Code Template for ACM-ICPC

wodesuck
SYSU_Braid
Sun Yat-sen University

January 1, 2016

Contents

Chapter 1

Graph/Tree Theory

1.1 Shortest Path

1.1.1 Dijkstra

```
void dijkstra(int s)
{
	typedef pair<int, int> T;
	priority_queue<T, vector<T>, greater<T> > h;
	memset(d, 0x3f, sizeof(d));
	memset(v, 0, sizeof(v));
	h.push(T(d[s] = 0, s));
	while (!h.empty()) {
		int w = h.top().first, u = h.top().second;
		h.pop();
		if (w > d[u]) continue;
		for (edge *i = e[u]; i; i = i->next) {
			int dis = d[u] + i->w;
			if (dis < d[i->t]) h.push(T(d[i->t] = dis, i->t));
		}
	}
}
```

1.1.2 SPFA

```
void spfa(int s)
{
	queue<int> q;
	memset(d, 0x3f, sizeof(d));
	memset(v, 0, sizeof(v));
	q.push(s); d[s] = 0; v[s] = true;
	while (!q.empty()) {
		int u = q.front(); q.pop(); v[u] = false;
		for (edge *i = e[u]; i; i = i->next) {
			if (d[u] + i->w < d[i->t]) {
				d[i->t] = d[u] + i->w;
				if (!v[i->t]) {
					q.push(i->t);
					v[i->t] = true;
				}
			}
		}
	}
}
```

1.1.3 Minimum-weight Cycle(Folyd)

```
// for undirected graph
const int INF = 0x2a2a2a2a;

int folyd()
{
	int ans = INF;
	for (int k = 0; k < n; ++k) {
		for (int i = 0; i < k; ++i) {
			for (int j = 0; j < i; ++j) {
				ans = min(ans, f[i][j] + g[j][k] + g[k][i]);
			}
		}
		for (int i = 0; i < n; ++i) {
			for (int j = 0; j < n; ++j) {
				f[i][j] = min(f[i][j], f[i][k] + f[k][j]);
			}
		}
	}
	return ans;
}

/*
Initialize:
	memset(g, 0x2a, sizeof(g));
	memset(f, 0x2a, sizeof(f));
*/
```

1.2 Bridge/Cutvertex-Finding(Tarjan)

```
void tarjan(int u, int fa)
{
	dfn[u] = low[u] = ++stamp;
	int ch = 0;
	for (edge *i = e[u]; i; i = i->next) {
		int v = i->t;
		if (!dfn[v]) {
			tarjan(v, u);
			low[u] = min(low[u], low[v]);
			if (u ? low[v] >= dfn[u] : ++ch > 1) cut[u] = true;
			if (low[v] > dfn[u]) bridge[u][v] = true;
		} else if (v != fa) {
			low[u] = min(low[u], dfn[v]);
		}
	}
}
```

1.3 Strongly Connected Components(Tarjan)

```
void tarjan(int u)
{
	dfn[u] = low[u] = ++stamp;
	sta[top++] = u; ins[u] = true;
	for (edge *i = e[u]; i; i = i->next) {
		int v = i->t;
		if (!dfn[v]) {
			tarjan(v);
			low[u] = min(low[u], low[v]);
		} else if (ins[v]) {
			low[u] = min(low[u], dfn[v]);
		}
```

```
		}
		if (dfn[u] == low[u]) {
			int v;
			do {
				v = sta[--top];
				ins[v] = false;
				scc[v] = cnt;
			} while (v != u);
			++cnt;
		}
}
```

1.4 Lowest Common Ancestor(Tarjan)

```
void tarjan(int u)
{
		anc[u] = u; v[u] = 1;
		for (edge *i = e[u]; i; i = i->next) {
			if (!v[i->t]) {
				tarjan(i->t);
				join(u, i->t);
				anc[find(u)] = u;
			}
		}
		v[u] = 2;
		for (quest *i = q[u]; i; i = i->next) {
			if (v[i->t] == 2) lca[i->id] = anc[find(i->t)];
		}
}
```

1.5 Network Flow

1.5.1 Maximum Flow(Improved-SAP)

```
struct edge {
		int t, u;
		edge *next, *pair;
}epool[MAXE * 2], *e[MAXV];
int esz, psz, s, t;
int h[MAXV], vh[MAXV + 1];

void addedge(int u, int v, int c)
{
		edge *e1 = epool + psz++, *e2 = epool + psz++;
		*e1 = (edge){v, c, e[u], e2}, e[u] = e1;
		*e2 = (edge){u, 0, e[v], e1}, e[v] = e2;
}

int aug(int u, int m)
{
		if (u == t) return m;
		int d = m;
		for (edge *i = e[u]; i; i = i->next) {
			if (i->u && h[u] == h[i->t] + 1) {
				int f = aug(i->t, min(i->u, d));
				i->u -= f, i->pair->u += f, d -= f;
				if (h[s] == esz || !d) return m - d;
			}
		}
		int w = d < m ? min(esz, h[u] + 2) : esz;
```

```
        for (edge *i = e[u]; i; i = i->next) {
                if (i->u) w = min(w, h[i->t] + 1);
        }
        ++vh[w];
        --vh[h[u]] ? h[u] = w : h[s] = esz;
        return m - d;
}

void maxflow()
{
        flow = 0;
        memset(h, 0, sizeof(h));
        memset(vh, 0, sizeof(vh));
        vh[0] = esz;
        while (h[s] != esz) flow += aug(s, INT_MAX);
}
```

1.5.2 Minimum Cost Maximum Flow(Primal-Dual)

```
struct edge {
        int t, u, c;
        edge *next, *pair;
}epool[MAXE * 2], *e[MAXV];
int psz, s, t;
int cost, dist, d[MAXV];
bool vis[MAXV];

void addedge(int u, int v, int c, int w)
{
        edge *e1 = epool + psz++, *e2 = epool + psz++;
        *e1 = (edge){v, c,  w, e[u], e2}, e[u] = e1;
        *e2 = (edge){u, 0, -w, e[v], e1}, e[v] = e2;
}

int aug(int u, int m)
{
        if (u == t) return cost += dist * m, m;
        int d = m; vis[u] = true;
        for (edge *i = e[u]; i; i = i->next) {
                if (i->u && !i->c && !vis[i->t]) {
                        int f = aug(i->t, min(d, i->u));
                        i->u -= f, i->pair->u +=f, d -= f;
                        if (!d) return m;
                }
        }
        return m - d;
}

bool modlabel()
{
        deque<int> q;
        memset(vis, 0, sizeof(vis));
        memset(d, 0x3f, sizeof(d));
        q.push_back(s); d[s] = 0; vis[s] = true;
        while (!q.empty()) {
                int u = q.front(); q.pop_front(); vis[u] = false;
                for (edge *i = e[u]; i; i = i->next) {
                        int v = i->t;
                        if (i->u && d[u] + i->c < d[v]) {
                                d[v] = d[u] + i->c;
                                if (vis[v]) continue;
```

```
                                vis[v] = true;
                                if (q.size() && d[v] < d[q[0]]) q.push_front(v);
                                else q.push_back(v);
                        }
                }
        }
        for (edge *i = epool; i < epool + psz; ++i) {
                i->c -= d[i->t] - d[i->pair->t];
        }
        dist += d[t];
        return d[t] < inf;
}

void costflow()
{
        cost = dist = 0;
        while (modlabel()) {
                do memset(vis, 0, sizeof(vis));
                while (aug(s, INT_MAX));
        }
}
```

1.5.3 Minimum Cost Maximum Flow(Cycle Canceling)

```
struct edge {
        int t, u, c;
        edge *next, *pair;
}epool[MAXE * 2], *e[MAXV];
int psz, s, t;
int d[MAXV];
bool vis[MAXV];
edge *fa[MAXV];

void addedge(int u, int v, int c, int w)
{
        edge *e1 = epool + psz++, *e2 = epool + psz++;
        *e1 = (edge){v, c,  w, e[u], e2}, e[u] = e1;
        *e2 = (edge){u, 0, -w, e[v], e1}, e[v] = e2;
}

void cancelcycle(int u)
{
        int i = u;
        do {
                --fa[i]->u, ++fa[i]->pair->u, cost += fa[i]->c;
                i = fa[i]->pair->t;
        } while (i != u);
}

bool aug(int u)
{
        vis[u] = true;
        for (edge *i = e[u]; i; i = i->next) {
                int v = i->t;
                if (i->u && d[u] + i->c < d[v]) {
                        d[v] = d[u] + i->c;
                        fa[v] = i;
                        if (vis[v]) cancelcycle(v);
                        if (vis[v] || aug(v)) return true;
                }
        }
```

```
        vis[u] = false;
        return false;
}

void costflow()
{
        cost = 0;
        for (;;) {
                memset(d, 0, sizeof(d));
                memset(vis, 0, sizeof(vis));
                bool flag = false;
                for (int i = 0; i < esz; ++i) {
                        if (aug(i)) { flag = true; break; }
                }
                if (!flag) return;
        }
}

/*
Initialize:
        addedge(t, s, inf, -inf);
        CAUTION: maybe OVERFLOW
*/
```

1.6 Matching

1.6.1 Maximum Bipartite Matching(Hungarian)

```
int n, m;
bool g[MAXN][MAXM];
int match[MAXM];
bool v[MAXN];

bool dfs(int i)
{
        for (int j = 0; j < m; ++j) {
                if (g[i][j] && !v[j]) {
                        v[j] = true;
                        if (match[j] < 0 || dfs(match[j])) {
                                match[j] = i;
                                return true;
                        }
                }
        }
        return false;
}

int hungarian()
{
        int c = 0;
        memset(match, -1, sizeof(match));
        for (int i = 0; i < n; ++i) {
                memset(v, 0, sizeof(v));
                if (dfs(i)) ++c;
        }
        return c;
}
```

