1 Math

1.1 Gauss-Jordan Elimination

```
#include <bits/stdc++.h>

using namespace std;

typedef struct _Matrix{
    int N;
    vector<vector<double>> matrix;

    _Matrix(int X) {
        N = X;
        matrix.resize(N, vector<double>(N + 1));
    } // N by N + 1 matrix
} Matrix;

void row_swap(Matrix& A, int i) {
    vector<double> temp = A.matrix[i];
    A.matrix.erase(A.matrix.begin() + i);
    A.matrix.push_back(temp);
}

//Gauss-Jordan Elmination
void gauss_jordan(Matrix& A) {
    for(int i = 0; i < A.N; ++i) {
        while(A.matrix[i][i] == 0) row_swap(A, i); //check diagonal components
          are non-zero, when if, rotate row(swap)

        for(int j = 0; j < A.N; ++j) { //make RREF
            if(i != j) {
                double ratio = A.matrix[j][i] / A.matrix[i][i];
                for(int k = 0; k <= A.N; ++k) {
                    A.matrix[j][k] = A.matrix[j][k] - ratio * A.matrix[i][k];
                }
            }
        }
    }
}
```

1.2 Fast-Fourier-Transform

```
#include <bits/stdc++.h>

using namespace std;
using cpx = complex<double>;
//Cooley-Tukey FFT
void FFT(vector<cpx>& A, cpx w) {
    int n = (int)A.size();
    if(n == 1) return;

    vector<cpx> even(n / 2), odd(n / 2);
    for(int i = 0; i < n; ++i) {
        if(i & 1) odd[i / 2] = A[i];
        else even[i / 2] = A[i];
    }

    FFT(even, w * w);
    FFT(odd, w * w);

    cpx w_e(1, 0);

    for(int i = 0; i < n / 2; ++i) {
        A[i] = even[i] + w_e * odd[i];
        A[i + n / 2] = even[i] - w_e * odd[i];
        w_e *= w;
    }
}

void product(vector<cpx>& A, vector<cpx>& B) {
    int n = (A.size() <= B.size()) ? ceil(log2((double)B.size())) : ceil(log2((
      double)A.size()));
    n = pow(2, n + 1);

    A.resize(n);
    B.resize(n);
    vector<cpx> C(n);

    cpx w(cos(2 * acos(-1) / n), sin(2 * acos(-1) / n));
    FFT(A, w);
    FFT(B, w);

    for(int i = 0; i < n; ++i) C[i] = A[i] * B[i];

    FFT(C, cpx(1, 0) / w);
    for(int i = 0; i < n; ++i) {
        C[i] /= cpx(n, 0);
        C[i] = cpx(round(C[i].real()), round(C[i].imag()));
    }
}

/*
void FFT(vector<cpx>& A, bool invert) {
    int n = (int)A.size();

    for(int i = 1, j = 0; i < n; ++i) {
        int bit = n >> 1;

        while(j >= bit) {
            j -= bit;
            bit >>= 1;
        }
        j += bit;

        if(i < j) swap(A[i], A[j]);
    }

    for(int length = 2; length <= n; length <<= 1) {
        double ang = 2 * PI / length * (invert ? -1 : 1);
```

```
        cpx w(cos(ang), sin(ang));

        for(int i = 0; i < n; i += length) {
            cpx w_i(1, 0);
            for(int j = 0; j < length / 2; ++j) {
                cpx u = A[i + j], v = A[i + j + length / 2] * w_i;
                A[i + j] = u + v, A[i + j + length / 2] = u - v;
                w_i *= w;
            }
        }
    }

    if(invert) {
        for(int i = 0; i < n; ++i) {
            A[i] /= cpx(n, 0);
            A[i] = cpx(round(A[i].real()), round(A[i].imag()));
        }
    }
} // referenced from https://blog.myungwoo.kr/54
*/ //faster version of FFT
```

2 Segment Tree

2.1 Dynamic Segment Tree

```
#include <bits/stdc++.h>

using namespace std;
using ll = long long;
using pii = pair<int, int>;
using ppii = pair<int, pii>;

struct node {
    node *l, *r;
    ll s;

    node() {
        l = r = NULL;
        s = 0;
    }
};

class dynamic_segment_tree {
    private:
        node* root;

        void delete_nodes(node* v) {
            if(!v) return;
            delete_nodes(v->l);
            delete_nodes(v->r);
            delete v;
        }

    public:
        dynamic_segment_tree() {
            root = new node();
        }

        node* get_root() {
            return root;
        }

        void update(int start, int end, node* cur_node, int idx, ll val) {
            if(idx < start || idx > end) return;

            if(start == end) {
                cur_node->s = val;
                return;
            }

            int mid = (start + end) / 2;
            if(idx <= mid) {
                if(!cur_node->l) cur_node->l = new node();
                update(start, mid, cur_node->l, idx, val);
            }
            else {
                if(!cur_node->r) cur_node->r = new node();
                update(mid + 1, end, cur_node->r, idx, val);
            }

            ll l_val = (cur_node->l ? cur_node->l->s : 0);
            ll r_val = (cur_node->r ? cur_node->r->s : 0);
            cur_node->s = l_val + r_val;
        }

        ll query(int start, int end, node* cur_node, int left, int right) {
            if(!cur_node) return 0;

            if(right < start || left > end) return 0;

            if(left <= start && end <= right) return cur_node->s;

            int mid = (start + end) / 2;

            return query(start, mid, cur_node->l, left, right) + query(mid + 1,
              end, cur_node->r, left, right);
        }

        ~dynamic_segment_tree() {
            delete_nodes(root);
        }
};
```

