

# Contents

|                                  |    |
|----------------------------------|----|
| <b>1 Basic</b>                   |    |
| 1.1 Default Code                 | 1  |
| 1.2 .vimrc                       | 1  |
| <b>2 Data Structure</b>          |    |
| 2.1 Heavy-Light Decomposition    | 1  |
| 2.2 Li-Chao Tree                 | 2  |
| 2.3 Link Cut Tree                | 2  |
| 2.4 Treap                        | 3  |
| <b>3 Flow Matching</b>           |    |
| 3.1 Bounded Flow                 | 3  |
| 3.2 Dinic                        | 3  |
| 3.3 Gomory Hu                    | 4  |
| 3.4 Hungarian Algorithm          | 4  |
| 3.5 ISAP Algorithm               | 5  |
| 3.6 KM Algorithm                 | 5  |
| 3.7 Max Simple Graph Matching    | 5  |
| 3.8 MCMF                         | 6  |
| 3.9 Min Cost Circulation         | 6  |
| 3.10 SW Mincut                   | 6  |
| <b>4 Geometry</b>                |    |
| 4.1 Geometry Template            | 7  |
| 4.2 Convex Hull                  | 7  |
| 4.3 Minimum Enclosing Circle     | 7  |
| 4.4 Minkowski Sum                | 7  |
| 4.5 Half Plane Intersection      | 8  |
| <b>5 Graph</b>                   |    |
| 5.1 Block Cut Tree               | 8  |
| 5.2 2-SAT                        | 8  |
| 5.3 Dominator Tree               | 9  |
| 5.4 Virtual Tree                 | 9  |
| <b>6 Math</b>                    |    |
| 6.1 Extended Euclidean Algorithm | 9  |
| 6.2 Floor & Ceil                 | 9  |
| 6.3 Legendre                     | 9  |
| 6.4 Simplex                      | 10 |
| <b>7 Polynomial</b>              |    |
| 7.1 FFT                          | 10 |
| 7.2 NTT                          | 10 |
| <b>8 String</b>                  |    |
| 8.1 KMP Algorithm                | 11 |
| 8.2 Manacher Algorithm           | 11 |
| 8.3 MCP                          | 11 |
| 8.4 Suffix Array                 | 11 |
| 8.5 Suffix Array Automaton       | 11 |
| 8.6 Z-value Algorithm            | 12 |
| <b>9 Formula</b>                 |    |
| 9.1 Recurrences                  | 12 |
| 9.2 Trigonometry                 | 12 |
| 9.3 Geometry                     | 12 |
| 9.3.1 Triangles                  | 12 |
| 9.3.2 Quadrilaterals             | 12 |
| 9.3.3 Spherical coordinates      | 12 |
| 9.4 Derivatives/Integrals        | 13 |
| 9.5 Sums                         | 13 |
| 9.6 Series                       | 13 |
| 9.7 Probability theory           | 13 |
| 9.7.1 Discrete distributions     | 13 |
| 9.7.2 Continuous distributions   | 13 |
| 9.8 Markov chains                | 13 |

## 1 Basic

### 1.1 Default Code

```
//Challenge: Accepted
#pragma GCC optimize("Ofast")
#include <bits/stdc++.h>
using namespace std;
#ifdef zisk
void debug(){cout << endl;}
template<class T, class ... U> void debug(T a, U ... b){
    cout << a << " ", debug(b...);}
template<class T> void pary(T l, T r) {
    while (l != r) cout << *l << " ", l++;
    cout << endl;
}
#else
#define debug(...) 0
#define pary(...) 0
#endif
#define ll long long
#define maxn 50005
#define pii pair<int, int>
#define ff first
#define ss second
#define io ios_base::sync_with_stdio(0);cin.tie(0);
#define iter(v) v.begin(),v.end()
#define SZ(v) (int)v.size()
#define pb emplace_back
int main() {
    io
}
```

### 1.2 .vimrc

```
sy on
se nu rnu bs=2 sw=4 ts=4 hls ls=2 si acd bo=all mouse=a
map <F9> :w<bar>!g++ "%" -o %:r -std=c++17 -Wall -Wshadow -
    Dzisk -g -fsanitize=undefined,address<CR>
map <F8> :!./%:r<CR>
map <C-a> <ESC>ggVG
```

## 2 Data Structure

### 2.1 Heavy-Light Decomposition

```
struct Heavy_light_Decomposition { // 1-base
    int n, up[maxn], dep[maxn], to[maxn], siz[maxn], pa[maxn];
};
int C, ti[maxn], ord[maxn], wdown[maxn], edge[maxn], et = 0;
vector<pii> G[maxn];
void init(int _n) {
    n = _n, C = 0, et = 1;
    for (int i = 1; i <= n; i++)
        G[i].clear(), to[i] = 0;
}
void add_edge(int a, int b, int w) {
    G[a].push_back(pii(b, et)), G[b].push_back(pii(a, et));
    edge[et++] = w;
}
void dfs(int u, int f, int d) {
    siz[u] = 1, pa[u] = f, dep[u] = d;
    for (auto &v: G[u])
        if (v.ff != f) {
            dfs(v.ff, u, d+1), siz[u] += siz[v];
            if (siz[to[u]] < siz[v]) to[u] = v;
        }
}
void cut(int u, int link) {
    ti[u] = C;
    ord[C++] = u, up[u] = link;
}
```

```

    if (!to[u]) return;
    cut(to[u], link);
    for (auto v:G[u]) {
        if (v.ff != pa[u] && v.ff != to[u]) cut(v.ff, v.ff);
    }
}

void build() { dfs(1, 1, 1), cut(1, 1); }
int query(int a, int b) {
    int ta = up[a], tb = up[b], re = 0;
    while (ta != tb)
        if (dep[ta] < dep[tb])
            /*query*/, tb = up[b = pa[tb]];
        else /*query*/, ta = up[a = pa[ta]];
    if (a == b) return re;
    if (ti[a] > ti[b]) swap(a, b);
    /*query*/
    return re;
}
};

```

## 2.2 Li-Chao Tree

```

struct LiChao_min {
    struct line {
        ll m, c;
        line(ll _m = 0, ll _c = 0) {
            m = _m;
            c = _c;
        }
        ll eval(ll x) { return m * x + c; }
    };
    struct node {
        node *l, *r;
        line f;
        node(line v) {
            f = v;
            l = r = NULL;
        }
    };
    typedef node *pnode;
    pnode root;
    int sz;
#define mid ((l + r) >> 1)
    void insert(line &v, int l, int r, pnode &nd) {
        if (!nd) {
            nd = new node(v);
            return;
        }
        ll trl = nd->f.eval(l), trr = nd->f.eval(r);
        ll vl = v.eval(l), vr = v.eval(r);
        if (trl <= vl && trr <= vr) return;
        if (trl > vl && trr > vr) {
            nd->f = v;
            return;
        }
        if (trl > vl) swap(nd->f, v);
        if (nd->f.eval(mid) < v.eval(mid))
            insert(v, mid + 1, r, nd->r);
        else swap(nd->f, v), insert(v, l, mid, nd->l);
    }
    ll query(int x, int l, int r, pnode &nd) {
        if (!nd) return inf;
        if (l == r) return nd->f.eval(x);
        if (mid >= x)
            return min(
                nd->f.eval(x), query(x, l, mid, nd->l));
        return min(
            nd->f.eval(x), query(x, mid + 1, r, nd->r));
    }
    /* -sz <= query_x <= sz */
    void init(int _sz) {
        sz = _sz + 1;
        root = NULL;
    }
    void add_line(ll m, ll c) {
        line v(m, c);

```

```

        insert(v, -sz, sz, root);
    }
    ll query(ll x) { return query(x, -sz, sz, root); }
};