1.6.2 Maximum Weight Perfect Biparite Matching(KM)

```
int n;
```

```
int w[MAXN][MAXN];
int lx[MAXN], ly[MAXN], match[MAXN], slack[MAXN];
bool vx[MAXN], vy[MAXN];

bool dfs(int i)
{
	vx[i] = true;
	for (int j = 0; j < n; ++j) {
		if (lx[i] + ly[j] > w[i][j]) {
			slack[j] = min(slack[j], lx[i] + ly[j] - w[i][j]);
		} else if (!vy[j]) {
			vy[j] = true;
			if (match[j] < 0 || dfs(match[j])) {
				match[j] = i;
				return true;
			}
		}
	}
	return false;
}

void km()
{
	memset(match, -1, sizeof(match));
	memset(ly, 0, sizeof(ly));
	for (int i = 0; i < n; ++i) lx[i] = *max_element(w[i], w[i] + n);
	for (int i = 0; i < n; ++i) {
		for (;;) {
			memset(vx, 0, sizeof(vx));
			memset(vy, 0, sizeof(vy));
			memset(slack, 0x3f, sizeof(slack));
			if (dfs(i)) break;
			int d = inf;
			for (int i = 0; i < n; ++i) {
				if (!vy[i]) d = min(d, slack[i]);
			}
			for (int i = 0; i < n; ++i) {
				if (vx[i]) lx[i] -= d;
				if (vy[i]) ly[i] += d;
			}
		}
	}
}
```

1.6.3 Maximum Matching on General Graph(Blossom Algorithm)

```
int n;
int next[MAXN], match[MAXN], v[MAXN], f[MAXN];
int que[MAXN], head, tail;

int find(int p)
{
	return f[p] < 0 ? p : f[p] = find(f[p]);
}

void join(int x, int y)
{
	x = find(x); y = find(y);
	if (x != y) f[x] = y;
}

```

```
int lca(int x, int y)
{
	static int v[MAXN], stamp = 0;
	++stamp;
	for (;;) {
		if (x >= 0) {
			x = find(x);
			if (v[x] == stamp) return x;
			v[x] = stamp;
			if (match[x] >= 0) x = next[match[x]];
			else x = -1;
		}
		swap(x, y);
	}
}

void group(int a, int p)
{
	while (a != p) {
		int b = match[a], c = next[b];
		if (find(c) != p) next[c] = b;
		if (v[b] == 2) v[que[tail++] = b] = 1;
		if (v[c] == 2) v[que[tail++] = c] = 1;
		join(a, b); join(b, c);
		a = c;
	}
}

void aug(int s)
{
	memset(v, 0, sizeof(v));
	memset(f, -1, sizeof(f));
	memset(next, -1, sizeof(next));
	que[0] = s; head = 0; tail = 1; v[s] = 1;
	while (head < tail && match[s] < 0) {
		int x = que[head++];
		for (edge *i = e[x]; i; i = i->next) {
			int y = i->t;
			if (match[x] == y || v[y] == 2 || find(x) == find(y)) {
				continue;
			} else if (v[y] == 1) {
				int p = lca(x, y);
				if (find(x) != p) next[x] = y;
				if (find(y) != p) next[y] = x;
				group(x, p);
				group(y, p);
			} else if (match[y] < 0) {
				next[y] = x;
				while (~y) {
					int z = next[y];
					int p = match[z];
					match[y] = z; match[z] = y;
					y = p;
				}
				break;
			} else {
				next[y] = x;
				v[que[tail++] = match[y]] = 1;
				v[y] = 2;
			}
		}
```

```
        }
}

void blossom()
{
        memset(match, -1, sizeof(match));
        for (int i = 0; i < n; ++i) {
                if (match[i] < 0) aug(i);
        }
}
```

1.6.4 Maximum Weight Perfect Matching on General Graph(Randomize Greedy Matching)

```
int n;
int w[MAXN][MAXN];
int match[MAXN], p[MAXN], d[MAXN];
int path[MAXN], len;
bool v[MAXN];
const int inf = 0x3f3f3f3f;

bool dfs(int i)
{
        path[len++] = i;
        if (v[i]) return true;
        v[i] = true;
        for (int j = 0; j < n; ++j) {
                if (i != j && match[i] != j && !v[j]) {
                        int k = match[j];
                        if (d[k] < d[i] + w[i][j] - w[j][k]) {
                                d[k] = d[i] + w[i][j] - w[j][k];
                                if (dfs(k)) return true;
                        }
                }
        }
        --len;
        v[i] = false;
        return false;
}

int matching()
{
        for (int i = 0; i < n; ++i) p[i] = i, match[i] = i^1;
        int cnt = 0;
        for (;;) {
                len = 0;
                bool flag = false;
                memset(d, 0, sizeof(d));
                memset(v, 0, sizeof(v));
                for (int i = 0; i < n; ++i) {
                        if (dfs(p[i])) {
                                flag = true;
                                int t = match[path[len - 1]], j = len - 2;
                                while (path[j] != path[len - 1]) {
                                        match[t] = path[j];
                                        swap(t, match[path[j]]);
                                        --j;
                                }
                                match[t] = path[j];
                                match[path[j]] = t;
                                break;
```

```
                        }
                }
                if (!flag) {
                        if (++cnt >= 3) break;
                        random_shuffle(p, p + n);
                }
        }
}
```

1.7 2-SAT

```
// n vars
// sat variable Bi and !Bi are encoded as i<<1 and i<<1^1
// add edge (i<<1 -> j<<1^1) for Bi -> !Bj
// Bi is true if scc[i<<1] < scc[i<<1^1]

bool twosat()
{
        cnt = stamp = 0;
        memset(dfn, 0, sizeof(dfn));
        for (int i = 0; i < n<<1; ++i) if (!dfn[i]) tarjan(i);
        for (int i = 0; i < n; ++i) {
                if (scc[i<<1] == scc[i<<1^1]) return false;
                ans[i] = scc[i<<1] < scc[i<<1^1];
        }
        return true;
}
```

1.8 Centroid of a Tree

```
int getsize(int u, int fa)
{
        size[u] = 1;
        for (edge *i = e[u]; i; i = i->next) {
                if (i->t != fa) size[u] += getsize(i->t, u);
        }
        return size[u];
}

int divide(int u)
{
        for (edge *i = e[u]; i; i = i->next) {
                if (size[i->t] > size[u] / 2) {
                        size[u] -= size[i->t], size[i->t] += size[u];
                        return divide(i->t);
                }
        }
        return u;
}

void solve(int u) // Divide and Conquer for Tree
{
        u = divide(u);
        size[u] = 0; // delete
        for (edge *i = e[u]; i; i = i->next) {
                if (size[i->t]) {
                        dfs1(i->t, u); // calculate answer
                        dfs2(i->t, u); // update
                }
        }
```

```
        // calculate answer with root
        for (edge *i = e[u]; i; i = i->next) {
                if (size[i->t]) solve(i->t);
        }
}
```

1.9 Heavy-Light Decomposition

```
int fa[MAXN], dep[MAXN], size[MAXN], hson[MAXN], top[MAXN];
int dfn[MAXN], stamp;

void dfs1(int u)
{
        size[u] = 1, hson[u] = 0;
        for (edge *i = e[p]; i; i = i->next) {
                int v = i->t;
                if (v == fa[u]) continue;
                fa[v] = u;
                dep[v] = dep[u] + 1;
                dfs1(v);
                size[u] += size[v];
                if (!hson[u] || size[v] > size[hson[u]]) hson[u] = v;
        }
}

void dfs2(int u, int anc)
{
        dfn[u] = stamp++;
        top[u] = anc;
        if (hson[u]) dfs2(hson[u], anc);
        for (edge *i = e[p]; i; i = i->next) {
                int v = i->t;
                if (v != fa[u] && v != hson[u]) dfs2(v, v);
        }
}

int lca(int u, int v)
{
        while (top[u] != top[v]) {
                if (dep[top[u]] < dep[top[v]]) swap(u, v);
                // query(dfn[top[u]], dfn[u])
                u = fa[top[u]];
        }
        if (dep[u] > dep[v]) swap(u, v);
        // query(dfn[u], dfn[v]) -- include LCA
        // if (u != v) query(dfn[u] + 1, dfn[v]) -- exclude LCA
        return u;
}
```

1.10 Virtual Tree

```
bool cmp(int i, int j) { return dfn[i] < dfn[j]; }

int vtree(int h[], int m, int T[], int fa[])
{
        static int sta[MAXN];
        int tot = 0, top = 0;
        sort(h, h + m, cmp);
        sta[top++] = 0; // d[0] == -1
        for (int i = 0; i < m; ++i) {
```

```
            if (top <= 1) {
                sta[top++] = h[i];
                fa[h[i]] = 0;
            } else {
                int g = lca(h[i], sta[top - 1]);
                while (d[sta[top - 1]] > d[g]) {
                    --top;
                    if (d[sta[top - 1]] <= d[g]) fa[sta[top]] = g;
                }
                if (sta[top - 1] != g) {
                    T[tot++] = g;
                    fa[g] = sta[top - 1];
                    sta[top++] = g;
                }
                fa[h[i]] = g;
                sta[top++] = h[i];
            }
            T[tot++] = h[i];
        }
        sort(T, T + tot, cmp);
        return tot;
}

// return the number of nodes in virtual tree
// T[] -- nodes, fa[] -- father in virtual tree
```

Chapter 2

Data Structures

2.1 Segment Tree

2.1.1 Segment Tree(Non-recursive Implement)

```
int n, h;
S T[MAXN * 2]; // val
int d[MAXN * 2]; // lazy flag

void push(int p)
{
        for (int s = h, k = 1<<(h-1); s; --s, k >>= 1) {
                int i = p >> s;
                if (d[i]) {
                        apply(i<<1,   k, d[i]);
                        apply(i<<1|1, k, d[i]);
                        d[i] = 0;
                }
        }
}

void build()
{
        for (int i = n - 1; i; --i) update(i);
}

S query(int l, int r)
{
        S L, R;
        push(l += n), push(r += n);
        for (; l <= r; l >>= 1, r >>= 1) {
                if ( l&1) L = merge(L, T[l++]);
                if (~r&1) R = merge(T[r--], R);
        }
        return merge(L, R);
}

void modify(int l, int r, int x)
{
        bool cl = false, cr = false;
        push(l += n), push(r += n);
        for (int k = 1; l <= r; l >>= 1, r >>= 1, k <<= 1) {
                if (cl) update(l - 1);
                if (cr) update(r + 1);
                if ( l&1) apply(l++, k, x), cl = true;
                if (~r&1) apply(r--, k, x), cr = true;
```

```
        }
        for (--l, ++r; r; l >>= 1, r >>= 1) {
                if (cl) update(l);
                if (cr && (!cl || l != r)) update(r);
        }
}

// h = sizeof(int) * 8 - __builtin_clz(n);
```

