Li Chao Tree

$$\mathrm{dp}(n) = \max_{i<n} \{m(i) \cdot a(n) + c(i)\} + cc(n)$$

```
//dot product of (m, c).(x, 1)
ll f(pi a, pi x){return a.F*x.F + a.S*x.S;}

struct node{
    ll s, e, m;
    pi curw;
    bool assigned;
    node *l, *r;
    node (ll _s, ll _e): s(_s), e(_e), l(NULL), assigned(false), r(NULL), curw(make_pair(-1, -1)){
        m = (s + e)/2;
        if (m - s > e - m) m--;
    }

    // Add line of y = m(i)*x + c(i); nw = MP(m(i), c(i))
    void add_line(pi nw){
        if (!assigned){
            assigned = true;
            curw = nw;
            return;
        }

        bool lef = f(nw, MP(s, 1)) > f(curw, MP(s, 1)); // FLIP SIGNS FOR MIN
        bool mid = f(nw, MP(m + 1, 1)) > f(curw, MP(m + 1, 1)); // FLIP SIGNS FOR MIN
        if (m > e || s > m) return;
        if (mid) swap(curw, nw);
        if (e - s == 0) return;
        if (lef != mid){
            if (l == NULL) l = new node(s, m);
            l -> add_line(nw);
        } else {
            if (r == NULL) r = new node(m + 1, e);
            r -> add_line(nw);
        }
    }

    ll solve(ll x){
        if (l == NULL && r == NULL) return f(curw, MP(x, 1));
        if (x <= m){
            if (l == NULL) return f(curw, MP(x, 1));
            return max(f(curw, MP(x, (ll)1)), l -> solve(x)); // CHANGE MAX TO MIN
        } else {
            if (r == NULL) return f(curw, MP(x, 1));
            return max(f(curw, MP(x, (ll)1)), r -> solve(x)); // CHANGE MAX TO MIN
        }
    }
} *root;

int main(){
    //create with ALL possible query points
    root = new node(-10000000000001, 10000000000001);
    root -> add_line(MP(0, 0)); // CHANGE BASE CASE IF NECESSARY

    REP(i, 1, n + 1){
        dp[i] = cc(i) + root -> solve(a(i)); // get answer for dp(i) = cc(i) + max_{j < i}{m(j)*a(i) + c(j)}
        root -> add_line(MP(m(i), c(i))); // add line for this iteration y = m(i)*x + c(i)
    }
}
```

Divide and Conquer

$$\mathrm{dp}(n,k) = \begin{cases} \min\limits_{0\le i\le k} dp(n-1, i-1) + C(i,k), & k \ge 0 \\ 0, & k < 0 \end{cases}; \quad \mathrm{opt}(n,k) \le \mathrm{opt}(n,k+1)$$

```
// on the fly if necessary
ll memo[MAX_K][MAX_N];
int opt[MAX_N];

//l and r inclusive
//optl and optr inclusive
void dnc(int l, int r, int optl, int optr, int k){
    if (l > r) return;
    int mid = (l + r)/2;

    //do NOT set INF as base, always set to a possible value
    pi cur;
    if (mid == 0) cur = MP(0, 0);
    else cur = MP(memo[k - 1][mid - 1], mid - 1);

    REP(i, optl, min(mid - 1, optr) + 1){
        // cur = min(cur, MP(cost function, opt index))
        cur = min(cur, MP(memo[k - 1][i] + (rs[mid][mid] - rs[i][mid] - rs[mid][i] + rs[i][i])/2, (ll)i));
    }

    opt[mid] = cur.S;
```

```
    memo[k][mid] = cur.F;
    dnc(l, mid - 1, optl, opt[mid], k);
    dnc(mid + 1, r, opt[mid], optr, k);
}

int main(){
    //prepare cost function
    REP(i, 0, k + 1){
        if (i == 0){
            memo[0][0] = 0;
            REP(j, 1, n + 1) memo[0][j] = INF;
            continue;
        }
        dnc(0, n, 0, n, i);
    }
    cout << memo[k][n] << endl;
}
```

