function is defined as
     $\phi(n):=\#$ of positive integers $\leq n$ that are coprime with
     $n$.
 * $\phi(1)=1$, $p$ prime $\Rightarrow \phi(p^k)=(p-1)p^{k-1}$, $m,n$
     coprime $\Rightarrow \phi(mn)=\phi(m)\phi(n)$.
 * If $n=p_1^{k_1}p_2^{k_2} ... p_r^{k_r}$ then $\phi(n) =
     (p_1-1)p_1^{k_1-1}...(p_r-1)p_r^{k_r-1}$.
 * $\phi(n)=n \cdot \prod_{p|n}(1-1/p)$.
 *
 * $\sum_{d|n} \phi(d) = n$, $\sum_{1\leq k \leq n, \gcd(k,n)=1} k = n
     \phi(n)/2, n>1$
 *
 * \textbf{Euler's thm}: $a,n$ coprime $\Rightarrow a^{\phi(n)} \equiv 1
     \pmod{n}$.
 *
 * \textbf{Fermat's little thm}: $p$ prime $\Rightarrow a^{p-1} \equiv 1
     \pmod{p}$ $\forall a$.
 */
#pragma once

const int LIM = 5000000;
int phi[LIM];

void calculatePhi() {
	rep(i,0,LIM) phi[i] = i&1 ? i : i/2;
	for (int i = 3; i < LIM; i += 2) if(phi[i] == i)
		for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
}
```

6 numerical

6.1 FastFourierTransform

```
/**
 * Description: fft(a) computes $\hat f(k) = \sum_x a[x] \exp(2\pi i
     \cdot k x / N)$ for all $k$. N must be a power of 2.
   Useful for convolution:
   \texttt{conv(a, b) = c}, where $c[x] = \sum a[i]b[x-i]$.
   For convolution of complex numbers or more than two vectors: FFT,
       multiply
   pointwise, divide by n, reverse(start+1, end), FFT back.
   Rounding is safe if $(\sum a_i^2 + \sum b_i^2)\log_2{N} <
       9\cdot10^{14}$
   (in practice $10^{16}$; higher for random inputs).
   Otherwise, use NTT/FFTMod.
 * Time: O(N \log N) with $N = |A|+|B|$ ($\tilde 1s$ for $N=2^{22}$)
 */
#pragma once

typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C>& a) {
	int n = sz(a), L = 31 - __builtin_clz(n);
```

```
    static vector<complex<long double>> R(2, 1);
    static vector<C> rt(2, 1); // (^ 10% faster if double)
    for (static int k = 2; k < n; k *= 2) {
        R.resize(n); rt.resize(n);
        auto x = polar(1.0L, acos(-1.0L) / k);
        rep(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
    }
    vi rev(n);
    rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
    rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
    for (int k = 1; k < n; k *= 2)
        for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
            // C z = rt[j+k] * a[i+j+k]; // (25% faster if
                hand-rolled) /// include-line
            auto x = (double *)&rt[j+k], y = (double
                *)&a[i+j+k];  /// exclude-line
            C z(x[0]*y[0] - x[1]*y[1], x[0]*y[1] + x[1]*y[0]);
                /// exclude-line
            a[i + j + k] = a[i + j] - z;
            a[i + j] += z;
        }
}
vd conv(const vd& a, const vd& b) {
    if (a.empty() || b.empty()) return {};
    vd res(sz(a) + sz(b) - 1);
    int L = 32 - __builtin_clz(sz(res)), n = 1 << L;
    vector<C> in(n), out(n);
    copy(all(a), begin(in));
    rep(i,0,sz(b)) in[i].imag(b[i]);
    fft(in);
    for (C& x : in) x *= x;
    rep(i,0,n) out[i] = in[-i & (n - 1)] - conj(in[i]);
    fft(out);
    rep(i,0,sz(res)) res[i] = imag(out[i]) / (4 * n);
    return res;
}
```

6.2 FastFourierTransformMod

```
/**
 * Description: Higher precision FFT, can be used for convolutions modulo
     arbitrary integers
 * as long as $N\log_2N\cdot \text{mod} < 8.6 \cdot 10^{14}$ (in practice
     $10^{16}$ or higher).
 * Inputs must be in $[0, \text{mod})$.
 * Time: O(N \log N), where $N = |A|+|B|$ (twice as slow as NTT or FFT)
 */
#pragma once