2.1.2 Functional Segment Tree

```
struct sgt {
        int sum;
        sgt *left, *right;
}tpool[PSZ];
int tpsz;

sgt *new_node(int sum)
{
        sgt *p = tpool + tpsz++;
        p->sum = sum;
        p->left = p->right = 0;
        return p;
}

sgt *merge(sgt *l, sgt *r)
{
        sgt *p = tpool + tpsz++;
        p->sum = l->sum + r->sum;
        p->left = l; p->right = r;
        return p;
}

sgt *build(int l, int r)
{
        if (l == r) return new_node(0);
        int mid = (l + r) >> 1;
        return merge(build(l, mid), build(mid + 1, r));
}

sgt *add(sgt *p, int l, int r, int x)
{
        if (l == r) return new_node(p->sum + 1);
        int mid = (l + r) >> 1;
        return x <= mid ? merge(add(p->left, l, mid, x), p->right)
                        : merge(p->left, add(p->right, mid + 1, r, x));
}

int kth(sgt *a, sgt *b, int l, int r, int k)
{
        if (l == r) return l;
        int mid = (l + r) >> 1;
        int lsum = a->left->sum - b->left->sum;
        return k <= lsum ? kth(a->left, b->left, l, mid, k)
                         : kth(a->right, b->right, mid + 1, l, k - lsum);
}
```

2.2 Self-balancing BST

2.2.1 Size Balanced Tree

```
struct sbt {
        int k, sz;
        sbt *ch[2];
}pool[MAXN], *null;
int psz;

sbt *new_sbt(int v)
{
        sbt *t = pool + psz++;
        t->k = v; t->sz = 1;
        t->ch[0] = t->ch[1] = null;
        return t;
}

void rot(sbt *&t, int i)
{
        sbt *k = t->ch[i^1];
        t->ch[i^1] = k->ch[i]; k->ch[i] = t;
        k->sz = t->sz; t->sz = t->ch[0]->sz + t->ch[1]->sz + 1;
        t = k;
}

void maintain(sbt *&t, int i)
{
        if (t->ch[i]->ch[i]->sz > t->ch[i^1]->sz) {
                rot(t, i^1);
        } else if (t->ch[i]->ch[i^1]->sz > t->ch[i^1]->sz) {
                rot(t->ch[i], i), rot(t, i^1);
        } else return;
        maintain(t->ch[0], 0);
        maintain(t->ch[1], 1);
        maintain(t, 0);
        maintain(t, 1);
}

void insert(sbt *&t, int v)
{
        if (t == null) { t = new_sbt(v); return; }
        ++t->sz;
        insert(t->ch[v > t->k], v);
        maintain(t, v > t->k);
}

int erase(sbt *&t, int v)
{
        --t->sz;
        if (v == t->k || t->ch[v > t->k] == null) {
                v = t->k;
                if (t->ch[0] == null) t = t->ch[1];
                else if (t->ch[1] == null) t = t->ch[0];
                else t->k = erase(t->ch[0], v + 1);
                return v;
        }
        return erase(t->ch[v > t->k], v);
}

sbt *find(sbt *t, int v)
{
        if (t == null) return 0;
        if (v == t->k) return t;
        return find(t->ch[v > t->k], v);
```

```
}

int rank(sbt *t, int v)
{
        if (t == null) return 0;
        else if (v < t->k) return rank(t->ch[0], v);
        else return t->ch[0]->sz + 1 + rank(t->ch[1], v);
}

sbt *select(int t, int k)
{
        if (k == t->ch[0]->sz + 1) return t;
        else if (k <= t->ch[0]->sz) return select(t->ch[0], k);
        else return select(t->ch[1], k - t->ch[0]->sz - 1);
}

void init()
{
        psz = 0;
        null = new_sbt(0);
        null->sz = 0, null->ch[0] = null->ch[1] = null;
}
```

2.2.2 Treap

```
struct treap {
        treap *l, *r;
        int w, size;
        void update() { size = l->size + r->size + 1; }
        void sink() {}
}node[MAXN], *null;

void split(treap *t, int k, treap *&l, treap *&r) // split first k elements
{
        if (k == 0) { l = null, r = t; return; }
        t->sink();
        if (k <= t->l->size) {
                split(t->l, k, l, r);
                t->l = r, r = t;
        } else {
                split(t->r, k - t->l->size - 1, l, r);
                t->r = l, l = t;
        }
        t->update();
}

treap *merge(treap *l, treap *r)
{
        if (l == null) return r;
        if (r == null) return l;
        l->sink(), r->sink();
        treap *t;
        if (l->w < r->w) {
                t = l, l->r = merge(l->r, r);
        } else {
                t = r, r->l = merge(l, r->l);
        }
        t->update();
        return t;
}
```

2.2.3 Splay

```
struct node_t *null, *root;
struct node_t {
	node_t *ch[2], *fa;
	int size;

	int dir() { return fa->ch[0] == this ? 0 : 1; }
	void setc(node_t *c, int d) { ch[d] = c; if (c != null) c->fa = this; }
	void update() { size = ch[0]->size + ch[1]->size + 1; }
	void sink() {}

	void rot()
	{
		node_t *p = fa;
		int d = dir();
		if (p->fa == null) fa = null, root = this;
		else p->fa->setc(this, p->dir());
		p->setc(ch[d^1], d), setc(p, d^1);
		p->update(), update();
	}

	void splay(node_t *header = null)
	{
		for (; fa != header; rot()) {
			if (fa->fa != header) {
				if (dir() == fa->dir()) fa->rot();
				else rot();
			}
		}
	}

	node_t *select(int k)
	{
		node_t *t = this;
		while (t->sink(), k != t->ch[0]->size + 1) {
			if (k <= t->ch[0]->size) t = t->ch[0];
			else k -= t->ch[0]->size + 1, t = t->ch[1];
		}
		t->splay(fa);
		return t;
	}

	node_t *select(int l, int r)
	{
		return select(r + 1)->ch[0]->select(l - 1)->ch[1];
	}
}node[MAXN];
```

2.2.4 Functional Treap

```
struct node {
	int k, w; // key, weight
	node *l, *r;
}pool[PSZ];
int psz;

node *new_node(int key, int weight, node *left, node *right)
{
	node *t = pool + psz++;
	t->k = key; t->w = weight; t->l = left; t->r = right;
```

```
        return t;
}

node *split_l(node *t, int key)
{
        return !t ? 0 : (key < t->k ? split_l(t->l, key) :
                new_node(t->k, t->w, t->l, split_l(t->r, key)));
}

node *split_r(node *t, int key)
{
        return !t ? 0 : (key >= t->k ? split_r(t->r, key) :
                new_node(t->k, t->w, split_r(t->l, key), t->r));
}

node *merge(node *a, node *b)
{
        return (!a || !b) ? (a ? a : b) : (a->w < b->w ?
                new_node(a->k, a->w, a->l, merge(a->r, b)) :
                new_node(b->k, b->w, merge(a, b->l), b->r));
}

node *insert(node *t, int key)
{
        return merge(merge(split_l(t, key), new_node(key, rand(), 0, 0)),
                        split_r(t, key));
}
```

2.2.5 Functional Treap(Range Operation)

```
struct node {
        int v, w, sz; // value, weight, size
        node *l, *r;
}pool[PSZ];
int psz;

inline int sz(node *t) { return t ? t->sz : 0; }

node *new_node(int val, int weight, node *left, node *right)
{
        node *t = pool + psz++;
        t->v = val; t->w = weight; t->l = left; t->r = right;
        t->sz = sz(left) + sz(right) + 1;
        return t;
}

node *split_l(node *t, int k) // get the first k elements
{
        return !t ? 0 :
                 (k <= sz(t->l) ? split_l(t->l, k) :
                  new_node(t->v, t->w, t->l, split_l(t->r, k - sz(t->l) - 1)));
}

node *split_r(node *t, int k)
{
        return !t ? 0 :
                 (k > sz(t->l) ? split_r(t->r, k - sz(t->l) - 1) :
                  new_node(t->v, t->w, split_r(t->l, k), t->r));
}

node *merge(node *a, node *b)
```

```
{
        return (!a || !b) ? (a ? a : b) :
                (a->w < b->w ?
                 new_node(a->v, a->w, a->l, merge(a->r, b)) :
                 new_node(b->v, b->w, merge(a, b->l), b->r));
}

node *insert(node *t, int pos, int *val, int n) // insert before pos
{
        node *l = split_l(t, pos), *r = split_r(t, pos);
        for (int i = 0; i < n; ++i) {
                l = merge(l, new_node(val[i], rand(), 0, 0));
        }
        return merge(l, r);
}

node *fetch(node *t, int l, int r) // fetch [l, r]
{
        return split_l(split_r(t, l), r - l + 1);
}

// index from 0
```

2.3 Leftist Tree

```
int n;
int key[MAXN], left[MAXN], right[MAXN], dist[MAXN];

int merge(int a, int b)
{
        if (!a) return b;
        if (!b) return a;
        if (key[b] > key[a]) swap(a, b);
        right[a] = merge(right[a], b);
        if (dist[left[a]] < dist[right[a]]) swap(left[a], right[a]);
        dist[a] = dist[right[a]] + 1;
        return a;
}

/*
Initialize:
        memset(left, 0, sizeof(left));
        memset(right, 0, sizeof(left));
        dist[0] = -1;
*/
```

2.4 Dynamic Tree

2.4.1 Link-cut Tree

```
struct node_t {
        node_t *ch[2], *fa;
        int val, mx;
        bool rev;

        bool isroot() { return !fa || (fa->ch[0] != this && fa->ch[1] != this); }
        int dir() { return fa->ch[0] == this ? 0 : 1; }
        void setc(node_t *c, int d) { ch[d] = c; if (c) c->fa = this; }
        void reverse() { rev ^= 1; swap(ch[0], ch[1]); }