All-in-One Segment Tree

```
struct node {
    int s, e;
    ll mn, mx, sum;
    bool lset;
    ll add_val, set_val;
    node *l, *r;
    node (int _s, int _e, int A[] = NULL): s(_s), e(_e), mn(0), mx(0), sum(0), lset(0), add_val(0), set_val(0),
l(NULL), r(NULL) {
        if (A == NULL) return;
        if (s == e) mn = mx = sum = A[s];
        else {
            l = new node(s, (s+e)>>1, A), r = new node((s+e+2)>>1, e, A);
            combine();
        }
    }
    void create_children() {
        if (s == e) return;
        if (l != NULL) return;
        int m = (s+e)>>1;
        l = new node(s, m);
        r = new node(m+1, e);
    }
    void self_set(ll v) {
        lset = 1;
        mn = mx = set_val = v;
        sum = v * (e-s+1);
        add_val = 0;
    }
    void self_add(ll v) {
        if (lset) { self_set(v + set_val); return; }
        mn += v, mx += v, add_val += v;
        sum += v*(e-s+1);
    }
    void lazy_propagate() {
        if (s == e) return;
        if (lset) {
            l->self_set(set_val), r->self_set(set_val);
            lset = set_val = 0;
        }
        if (add_val != 0) {
            l->self_add(add_val), r->self_add(add_val);
            add_val = 0;
        }
    }
    void combine() {
        if (l == NULL) return;
        sum = l->sum + r->sum;
        mn = min(l->mn, r->mn);
        mx = max(l->mx, r->mx);
    }
    void add(int x, int y, ll v) {
        if (s == x && e == y) { self_add(v); return; }
        int m = (s+e)>>1;
        create_children(); lazy_propagate();
        if (x <= m) l->add(x, min(y, m), v);
        if (y > m) r->add(max(x, m+1), y, v);
        combine();
    }
    void set(int x, int y, ll v) {
        if (s == x && e == y) { self_set(v); return; }
        int m = (s+e)>>1;
        create_children(); lazy_propagate();
        if (x <= m) l->set(x, min(y, m), v);
        if (y > m) r->set(max(x, m+1), y, v);
        combine();
    }
    ll range_sum(int x, int y) {
        if (s == x && e == y) return sum;
        if (l == NULL || lset) return (sum/(e-s+1))*(y-x+1);
        int m = (s+e)>>1;
```

```
        lazy_propagate();
        if (y <= m) return l->range_sum(x, y);
        if (x > m) return r->range_sum(x, y);
        return (l->range_sum(x, m) + r->range_sum(m+1, y));
    }
    ll range_min(int x, int y) {
        if (s == x && e == y) return mn;
        if (l == NULL || lset) return mn;
        int m = (s+e)>>1;
        lazy_propagate();
        if (y <= m) return l->range_min(x, y);
        if (x > m) return r->range_min(x, y);
        return min(l->range_min(x, m), r->range_min(m+1, y));
    }
    ll range_max(int x, int y) {
        if (s == x && e == y) return mx;
        if (l == NULL || lset) return mx;
        int m = (s+e)>>1;
        lazy_propagate();
        if (y <= m) return l->range_max(x, y);
        if (x > m) return r->range_max(x, y);
        return max(l->range_max(x, m), r->range_max(m+1, y));
    }
    ~node() {
        if (l != NULL) delete l;
        if (r != NULL) delete r;
    }
} *root;

int main(){
    node* root = new node(0, n - 1); // inclusive
    cout << root -> range_sum(a - 1, b - 1) << endl; // don't forget MOD if necessary and in the tree
    root -> add(a - 1, b - 1, x);
}
```

2k Decomposition

```
int twok[MAXN][LOGN];
// twok[node][k] stores the (2^k)th parent of node i.e. twok[node][0] stores the direct parent of node
int d[MAXN];

vector<int> adjList[MAXN];

//Assign a parent-node relationship
void assignParent(int node, int parent){
    twok[node][0] = parent;
    REP(i, 1, LOGN){
        if (twok[node][i-1] == -1) break;
        else twok[node][i] = twok[twok[node][i-1]][i-1];
    }
}

//Returns the counterth parent of node
int findParent(int node, int c){
    if (c == 0) return node;
    for (int i = LOGN - 1; i >= 0; i--){
        if (twok[node][i] == -1) continue;
        if ((1 << i) > c) continue;
        return findParent(twok[node][i], c - (1 << i));
    }
}

//Assigns height of each node and calculates the 2k decomp
void dfs(int x){
    VREP(it, adjList[x]){
        if (d[*it] != -1) continue;
        d[*it] = 1 + d[x];
        assignParent(*it, x);
        dfs(*it);
    }
}

int lca(int x, int y) {
    if (d[x] < d[y]) swap(x,y);
    if (d[x] > d[y]) {
        int diff = d[x] - d[y];
        REP(i, 0, LOGN) if (diff & (1 << i)) x = twok[x][i];
    }
    if (x == y) return x;
    for (int i = LOGN - 1; i >= 0; i--) {
        if (twok[x][i] != twok[y][i]) {
            x = twok[x][i]; y = twok[y][i];
        }
    }
    return twok[x][0];
}

int main(){
    memset(twok, -1, sizeof(twok));
```

```
    memset(d, -1, sizeof(d));
    int root = 0;
    d[root] = 0;
    dfs(root);
    cout << lca(a, b) << endl;
}
```