```

## 2.3 Link Cut Tree

```

struct Splay { // xor-sum
    static Splay nil;
    Splay *ch[2], *f;
    int val, sum, rev, size;
    Splay(int _val = 0)
        : val(_val), sum(_val), rev(0), size(1) {
        f = ch[0] = ch[1] = &nil;
    }
    bool isr() {
        return f->ch[0] != this && f->ch[1] != this;
    }
    int dir() { return f->ch[0] == this ? 0 : 1; }
    void setCh(Splay *c, int d) {
        ch[d] = c;
        if (c != &nil) c->f = this;
        pull();
    }
    void push() {
        if (!rev) return;
        swap(ch[0], ch[1]);
        if (ch[0] != &nil) ch[0]->rev ^= 1;
        if (ch[1] != &nil) ch[1]->rev ^= 1;
        rev = 0;
    }
    void pull() {
        // take care of the nil!
        size = ch[0]->size + ch[1]->size + 1;
        sum = ch[0]->sum ^ ch[1]->sum ^ val;
        if (ch[0] != &nil) ch[0]->f = this;
        if (ch[1] != &nil) ch[1]->f = this;
    }
} Splay::nil;
Splay *nil = &Splay::nil;
void rotate(Splay *x) {
    Splay *p = x->f;
    int d = x->dir();
    if (!p->isr()) p->f->setCh(x, p->dir());
    else x->f = p->f;
    p->setCh(x->ch[!d], d);
    x->setCh(p, !d);
    p->pull(), x->pull();
}
void splay(Splay *x) {
    vector<Splay *> splayVec;
    for (Splay *q = x;; q = q->f) {
        splayVec.pb(q);
        if (q->isr()) break;
    }
    reverse(ALL(splayVec));
    for (auto it : splayVec) it->push();
    while (!x->isr()) {
        if (x->f->isr()) rotate(x);
        else if (x->dir() == x->f->dir())
            rotate(x->f), rotate(x);
        else rotate(x), rotate(x);
    }
}
Splay *access(Splay *x) {
    Splay *q = nil;
    for (; x != nil; x = x->f)
        splay(x), x->setCh(q, 1), q = x;
    return q;
}
void root_path(Splay *x) { access(x), splay(x); }
void chroot(Splay *x) {
    root_path(x), x->rev ^= 1;
    x->push(), x->pull();
}
void split(Splay *x, Splay *y) {

```

```

    chroot(x), root_path(y);
}
void link(Splay *x, Splay *y) {
    root_path(x), chroot(y);
    x->setCh(y, 1);
}
void cut(Splay *x, Splay *y) {
    split(x, y);
    if (y->size != 5) return;
    y->push();
    y->ch[0] = y->ch[0]->f = nil;
}
Splay *get_root(Splay *x) {
    for (root_path(x); x->ch[0] != nil; x = x->ch[0])
        x->push();
    splay(x);
    return x;
}
bool conn(Splay *x, Splay *y) {
    return get_root(x) == get_root(y);
}
Splay *lca(Splay *x, Splay *y) {
    access(x), root_path(y);
    if (y->f == nil) return y;
    return y->f;
}
void change(Splay *x, int val) {
    splay(x), x->val = val, x->pull();
}
int query(Splay *x, Splay *y) {
    split(x, y);
    return y->sum;
}
}

```

## 2.4 Treap

```

struct node {
    int data, sz;
    node *l, *r;
    node(int k) : data(k), sz(1), l(0), r(0) {}
    void up() {
        sz = 1;
        if (l) sz += l->sz;
        if (r) sz += r->sz;
    }
    void down() {}
};
int sz(node *a) { return a ? a->sz : 0; }
node *merge(node *a, node *b) {
    if (!a || !b) return a ? a : b;
    if (rand() % (sz(a) + sz(b)) < sz(a))
        return a->down(), a->r = merge(a->r, b), a->up(),
            a;
    return b->down(), b->l = merge(a, b->l), b->up(), b;
}
void split(node *o, node *&a, node *&b, int k) {
    if (!o) return a = b = 0, void();
    o->down();
    if (o->data <= k)
        a = o, split(o->r, a->r, b, k), a->up();
    else b = o, split(o->l, a, b->l, k), b->up();
}
void split2(node *o, node *&a, node *&b, int k) {
    if (sz(o) <= k) return a = o, b = 0, void();
    o->down();
    if (sz(o->l) + 1 <= k)
        a = o, split2(o->r, a->r, b, k - sz(o->l) - 1);
    else b = o, split2(o->l, a, b->l, k);
    o->up();
}
node *kth(node *o, int k) {
    if (k <= sz(o->l)) return kth(o->l, k);
    if (k == sz(o->l) + 1) return o;
    return kth(o->r, k - sz(o->l) - 1);
}
int Rank(node *o, int key) {

```

```

    if (!o) return 0;
    if (o->data < key)
        return sz(o->l) + 1 + Rank(o->r, key);
    else return Rank(o->l, key);
}
bool erase(node *&o, int k) {
    if (!o) return 0;
    if (o->data == k) {
        node *t = o;
        o->down(), o = merge(o->l, o->r);
        delete t;
        return 1;
    }
    node *&t = k < o->data ? o->l : o->r;
    return erase(t, k) ? o->up(), 1 : 0;
}
void insert(node *&o, int k) {
    node *a, *b;
    split(o, a, b, k),
        o = merge(a, merge(new node(k), b));
}
void interval(node *&o, int l, int r) {
    node *a, *b, *c;
    split2(o, a, b, l - 1), split2(b, b, c, r);
    // operate
    o = merge(a, merge(b, c));
}

```

## 3 Flow Matching

### 3.1 Bounded Flow

```

struct BoundedFlow { // 0-base
    struct edge {
        int to, cap, flow, rev;
    };
    vector<edge> G[N];
    int n, s, t, dis[N], cur[N], cnt[N];
    void init(int _n) {
        n = _n;
        for (int i = 0; i < n + 2; ++i)
            G[i].clear(), cnt[i] = 0;
    }
    void add_edge(int u, int v, int lcap, int rcap) {
        cnt[u] -= lcap, cnt[v] += lcap;
        G[u].pb(edge{v, rcap, lcap, SZ(G[v])});
        G[v].pb(edge{u, 0, 0, SZ(G[u]) - 1});
    }
    void add_edge(int u, int v, int cap) {
        G[u].pb(edge{v, cap, 0, SZ(G[v])});
        G[v].pb(edge{u, 0, 0, SZ(G[u]) - 1});
    }
    int dfs(int u, int cap) {
        if (u == t || !cap) return cap;
        for (int &i = cur[u]; i < SZ(G[u]); ++i) {
            edge &e = G[u][i];
            if (dis[e.to] == dis[u] + 1 && e.cap != e.flow) {
                int df = dfs(e.to, min(e.cap - e.flow, cap));
                if (df) {
                    e.flow += df, G[e.to][e.rev].flow -= df;
                    return df;
                }
            }
        }
        dis[u] = -1;
        return 0;
    }
    bool bfs() {
        fill_n(dis, n + 3, -1);
        queue<int> q;
        q.push(s), dis[s] = 0;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (edge &e : G[u])

```

```

    if (!~dis[e.to] && e.flow != e.cap)
        q.push(e.to), dis[e.to] = dis[u] + 1;
}
return dis[t] != -1;
}
int maxflow(int _s, int _t) {
    s = _s, t = _t;
    int flow = 0, df;
    while (bfs()) {
        fill_n(cur, n + 3, 0);
        while ((df = dfs(s, INF))) flow += df;
    }
    return flow;
}
bool solve() {
    int sum = 0;
    for (int i = 0; i < n; ++i)
        if (cnt[i] > 0)
            add_edge(n + 1, i, cnt[i]), sum += cnt[i];
        else if (cnt[i] < 0) add_edge(i, n + 2, -cnt[i]);
    if (sum != maxflow(n + 1, n + 2)) sum = -1;
    for (int i = 0; i < n; ++i)
        if (cnt[i] > 0)
            G[n + 1].pop_back(), G[i].pop_back();
        else if (cnt[i] < 0)
            G[i].pop_back(), G[n + 2].pop_back();
    return sum != -1;
}
int solve(int _s, int _t) {
    add_edge(_t, _s, INF);
    if (!solve()) return -1; // invalid flow
    int x = G[_t].back().flow;
    return G[_t].pop_back(), G[_s].pop_back(), x;
}
};