```

```
	void init(int v)
	{
		ch[0] = ch[1] = fa = 0;
		rev = false;
		val = mx = v;
	}

	void update()
	{
		mx = val;
		if (ch[0]) mx = max(mx, ch[0]->mx);
		if (ch[1]) mx = max(mx, ch[1]->mx);
	}

	void sink()
	{
		if (rev) {
			if (ch[0]) ch[0]->reverse();
			if (ch[1]) ch[1]->reverse();
			rev = 0;
		}
	}

	void rot()
	{
		node_t *p = fa;
		int d = dir();
		if (p->isroot()) fa = p->fa;
		else p->fa->setc(this, p->dir());
		p->setc(ch[d^1], d), setc(p, d^1);
		p->update(), update();
	}

	void sinkdown()
	{
		if (!isroot()) fa->sinkdown();
		sink();
	}

	void splay()
	{
		sinkdown();
		for (; !isroot(); rot()) {
			if (!fa->isroot()) {
				if (dir() == fa->dir()) fa->rot();
				else rot();
			}
		}
	}

	node_t *expose()
	{
		node_t *u = 0, *t = this;
		for (; t; u = t, t = t->fa) {
			t->splay();
			t->ch[1] = u;
			t->update();
		}
		return u;
	}

```

```
        node_t *root()
        {
                node_t *t = expose();
                while (t->sink(), t->ch[0]) t = t->ch[0];
                return t;
        }

        void setroot()
        {
                expose()->reverse();
        }

        void link(node_t *p)
        {
                setroot(); // have no nedd for rooted tree
                expose()->fa = p;
        }

        void cut(node_t *p)
        {
                p->setroot(); // have no nedd for rooted tree
                expose();
                splay();
                ch[0] = ch[0]->fa = 0;
                update();
        }

        int query(node_t *p)
        {
                p->setroot();
                return expose()->mx;
        }

        int query(node_t *t) // without setroot
        {
                expose();
                t = t->expose(); // lca
                int ret = t->val; // analysis lca
                if (t->ch[1]) ret = max(ret, t->ch[1]->mx); // lca -> v
                if (t != this) {
                        splay();
                        ret = max(ret, mx); // lca -> u
                }
                return ret;
        }
}node[MAXN];
```

2.4.2 Euler Tour Tree

```
struct node_t {
        // splay tree ...

        node_t *walkdown(int d)
        {
                node_t *t = this;
                while (t->ch[d] != null) t = t->ch[d];
                return t;
        }

        node_t *adj(int d) // 0 -- prev, 1 -- succ
        {
```

```
        if (ch[d] != null) return ch[d]->walkdown(d^1);
        node_t *t = this;
        while (t->dir() == d) t = t->fa;
        return t->fa;
    }
}node[MAXN * 2]; // each node split into 2 nodes. i --> (i<<1) && (i<<1)^1

void cut(int t)
{
    node_t *x = node[t<<1].adj(0), *y = node[t<<1^1].adj(1);
    x->splay(), y->splay(x);
    y->ch[0]->fa = null, y->setc(null, 0);
    y->update(), x->update();
}

void link(int t, int p) // link subtree t to p
{
    node_t *x = &node[p<<1], *y = &node[t<<1];
    x->splay(), x->adj(1)->splay(x), y->splay();
    x->ch[1]->setc(y, 0);
    x->ch[1]->update(), x->update();
}
```

2.4.3 Top Tree

```
struct info_t {
    int min, max, sum, size;

    info_t(int min = inf, int max = -inf, int sum = 0, int size = 0)
        : min(min), max(max), sum(sum), size(size) {}

    info_t& operator+=(const info_t& a)
    {
        if(a.size == 0) return *this;
        if(size == 0) return *this = a;
        min = std::min(min, a.min);
        max = std::max(max, a.max);
        sum = sum + a.sum;
        size = size + a.size;
        return *this;
    }
};

struct flag_t {
    int mul, add;

    flag_t(int mul = 1, int add = 0)
        : mul(mul), add(add) {}

    flag_t operator+=(const flag_t& a)
    {
        mul = mul * a.mul;
        add = add * a.mul + a.add;
        return *this;
    }

    bool empty() const
    {
        return mul == 1 && add == 0;
    }
};
```

```
info_t operator+(const info_t& a, const flag_t& b)
{
        if(!a.size) return a;

        info_t z;
        z.max = a.max * b.mul + b.add;
        z.min = a.min * b.mul + b.add;
        z.sum = a.sum * b.mul + b.add * a.size;
        z.size = a.size;
        return z;
}

struct node_t;
node_t* newnode();
void delnode(node_t *n);
struct node_t
{
        int val;
        node_t *fa, *ch[4];
        info_t chain, tree, all;
        flag_t chain_flag, tree_flag;
        bool is_rev, is_inner;

        bool is_root(int z) const
        {
                if(z == 0)
                {
                        if(!fa) return true;
                        return fa->ch[0] != this && fa->ch[1] != this;
                } else {
                        return !fa || !fa->is_inner || !is_inner;
                }
        }

        void setc(node_t *c, int z)
        {
                ch[z] = c;
                if(c) c->fa = this;
        }

        int where() const
        {
                for(int i = 0; i != 4; ++i)
                        if(fa->ch[i] == this) return i;
                return -1;  // throw 0;
        }

        int where(int z) const
        {
                return fa->ch[z] != this;
        }

        node_t* son(int i)
        {
                if(ch[i]) ch[i]->pushdown();
                return ch[i];
        }

        void pushup()
        {
```

```
			chain = tree = all = info_t();
			if(!is_inner) chain = all = info_t(val, val, val, 1);
			for(int i = 0; i != 4; ++i) if(ch[i]) all += ch[i]->all;
			for(int i = 2; i != 4; ++i) if(ch[i]) tree += ch[i]->all;
			for(int i = 0; i != 2; ++i) {
				if(ch[i]) {
					chain += ch[i]->chain;
					tree += ch[i]->tree;
				}
			}
		}

		void rotate(int z)
		{
			node_t *p = fa;
			int d = where(z);
			if(!p->fa) fa = 0;
			else p->fa->setc(this, p->where());
			p->setc(ch[z + !d], z + d);
			setc(p, z + !d);
			p->pushup();
		}

		void push_chain(const flag_t& r)
		{
			chain = chain + r, chain_flag += r;
			all = chain, all += tree;
			val = val * r.mul + r.add;
		}

		void push_tree(const flag_t& r, int virt)
		{
			all = all + r;
			tree = tree + r;
			tree_flag += r;
			if(virt) push_chain(r);
		}

		void make_rev()
		{
			is_rev ^= 1;
			std::swap(ch[0], ch[1]);
		}

		void pushdown()
		{
			if(is_rev) {
				is_rev = false;
				std::swap(ch[0], ch[1]);
				if(ch[0]) ch[0]->is_rev ^= 1;
				if(ch[1]) ch[1]->is_rev ^= 1;
			}

			if(!tree_flag.empty()) {
				for(int i = 0; i != 4; ++i)
					if(ch[i]) ch[i]->push_tree(tree_flag, i >> 1);
				tree_flag = flag_t();
			}

			if(!chain_flag.empty()) {
				for(int i = 0; i != 2; ++i)
```

```
                        if(ch[i]) ch[i]->push_chain(chain_flag);
                chain_flag = flag_t();
        }
    }

    void splay(int z)
    {
        for(; !is_root(z); rotate(z)) {
            if(!fa->is_root(z)) {
                if(where(z) == fa->where(z))
                    fa->rotate(z);
                else rotate(z);
            }
        }

        pushup();
    }

    void add(node_t *w)
    {
        w->pushdown();
        for(int i = 2; i != 4; ++i)
        {
            if(!w->ch[i])
            {
                w->setc(this, i);
                return;
            }
        }

        node_t *x = newnode(), *n = w;
        for(; n->ch[2]->is_inner; n = n->son(2));
        x->setc(n->ch[2], 2);
        x->setc(this, 3);
        n->setc(x, 2);
        x->splay(2);
    }

    void del()
    {
        if(fa->is_inner) {
            fa->fa->setc(fa->ch[5 - where()], fa->where());
            fa->fa->splay(2);
            delnode(fa);
        } else fa->setc(0, where());
        fa = 0;
    }

    void access()
    {
        static node_t *stack[MAXN];
        int st = 0;
        for(node_t *u = this; u; u = u->fa) stack[st++] = u;
        while(st) stack[--st]->pushdown();
        node_t *prev = 0, *now = this;
        while(now) {
            now->splay(0);
            if(now->ch[1]) now->ch[1]->add(now);
            if(prev) prev->del();
            now->setc(prev, 1);
            prev = now;
```

```
                        now->pushup();
                        for(now = now->fa; now && now->is_inner; now = now->fa);
                }

                splay(0);
        }

        void make_root()
        {
                access();
                is_rev ^= 1;
        }

        node_t *find_root()
        {
                node_t *z = this;
                for(; z->fa; z = z->fa);
                return z;
        }

        node_t *find_parent()
        {
                access();
                node_t *z = son(0);
                while(z && z->ch[1]) z = z->son(1);
                return z;
        }
};

node_t* cut(node_t *u)
{
        node_t *v = u->find_parent();
        if(v) v->access(), u->del(), v->pushup();
        return v;
}

void link(node_t *u, node_t *v)
{
        node_t *p = cut(u);
        if(u->find_root() != v->find_root()) p = v;
        if(p) p->access(), u->add(p), p->pushup();
}

int n, m;
int u[MAXN], v[MAXN];
int root;
int node_used;
node_t node[MAXN];
stack<node_t*> garbages;

void delnode(node_t *n)
{
        garbages.push(n);
}

node_t* newnode()
{
        node_t* z;
        if(!garbages.empty()) {
                z = garbages.top();
                garbages.pop();
```

```
        } else z = node + node_used++;
        z->chain = z->all = info_t();
        z->chain_flag = z->tree_flag = flag_t();
        z->val = z->is_rev = 0;
        z->is_inner = 1; z->fa = 0;
        for(int i = 0; i != 4; ++i) z->ch[i] = 0;
        return z;
}

void init()
{
        scanf("%d%d", &n, &m);
        node_used = n + 1;
        for(int i = 1; i < n; ++i) scanf("%d%d", &u[i], &v[i]);
        for(int i = 1; i <= n; ++i) {
                scanf("%d", &node[i].val);
                node[i].pushup();
        }

        for(int i = 1; i != n; ++i) {
                node[v[i]].make_root();
                link(&node[v[i]], &node[u[i]]);
        }

        scanf("%d", &root);
        node[root].make_root();
}

void query()
{
        // 0 -- subtree modify
        // 1 -- change root
        // 2 -- chain modify
        // 3 -- subtree query minimum
        // 4 -- subtree query maximum
        // 5 -- subtree add
        // 6 -- chain add
        // 7 -- chain query minimum
        // 8 -- chain query maximum
        // 9 -- change x's father to y
        // 10 -- chain query sum
        // 11 -- subtree query sum
        int k, x;
        scanf("%d %d", &k, &x);
        node_t *u = node + x;
        if(k == 0 || k == 3 || k == 4 || k == 5 || k == 11) {
                u->access();
                if(k == 3 || k == 4 || k == 11) {
                        int ans = u->val;
                        for(int i = 2; i != 4; ++i) {
                                if(u->ch[i]) {
                                        info_t info = u->ch[i]->all;
                                        if(k == 3) ans = min(ans, info.min);
                                        else if(k == 4) ans = max(ans, info.max);
                                        else if(k == 11) ans += info.sum;
                                }
                        }

                        printf("%d\n", ans);
                } else {
                        int y;
```

```
                            scanf("%d", &y);
                            flag_t flag(k == 5, y);
                            u->val = u->val * flag.mul + flag.add;
                            for(int i = 2; i != 4; ++i)
                                    if(u->ch[i]) u->ch[i]->push_tree(flag, 1);
                            u->pushup();
                    }
            } else if(k == 9) {
                    int v;
                    scanf("%d", &v);
                    link(u, node + v);
            } else if(k == 1) {
                    u->make_root();
                    root = x;
            } else {
                    int y;
                    scanf("%d", &y);
                    u->make_root();
                    node[y].access();
                    u->splay(0);
                    if(k == 7 || k == 8 || k == 10)
                    {
                            info_t ans = u->chain;
                            if(k == 7) printf("%d\n", ans.min);
                            else if(k == 8) printf("%d\n", ans.max);
                            else printf("%d\n", ans.sum);
                    } else {
                            int v;
                            scanf("%d", &v);
                            u->push_chain(flag_t(k == 6, v));
                    }

                    node[root].make_root();
            }
}
```