Augmenting Path Algorithm

```
struct AugPath {
    int A, B;   // size of left, right groups
    vector<vector<int> > G; // size A
    vector<bool> visited;   // size A
    vector<int> P;          // size B

    AugPath(int _A, int _B): A(_A), B(_B), G(_A), P(_B, -1){}

    void addEdge(int a, int b){
        // just need edges from A to B; a from left, b from right
        G[a].PB(b);
    }

    bool aug(ll x) {
        if (visited[x]) return false;
        visited[x] = true;
        VREP(it, G[x]){
            if (P[*it] == -1 || aug(P[*it])){
                P[*it] = x;
                return true;
            }
        }
        return false;
    }

    int mcbm(){
        int matchings = 0;
        REP(i, 0, A){
            visited.resize(A, 0);
            matchings += aug(i);
            visited.clear();
        }
        return matchings;
    }

    vector<pi> getMatchings() {
        vector<pi> matchings;
        REP(i, 0, B) if (P[i] != -1) matchings.emplace_back(P[i], i);
        return matchings;
    }
};


int main(){
    AugPath graph = AugPath(n, n);
    REP(i, 0, m) graph.addEdge(u, v)
    cout << graph.mcbm() << endl;
}
```

Dinic's Algorithm

```
struct flowEdge{
    int u, v; // vertices of this edge
    ll cap, flow = 0; // capacity and flow
    flowEdge(int u, int v, ll cap): u(u), v(v), cap(cap) {}
};

struct dinic{
    const ll flow_inf = 1e18;
    vector<flowEdge> edgeList;
    vector<vector<int> > adj;
    int n, m = 0, s, t;
    vector<int> level, ptr;
    queue<int> q;

    dinic(int n, int s, int t): n(n), s(s), t(t){
        // n: total nodes, s: source node, t: sink node
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int u, int v, ll cap){
        edgeList.PB(flowEdge(u, v, cap));
        edgeList.PB(flowEdge(v, u, 0)); //residue edge
        adj[u].PB(m);
        adj[v].PB(m + 1);
        m+=2;
    }
```

```
    bool bfs(){
        while (!q.empty()){
            int x = q.front();
            q.pop();
            for (int id: adj[x]){
                if (edgeList[id].cap - edgeList[id].flow < 1) continue;
                if (level[edgeList[id].v] != -1) continue;
                level[edgeList[id].v] = level[x] + 1;
                q.push(edgeList[id].v);
            }
        }
        return (level[t] != -1);
    }

    ll dfs(int x, ll pushed){
        if (pushed == 0) return 0;
        if (x == t) return pushed;
        for (int& cid = ptr[x]; cid < (int)adj[x].size(); cid++){
            int id = adj[x][cid], v = edgeList[id].v;
            if (level[x] + 1 != level[v] || edgeList[id].cap - edgeList[id].flow < 1) continue;
            ll tr = dfs(v, min(pushed, edgeList[id].cap - edgeList[id].flow));
            if (tr == 0) continue;
            edgeList[id].flow += tr;
            edgeList[id^1].flow -= tr;
            return tr;
        }
        return 0;
    }

    ll flow(){
        ll f = 0;
        while (true){
            fill (level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs()) break;
            fill(ptr.begin(), ptr.end(), 0);
            while (ll pushed = dfs(s, flow_inf)) f += pushed;
        }
        return f;
    }

    vector<vector<int> > residual_graph;
    vector<bool> mincut_visited;

    void mincut_dfs(int node){
        mincut_visited[node] = true;
        for (auto it = residual_graph[node].begin(); it != residual_graph[node].end(); it++){
            if (mincut_visited[*it]) continue;
            mincut_dfs(*it);
        }
    }

    vector<ll> mincut(){ // returns set S of mincut (S, T)
        // run maxflow first
        residual_graph.resize(n);
        mincut_visited.resize(n, false);

        for (auto it = edgeList.begin(); it != edgeList.end(); it++){
            if (it -> cap > 0 && it -> flow == it -> cap) continue;
            if (it -> cap == 0 && it -> flow == 0) continue;
            residual_graph[it -> u].PB(it -> v);
        }

        mincut_dfs(s);
        vector<ll> res;
        for (int i = 0; i < n; i++){
            if (mincut_visited[i]) res.PB(i);
        }
        return res;
    }
};

int main(){
    dinic graph = dinic(n + 2, 0, n + 1); // usually, create n + 2 nodes with one sink and one source
    graph.add_edge(0, i, b[i]); // add edge between source and node i with capacity b[i]
    graph.add_edge(i, n + 1, c[i]); // add edge between node i and sink with capacity c[i]

    cout << graph.flow() << endl; // max flow

    VREP(it, graph.edgeList){
        if (it -> flow == it -> cap && it -> flow > 0){
            // edge is in use and maximum capacity
        } else if (it -> flow > 0){
            // edge is in use
        } else {
            // not in use edges or residue edges
        }
    }
```

```
    vector<ll> res = graph.mincut();
    cout << res.size() << endl;
    VREP(it, res){
        cout << *it << endl;
    }
}
```