```

### 3.2 Dinic

```

struct MaxFlow { // 1-base
    struct edge {
        int to, cap, flow, rev;
    };
    vector<edge> g[maxn];
    int s, t, dis[maxn], ind[maxn], n;

    void init(int _n) {
        n = _n + 2;
        s = _n + 1, t = _n + 2;
        for (int i = 0; i <= n; ++i) g[i].clear();
    }
    void reset() {
        for (int i = 0; i <= n; ++i)
            for (auto &j : g[i]) j.flow = 0;
    }
    void add_edge(int u, int v, int cap) {
        g[u].pb(edge{v, cap, 0, (int)g[v].size()});
        g[v].pb(edge{u, 0, 0, (int)g[u].size() - 1});
        //change g[v] to cap for undirected graphs
    }
    bool bfs() {
        fill(dis, dis+n+1, -1);
        queue<int> q;
        q.push(s), dis[s] = 0;
        while (!q.empty()) {
            int cur = q.front(); q.pop();
            for (auto &e : g[cur]) {
                if (dis[e.to] == -1 && e.flow != e.cap) {
                    q.push(e.to);
                    dis[e.to] = dis[cur] + 1;
                }
            }
        }
        return dis[t] != -1;
    }
    int dfs(int u, int cap) {
        if (u == t || !cap) return cap;

```

```

        for (int &i = ind[u]; i < (int)g[u].size(); ++i) {
            edge &e = g[u][i];
            if (dis[e.to] == dis[u] + 1 && e.flow != e.cap) {
                int df = dfs(e.to, min(e.cap - e.flow, cap));
                if (df) {
                    e.flow += df;
                    g[e.to][e.rev].flow -= df;
                    return df;
                }
            }
        }
        dis[u] = -1;
        return 0;
    }
    int maxflow() {
        int flow = 0, df;
        while (bfs()) {
            fill(ind, ind+n+1, 0);
            while ((df = dfs(s, inf))) flow += df;
        }
        return flow;
    }
} flow;

```

### 3.3 Gomory Hu

```

MaxFlow Dinic;
int g[MAXN];
void GomoryHu(int n) { // 0-base
    fill_n(g, n, 0);
    for (int i = 1; i < n; ++i) {
        Dinic.reset();
        add_edge(i, g[i], Dinic.maxflow(i, g[i]));
        for (int j = i + 1; j <= n; ++j)
            if (g[j] == g[i] && ~Dinic.dis[j])
                g[j] = i;
    }
}

```

### 3.4 Hungarian Algorithm

```

int c[maxn][maxn]; //hungarian algorithm in O(n^3)
//1 base
int lx[maxn], ly[maxn], mx[maxn], my[maxn];
bool vx[maxn], vy[maxn];
int slack[maxn];
int tot;
bool dfs(int n, bool ch) {
    if (vx[n]) return false;
    vx[n] = 1;
    for (int v = 1; v <= tot; v++) {
        slack[v] = min(slack[v], lx[n] + ly[v] - c[n][v]);
        if (lx[n] + ly[v] - c[n][v] > 0) continue;
        vy[v] = 1;
        if (!my[v] || dfs(my[v], ch)) {
            if (ch) mx[n] = v, my[v] = n;
            return true;
        }
    }
    return false;
}
int main() {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) vx[j] = vy[j] = 0;
        for (int j = 1; j <= n; j++) slack[j] = 1<<30;
        if (dfs(i, 1)) continue;
        bool aug = 0;
        while (!aug) {
            for (int j = 1; j <= n; j++) {
                if (!vy[j] && slack[j] == 0) {
                    vy[j] = 1;
                    if (dfs(my[j], 0)) {
                        aug = 1;
                        break;
                    }
                }
            }
        }
    }
}

```

```

    }
}
if (aug) break;
int delta = 1<<30;
for (int j = 1; j <= n; j++) {
    if (!vy[j]) delta = min(delta, slack[j]);
}
for (int j = 1; j <= n; j++) {
    if (vx[j]) lx[j] -= delta;
    if (vy[j]) ly[j] += delta;
    else {
        slack[j] -= delta;
        if (slack[j] == 0 && !my[j]) aug = 1;
    }
}
}
for (int j = 1; j <= n; j++) vx[j] = vy[j] = 0;
dfs(i, 1);
}
}

```

### 3.5 ISAP Algorithm

```

struct Maxflow { //to be modified
    static const int MAXV = 20010;
    static const int INF = 10000000;
    struct Edge {
        int v, c, r;
        Edge(int _v, int _c, int _r)
            : v(_v), c(_c), r(_r) {}
    };
    int s, t;
    vector<Edge> G[MAXV * 2];
    int iter[MAXV * 2], d[MAXV * 2], gap[MAXV * 2], tot;
    void init(int x) {
        tot = x + 2;
        s = x + 1, t = x + 2;
        for (int i = 0; i <= tot; i++) {
            G[i].clear();
            iter[i] = d[i] = gap[i] = 0;
        }
    }
    void addEdge(int u, int v, int c) {
        G[u].push_back(Edge(v, c, SZ(G[v])));
        G[v].push_back(Edge(u, 0, SZ(G[u]) - 1));
    }
    int dfs(int p, int flow) {
        if (p == t) return flow;
        for (int &i = iter[p]; i < SZ(G[p]); i++) {
            Edge &e = G[p][i];
            if (e.c > 0 && d[p] == d[e.v] + 1) {
                int f = dfs(e.v, min(flow, e.c));
                if (f) {
                    e.c -= f;
                    G[e.v][e.r].c += f;
                    return f;
                }
            }
        }
    }
    if ((--gap[d[p]]) == 0) d[s] = tot;
    else {
        d[p]++;
        iter[p] = 0;
        ++gap[d[p]];
    }
    return 0;
}
int solve() {
    int res = 0;
    gap[0] = tot;
    for (res = 0; d[s] < tot; res += dfs(s, INF))
        ;
    return res;
}
} flow;

```

### 3.6 KM Algorithm

```

int n, m; //1-base, max matching
int mx[maxn], my[maxn];
bool adj[maxn][maxn], vis[maxn];
bool dfs(int n) {
    if (vis[n]) return false;
    vis[n] = 1;
    for (int v = 1; v <= n; v++) {
        if (!adj[n][v]) continue;
        if (!my[v] || (my[v] && dfs(my[v]))) {
            mx[n] = v, my[v] = n;
            return true;
        }
    }
    return false;
}
//min vertex cover: take unmatched vertex in L and find
//alternating tree,
//ans is not reached in L + reached in R

```

### 3.7 Max Simple Graph Matching

```

struct GenMatch { // 1-base
    int V, pr[N];
    bool el[N][N], inq[N], inp[N], inb[N];
    int st, ed, nb, bk[N], djs[N], ans;
    void init(int _V) {
        V = _V;
        for (int i = 0; i <= V; ++i) {
            for (int j = 0; j <= V; ++j) el[i][j] = 0;
            pr[i] = bk[i] = djs[i] = 0;
            inq[i] = inp[i] = inb[i] = 0;
        }
    }
    void add_edge(int u, int v) {
        el[u][v] = el[v][u] = 1;
    }
    int lca(int u, int v) {
        fill_n(inp, V + 1, 0);
        while (1)
            if (u = djs[u], inp[u] = true, u == st) break;
            else u = bk[pr[u]];
        while (1)
            if (v = djs[v], inp[v]) return v;
            else v = bk[pr[v]];
        return v;
    }
    void upd(int u) {
        for (int v; djs[u] != nb;) {
            v = pr[u], inb[djs[u]] = inb[djs[v]] = true;
            u = bk[v];
            if (djs[u] != nb) bk[u] = v;
        }
    }
    void blo(int u, int v, queue<int> &qe) {
        nb = lca(u, v), fill_n(inb, V + 1, 0);
        upd(u), upd(v);
        if (djs[u] != nb) bk[u] = v;
        if (djs[v] != nb) bk[v] = u;
        for (int tu = 1; tu <= V; ++tu)
            if (inb[djs[tu]])
                if (djs[tu] = nb, !inq[tu])
                    qe.push(tu), inq[tu] = 1;
    }
    void flow() {
        fill_n(inq + 1, V, 0), fill_n(bk + 1, V, 0);
        iota(djs + 1, djs + V + 1, 1);
        queue<int> qe;
        qe.push(st), inq[st] = 1, ed = 0;
        while (!qe.empty()) {
            int u = qe.front();
            qe.pop();
            for (int v = 1; v <= V; ++v)
                if (el[u][v] && djs[u] != djs[v] &&
                    pr[u] != v) {