3 Graph

3.1 Heavy-Light Decomposition

```
#include <bits/stdc++.h>
```

```cpp
#define MAX 100'000
#define INF 987654321

using namespace std;
using ll = long long;
using pii = pair<int, int>;
using ppii = pair<int, pii>;

class segment_tree { /* segment tree implementation */ };

class heavy_light_decomposition {
    private:
        int group_cnt;
        int tree_size[MAX], depth[MAX], parent[MAX], top_chain[MAX], in[MAX],
          out[MAX];
        bool visit[MAX];
        vector<int> child[MAX];
        vector<int> adj[MAX];
        segment_tree tree;

    public:
        void init() {
            group_cnt = 0;

            int N; cin >> N;
            tree.resize(MAX);

            for(int i = 0; i < N - 1; ++i) {
                int u, v; cin >> u >> v;
                adj[u].push_back(v);
                adj[v].push_back(u);
            }

            dfs_child_set();
            dfs_size();
            dfs_grouping();
        }

        void dfs_child_set(int v = 1) {
            visit[v] = true;
            for(auto& next : adj[v]) {
                if(visit[next]) continue;
                visit[next] = true;
                child[v].push_back(next);
                dfs_child_set(next);
            }
        }

        void dfs_size(int v = 1) {
            tree_size[v] = 1;
            for(auto& next : child[v]) {
                depth[next] = depth[v] + 1;
                parent[next] = v;

                dfs_size(next);
                tree_size[v] += tree_size[next];
                if(tree_size[next] > tree_size[child[v][0]]) swap(child[v][0],
                  next);
            }
        }

        void dfs_grouping(int v = 1) {
            in[v] = ++group_cnt;
            for(auto& next : child[v]) {
                top_chain[next] = (next == child[v][0] ? top_chain[v] : next);
                dfs_grouping(next);
            }
            out[v] = group_cnt;
        }

        void update(int v, int w) {
            tree.update(1, MAX, 1, in[v], w);
        }

        int query(int a, int b) {
            int ret = 0;
            while(top_chain[a] != top_chain[b]) {
                if(depth[top_chain[a]] < depth[top_chain[b]]) swap(a, b);
                int v = top_chain[a];
                ret += tree.query(1, MAX, 1, in[v], in[a]);
                a = parent[v];
            }

            if(depth[a] > depth[b]) swap(a, b);
            ret += tree.query(1, MAX, 1, in[a], in[b]);

            return ret;
        }
};
```

4 Geometry

4.1 Convex Hull

```cpp
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;

typedef struct _Point {
    int x;
    int y;
} Point;

//Standard Point to Sort
Point S;

Point get_vector(const Point& A, const Point& B) {
    Point v = {B.x - A.x, B.y - A.y};
```

```
    return v;
}

//ccw test
int ccw(const Point& v, const Point& u) {
    ll val = (ll)v.x * u.y - (ll)v.y * u.x;
    if(val > 0) return 1;
    else if(val < 0) return -1;
    else return 0;
}

int ccw(const Point& A, const Point& B, const Point& C) {
    Point v = get_vector(A, B);
    Point u = get_vector(B, C);
    return ccw(v, u);
}

//to sort by ccw
bool comp(const Point& A, const Point& B) {
    Point v = get_vector(S, A);
    Point u = get_vector(S, B);

    if(ccw(v, u) > 0) return true;
    else if(ccw(v, u) < 0) return false;

    return (v.x == u.x) ? (v.y < u.y) : (v.x < u.x);
}

bool operator<(const Point& A, const Point& B) {
    return (A.x == B.x) ? (A.y < B.y) : (A.x < B.x);
}

//Graham's Scan Method
vector<Point> get_convex_hull(vector<Point>& A) {
    S = *min_element(A.begin(), A.end());
    sort(A.begin(), A.end(), comp);
    int n = (int)A.size();

    vector<Point> convex_hull;

    //get Convex Hull
    for(int i = 0; i < n; ++i) {
        while((int)convex_hull.size() > 1
        && ccw(convex_hull[(int)convex_hull.size() - 2], convex_hull.back(), A[i
          ]) <= 0) {
            convex_hull.pop_back();
        }
        convex_hull.push_back(A[i]);
    }

    return convex_hull;
}
```

4.2 Point in Convex Hull Test

```
#include <bits/stdc++.h>

using namespace std;

typedef struct _Point {
    int x;
    int y;
} Point;

Point get_vector(const Point& A, const Point& B);
int ccw(const Point& v, const Point& u);
int ccw(const Point& A, const Point& B, const Point& C);

//convext_hull size >= 3
bool isInside(vector<Point>& convext_hull, Point& A) {
    int O = 0;
    int L = 1, R = (int)convext_hull.size() - 1;
    int M = (L + R) / 2;

    Point vecOL = get_vector(convext_hull[O], convext_hull[L]);
    Point vecOA = get_vector(convext_hull[O], A);
    Point vecOR = get_vector(convext_hull[O], convext_hull[R]);
    Point vecOM = get_vector(convext_hull[O], convext_hull[M]);

    if(ccw(vecOL, vecOA) < 0) return false;
    if(ccw(vecOR, vecOA) > 0) return false;

    while(L + 1 != R) {
        M = (L + R) / 2;
        vecOM = get_vector(convext_hull[O], convext_hull[M]);

        if(ccw(vecOM, vecOA) > 0) L = M;
        else R = M;
    }

    if(ccw(convext_hull[L], A, convext_hull[R]) <= 0) return true;
    else return false;
}
```