Tarjan's Algorithm

```
struct tarjan{
    int n, m = 0, vcount = 0; // number of nodes, number of sccs
    vector<vector<int> > adjList, scc;
    vector<set<int> > sccDag; // DAG of SCCs
    vector<int> sccIndex, topo;
    vector<bool> visited;
    vector<pi> vv;
    stack<int> st;

    tarjan(int n): n(n){
        adjList.resize(n);
        visited.resize(n);
        vv.resize(n);
        sccIndex.resize(n);
        sccDag.resize(n);
        scc.resize(n);
        REP(i, 0, n){
            visited[i] = false;
            vv[i] = MP(-1, -1);
            sccIndex[i] = -1;
        }
        m = 0; vcount = 0;
    }

    void addEdge(int a, int b){
        adjList[a].PB(b);
    }

    void dfs(int x){
        if (vv[x].first != -1) return;
        st.push(x);
        visited[x] = true;
        vv[x] = MP(vcount, vcount);
        vcount++;
        VREP(it, adjList[x]){
            if (vv[*it].first == -1) dfs(*it);
            if (visited[*it]) vv[x].second = min(vv[*it].second, vv[x].second);
        }

        if (vv[x].first == vv[x].second){
            while (true){
                int cur = st.top();
                st.pop();
                sccIndex[cur] = m;
                scc[m].PB(cur);
                visited[cur] = false;
                if (x == cur) break;
            }
            m++;
        }
    }

    void dfs_topo(int x){
        if (visited[x]) return;
        visited[x] = 1;
        VREP(it, sccDag[x]){
            if (visited[*it]) continue;
            dfs_topo(*it);
        }
        topo.PB(x);
    }

    void compute(){
        REP(i, 0, n) dfs(i);
        REP(i, 0, n){
            VREP(it, adjList[i]){
                if (sccIndex[i] != sccIndex[*it]) sccDag[sccIndex[i]].insert(sccIndex[*it]);
            }
        }
        REP(i, 0, m) visited[i] = false;
        REP(i, 0, m) dfs_topo(i);
        reverse(topo.begin(), topo.end());
    }
};

int main(){
    tarjan x = tarjan(n);
    x.addEdge(0, 1); // add some edges
    x.compute();
```

```
    cout << x.m << endl;
    VREP(it, x.topo){
        cout << sz(x.scc[*it]);
        REP(j, 0, sz(x.scc[*it])) cout << " " << x.scc[*it][j];
        cout << endl;
    }
}
```