```

```

    if ((v == st) ||
        (pr[v] > 0 && bk[pr[v]] > 0)) {
        blo(u, v, qe);
    } else if (!bk[v]) {
        if (bk[v] = u, pr[v] > 0) {
            if (!inq[pr[v]]) qe.push(pr[v]);
        } else {
            return ed = v, void();
        }
    }
}
}
}
void aug() {
    for (int u = ed, v, w; u > 0;)
        v = bk[u], w = pr[v], pr[v] = u, pr[u] = v,
        u = w;
}
int solve() {
    fill_n(pr, V + 1, 0), ans = 0;
    for (int u = 1; u <= V; ++u)
        if (!pr[u])
            if (st = u, flow(), ed > 0) aug(), ++ans;
    return ans;
}
};

```

### 3.8 MCMF

```

struct MCMF { // 0-base
    struct edge {
        ll from, to, cap, flow, cost, rev;
    } * past[maxn];
    vector<edge> G[maxn];
    bitset<maxn> inq;
    ll dis[maxn], up[maxn], s, t, mx, n;
    bool BellmanFord(ll &flow, ll &cost) {
        fill(dis, dis + n, inf);
        queue<ll> q;
        q.push(s), inq.reset(), inq[s] = 1;
        up[s] = mx - flow, past[s] = 0, dis[s] = 0;
        while (!q.empty()) {
            ll u = q.front();
            q.pop(), inq[u] = 0;
            if (!up[u]) continue;
            for (auto &e : G[u])
                if (e.flow != e.cap &&
                    dis[e.to] > dis[u] + e.cost) {
                    dis[e.to] = dis[u] + e.cost, past[e.to] = &e;
                    up[e.to] = min(up[u], e.cap - e.flow);
                    if (!inq[e.to]) inq[e.to] = 1, q.push(e.to);
                }
        }
        if (dis[t] == inf) return 0;
        flow += up[t], cost += up[t] * dis[t];
        for (ll i = t; past[i]; i = past[i]->from) {
            auto &e = *past[i];
            e.flow += up[t], G[e.to][e.rev].flow -= up[t];
        }
        return 1;
    }
    ll MinCostMaxFlow(ll _s, ll _t, ll &cost) {
        s = _s, t = _t, cost = 0;
        ll flow = 0;
        while (BellmanFord(flow, cost));
        return flow;
    }
    void init(ll _n, ll _mx) {
        n = _n, mx = _mx;
        for (int i = 0; i < n; ++i) G[i].clear();
    }
    void add_edge(ll a, ll b, ll cap, ll cost) {
        G[a].pb(edge{a, b, cap, 0, cost, G[b].size()});
        G[b].pb(edge{b, a, 0, 0, -cost, G[a].size() - 1});
    }
};

```

### 3.9 Min Cost Circulation

```

//to be modified
struct Edge { int to, cap, rev, cost; };
vector<Edge> g[kN];
int dist[kN], pv[kN], ed[kN];
bool mark[kN];
int NegativeCycle(int n) {
    memset(mark, false, sizeof(mark));
    memset(dist, 0, sizeof(dist));
    int upd = -1;
    for (int i = 0; i <= n; ++i) {
        for (int j = 0; j < n; ++j) {
            int idx = 0;
            for (auto &e : g[j]) {
                if (e.cap > 0 && dist[e.to] > dist[j] + e.cost) {
                    dist[e.to] = dist[j] + e.cost;
                    pv[e.to] = j, ed[e.to] = idx;
                    if (i == n) {
                        upd = j;
                        while (!mark[upd]) mark[upd] = true, upd = pv[upd];
                        return upd;
                    }
                }
            }
            idx++;
        }
    }
    return -1;
}
int Solve(int n) {
    int rt = -1, ans = 0;
    while ((rt = NegativeCycle(n)) >= 0) {
        memset(mark, false, sizeof(mark));
        vector<pair<int, int>> cyc;
        while (!mark[rt]) {
            cyc.emplace_back(pv[rt], ed[rt]);
            mark[rt] = true;
            rt = pv[rt];
        }
        reverse(cyc.begin(), cyc.end());
        int cap = kInf;
        for (auto &i : cyc) {
            auto &e = g[i.first][i.second];
            cap = min(cap, e.cap);
        }
        for (auto &i : cyc) {
            auto &e = g[i.first][i.second];
            e.cap -= cap;
            g[e.to][e.rev].cap += cap;
            ans += e.cost * cap;
        }
    }
    return ans;
}
}

```

### 3.10 SW Mincut

```

// stoer wagner algorithm: global min cut
const int maxn = 505;
struct SW { // O(V^3) 0-based
    int n, vis[maxn], del[maxn];
    int edge[maxn][maxn], wei[maxn];
    void init(int _n) {
        n = _n;
        fill(del, del + n, 0);
        for (int i = 0; i < n; ++i) fill(edge[i], edge[i] + n, 0);
    }
    void addEdge(int u, int v, int w) {
        edge[u][v] += w, edge[v][u] += w;
    }
    void search(int &s, int &t) {
        fill(vis, vis + n, 0);
        fill(wei, wei + n, 0);
    }
};

```

```

s = t = -1;
while (1) {
    int ma = -1, cur = 0;
    for (int i = 0; i < n; ++i)
        if (!del[i] && !vis[i] && ma < wei[i])
            cur = i, ma = wei[i];
    if (mx == -1) break;
    vis[cur] = 1, s = t, t = cur;
    for (int i = 0; i < n; ++i)
        if (!vis[i] && !del[i]) wei[i] += edge[cur][i];
}
int solve() {
    int ret = INF;
    for (int i = 0, x=0, y=0; i < n-1; ++i) {
        search(x, y), ret = min(res, wei[y]), del[y] = 1;
        for (int j = 0; j < n; ++j)
            edge[x][j] = (edge[j][x] += edge[y][j]);
    }
    return ret;
}
};

```

## 4 Geometry

### 4.1 Geometry Template

```

using NumType = ll;
// using NumType = ld;
using Pt = pair<NumType, NumType>;
using Line = pair<Pt, Pt>;
#define X first
#define Y second
// ld eps = 1e-7;

Pt operator+(Pt a, Pt b)
{ return {a.X + b.X, a.Y + b.Y}; }
Pt operator-(Pt a, Pt b)
{ return {a.X - b.X, a.Y - b.Y}; }
Pt operator*(NumType i, Pt v)
{ return {i * v.X, i * v.Y}; }
Pt operator/(Pt v, NumType i)
{ return {v.X / i, v.Y / i}; }
NumType dot(Pt a, Pt b)
{ return a.X * b.X + a.Y * b.Y; }
NumType cross(Pt a, Pt b)
{ return a.X * b.Y - a.Y * b.X; }
NumType abs2(Pt v)
{ return v.X * v.X + v.Y * v.Y; }
int sgn(NumType v)
{ return v > 0 ? 1 : (v < 0 ? -1 : 0); }
// int sgn(NumType v){ return v > eps ? 1 : ( v < -eps ? -1 : 0); }

int ori(Pt a, Pt b, Pt c)
{ return sgn(cross(b - a, c - a)); }
bool collinearity(Pt a, Pt b, Pt c)
{ return ori(a, b, c) == 0; }
bool btw(Pt p, Pt a, Pt b)
{ return collinearity(p, a, b) && sgn(dot(a - p, b - p)) <= 0; }

bool intersect(Line a, Line b){
    Pt p1, p2, p3, p4;
    tie(p1, p2) = a; tie(p3, p4) = b;
    if(btw(p1, p3, p4) || btw(p2, p3, p4) || btw(p3, p1, p2)
        || btw(p4, p1, p2))
        return true;
    return ori(p1, p2, p3) * ori(p1, p2, p4) < 0 &&
        ori(p3, p4, p1) * ori(p3, p4, p2) < 0;
}

```

### 4.2 Convex Hull

```
vector<int> getConvexHull(vector<Pt>& pts){
```

```

vector<int> id(SZ(pts));
iota(iter(id), 0);
sort(iter(id), [&](int x, int y){ return pts[x] < pts[y];
});
vector<int> hull;
for(int tt = 0; tt < 2; tt++){
    int sz = SZ(hull);
    for(int j : id){
        Pt p = pts[j];
        while(SZ(hull) - sz >= 2 &&
            cross(pts[hull.back()] - pts[hull[SZ(hull) - 2]],
                p - pts[hull[SZ(hull) - 2]]) <= 0)
            hull.pop_back();
        hull.pb(j);
    }
    hull.pop_back();
    reverse(iter(id));
}
return hull;
}