#include "FastFourierTransform.h"

typedef vector<ll> vl;
template<int M> vl convMod(const vl &a, const vl &b) {
    if (a.empty() || b.empty()) return {};
    vl res(sz(a) + sz(b) - 1);
    int B=32-__builtin_clz(sz(res)), n=1<<B, cut=int(sqrt(M));
    vector<C> L(n), R(n), outs(n), outl(n);
    rep(i,0,sz(a)) L[i] = C((int)a[i] / cut, (int)a[i] % cut);
    rep(i,0,sz(b)) R[i] = C((int)b[i] / cut, (int)b[i] % cut);
    fft(L), fft(R);
    rep(i,0,n) {
        int j = -i & (n - 1);
        outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
        outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
    }
    fft(outl), fft(outs);
    rep(i,0,sz(res)) {
        ll av = ll(real(outl[i])+.5), cv = ll(imag(outs[i])+.5);
        ll bv = ll(imag(outl[i])+.5) + ll(real(outs[i])+.5);
        res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
    }
    return res;
}
```

6.3 LinearRecurrence

```
/**
 * Description: Generates the $k$'th term of an $n$-order
 * linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$,
 * given $S[0 \ldots \ge n-1]$ and $tr[0 \ldots n-1]$.
 * Faster than matrix multiplication.
 * Useful together with Berlekamp--Massey.
 * Usage: linearRec({0, 1}, {1, 1}, k) // k'th Fibonacci number
 * Time: O(n^2 \log k)
 */
```

```cpp
#pragma once

const ll mod = 5; /** exclude-line */

typedef vector<ll> Poly;
ll linearRec(Poly S, Poly tr, ll k) {
	int n = sz(tr);

	auto combine = [&](Poly a, Poly b) {
		Poly res(n * 2 + 1);
		rep(i,0,n+1) rep(j,0,n+1)
			res[i + j] = (res[i + j] + a[i] * b[j]) % mod;
		for (int i = 2 * n; i > n; --i) rep(j,0,n)
			res[i - 1 - j] = (res[i - 1 - j] + res[i] * tr[j])
			    % mod;
		res.resize(n + 1);
		return res;
	};

	Poly pol(n + 1), e(pol);
	pol[0] = e[1] = 1;

	for (++k; k; k /= 2) {
		if (k % 2) pol = combine(pol, e);
		e = combine(e, e);
	}

	ll res = 0;
	rep(i,0,n) res = (res + pol[i + 1] * S[i]) % mod;
	return res;
}
```

6.4 MatrixInverse-mod

```cpp
/**
 * Description: Invert matrix $A$ modulo a prime.
 * Returns rank; result is stored in $A$ unless singular (rank < n).
 * For prime powers, repeatedly set $A^{-1} = A^{-1} (2I - AA^{-1})\
     (\text{mod }p^k)$ where $A^{-1}$ starts as
 * the inverse of A mod p, and k is doubled in each step.
 * Time: O(n^3)
 */
#pragma once

#include "../number-theory/ModPow.h"

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

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

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

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

6.5 MatrixInverse

```cpp
/**
 * Description: Invert matrix $A$. Returns rank; result is stored in $A$
     unless singular (rank < n).
 * Can easily be extended to prime moduli; for prime powers, repeatedly
 * set $A^{-1} = A^{-1} (2I - AA^{-1})\ (\text{mod }p^k)$ where $A^{-1}$
     starts as
 * the inverse of A mod p, and k is doubled in each step.
 * Time: O(n^3)
 */
#pragma once

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

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

	/// forget A at this point, just eliminate tmp backward
	for (int i = n-1; i > 0; --i) rep(j,0,i) {
		double v = A[j][i];
		rep(k,0,n) tmp[j][k] -= v*tmp[i][k];
	}

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

6.6 NumberTheoreticTransform

```cpp
/**
 * Description: ntt(a) computes $\hat f(k) = \sum_x a[x] g^{xk}$ for all
     $k$, where $g=\text{root}^{(mod-1)/N}$.
 * N must be a power of 2.
 * Useful for convolution modulo specific nice primes of the form $2^a
     b+1$,
 * where the convolution result has size at most $2^a$. For arbitrary
     modulo, see FFTMod.
   \texttt{conv(a, b) = c}, where $c[x] = \sum a[i]b[x-i]$.
   For manual convolution: NTT the inputs, multiply
   pointwise, divide by n, reverse(start+1, end), NTT back.
 * Inputs must be in [0, mod).
 * Time: O(N \log N)
 */
#pragma once