2SAT Algorithm

```
struct TwoSat{
    int n; //number of real and conjugate nodes
    vector<vector<int> > g, gt; //graph of implication and transposed
    vector<bool> used;
    vector<int> order, comp;
    vector<bool> assignment; //assignment of true/false for real nodes

    //all even nodes are real
    //all odd nodes are conjugate
    //conjugate of 2*x is 2*x + 1 (or equivalently (2x)^1)

    TwoSat(int nn){
        //nn - number of nodes (not including conjugate)
        n = 2*nn; //real node and conjugate node
        g.resize(n);
        gt.resize(n);
    }

    void addOrEdge(int x, int y){
        //(x or y), equivalently if not x then y, if not y then x
        int notx = (x^1), noty = (y^1);
        g[noty].PB(x); g[notx].PB(y);
        gt[x].PB(noty); gt[y].PB(notx);
    }

    void addImplicationEdge(int x, int y){
        //if x then y
        int notx = (x^1), noty = (y^1);
        g[x].PB(y); gt[y].PB(x);
        g[noty].PB(notx); gt[notx].PB(noty);
    }

    void dfs1(int v){
        used[v] = true;
        VREP(it, g[v]){
            if (!used[*it]) dfs1(*it);
        }
        order.PB(v);
    }

    void dfs2(int v, int cl){
        comp[v] = cl;
        VREP(it, gt[v]){
            if (comp[*it] == -1) dfs2(*it, cl);
        }
    }

    bool solve(){
        used.assign(n, false);
        comp.assign(n, -1);
        REP(i, 0, n){
            if (!used[i]) dfs1(i);
        }
        int j = 0;
        REP(i, 0, n){
            int v = order[n - 1 - i];
            if (comp[v] == -1) dfs2(v, j++);
        }
        assignment.assign(n/2, false);
        for (int i = 0; i < n; i+=2){
            if (comp[i] == comp[i + 1]) return false;
            assignment[i/2] = (comp[i] > comp[i + 1]);
        }
        return true;
    }
};

int main(){
    TwoSat h = TwoSat(n);

    h.addOrEdge(x, y); //(x OR y)
    h.addImplicationEdge(x, y) //x implies y

    bool ans = h.solve();
    if (ans) cout << "Yes" << endl;
    else cout << "No" << endl;

    REP(i, 0, n) cout << h.assignment[i] << endl;
```

```
}
```

Template

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define PB push_back
#define MP make_pair
#define REP(i, a, b) for (int i = (int)a; i < (int)b; i++)
typedef pair<ll,ll> pi;
const ll MOD = 1000000009;
const ll INF = 2e18;
const long double EPS = 1e-9;
const double PI = acos(-1);
int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0);
}
//cout.flush()
//cout << fixed << setprecision(9) << x << endl;
```

Dijkstra

```
const int N = 200005;
vector<pi> adjList[N];
ll dist[N];
priority_queue<pi, vector<pi>, greater<pi> > pq;

int main(){
    memset(dist, -1, sizeof(dist));
    dist[S] = 0;
    pq.push(MP(0, S));
    while (!pq.empty()) {
        pi cur = pq.top();
        pq.pop();
        ll x = cur.second, d = cur.first;
        if (d > dist[x]) continue;
        VREP(it, adjList[x]){
            ll nx = it->first, nd = d+it->second;
            if (dist[nx] != -1 && dist[nx] <= nd) continue;
            dist[nx] = nd;
            pq.push(MP(nd, nx));
        }
    }
}
```

Centroid Decomposition

```
const int MAX_N = 100005;

set<int> adjList[MAX_N]; //original adjList
set<int> temp[MAX_N]; //temporary adjList to erase edges

int crank[MAX_N]; //rank of centroids, crank = 0 is the root
vector<int> cc[MAX_N]; //centroid tree O(N)
vector<int> cr[MAX_N]; //centroid rank vector, cr[0] stores root, cr[1] stores centroid dist 1 to root
map<int, ll> ma[MAX_N]; //map for centroid decomp O(NlogN)

int p[MAX_N]; //stores the centroid parent
int sts[MAX_N]; //stores subtree size

int dfs(int x, int par){
    //dfs to obtain subtree size
    int ans = 1;
    VREP(it, temp[x]){
        if (*it == par) continue;
        ans += dfs(*it, x);
    }
    return sts[x] = ans;
}

int centroid(int x){
    int t = dfs(x, -1), cur = x, par = -1;
    while (true){
        pair<int,int> m = MP(-1, -1); //<max subtree size, node to move to>
        bool isCentroid = true;
        VREP(it, temp[cur]){
            if (*it == par) continue;
            if (2*sts[*it] > t) isCentroid = false;
            m = max(m, MP(sts[*it], *it));
        }
        if (isCentroid) return cur;
        par = cur; cur = m.second;
    }
}

void centroid_decomp(int x, int pc){
    int c = centroid(x);
```

```
    p[c] = pc;

    if (pc != -1){
        cc[pc].PB(c); cc[c].PB(pc);
        crank[c] = crank[pc] + 1;
    } else crank[c] = 0;

    cr[crank[c]].PB(c);