```

### 4.3 Minimum Enclosing Circle

```

using NumType = ld;
pair<Pt, ld> MinimumEnclosingCircle(vector<Pt> &pts){
    random_shuffle(iter(pts));
    Pt c = pts[0];
    ld r = 0;
    for(int i = 1; i < SZ(pts); i++){
        if(abs(pts[i] - c) <= r) continue;
        c = pts[i]; r = 0;
        for(int j = 0; j < i; j++){
            if(abs(pts[j] - c) <= r) continue;
            c = (pts[i] + pts[j]) / 2;
            r = abs(pts[i] - c);
            for(int k = 0; k < j; k++){
                if(abs(pts[k] - c) > r)
                    c = circumcenter(pts[i], pts[j], pts[k]);
            }
        }
    }
    return {c, r};
}

```

### 4.4 Minkowski Sum

```

void reorder_poly(vector<Pt>& pnts){
    int mn = 0;
    for(int i = 1; i < (int)pnts.size(); i++){
        if(pnts[i].Y < pnts[mn].Y || (pnts[i].Y == pnts[mn].Y
            && pnts[i].X < pnts[mn].X))
            mn = i;
    }
    rotate(pnts.begin(), pnts.begin() + mn, pnts.end());
}

vector<Pt> minkowski(vector<Pt> P, vector<Pt> Q){
    reorder_poly(P);
    reorder_poly(Q);
    int psz = P.size();
    int qsz = Q.size();
    P.pb(P[0]);
    P.pb(P[1]);
    Q.pb(Q[0]);
    Q.pb(Q[1]);
    vector<Pt> ans;
    int i = 0, j = 0;
    while(i < psz || j < qsz){
        ans.pb(P[i] + Q[j]);
        int t = sgn(cross(P[i + 1] - P[i], Q[j + 1] - Q[j]));
        if(t >= 0) i++;
        if(t <= 0) j++;
    }
    return ans;
}

```

## 4.5 Half Plane Intersection

```
// copy from 8BQube
bool isin( Line l0, Line l1, Line l2 ) {
    // Check inter(l1, l2) in l0
    pdd p = intersect(l1.X, l1.Y, l2.X, l2.Y);
    return sign(cross(l0.Y - l0.X, p - l0.X)) > 0;
}
/* Having solution, check intersect(ret[0], ret[1])
 * in all the lines. (use (l.Y - l.X) ^ (p - l.X) > 0
 */
/* --- Line.X --- Line.Y --- */
vector<Line> halfPlaneInter(vector<Line> lines) {
    vector<double> ata(SZ(lines)), ord(SZ(lines));
    for(int i = 0; i < SZ(lines); ++i) {
        ord[i] = i;
        pdd d = lines[i].Y - lines[i].X;
        ata[i] = atan2(d.Y, d.X);
    }
    sort(ALL(ord), [&](int i, int j) {
        if (fabs(ata[i] - ata[j]) >= eps)
            return ata[i] < ata[j];
        return ori(lines[i].X, lines[i].Y, lines[j].Y) < 0;
    });
    vector<Line> fin(1, lines[ord[0]]);
    for (int i = 1; i < SZ(lines); ++i)
        if (fabs(ata[ord[i]] - ata[ord[i - 1]]) > eps)
            fin.pb(lines[ord[i]]);
    deque<Line> dq;
    for (int i = 0; i < SZ(fin); ++i) {
        while (SZ(dq) >= 2 && !isin(fin[i], dq[SZ(dq) - 2], dq.back()))
            dq.pop_back();
        while (SZ(dq) >= 2 && !isin(fin[i], dq[0], dq[1]))
            dq.pop_front();
        dq.pb(fin[i]);
    }
    while (SZ(dq) >= 3 && !isin(dq[0], dq[SZ(dq) - 2], dq.back()))
        dq.pop_back();
    while (SZ(dq) >= 3 && !isin(dq.back(), dq[0], dq[1]))
        dq.pop_front();
    return vector<Line>(ALL(dq));
}
```

## 5 Graph

### 5.1 Block Cut Tree

```
struct BlockCutTree{
    vector<vector<int>> tree; // 1-based
    vector<int> node;
    vector<int> type; // 0:square, 1:circle

    bool iscut(int v){
        return type[node[v]] == 1;
    }

    vector<int> getbcc(int v){
        if(!iscut(v)) return {node[v]};
        vector<int> ans;
        for(int i : tree[node[v]])
            ans.pb(i);
        return ans;
    }

    void build(int n, vector<vector<int>>& g){
        tree.resize(2 * n + 1);
        type.resize(2 * n + 1);
        node.resize(n + 1, -1);
        vector<int> in(n + 1);
        vector<int> low(n + 1);
        stack<int> st;

        int ts = 1;
        int bcc = 1;
```

```
auto addv = [&](int id, int v){
    if(node[v] == -1){
        node[v] = id;
        return;
    }
    if(type[node[v]] == 0){
        int o = node[v];
        node[v] = bcc++;
        type[node[v]] = 1;
        tree[o].pb(node[v]);
        tree[node[v]].pb(o);
    }
    tree[id].pb(node[v]);
    tree[node[v]].pb(id);
};

function<void(int, int)> dfs = [&](int now, int p){
    in[now] = low[now] = ts++;
    st.push(now);
    int child = 0;
    for(int i : g[now]){
        if(i == p) continue;
        if(in[i]){
            low[now] = min(low[now], in[i]);
            continue;
        }
        child++;
        dfs(i, now);
        low[now] = min(low[now], low[i]);

        if(low[i] >= in[now]){
            int nowid = bcc++;
            while(true){
                int x = st.top();
                st.pop();
                addv(nowid, x);
                if(x == i) break;
            }
            addv(nowid, now);
        }
    }
    if(child == 0 && now == p) addv(bcc++, now);
};
dfs(1, 1);
};
```

### 5.2 2-SAT

```
struct SAT{ // 1-based
    int n;
    vector<vector<int>> g, rg;
    bool ok = true;
    vector<bool> ans;

    void init(int _n){
        n = _n;
        g.resize(2 * n + 1);
        rg.resize(2 * n + 1);
        ans.resize(n + 1);
    }

    int neg(int v){
        return v <= n ? v + n : v - n;
    }

    void addEdge(int u, int v){
        g[u].eb(v);
        rg[v].eb(u);
    }

    void addClause(int a, int b){
        addEdge(a, b);
        addEdge(neg(b), neg(a));
    }

    void build(){
        vector<bool> vst(n + 1);
        vector<int> tmp, scc(n + 1, -1);
        int cnt = 1;
        function<void(int)> dfs = [&](int now){
```



```

    vst[now] = true;
    for(int i : rg[now]){
        if(vst[i]) continue;
        dfs(i);
    }
    tmp.pb(now);
};
for(int i = 1; i <= 2 * n; i++){
    if(!vst[i]) dfs(i);
}
reverse(iter(tmp));
function<void(int, int)> dfs2 = [&](int now, int id){
    scc[now] = id;
    for(int i : g[now]){
        if(scc[i] != -1) continue;
        dfs2(i, id);
    }
};
for(int i : tmp){
    if(scc[i] == -1) dfs2(i, cnt++);
}
for(int i = 1; i <= n; i++){
    if(scc[i] == scc[neg(i)]){
        ok = false;
        return;
    }
    if(scc[i] < scc[neg(i)]) ans[i] = true;
    else ans[i] = false;
}
};

```

### 5.3 Dominator Tree

```

// copy from 8BQube
struct dominator_tree { // 1-base
    vector<int> G[N], rG[N];
    int n, pa[N], dfn[N], id[N], Time;
    int semi[N], idom[N], best[N];
    vector<int> tree[N]; // dominator_tree
    void init(int _n) {
        n = _n;
        for (int i = 1; i <= n; ++i)
            G[i].clear(), rG[i].clear();
    }
    void add_edge(int u, int v) {
        G[u].pb(v), rG[v].pb(u);
    }
    void dfs(int u) {
        id[dfn[u]] = ++Time;
        for (auto v : G[u])
            if (!dfn[v]) dfs(v), pa[dfn[v]] = dfn[u];
    }
    int find(int y, int x) {
        if (y <= x) return y;
        int tmp = find(pa[y], x);
        if (semi[best[y]] > semi[best[pa[y]]])
            best[y] = best[pa[y]];
        return pa[y] = tmp;
    }
    void tarjan(int root) {
        Time = 0;
        for (int i = 1; i <= n; ++i) {
            dfn[i] = idom[i] = 0;
            tree[i].clear();
            best[i] = semi[i] = i;
        }
        dfs(root);
        for (int i = Time; i > 1; --i) {
            int u = id[i];
            for (auto v : rG[u])
                if (v = dfn[v]) {
                    find(v, i);
                    semi[i] = min(semi[i], semi[best[v]]);
                }
        }
    }
};