2.5 KD Tree

```
const int K = 2;
struct kd {
        double x[K];
        int id;
}t[MAXN];

double dis(const kd &a, const kd &b)
{
        double s = 0;
        for (int i = 0; i < K; ++i) s += sqr(a.x[i] - b.x[i]);
        return sqrt(s);
}

struct cmpk {
        int k;
        cmpk(int k): k(k) {}
        bool operator()(const kd &a, const kd &b)
        { return a.x[k] < b.x[k]; }
};

void build(int l, int r, int d)
{
```

```
        if (r - l <= 1) return;
        int mid = (l + r) >> 1;
        nth_element(t + l, t + mid, t + r, cmpk(d));
        if (++d == K) d = 0;
        build(l, mid, d); build(mid + 1, r, d);
}

typedef priority_queue<pair<double, int> > heap;
void knn(int l, int r, int d, const kd &p, size_t k, heap &h)
{
        if (r - l < 1) return;
        int mid = (l + r) >> 1;
        h.push(make_pair(dis(p, t[mid]), t[mid].id));
        if (h.size() > k) h.pop();
        double dx = p.x[d] - t[mid].x[d];
        if (++d == K) d = 0;
        if (dx < 0) {
                knn(l, mid, d, p, k, h);
                if (h.top().first > dx) knn(mid + 1, r, d, p, k, h);
        } else {
                knn(mid + 1, r, d, p, k, h);
                if (h.top().first > dx) knn(l, mid, d, p, k, h);
        }
}

/*
Usage:
        build(0, n, 0);
        knn(0, n, 0, pos, ans_heap);
*/
```

2.6 Sparse Table

```
int rmq[MAXN][LOGN];

void initRMQ(int a[], int n)
{
        for (int i = 0; i < n; ++i) rmq[i][0] = a[i];
        for (int j = 1; (1<<j) <= n; ++j) {
                for (int i = 0; i + (1<<j) <= n; ++i) {
                        rmq[i][j] = min(rmq[i][j-1], rmq[i+(1<<(j-1))][j-1]);
                }
        }
}

int RMQ(int l, int r)
{
        int k = sizeof(int) * 8 - __builtin_clz(r - l + 1) - 1;
        return min(rmq[l][k], rmq[r-(1<<k)+1][k]);
}
```

Chapter 3

Stringology

3.1 KMP Algorithm

```
void getf(char *s, int *f)
{
        int n = strlen(s);
        f[0] = 0; f[1] = 0;
        for (int i = 1; i < n; ++i) {
                int j = f[i];
                while (j && s[i] != s[j]) j = f[j];
                f[i + 1] = s[i] == s[j] ? j + 1 : 0;
        }
}

int match(char *s, char *p, int *f)
{
        int n = strlen(s), m = strlen(p);
        int j = 0;
        for (int i = 0; i < n; ++i) {
                while (j && s[i] != p[j]) j = f[j];
                if (s[i] == p[j]) ++j;
                if (j == m) return i - m + 1;
        }
}
```

3.2 Extend-KMP Algorithm

```
void getf(char *s, int *f)
{
        int n = strlen(s), j = 0, k = 1;
        while (j + 1 < n && s[j] == s[j + 1]) ++j;
        f[0] = n; f[1] = j;
        for (int i = 2; i < n; ++i) {
                int len = k + f[k] - 1, t = f[i - k];
                if (i + t <= len) {
                        f[i] = t;
                } else {
                        j = max(0, len - i + 1);
                        while (i + j < n && s[i + j] == s[j]) ++j;
                        f[i] = j; k = i;
                }
        }
}

void match(char *s, char *p, int *f, int *ex)
{
```

```
        int n = strlen(s), j = 0, k = 0;
        while (j < n && s[j] == p[j]) ++j;
        ex[0] = j;
        for (int i = 1; i < n; ++i) {
                int len = k + ex[k] - 1, t = f[i - k];
                if (i + t <= len) {
                        ex[i] =  t;
                } else {
                        j = max(0, len - i + 1);
                        while (i + j < n && s[i + j] == p[j]) ++j;
                        ex[i] = j; k = i;
                }
        }
}
```

3.3 Aho-Corasick Automation

```
const int PSZ = MAXN * LEN;
struct trie {
        trie *ch[SIGMA], *f;
        // trie *last;
        int val;
}pool[PSZ], *dict;
int psz;
int head, tail;
trie *que[PSZ];

void insert(trie *t, const char *s)
{
        for (; *s; ++s) {
                int c = *s - 'a';
                if (!t->ch[c]) memset(t->ch[c] = pool + psz++, 0, sizeof(trie));
                t = t->ch[c];
        }
        ++t->val;
}

void build_fail(trie *t)
{
        head = tail = 0;
        for  (int i = 0; i < SIGMA; ++i) {
                if (t->ch[i]) (que[tail++] = t->ch[i])->f= t;
                else t->ch[i] = t;
        }
        while (head < tail) {
                t = que[head++];
                // t->val += t->f->val;                          # method 1
                // t->last = t->f->val ? t->f : t->f->last;  # method 2
                for (int i = 0; i < SIGMA; ++i) {
                        if (t->ch[i]) (que[tail++] = t->ch[i])->f = t->f->ch[i];
                        else t->ch[i] = t->f->ch[i];
                }
        }
}

int find(trie *t, const char *s)
{
        int sum = 0;
        for (; *s; ++s) {
                int c = *s - 'a';
```

```
                t = t->ch[c];
                // sum += i->val;  # method 1
                // for (trie *i = t; i && i->val != -1; i = i->last) {
                //      sum += i->val, i->val = -1; // mark as visited
                // }                # method 2
        }
        return sum;
}

/*
Initialize:
        psz = 1; memset(dict = pool, 0, sizeof(trie));
Method 1: counting matching times
Method 2: counting the number of pattern matched
** duplicate patterns counted only once in method 2
*/
```

3.4 Suffix Array

```
int n;
char s[MAXN];
int sa[MAXN], rank[MAXN], height[MAXN];
int c[MAXN], wx[MAXN], wy[MAXN];

void build_sa(int m)
{
        int *x = wx, *y = wy;
        for (int i = 0; i < m; ++i) c[i] = 0;
        for (int i = 0; i < n; ++i) ++c[x[i] = s[i]];
        for (int i = 1; i < m; ++i) c[i] += c[i - 1];
        for (int i = n - 1; i >= 0; --i) sa[--c[x[i]]] = i;
        for (int k = 1; k <= n; k <<= 1) {
                int p = 0;
                for (int i = n - k; i < n; ++i) y[p++] = i;
                for (int i = 0; i < n; ++i) if (sa[i] >= k) y[p++] = sa[i] - k;
                for (int i = 0; i < m; ++i) c[i] = 0;
                for (int i = 0; i < n; ++i) ++c[x[y[i]]];
                for (int i = 1; i < m; ++i) c[i] += c[i - 1];
                for (int i = n - 1; i >= 0; --i) sa[--c[x[y[i]]]] = y[i];
                swap(x, y);
                p = 1; x[sa[0]] = 0;
                for (int i = 1; i < n; ++i) {
                        x[sa[i]] = y[sa[i - 1]] == y[sa[i]] &&
                                   y[sa[i - 1] + k] == y[sa[i] + k] ?
                                   p - 1 : p++;
                }
                if (p == n) break;
                m = p;
        }
}

void build_height()
{
        for (int i = 0; i < n; ++i) rank[sa[i]] = i;
        for (int i = 0, k = 0; i < n; ++i) {
                if (k) --k;
                if (!rank[i]) continue;
                int j = sa[rank[i] - 1];
                while (s[i + k] == s[j + k]) ++k;
                height[rank[i]] = k;
```

```
        }
}

// height[i] == lcp(suffix(sa[i-1]), suffix(sa[i]))
// REMEBER: add '$' after the string
```