    VREP(it, temp[c]){
        temp[*it].erase(c);
        centroid_decomp(*it, c);
    }
    temp[c].clear();
}

void dfs2(int x, int par, int v, int cn){
    //x, p[x] or -1, value (in this case the distance), centroid --- dfs to obtain half path value
    VREP(it, adjList[x]){
        if (crank[*it] <= crank[cn] || (*it)==par) continue;
        ma[cn][*it] = v; //store half path value here
        dfs2(*it, x, v + 1, cn);
    }
}

int main(){
    REP(i, 0, n - 1){
        adjList[aa - 1].insert(bb - 1); adjList[bb - 1].insert(aa - 1);
        temp[aa - 1].insert(bb - 1); temp[bb - 1].insert(aa - 1);
    }
    memset(crank, -1, sizeof(crank));
    centroid_decomp(0, -1);

    REP(i, 0, n){
        if (sz(cr[i]) == 0) break; //no more
        VREP(it, cr[i]) dfs2(*it, -1, 1, *it); //start with distance of 0
    }

}
```

Graham Scan

```
struct pt {
    int x, y;
    pt(int _x, int _y): x(_x), y(_y){}
};

bool comparePt(pt a, pt b){
    return MP(a.x, a.y) < MP(b.x, b.y);
}

int orientation(pt a, pt b, pt c) {
    int v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}

bool ccw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    if (a.size() == 1) return;

    sort(a.begin(), a.end(), comparePt);
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.PB(p1); down.PB(p1);
    REP(i, 1, a.size()){
        if (i == a.size() - 1 || cw(p1, a[i], p2, include_collinear)) {
            while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1], a[i], include_collinear))
                up.pop_back();
            up.PB(a[i]);
        }
        if (i == a.size() - 1 || ccw(p1, a[i], p2, include_collinear)) {
            while (down.size() >= 2 && !ccw(down[down.size()-2], down[down.size()-1], a[i], include_collinear))
                down.pop_back();
            down.PB(a[i]);
        }
    }

    if (include_collinear && up.size() == a.size()) {
```

```
        reverse(a.begin(), a.end());
        return;
    }
    a.clear();
    REP(i, 0, up.size()) a.PB(up[i]);
    for (int i = down.size() - 2; i > 0; i--) a.PB(down[i]);
}

set<pi> s; // removes all repeated points

int n, xx, yy;
vector<pt> v;

int main(){
    cin >> n; v.clear(); s.clear();

    // setting up the points
    REP(i, 0, n){
        cin >> xx >> yy;
        if (s.find(MP(xx, yy)) != s.end()) continue;
        s.insert(MP(xx, yy));
        v.PB(pt(xx, yy));
    }

    convex_hull(v, false);
    reverse(v.begin(), v.end()); // in counter clockwise order
    cout << v.size() << endl;
    VREP(it, v) cout << it -> x << " " << it -> y << endl;
}
```

Floyd Warshall

```
int adjMat[N][N];
memset(adjMat, -1, sizeof(adjMat));

REP(k, 0, N){
    REP(i, 0, N){
        REP(j, 0, N){
            if (adjMat[i][k] == -1 || adjMat[k][j] == -1) continue;
            if (adjMat[i][j] == -1 || adjMat[i][j] > adjMat[i][k] + adjMat[k][j]) adjMat[i][j] = adjMat[i][k] +
adjMat[k][j];
        }
    }
}
```

Simplex Algorithm

```
// maximize     c^T x       subject to   Ax <= b, x >= 0
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)

const double EPS = 1e-9;
typedef double DOUBLE; // change to long double as necessary
typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<long long> VI;

struct LPSolver {
    int m, n;
    VI B, N;
    VVD D;

    LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
        for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
        for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[i][n + 1] = b[i]; }
        for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
        N[n] = -1; D[m + 1][n] = 1;
    }

    void Pivot(int r, int s) {
        double inv = 1.0 / D[r][s];
        for (int i = 0; i < m + 2; i++){
            if (i != r){
                for (int j = 0; j < n + 2; j++){
                    if (j != s) D[i][j] -= D[r][j] * D[i][s] * inv;
                }
            }
        }
        for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] *= inv;
        for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] *= -inv;
        D[r][s] = inv;
        swap(B[r], N[s]);
```