```

```

    tree[semi[i]].pb(i);
    for (auto v : tree[pa[i]]) {
        find(v, pa[i]);
        idom[v] =
            semi[best[v]] == pa[i] ? pa[i] : best[v];
    }
    tree[pa[i]].clear();
}
for (int i = 2; i <= Time; ++i) {
    if (idom[i] != semi[i]) idom[i] = idom[idom[i]];
    tree[id[idom[i]]].pb(id[i]);
}
};

```

### 5.4 Virtual Tree

```

// copy from 8BQube
vector<int> vG[N];
int top, st[N];

void insert(int u) {
    if (top == -1) return st[++top] = u, void();
    int p = LCA(st[top], u);
    if (p == st[top]) return st[++top] = u, void();
    while (top >= 1 && dep[st[top - 1]] >= dep[p])
        vG[st[top - 1]].pb(st[top]), --top;
    if (st[top] != p)
        vG[p].pb(st[top]), --top, st[++top] = p;
    st[++top] = u;
}

void reset(int u) {
    for (int i : vG[u]) reset(i);
    vG[u].clear();
}

void solve(vector<int> &v) {
    top = -1;
    sort(ALL(v),
        [&](int a, int b) { return dfn[a] < dfn[b]; });
    for (int i : v) insert(i);
    while (top > 0) vG[st[top - 1]].pb(st[top]), --top;
    // do something
    reset(v[0]);
}

```

## 6 Math

### 6.1 Extended Euclidean Algorithm

```

// ax+ny = 1, ax+ny == ax == 1 (mod n)
void extgcd(ll x, ll y, ll &g, ll &a, ll &b) {
    if (y == 0) g = x, a = 1, b = 0;
    else extgcd(y, x % y, g, b, a), b -= (x / y) * a;
}

```

### 6.2 Floor & Ceil

```

int floor_div(int a, int b){
    return a/b - (a%b && a < 0 ^ b < 0);
}
int ceil_div(int a, int b){
    return a/b + (a%b && a < 0 ^ b > 0);
}

```

### 6.3 Legendre

```

// the Jacobi symbol is a generalization of the Legendre
// symbol,
// such that the bottom doesn't need to be prime.

```

```
// (n|p) -> same as legendre
// (n|ab) = (n|a)(n|b)
// work with long long
int Jacobi(int a, int m) {
    int s = 1;
    for (; m > 1; ) {
        a %= m;
        if (a == 0) return 0;
        const int r = __builtin_ctz(a);
        if ((r & 1) && ((m + 2) & 4)) s = -s;
        a >>= r;
        if (a & m & 2) s = -s;
        swap(a, m);
    }
    return s;
}

// 0: a == 0
// -1: a isn't a quad res of p
// else: return X with X^2 % p == a
// doesn't work with long long
int QuadraticResidue(int a, int p) {
    if (p == 2) return a & 1;
    const int jc = Jacobi(a, p);
    if (jc == 0) return 0;
    if (jc == -1) return -1;
    int b, d;
    for (; ; ) {
        b = rand() % p;
        d = (1LL * b * b + p - a) % p;
        if (Jacobi(d, p) == -1) break;
    }
    int f0 = b, f1 = 1, g0 = 1, g1 = 0, tmp;
    for (int e = (1LL + p) >> 1; e; e >>= 1) {
        if (e & 1) {
            tmp = (1LL * g0 * f0 + 1LL * d * (1LL * g1 * f1 % p))
                % p;
            g1 = (1LL * g0 * f1 + 1LL * g1 * f0) % p;
            g0 = tmp;
        }
        tmp = (1LL * f0 * f0 + 1LL * d * (1LL * f1 * f1 % p)) %
            p;
        f1 = (2LL * f0 * f1) % p;
        f0 = tmp;
    }
    return g0;
}
```

## 6.4 Simplex

```
#pragma once
```

```
typedef double T; // long double, Rational, double + mod<P>
>...
typedef vector<T> vd;
typedef vector<vd> vvd;

const T eps = 1e-8, inf = 1/.0;
#define MP make_pair
#define ltj(X) if(s == -1 || MP(X[j],N[j]) < MP(X[s],N[s]))
    s=j

struct LPSolver {
    int m, n;
    vi N, B;
    vvd D;

    LPSolver(const vvd& A, const vd& b, const vd& c) :
        m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, vd(n+2)) {
        rep(i,0,m) rep(j,0,n) D[i][j] = A[i][j];
        rep(i,0,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i]; }
        rep(j,0,n) { N[j] = j; D[m][j] = -c[j]; }
        N[n] = -1; D[m+1][n] = 1;
    }
}
```

```
void pivot(int r, int s) {
    T *a = D[r].data(), inv = 1 / a[s];
    rep(i,0,m+2) if (i != r && abs(D[i][s]) > eps) {
        T *b = D[i].data(), inv2 = b[s] * inv;
        rep(j,0,n+2) b[j] -= a[j] * inv2;
        b[s] = a[s] * inv2;
    }
    rep(j,0,n+2) if (j != s) D[r][j] *= inv;
    rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
}

bool simplex(int phase) {
    int x = m + phase - 1;
    for (;) {
        int s = -1;
        rep(j,0,n+1) if (N[j] != -phase) ltj(D[x]);
        if (D[x][s] >= -eps) return true;
        int r = -1;
        rep(i,0,m) {
            if (D[i][s] <= eps) continue;
            if (r == -1 || MP(D[i][n+1] / D[i][s], B[i])
                < MP(D[r][n+1] / D[r][s], B[r])) r = i;
        }
        if (r == -1) return false;
        pivot(r, s);
    }
}

T solve(vd &x) {
    int r = 0;
    rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] < -eps) {
        pivot(r, n);
        if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
        rep(i,0,m) if (B[i] == -1) {
            int s = 0;
            rep(j,1,n+1) ltj(D[i]);
            pivot(i, s);
        }
    }
    bool ok = simplex(1); x = vd(n);
    rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return ok ? D[m][n+1] : inf;
}
};
```

## 7 Polynomial

### 7.1 FFT

```
template<int MAXN>
struct FFT {
    using val_t = complex<double>;
    const double PI = acos(-1);
    val_t w[MAXN];
    FFT() {
        for (int i = 0; i < MAXN; ++i) {
            double arg = 2 * PI * i / MAXN;
            w[i] = val_t(cos(arg), sin(arg));
        }
    }
    void bitrev(val_t *a, int n); // see NTT
    void trans(val_t *a, int n, bool inv = false); // see NTT
};
```

### 7.2 NTT

```
//to be modified
//(2^16)+1, 65537, 3
//7*17*(2^23)+1, 998244353, 3
//1255*(2^20)+1, 1315962881, 3
```

```
//51*(2^25)+1, 1711276033, 29
template<int MAXN, ll P, ll RT> //MAXN must be 2^k
struct NTT {
    ll w[MAXN];
    ll mpow(ll a, ll n);
    ll minv(ll a) { return mpow(a, P - 2); }
    NTT() {
        ll dw = mpow(RT, (P - 1) / MAXN);
        w[0] = 1;
        for (int i = 1; i < MAXN; ++i) w[i] = w[i - 1] * dw % P;
    }
    void bitrev(ll *a, int n) {
        int i = 0;
        for (int j = 1; j < n - 1; ++j) {
            for (int k = n >> 1; (i ^= k) < k; k >>= 1);
            if (j < i) swap(a[i], a[j]);
        }
    }
    void operator()(ll *a, int n, bool inv = false) { //0 <= a[i] < P
        bitrev(a, n);
        for (int L = 2; L <= n; L <= 1) {
            int dx = MAXN / L, dl = L >> 1;
            for (int i = 0; i < n; i += L) {
                for (int j = i, x = 0; j < i + dl; ++j, x += dx) {
                    ll tmp = a[j + dl] * w[x] % P;
                    if ((a[j + dl] = a[j] - tmp) < 0) a[j + dl] += P;
                    if ((a[j] += tmp) >= P) a[j] -= P;
                }
            }
        }
        if (inv) {
            reverse(a + 1, a + n);
            ll invn = minv(n);
            for (int i = 0; i < n; ++i) a[i] = a[i] * invn % P;
        }
    };
};
```