3.5 Suffix Automation

```
struct sam {
        int l;
        sam *f, *ch[SIGMA];
}pool[LEN * 2], *root, *tail;
int psz;

sam *init_node(sam *p)
{
        memset(p->ch, 0, sizeof(p->ch));
        p->f = 0; p->l = 0;
        return p;
}

void sam_add(int v)
{
        sam *p = init_node(pool + psz++), *i;
        p->l = tail->l + 1;
        for (i = tail; i && !i->ch[v]; i = i->f) i->ch[v] = p;
        if (!i) {
                p->f = root;
        } else if (i->ch[v]->l == i->l + 1) {
                p->f = i->ch[v];
        } else {
                sam *q = pool + psz++, *r = i->ch[v];
                *q = *r;
                q->l = i->l + 1;
                p->f = r->f = q;
                for (; i && i->ch[v] == r; i = i->f) i->ch[v] = q;
        }
        tail = p;
}

int match(sam *root, char *s)
{
        int k = 0, ret = 0;
        sam *p = root;
        for (; *s; ++s) {
                int c = *s - 'a';
                if (p->ch[c]) {
                        ++k, p = p->ch[c];
                } else {
                        while (p && !p->ch[c]) p = p->f;
                        if (p) k = p->l + 1, p = p->ch[c];
                        else p = root; k = 0;
                }
                ret = max(ret, k);
                // p->match = max(p->match, k);
        }
        return ret;
}

// Initalize: init_node(root = tail = pool); psz = 1;
```

3.6 Longest Palindorme Substring(Manacher)

```
char s[MAXN], t[MAXN + MAXN + 3];
int rad[MAXN + MAXN + 3];

void manacher(char *s)
{
        int n = strlen(s), len = 0;
        t[len++] = '^'; t[len++] = '#';
        for (int i = 0; i < n; ++i) {
                t[len++] = s[i];
                t[len++] = '#';
        }
        t[len] = 0;
        int i = 1, j = 1, k;
        while (i < len) {
                while (t[i - j] == t[i + j]) ++j;
                rad[i] = j;
                for (k = 1; k < j && rad[i - k] != rad[i] - k; ++k) {
                        rad[i + k] = min(rad[i - k], rad[i] - k);
                }
                i += k; j = max(j - k, 1);
        }
}

/*
s: abaaba
t:    ^ # a # b # a # a # b # a # \0
rad: 0 1 2 1 4 1 2 7 2 1 4 1 2 1
*/
```

3.7 Palindromic Tree

```
char s[MAXN];
struct node {
        int len;
        node *ch[2], *fail;
}pool[MAXN], *root, *last;
int psz;

node *newnode(int len)
{
        node *t = pool + psz++;
        t->len = len;
        t->ch[0] = t->ch[1] = t->fail = 0;
        return t;
}

node *find(node *t, int i)
{
        while (i <= t->len || s[i - t->len - 1] != s[i]) t = t->fail;
        return t;
}

bool add(int i) // return whether add a new palindrome or not
{
        int c = s[i] - 'a';
        node *t = find(last, i);
        if (t->ch[c]) { last = t->ch[c]; return false; }
        last = t->ch[c] = newnode(t->len + 2);
```

```
        last->fail = t->fail ? find(t->fail, i)->ch[c] : root;
        return true;
}

void init()
{
        n = strlen(s);
        psz = 0;
        node *t0 = newnode(-1), *t1 = newnode(0);
        t1->fail = t0;
        root = last = t1;
        for (int i = 0; i < n; ++i) add(i);
}
```

3.8 Minimum Representation

```
int minrep(char *s, int n)
{
        int i = 0, j = 1, k = 0, t;
        while (i < n && j < n && k < n) {
                t = s[(i + k) % n] - s[(j + k) % n];
                if (!t) { ++k; continue; }
                if (t > 0) i = i + k + 1;
                else j = j + k + 1;
                if (i == j) ++j;
                k = 0;
        }
        return min(i, j);
}
```

Chapter 4

Computational Geometry

4.1 Basic Operations

```
typedef complex<double> point;
typedef point vec;
#define X real()
#define Y imag()

const double eps = 1e-8;

int  dcmp(double x) { return x < -eps ? -1 : x > eps;  }
bool zero(vec v)    { return !dcmp(v.X) && !dcmp(v.Y); }

double sqr(double x)          { return x * x;      }
double dis(point a, point b) { return abs(a - b); }

double cross(vec a, vec b)               { return a.X * b.Y - a.Y * b.X; }
double cross(point a, point b, point c) { return cross(b - a, c - a);   }
double dot(vec a, vec b)                 { return a.X * b.X + a.Y * b.Y; }
double dot(point a, point b, point c)   { return dot(b - a, c - a);     }

vec dir(line ln)  { return ln.t - ln.s;    }
vec normal(vec v) { return vec(-v.Y, v.X); }
vec unit(vec v)   { return v / abs(v);     }

vec   proj(vec v, vec n)        { return n * dot(v, n) / norm(n);          }
point proj(point p, line ln)    { return ln.s + proj(p - ln.s, dir(ln));    }
vec   reflect(vec v, vec n)     { return proj(v, n) * 2. - v;              }
point reflect(point p, line ln) { return ln.s + reflect(p - ln.s, dir(ln)); }

vec    rotate(vec v, double a) { return v * polar(1., a); }
double angle(vec a, vec b)     { return arg(b / a);      }
```

4.1.1 Line

```
double dis(point p, line ln) { return fabs(cross(p, ln.s, ln.t)) / len(ln); }

bool onseg(point p, line ln)
{ return dcmp(cross(p, ln.s, ln.t)) == 0 && dcmp(dot(p, ln.s, ln.t)) <= 0; }

double dtoseg(point p, line ln)
{
        if (dcmp(dot(ln.s, ln.t, p)) <= 0) return dis(p, ln.s);
        if (dcmp(dot(ln.t, ln.s, p)) <= 0) return dis(p, ln.t);
        return dis(p, ln);
}
```

```

bool inter(line a, line b, point &p)
{
        double s1 = cross(a.s, a.t, b.s);
        double s2 = cross(a.s, a.t, b.t);
        if (!dcmp(s1 - s2)) return false;
        p = (s1 * b.t - s2 * b.s) / (s1 - s2);
        return true;
}

bool seginter(line a, line b, point &p) // segment intersection(strict)
{
        double s1 = cross(a.s, a.t, b.s);
        double s2 = cross(a.s, a.t, b.t);
        if ((dcmp(s1) ^ dcmp(s2)) != -2) return false;
        double s3 = cross(b.s, b.t, a.s);
        double s4 = cross(b.s, b.t, a.t);
        if ((dcmp(s3) ^ dcmp(s4)) != -2) return false;
        p = (s1 * b.t - s2 * b.s) / (s1 - s2);
        return true;
}
```

4.1.2 Triangle

```
double area(double a, double b, double c) // Heron's Formula
{
        double p = (a + b + c) * 0.5;
        return sqrt(p * (p - a) * (p - b) * (p - c));
}

double angle(double a, double b, double c) // Law of Cosines
{
        return acos((sqr(a) + sqr(b) - sqr(c)) / (2 * a * b));
}

point center(point A, point B, point C) // Circumcenter
{
        double d1 = dot(A, B, C), d2 = dot(B, C, A), d3 = dot(C, A, B);
        double c1 = d2 * d3, c2 = d1 * d3, c3 = d1 * d2, c = c1 + c2 + c3;
        if (!dcmp(c)) return A; // coincident
        return ((c2 + c3) * A + (c1 + c3) * B + (c1 + c2) * C) / (2 * c);
}

point incenter(point A, point B, point C)
{
        double a = abs(B - C), b = abs(C - A), c = abs(A - B);
        if (!dcmp(a + b + c)) return A; // coincident
        return (a * A + b * B + c * C) / (a + b + c);
}

point centroid(point A, point B, point C)
{
        return (A + B + C) / 3;
}

point orthocenter(point A, point B, point C)
{
        double d1 = dot(A, B, C), d2 = dot(B, C, A), d3 = dot(C, A, B);
        double c1 = d2 * d3, c2 = d1 * d3, c3 = d1 * d2, c = c1 + c2 + c3;
        if (!dcmp(c)) return A; // coincident
        return (c1 * A + c2 * B + c3 * C) / c;
```

```
}

point fermat(point A, point B, point C)
{
        double a = abs(B - C), b = abs(C - A), c = abs(A - B);
        if (dot(A, B, C) / b / c < -.5) return A;
        if (dot(B, C, A) / c / a < -.5) return B;
        if (dot(C, A, B) / a / b < -.5) return C;
        if (cross(A, B, C) < 0) swap(B, C);
        point CC = (B - A) * polar(1., -pi / 3) + A;
        point BB = (C - A) * polar(1.,  pi / 3) + A;
        return inter(line(B, BB), line(C, CC));
}
```