#include "../number-theory/ModPow.h"

const ll mod = (119 << 23) + 1, root = 62; // = 998244353
// For p < 2^30 there is also e.g. 5 << 25, 7 << 26, 479 << 21
// and 483 << 21 (same root). The last two are > 10^9.
typedef vector<ll> vl;
void ntt(vl &a) {
	int n = sz(a), L = 31 - __builtin_clz(n);
	static vl rt(2, 1);
	for (static int k = 2, s = 2; k < n; k *= 2, s++) {
		rt.resize(n);
		ll z[] = {1, modpow(root, mod >> s)};
		rep(i,k,2*k) rt[i] = rt[i / 2] * z[i & 1] % mod;
	}
	vi rev(n);
	rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
	rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
	for (int k = 1; k < n; k *= 2)
		for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
```

```
                ll z = rt[j + k] * a[i + j + k] % mod, &ai = a[i +
                    j];
                a[i + j + k] = ai - z + (z > ai ? mod : 0);
                ai += (ai + z >= mod ? z - mod : z);
            }
}
vl conv(const vl &a, const vl &b) {
    if (a.empty() || b.empty()) return {};
    int s = sz(a) + sz(b) - 1, B = 32 - __builtin_clz(s), n = 1 << B;
    int inv = modpow(n, mod - 2);
    vl L(a), R(b), out(n);
    L.resize(n), R.resize(n);
    ntt(L), ntt(R);
    rep(i,0,n) out[-i & (n - 1)] = (ll)L[i] * R[i] % mod * inv % mod;
    ntt(out);
    return {out.begin(), out.begin() + s};
}
```

7 strings

7.1 hashing$_rabin_karp$

```
// Note: Might have overflow bugs, need to stress test. Use with #define
    int ll just in case.
// Note2: Useful but also memory heavy as it maintains rabinkarp array
    for all strings.

class hstring{
private:
    int n;
    string s;
    vector<pii> h;
    const static int p1 = 137;
    const static int p2 = 277;
    const static int m1 = 127657753;
    const static int m2 = 987654319;
    static vector<pii> pow_p;
    static vector<pii> ipow_p;

    void prec(){
        pow_p[0] = ipow_p[0] = {1, 1};
        int ip1 = 11181701;
        int ip2 = 802246288;
        for(int i=1; i<sz(pow_p); i++) {
            pow_p[i].ff = (1ll*pow_p[i-1].ff*p1)%m1;
            ipow_p[i].ff = (1ll*ipow_p[i-1].ff*ip1)%m1;
            pow_p[i].ss = (1ll*pow_p[i-1].ss*p2)%m2;
            ipow_p[i].ss = (1ll*ipow_p[i-1].ss*ip2)%m2;
        }
    }

    void _hash(){
        if(pow_p[0].ff == 0) prec();

        n = sz(s);
        h.resize(n+1);
        for(int i=0; i<sz(s); i++){
            h[i+1].ff = (h[i].ff + (1ll*s[i]*pow_p[i].ff)%m1)%m1;
            h[i+1].ss = (h[i].ss + (1ll*s[i]*pow_p[i].ss)%m2)%m2;
        }
    }

public:
    hstring() = default;
    hstring(string &t) : s(t){ _hash(); }
    hstring(string t) : s(t){ _hash(); }

    pii hash() { return h[n]; }
    pii hash(int l, int r) {
        int fi = ((h[r+1].ff - h[l].ff + m1) * 1ll * ipow_p[l].ff)%m1;
        int se = ((h[r+1].ss - h[l].ss + m2) * 1ll * ipow_p[l].ss)%m2;
        return {fi, se};
    }

    size_t size() { return sz(s); }
    string& str() { return s; }

    bool operator ==(hstring &t) { return t.hash() == hash(); }
    friend istream &operator>>(istream &in, hstring &hs) {
        in>>hs.s; hs._hash();
        return in;
    }
};

vector<pii> hstring::pow_p(1e7);
vector<pii> hstring::ipow_p(1e7);
```