## 8 String

### 8.1 KMP Algorithm

```
void kmp(string s){
    int siz = s.size();
    vector<int> f(siz, 0);
    f[0] = 0;
    for (int i = 1; i < siz; ++i) {
        f[i] = f[i-1];
        bool zero = 0;
        while (s[f[i]] != s[i]) {
            if (f[i] == 0) {
                zero = 1;
                break;
            }
            f[i] = f[f[i]-1];
        }
        if (!zero) f[i]++;
    }
}
```

### 8.2 Manacher Algorithm

```
vector<int> manacher(string s) {
    int n = s.size();
    vector<int> v(n);
    int pnt = -1, len = 1;
    for (int i = 0; i < n; ++i) {
        int cor = 2 * pnt - i;
        if (cor >= 0) v[i] = min(v[cor], cor - pnt + len);
        while (i+v[i] < n && i-v[i] >= 0 && s[i+v[i]] == s[i-v[i]-1]) v[i]++;
        if (i + v[i] >= pnt + len) pnt = i, len = v[i];
    }
}
```

```
}
for (int i = 0; i < n; ++i) v[i] = 2 * v[i] - 1;
return v;
}
```

### 8.3 MCP

```
string mcp(string s) { //Duval algorithm for Lyndon factorization
    s += s;
    int n = s.size(), i = 0, ans = 0;
    while (i < n/2) {
        ans = i;
        int j = i+1, k=i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j]) k = i;
            else k++;
            j++;
        }
        while (i <= k) i += j - k;
    }
    return s.substr(ans, n/2);
}
```

### 8.4 Suffix Array

```
struct SuffixArray { //tested
    vector<int> sa, lcp, rank; //lcp[i] is lcp of sa[i] and sa[i-1]
    SuffixArray(string& s, int lim=256) { // or basic_string<int>
        int n = s.size() + 1, k = 0, a, b;
        vector<int> x(n, 0), y(n), ws(max(n, lim));
        rank.resize(n);
        for (int i = 0; i < n-1; ++i) x[i] = (int)s[i];
        sa = lcp = y, iota(sa.begin(), sa.end(), 0);
        for (int j = 0, p = 0; p < n; j = max(1, j * 2), lim = p) {
            p = j, iota(y.begin(), y.end(), n - j);
            for (int i = 0; i < n; ++i) if (sa[i] >= j) y[p++] = sa[i] - j;
            for (int &i : ws) i = 0;
            for (int i = 0; i < n; ++i) ws[x[i]]++;
            for (int i = 1; i < lim; ++i) 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;
            for (int i = 1; i < n; ++i) a = sa[i - 1], b = sa[i], x[b] = (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p++;
        }
        for (int i = 1; i < n; ++i) rank[sa[i]] = i;
        for (int i = 0, j; i < n - 1; lcp[rank[i+1]] = k)
            for (k && k--, j = sa[rank[i] - 1]; s[i + k] == s[j + k]; k++);
    }
};
```

### 8.5 Suffix Array Automaton

```
//to be modified
const int MAXM = 1000010;
struct SAM {
    int tot, root, lst, mom[MAXM], mx[MAXM];
    int acc[MAXM], nxt[MAXM][33];
    int newNode() {
        int res = ++tot;
        fill(nxt[res], nxt[res] + 33, 0);
        mom[res] = mx[res] = acc[res] = 0;
        return res;
    }
    void init() {
        tot = 0;
    }
};
```

```

root = newNode();
mom[root] = 0, mx[root] = 0;
lst = root;
}
void push(int c) {
    int p = lst;
    int np = newNode();
    mx[np] = mx[p] + 1;
    for (; p && nxt[p][c] == 0; p = mom[p])
        nxt[p][c] = np;
    if (p == 0) mom[np] = root;
    else {
        int q = nxt[p][c];
        if (mx[p] + 1 == mx[q]) mom[np] = q;
        else {
            int nq = newNode();
            mx[nq] = mx[p] + 1;
            for (int i = 0; i < 33; i++)
                nxt[nq][i] = nxt[q][i];
            mom[nq] = mom[q];
            mom[q] = nq;
            mom[np] = nq;
            for (; p && nxt[p][c] == q; p = mom[p])
                nxt[p][c] = nq;
        }
    }
    lst = np;
}
void push(char *str) {
    for (int i = 0; str[i]; i++)
        push(str[i] - 'a' + 1);
}
} sam;

```

## 8.6 Z-value Algorithm

```

int z[maxn];
void make_z(const string &s) {
    int l = 0, r = 0;
    for (int i = 1; i < SZ(s); ++i) {
        for (z[i] = max(0, min(r - i + 1, z[i - l]));
             i + z[i] < SZ(s) && s[i + z[i]] == s[z[i]];
             ++z[i])
            ;
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
}

```

## 9 Formula

### 9.1 Recurrences

If  $a_n = c_1 a_{n-1} + \dots + c_k a_{n-k}$ , and  $r_1, \dots, r_k$  are distinct roots of  $x^k + c_1 x^{k-1} + \dots + c_k$ , there are  $d_1, \dots, d_k$  s.t.

$$a_n = d_1 r_1^n + \dots + d_k r_k^n.$$

Non-distinct roots  $r$  become polynomial factors, e.g.  $a_n = (d_1 n + d_2) r^n$ .

### 9.2 Trigonometry

$$\sin(v+w) = \sin v \cos w + \cos v \sin w$$

$$\cos(v+w) = \cos v \cos w - \sin v \sin w$$

$$\tan(v+w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$

$$\sin v + \sin w = 2 \sin \frac{v+w}{2} \cos \frac{v-w}{2}$$

$$\cos v + \cos w = 2 \cos \frac{v+w}{2} \cos \frac{v-w}{2}$$

$$(V+W) \tan(v-w)/2 = (V-W) \tan(v+w)/2$$

where  $V, W$  are lengths of sides opposite angles  $v, w$ .

$$a \cos x + b \sin x = r \cos(x - \phi)$$

$$a \sin x + b \cos x = r \sin(x + \phi)$$

where  $r = \sqrt{a^2 + b^2}$ ,  $\phi = \text{atan2}(b, a)$ .

## 9.3 Geometry

### 9.3.1 Triangles

Side lengths:  $a, b, c$

$$\text{Semiperimeter: } p = \frac{a+b+c}{2}$$

$$\text{Area: } A = \sqrt{p(p-a)(p-b)(p-c)}$$

$$\text{Circumradius: } R = \frac{abc}{4A}$$

$$\text{Inradius: } r = \frac{A}{p}$$

Length of median (divides triangle into two equal-area triangles):  
 $m_a = \frac{1}{2} \sqrt{2b^2 + 2c^2 - a^2}$

$$\text{Length of bisector (divides angles in two): } s_a = \sqrt{bc \left( 1 - \left( \frac{a}{b+c} \right)^2 \right)}$$

$$\text{Law of sines: } \frac{\sin \alpha}{a} = \frac{\sin \beta}{b} = \frac{\sin \gamma}{c} = \frac{1}{2R}$$

$$\text{Law of cosines: } a^2 = b^2 + c^2 - 2bc \cos \alpha$$

$$\text{Law of tangents: } \frac{a+b}{a-b} = \frac{\tan \frac{\alpha+\beta}{2}}{\tan \frac{\alpha-\beta}{2}}$$

Incenter:

$$P_1 = (x_1, y_1), P_2 = (x_2, y_2), P_3 = (x_3, y_3)$$

$$s_1 = \frac{P_2 P_3}{2}, s_2 = \frac{P_1 P_3}{2}, s_3 = \frac{P_1 P_2}{2}$$

$$\frac{s_1 P_1 + s_2 P_2 + s_3 P_3}{s_1 + s_2 + s_3}$$

Circumcenter:

$$P_0 = (0, 0), P_1 = (x_1, y_1), P_2 = (x_2, y_2)$$

$$x_c = \frac{1}{2} \times \frac{y_2(x_1^2 + y_1^2) - y_1(x_2^2 + y_2^2)}{-x_2 y_1 + x_1 y_2}$$

$$y_c = \frac{1}{2} \times \frac{x_2(x_1^2 + y_1^2) - x_1(x_2^2 + y_2^2)}{-x_1 y_2 + x_2 y_1}$$