4.1.3 Circle

```
double adjust(double a)
{
        while (a < -pi) a += 2 * pi;
        while (a >  pi) a -= 2 * pi;
        return a;
}

bool inter(circle c, line ln, point &p1, point &p2)
{
        point p = proj(c.c, ln);
        double d = dis(p, c.c);
        if (dcmp(d - c.r) > 0) return false;
        vec v = sqrt(c.r * c.r - d * d) * unit(dir(ln));
        p1 = p - v; p2 = p + v;
        return true;
}

bool inter(circle c, line ln, double &a1, double &a2)
{
        point p = proj(c.c, ln);
        double d = dis(p, c.c);
        if (dcmp(d - c.r) > 0) return false;
        double alpha = arg(p - c.c), beta = acos(d / c.r);
        a1 = adjust(alpha - beta); a2 = adjust(alpha + beta);
        return true;
}

bool inter(circle a, circle b, point &p1, point &p2)
{
        double d = dis(a.c, b.c);
        if (dcmp(d - (a.r + b.r)) > 0) return false;                 // disjoint
        if (!dcmp(d) || dcmp(d - fabs(a.r - b.r)) < 0) return false; // include
        double d1 = (sqr(d) + sqr(a.r) - sqr(b.r)) / (2 * d), d2 = d - d1;
        point p = (d1 * b.c + d2 * a.c) / d;
        vec v = sqrt(sqr(a.r) - sqr(d1)) * unit(normal(b.c - a.c));
        p1 = p - v; p2 = p + v;
        return true;
}

bool inter(circle a, circle b, double &a1, double &a2)
{
        double d = dis(a.c, b.c);
        if (dcmp(d - (a.r + b.r)) > 0) return false;                 // disjoint
        if (!dcmp(d) || dcmp(d - fabs(a.r - b.r)) < 0) return false; // include
        double alpha = arg(b.c - a.c), beta = angle(a.r, d, b.r);
```

```
        a1 = adjust(alpha - beta); a2 = adjust(alpha + beta);
        return true;
}

bool tan(circle c, point p, point &p1, point &p2)
{
        double d = dis(p, c.c);
        if (dcmp(d - c.r) < 0) return false;
        double d1 = c.r * c.r / d, d2 = d - d1;
        point p0 = (d1 * p + d2 * c.c) / d;
        vec v = sqrt(sqr(c.r) - sqr(d1)) * unit(normal(p - c.c));
        p1 = p0 - v; p2 = p0 + v;
        return true;
}

bool tan(circle c, point p, double &a1, double &a2)
{
        double d = dis(p, c.c);
        if (dcmp(d - c.r) < 0) return false;
        double alpha = arg(p - c.c), beta = acos(c.r / d);
        a1 = adjust(alpha - beta); a2 = adjust(alpha + beta);
        return true;
}

bool outertan(circle a, circle b, double &a1, double &a2)
{
        double d = dis(a.c, b.c);
        if (!dcmp(d) || dcmp(d - fabs(a.r - b.r)) < 0) return false; // include
        double alpha = arg(b.c - a.c), beta = acos((a.r - b.r) / d);
        a1 = adjust(alpha - beta); a2 = adjust(alpha + beta);
        return true;
}

bool innertan(circle a, circle b, double &a1, double &a2)
{
        double d = dis(a.c, b.c);
        if (!dcmp(d) || dcmp(d - (a.r + b.r)) < 0) return false;     // disjoint
        double alpha = arg(b.c - a.c), beta = acos((a.r + b.r) / d);
        a1 = adjust(alpha - beta); a2 = adjust(alpha + beta);
        return true;
}
```

4.2 Point in Polygon Problem

```
bool inpoly(point a, point *p, int n)
{
        int wn = 0;
        for (int i = 0; i < n; ++i) {
                point p1 = p[i], p2 = p[(i + 1) % n];
                int d = dcmp(cross(a, p1, p2));
                if (!s && dot(a, p1, p2) <= 0) return true;
                int d1 = dcmp(p1.Y - a.Y);
                int d2 = dcmp(p2.Y - a.Y);
                if (d > 0 && d1 <= 0 && d2 > 0) ++wn;
                if (d < 0 && d2 <= 0 && d1 > 0) --wn;
        }
        return wn != 0;
}
```

4.3 Convex Hull(Graham)

```
bool cmpx(point a, point b) { return dcmp(a.X - b.X) ? a.X < b.X : a.Y < b.Y; }

int graham(point p[], int n, point h[])
{
	int m = 0;
	sort(p, p + n, cmpx);
	for (int i = 0; i < n; ++i) {
		while (m > 1 && dcmp(cross(h[m - 2], h[m - 1], p[i])) <= 0) --m;
		h[m++] = p[i];
	}
	int k = m;
	for (int i = n - 2; i >= 0; --i) {
		while (m > k && dcmp(cross(h[m - 2], h[m - 1], p[i])) <= 0) --m;
		h[m++] = p[i];
	}
	if (n > 1) --m;
	return m;
}
```

4.4 Dynamic Convex Hull

```
struct cmpx {
	bool operator()(const point &a, point &b) const
	{ return dcmp(a.X - b.X) < 0; }
}

set<point, cmpx> lower, upper;

double insert(set<point, cmpx> &h, point p)
{
	double s = 0;
	set<point, cmpx>::iterator it = h.lower_bound(p);
	if (it != h.end() && !dcmp(p.X - it->X)) {
		if (dcmp(p.Y - it->Y) >= 0) return 0;
		if (it != h.begin())     s += cross(p, *it, *prev(it));
		if (next(it) != h.end()) s += cross(p, *next(it), *it);
		h.erase(it);
	} else if (it != h.begin() && it != h.end()) {
		double ds = cross(p, *it, *prev(it));
		if (dcmp(ds) <= 0) return 0;
		s += ds;
	}
	it = h.insert(p).first;
	while (it != h.begin() && prev(it) != h.begin()) {
		double ds = cross(p, *prev(it), *prev(prev(it)));
		if (dcmp(ds) < 0) break;
		h.erase(prev(it));
		s += ds;
	}
	while (next(it) != h.end() && next(next(it)) != h.end()) {
		double = cross(p, *next(it), *next(next(it)));
		if (dcmp(ds) > 0) break;
		h.erase(next(it));
		s -= ds;
	}
	return s * 0.5;
}

```

```
double insert(point p) // return area increment
{
        double s = 0;
        if (lower.size()) {
                s += max(0., cross(p, *lower.begin(), conj(*upper.begin())));
                s += max(0., cross(p, conj(*upper.rbegin()), *lower.rbegin()));
        }
        s += insert(lower, p);
        s += insert(upper, conj(p));
        return s;
}
```

4.5 Half-plane Intersection

```
bool inhp(point p, line hp) { return dcmp(cross(hp.s, hp.t, p)) >= 0; }

bool cmpang(line a, line b)
{ return dcmp(a.a - b.a) ? a.a < b.a : cross(a.s, a.t, b.s) < 0; }

int hpinter(line q[], int n, point h[])
{
        // line q[i] represent the half-plane on its left
        int head = 0, tail = 0, m = 0;
        for (int i = 0; i < n; ++i) q[i].a = arg(dir(q[i]));
        sort(q, q + n, cmpang);
        for (int i = 1; i < n; ++i) {
                if (!dcmp(q[i].a - q[i - 1].a)) continue;
                while (head < tail && !inhp(h[tail - 1], q[i])) --tail;
                while (head < tail && !inhp(h[head], q[i])) ++head;
                q[++tail] = q[i];
                if (head < tail) h[tail - 1] = inter(q[tail - 1], q[tail]);
        }
        while (head < tail && !inhp(h[tail - 1], q[head])) --tail;
        if (head < tail) h[tail] = inter(q[tail], q[head]);
        for (int i = head; i <= tail; ++i) h[m++] = h[i];
        return m;
}

line makehp(double a, double b, double c) // ax + by + c > 0
{
        point p1 = fabs(a) > fabs(b) ? point(-c / a, o) : point(0, -c / b);
        point p2 = p1 + vec(b, -a);
        return line(p1, p2);
}
```

4.6 Closest Pair(Divide and Conquer)

```
bool cmpx(point a, point b) { return a.X < b.X; }
bool cmpy(point a, point b) { return a.Y < b.Y; }

double mindis(point p[], int l, int r)
{
        static point t[MAXN];
        if (r - l <= 1) return inf;
        int mid = (l + r) >> 1, m = 0;
        double x = p[mid].X;
        double d = min(mindis(l, mid), mindis(mid, r));
        inplace_merge(p + l, p + mid, p + r, cmpy());
        for (int i = l; i < r; ++i) {
                if (fabs(x - p[i].X) < d) t[m++] = p[i];
```

```
        }
        for (int i = 0; i < m; ++i) {
                for (int j = i + 1; j < m; ++j) {
                        if (t[j].Y - t[i].Y >= d) break;
                        d = min(d, abs(t[i] - t[j]));
                }
        }
        return d;
}

/*
Initialize: sort(p, p + n, cmpx)
Usage: mindis(0, n)
*/
```

4.7 Farthest Pair(Rotating Caliper)

```
double maxdis(point *p, int n)
{
        int m = graham(p, n, h);
        if (m == 2) return abs(h[0] - h[1]);
        h[m] = h[0];
        double d = 0;
        for (int i = 0, j = 1; i < m; ++i) {
                while (dcmp(cross(h[i + 1] - h[i], h[j + 1] - h[j])) > 0) {
                        j = (j + 1) % m;
                }
                d = max(d, abs(h[i] - h[j]));
        }
        return d;
}
```

4.8 Minimum Distance Between Convec Hull(Rotating Caliper)

```
void mindis(point *p1, int n, point *p2, int m)
{
        int i = 0, j = 0;
        for (int k = 1; k < n; ++k) if (cmpx()(p1[k], p1[i])) i = k;
        for (int k = 1; k < m; ++k) if (cmpx()(p2[j], p2[k])) j = k;
        for (int t = 0; t < n + m; ++t) {
                if (dcmp(cross(p1[i + 1] - p1[i], p2[j + 1] - p2[j])) < 0) {
                        ans = min(ans, dtoseg(p2[j], line(p1[i], p1[i + 1])));
                        i = (i + 1) % n;
                } else {
                        ans = min(ans, dtoseg(p1[i], line(p2[j], p2[j + 1])));
                        j = (j + 1) % m;
                }
        }
}
```

4.9 Union Area of a Circle and a Polygon

```
double area(circle c, point a, point b)
{
        a -= c.c; b -= c.c;
        if (zero(a) || zero(b)) return 0;
        double s1 = .5 * arg(b / a) * sqr(c.r);
        double s2 = .5 * cross(a, b);
        return fabs(s1) < fabs(s2) ? s1 : s2;
```

```
}

double unionarea(circle c, point p[], int n)
{
        double s = 0;
        for (int i = 0; i < n; ++i) {
                point A = p[i], B = p[(i + 1) % n], p1, p2;
                line AB = line(A, B);
                if (inter(c, AB, p1, p2) && (onseg(p1, AB) || onseg(p2, AB))) {
                        s += area(c, A, p1) + area(c, p1, p2) + area(c, p2, B);
                } else {
                        s += area(c, A, B);
                }
        }
        return fabs(s);
}
```