```
    }

    bool Simplex(int phase) {
        int x = phase == 1 ? m + 1 : m;
        while (true) {
            int s = -1;
            for (int j = 0; j <= n; j++) {
                if (phase == 2 && N[j] == -1) continue;
                if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;
            }
            if (D[x][s] > -EPS) return true;
            int r = -1;
            for (int i = 0; i < m; i++) {
                if (D[i][s] < EPS) continue;
                if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
                  (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r]) r = i;
            }
            if (r == -1) return false;
            Pivot(r, s);
        }
    }

    DOUBLE Solve(VD &x) {
        int r = 0;
        for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
        if (D[r][n + 1] < -EPS) {
            Pivot(r, n);
            if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return -numeric_limits<DOUBLE>::infinity();
            for (int i = 0; i < m; i++) if (B[i] == -1) {
                int s = -1;
                for (int j = 0; j <= n; j++){
                    if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
                }
                Pivot(i, s);
            }
        }
        if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
        x = VD(n);
        for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
        return D[m][n + 1];
    }
};

int main(){
    vector<vector<double> > A(2 + 2*n, vector<double>(n));
    vector<double> b(2 + 2*n, 1);

    fill(A[0].begin(), A[0].end(), 1);
    fill(A[1].begin(), A[1].end(), -1);

    // handle equality constraints with epsilon room of space
    b[0] = R + EPS;
    b[1] = -R + EPS;

    LPSolver solver(A, b, c);
    vector<double> sol;
    double val = solver.Solve(sol);

    if (isnan(val) || isinf(val)){
        cout << "-1" << endl;
        return 0;
    }
}
```

Matrix Exponentiation

```
ll n, mod = 1e9 + 7;

struct matrix {

    ll siz, m[50][50];

    matrix(int n, ll arr[][50]){
        siz = n;
        REP(i, 0, n){
            REP(j, 0, n) m[i][j] = arr[i][j];
        }
    }

    matrix clone(){ return matrix(siz, m); }

    ll & operator()(int i, int j){ return m[i][j]; }

    matrix operator* (matrix b){
        matrix a = (*this).clone();
        matrix res = b;
        REP(i, 0, a.siz){
            REP(j, 0, b.siz){
                res.m[i][j] = 0;
```

```
                REP(k, 0, a.siz){
                    res.m[i][j] += a.m[i][k]*b.m[k][j];
                    res.m[i][j] %= mod;
                }
            }
        }
        return res;
    }
};

matrix expo(matrix a, ll n){
    if (n == 1) return a;
    matrix half = expo(a, n/2);
    half = half*half;
    if (n % 2 == 1) half = half*a;
    return half;
}

int main(){

    cin >> n;
    ll a[50][50]; //must initialise as a[][50]
    a[0][0] = 19; a[0][1] = 7; a[1][0] = 6; a[1][1] = 20;

    matrix fib = matrix(2, a);

    matrix ans = expo(fib, n);
    cout << ans(0, 0) << endl;
}
```

Determinant

```
//mat must be a square matrix
ll determinant(vector<vector<ll> > mat, ll mod){

    ll n = sz(mat), num1 = 0, num2 = 0, det = 1, index = 0, total = 1;
    ll temp[n + 1];

    REP(i, 0, n){
        index = i;
        while (index < n && mat[index][i] == 0) index++;
        if (index == n) return 0;

        if (index != i){
            REP(j, 0, n) swap(mat[index][j], mat[i][j]);
            det *= -1;
            det = ((det%mod)+mod)%mod;
        }

        REP(j, 0, n) temp[j] = mat[i][j];

        for (int j = i + 1; j < n; j++){
            num1 = temp[i];
            num2 = mat[j][i];

            REP(k, 0, n){
                mat[j][k] = (num1 * mat[j][k]) - (num2 * temp[k]);
                mat[j][k] = (mat[j][k] + mod)%mod;
            }

            total *= num1;
            total = (total%mod + mod)%mod;
        }
    }

    REP(i, 0, n){
        det *= mat[i][i];
        det = (det%mod + mod)%mod;
    }

    return ((det*power(total, mod - 2))%mod + mod)%mod;
}
```