Check if  $(x_0, y_0)$  is in the circumcircle:

$$\begin{vmatrix} x_1 - x_0 & y_1 - y_0 & (x_1^2 + y_1^2) - (x_0^2 + y_0^2) \\ x_2 - x_0 & y_2 - y_0 & (x_2^2 + y_2^2) - (x_0^2 + y_0^2) \\ x_3 - x_0 & y_3 - y_0 & (x_3^2 + y_3^2) - (x_0^2 + y_0^2) \end{vmatrix}$$

0: on edge, > 0: inside, < 0: outside

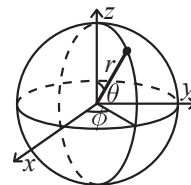
### 9.3.2 Quadrilaterals

With side lengths  $a, b, c, d$ , diagonals  $e, f$ , diagonals angle  $\theta$ , area  $A$  and magic flux  $F = b^2 + d^2 - a^2 - c^2$ :

$$4A = 2ef \cdot \sin \theta = F \tan \theta = \sqrt{4e^2 f^2 - F^2}$$

For cyclic quadrilaterals the sum of opposite angles is  $180^\circ$ ,  $ef = ac + bd$ , and  $A = \sqrt{(p-a)(p-b)(p-c)(p-d)}$ .

### 9.3.3 Spherical coordinates



$$\begin{aligned} x &= r \sin \theta \cos \phi & r &= \sqrt{x^2 + y^2 + z^2} \\ y &= r \sin \theta \sin \phi & \theta &= \arccos(z / \sqrt{x^2 + y^2 + z^2}) \\ z &= r \cos \theta & \phi &= \text{atan2}(y, x) \end{aligned}$$

## 9.4 Derivatives/Integrals

$$\begin{aligned}\frac{d}{dx} \arcsin x &= \frac{1}{\sqrt{1-x^2}} & \frac{d}{dx} \arccos x &= -\frac{1}{\sqrt{1-x^2}} \\ \frac{d}{dx} \tan x &= 1 + \tan^2 x & \frac{d}{dx} \arctan x &= \frac{1}{1+x^2} \\ \int \tan ax &= -\frac{\ln |\cos ax|}{a} & \int x \sin ax &= \frac{\sin ax - ax \cos ax}{a^2} \\ \int e^{-x^2} &= \frac{\sqrt{\pi}}{2} \operatorname{erf}(x) & \int x e^{ax} dx &= \frac{e^{ax}}{a^2} (ax - 1)\end{aligned}$$

Integration by parts:

$$\int_a^b f(x)g(x)dx = [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx$$

## 9.5 Sums

$$c^a + c^{a+1} + \dots + c^b = \frac{c^{b+1} - c^a}{c - 1}, c \neq 1$$

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

$$1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(2n+1)(n+1)}{6}$$

$$1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{n^2(n+1)^2}{4}$$

$$1^4 + 2^4 + 3^4 + \dots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

## 9.6 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots, (-\infty < x < \infty)$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots, (-1 < x \leq 1)$$

$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \dots, (-1 \leq x \leq 1)$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots, (-\infty < x < \infty)$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots, (-\infty < x < \infty)$$

## 9.7 Probability theory

Let  $X$  be a discrete random variable with probability  $p_X(x)$  of assuming the value  $x$ . It will then have an expected value (mean)  $\mu = \mathbb{E}(X) = \sum_x x p_X(x)$  and variance  $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)$  where  $\sigma$  is the standard deviation. If  $X$  is instead continuous it will have a probability density function  $f_X(x)$  and the sums above will instead be integrals with  $p_X(x)$  replaced by  $f_X(x)$ .

Expectation is linear:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

For independent  $X$  and  $Y$ ,

$$V(aX + bY) = a^2 V(X) + b^2 V(Y).$$

### 9.7.1 Discrete distributions

**Binomial distribution** The number of successes in  $n$  independent yes/no experiments, each which yields success with probability  $p$  is  $\text{Bin}(n, p)$ ,  $n = 1, 2, \dots$ ,  $0 \leq p \leq 1$ .

$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\mu = np, \sigma^2 = np(1-p)$$

$\text{Bin}(n, p)$  is approximately  $\text{Po}(np)$  for small  $p$ .

**First success distribution** The number of trials needed to get the first success in independent yes/no experiments, each which yields success with probability  $p$  is  $\text{Fs}(p)$ ,  $0 \leq p \leq 1$ .

$$p(k) = p(1-p)^{k-1}, k = 1, 2, \dots$$

$$\mu = \frac{1}{p}, \sigma^2 = \frac{1-p}{p^2}$$

**Poisson distribution** The number of events occurring in a fixed period of time  $t$  if these events occur with a known average rate  $\kappa$  and independently of the time since the last event is  $\text{Po}(\lambda)$ ,  $\lambda = t\kappa$ .

$$p(k) = e^{-\lambda} \frac{\lambda^k}{k!}, k = 0, 1, 2, \dots$$

$$\mu = \lambda, \sigma^2 = \lambda$$

### 9.7.2 Continuous distributions

**Uniform distribution** If the probability density function is constant between  $a$  and  $b$  and 0 elsewhere it is  $\text{U}(a, b)$ ,  $a < b$ .

$$f(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{a+b}{2}, \sigma^2 = \frac{(b-a)^2}{12}$$

**Exponential distribution** The time between events in a Poisson process is  $\text{Exp}(\lambda)$ ,  $\lambda > 0$ .

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$\mu = \frac{1}{\lambda}, \sigma^2 = \frac{1}{\lambda^2}$$

**Normal distribution** Most real random values with mean  $\mu$  and variance  $\sigma^2$  are well described by  $\mathcal{N}(\mu, \sigma^2)$ ,  $\sigma > 0$ .

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If  $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$  and  $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$  then

$$aX_1 + bX_2 + c \sim \mathcal{N}(\mu_1 + \mu_2 + c, a^2\sigma_1^2 + b^2\sigma_2^2)$$

## 9.8 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let  $X_1, X_2, \dots$  be a sequence of random variables generated by the Markov process. Then there is a transition matrix  $\mathbf{P} = (p_{ij})$ , with  $p_{ij} = \Pr(X_n = i | X_{n-1} = j)$ , and  $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$  is the probability distribution for  $X_n$  (i.e.,  $p_i^{(n)} = \Pr(X_n = i)$ ), where  $\mathbf{p}^{(0)}$  is the initial distribution.

$\pi$  is a stationary distribution if  $\pi = \pi \mathbf{P}$ . If the Markov chain is *irreducible* (it is possible to get to any state from any state), then  $\pi_i = \frac{1}{\mathbb{E}(T_i)}$  where  $\mathbb{E}(T_i)$  is the expected time between two visits in state  $i$ .  $\pi_j / \pi_i$  is the expected number of visits in state  $j$  between two visits in state  $i$ .

For a connected, undirected and non-bipartite graph, where the transition probability is uniform among all neighbors,  $\pi_i$  is proportional to node  $i$ 's degree.

A Markov chain is *ergodic* if the asymptotic distribution is independent of the initial distribution. A finite Markov chain is ergodic iff it is irreducible and *aperiodic* (i.e., the gcd of cycle lengths is 1).  $\lim_{k \rightarrow \infty} \mathbf{P}^k = \mathbf{1}\pi$ .

A Markov chain is an **A-chain** if the states can be partitioned into two sets **A** and **G**, such that all states in **A** are absorbing ( $p_{ii} = 1$ ), and all states in **G** leads to an absorbing state in **A**. The probability for absorption in state  $i \in \mathbf{A}$ , when the initial state is  $j$ , is  $a_{ij} = p_{ij} + \sum_{k \in \mathbf{G}} a_{ik} p_{kj}$ . The expected time until absorption, when the initial state is  $i$ , is  $t_i = 1 + \sum_{k \in \mathbf{G}} p_{ki} t_k$ .