4.10 Union Area of Circles

```
bool same(circle a, circle b)  { return zero(a.c - b.c) && !dcmp(a.r - b.r);  }
bool incir(circle a, circle b) { return dcmp(dis(a.c, b.c) + a.r - b.r) <= 0; }

void unionarea(circle c[], int n, double tot[])
{
        static pair<double, int> a[MAXN * 2];
        for (int i = 0; i <= n; ++i) tot[i] = 0;
        for (int i = 0; i < n; ++i) {
                int m = 0, k = 0;
                for (int j = 0; j < n; ++j) if (i != j) {
                        double a1, a2;
                        if (same(c[i], c[j]) && i < j) continue;
                        if (incir(c[i], c[j])) { ++k; continue; }
                        if (!inter(c[i], c[j], a1, a2)) continue;
                        a[m++] = make_pair(a1,  1);
                        a[m++] = make_pair(a2, -1);
                        if (a1 > a2) ++k;
                }
                sort(a, a + m);
                double a1 = a[m - 1].first - 2 * pi, a2, rad;
                for (int j = 0; j < m; ++j) {
                        a2 = a[j].first, rad = a2 - a1;
                        tot[k] += .5 * sqr(c[i].r) * (rad - sin(rad));
                        tot[k] += .5 * cross(c[i].p(a1), c[i].p(a2));
                        k += a[j].second;
                        a1 = a2;
                }
                if (!m) tot[k] += pi * sqr(c[i].r);
        }
}

/*
tot[0]              = the aera of union
tot[n-1]            = the aera of intersection
tot[k-1] - tot[k] = the aera covered k times
*/
```

4.11 Union Area of Polygons

```
double pos(point p, line ln)
{ return dot(p - ln.s, dir(ln)) / norm(dir(ln)); }
```

```

void unionarea(vector<point> p[], int n, double tot[])
{
        for (int i = 0; i <= n; ++i) tot[i] = 0;
        for (int i = 0; i < n; ++i)
        for (int ii = 0; ii < p[i].size(); ++ii) {
                point A = p[i][ii], B = p[i][(ii + 1) % p[i].size()];
                line AB = line(A, B);
                vector<pair<double, int> > c;
                for (int j = 0; j < n; ++j) if (i != j)
                for (int jj = 0; jj < p[j].size(); ++jj) {
                        point C = p[j][jj], D = p[j][(jj + 1) % p[j].size()];
                        line CD = line(C, D);
                        int f1 = dcmp(cross(A, B, C));
                        int f2 = dcmp(cross(A, B, D));
                        if (!f1 && !f2) {
                                if (i < j && dcmp(dot(dir(AB), dir(CD))) > 0) {
                                        c.push_back(make_pair(pos(C, AB),  1));
                                        c.push_back(make_pair(pos(D, AB), -1));
                                }
                                continue;
                        }
                        double s1 = cross(C, D, A);
                        double s2 = cross(C, D, B);
                        double t = s1 / (s1 - s2);
                        if (f1 >= 0 && f2 < 0) c.push_back(make_pair(t,  1));
                        if (f1 < 0 && f2 >= 0) c.push_back(make_pair(t, -1));
                }
                c.push_back(make_pair(0., 0));
                c.push_back(make_pair(1., 0));
                sort(c.begin(), c.end());
                double s = .5 * cross(A, B), z = min(max(c[0].s, 0.), 1.);
                for (int j = 1, k = c[0].second; j < c.size(); ++j) {
                        double w = min(max(c[j].first, 0.), 1.);
                        tot[k] += s * (w - z);
                        k += c[j].second;
                        z = w;
                }
        }
}

/*
tot[0]              = the aera of union
tot[n-1]            = the aera of intersection
tot[k-1] - tot[k] = the aera covered by k times
*/
```

4.12 Minimum Enclosing Circle(Randomized Incremental Method)

```
circle mincir(point *p, int n)
{
        point c;
        double r;
        random_shuffle(p, p + n);
        c = p[0]; r = 0;
        for (int i = 1; i < n; ++i) {
                if (dcmp(abs(p[i] - c) - r) <= 0) continue;
                c = p[i]; r = 0;
                for (int j = 0; j < i; ++j) {
                        if (dcmp(abs(p[j] - c) - r) <= 0) continue;
```

```
                        c = (p[i] + p[j]) * 0.5; r = dis(p[j], c);
                        for (int k = 0; k < j; ++k) {
                                if (dcmp(abs(p[k] - c) - r) <= 0) continue;
                                c = center(p[i], p[j], p[k]); r = dis(p[k], c);
                        }
                }
        }
        return circle(c, r);
}
```

4.13 Planar Strainght-line Graph(PSLG)

```
int pcnt, ecnt, fcnt;
point p[MAXN];
struct edge { int t; double ang; };
edge E[MAXN * MAXN * 2];
vector<int> G[MAXN * MAXN];
int co[MAXN * MAXN * 2];
int pre[MAXN * MAXN * 2];
vector<point> face[MAXN * MAXN * 2];

void addedge(int u, int v)
{
        G[u].push_back(ecnt);
        E[ecnt++] = (edge){v, arg(p[v] - p[u])};
        G[v].push_back(ecnt);
        E[ecnt++] = (edge){u, arg(p[u] - p[v])};
}

bool cmpang(int i, int j) { return E[i].ang < E[j].ang; }

void build_pslg()
{
        for (int u = 0; u < pcnt; ++u) {
                sort(G[u].begin(), G[u].end(), cmpang);
                int n = G[u].size();
                for (int j = 0; j < n; ++j) pre[G[u][(j + 1) % n]] = G[u][j];
        }
        memset(co, -1, sizeof(co));
        for (int u = 0; u < pcnt; ++u) {
                for (int i = 0; i < G[u].size(); ++i) {
                        int e = G[u][i];
                        if (co[e] != -1) continue;
                        while (co[e] == -1) {
                                co[e] = fcnt;
                                face[fcnt].push_back(p[E[e].t]);
                                e = pre[e^1];
                        }
                        face[++fcnt].clear();
                }
        }
}
```

4.14 3D Computational Geometry

```
bool zero(vec3 v)
{ return !dcmp(v.x) && !dcmp(v.y) && !dcmp(v.z); }

double dot(vec3 a, vec3 b)
{ return a.x * b.x + a.y * b.y + a.z * b.z; }
```

```

double abs(vec3 v)
{ return sqrt(dot(v, v)); }

vec3 unit(vec3 v)
{ return v / abs(v); }

vec3 cross(vec3 a, vec3 b)
{
        return vec3(a.y * b.z - a.z * b.y,
                    a.z * b.x - a.x * b.z,
                    a.x * b.y - a.y * b.x);
}

double area2(point3 a, point3 b, point3 c)
{ return abs(cross(b - a, c - a)); }

double vol6(point3 a, point3 b, point3 c, point3 d)
{ return dot(cross(b - a, c - a), d - a); }

double len(line3 ln)
{ return abs(ln.s - ln.t); }

vec3 dir(line3 ln)
{ return ln.t - ln.s; }

vec3 proj(vec3 v, vec3 d)
{ return d * dot(v, d) / dot(d, d); }

point3 proj(point3 p, line3 ln)
{ return ln.s + proj(p - ln.s, dir(ln)); }

point3 proj(point3 p, point3 p0, vec3 n) // projection on plane
{ return p - proj(p - p0, n); }

vec3 reflect(vec3 v, vec3 n)
{ return proj(v, n) * 2 - v; }

point3 reflect(point3 p, line3 ln)
{ return ln.s + reflect(p - ln.s, dir(ln)); }

point3 reflect(point3 p, point3 p0, vec3 n) // reflection to plane
{ return p - proj(p - p0, n) * 2; }

double angle(vec3 a, vec3 b)
{ return acos(dot(a, b) / abs(a) / abs(b)); }

vec3 rotate(vec3 v, vec3 n, double a)
{
        n = unit(n);
        double cosa = cos(a), sina = sin(a);
        return v * cosa + cross(n, v) * sina + n * dot(n, v) * (1 - cosa);
}
```

4.14.1 Line

```
double dis(point3 p, line3 ln)
{ return area2(p, ln.s, ln.t) / len(ln); }

double dtoseg(point3 p, line3 ln)
{
```

```
        if (dcmp(dot(p - ln.s, dir(ln))) <= 0) return dis(p, ln.s);
        if (dcmp(dot(p - ln.t, dir(ln))) >= 0) return dis(p, ln.t);
        return dis(p, ln);
}

bool onseg(point3 p, line3 ln)
{
        return zero(cross(p - ln.s, p - ln.t))
            && dcmp(dot(p - ln.s, p - ln.t)) <= 0;
}

bool inter(line3 ln, point3 p0, vec3 n, point3 &p) // line & plane intersection
{
        double d1 = dot(ln.s - p0, n);
        double d2 = dot(ln.t - p0, n);
        if (!dcmp(d1 - d2)) return false;
        p = (ln.t * d1 - ln.s * d2) / (d1 - d2);
        return true;
}

double dis(line3 a, line3 b)
{
        vec3 n = cross(dir(a), dir(b));
        if (zero(n)) return dis(a.s, b);
        return fabs(dot(a.s - b.s, n)) / abs(n);
}

bool approach(line3 a, line3 b, point3 &p) // clost approach point of 2 lines
{
        vec3 u = dir(a), v = dir(b), w = a.s - b.s;
        double d = dot(u, u) * dot(v, v) - dot(u, v) * dot(u, v);
        if (!dcmp(d)) return false; // parallel
        double c = dot(u, v) * dot(v, w) - dot(v, v) * dot(u, w);
        p = a.s + u * (c / d);
        return true;
}
```

4.14.2 Sphere

```
struct sphere {
        point3 c;
        double r;
        sphere() {}
        sphere(point3 c, double r): c(c), r(r) {}
};

bool inter(sphere s, line3 ln, point3 &p1, point3 &p2)
{
        point3 p = proj(s.c, ln);
        double d = abs(p - s.c);
        if (dcmp(d - s.r) > 0) return false;
        vec3 v = unit(dir(ln)) * sqrt(s.r * s.r - d * d);
        p1 = p - v; p2 = p + v;
        return true;
}
```

4.14.3 3D Transformation Matrix

Translate, $\overrightarrow{(x,y,z)}$ as offset:

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale, (x,y,z) as scale:

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Point reflection, (x,y,z) as center:

$$\begin{bmatrix} -1 & 0 & 0 & 2x \\ 0 & -1 & 0 & 2y \\ 0 & 0 & -1 & 2z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Line reflection, $\overline{O(x,y,z)}$ as axis:

$$\begin{bmatrix} 2x^2-1 & 2xy-1 & 2xz-1 & 0 \\ 2yx-1 & 2y^2-1 & 2yz-1 & 0 \\ 2zx-1 & 2zy-1 & 2z^2-1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\overrightarrow{(x,y,z)}$ should be a unit vector

Plane reflection, $\overrightarrow{(x,y,z)}$ as the normal vector of the reflection plane:

$$\begin{bmatrix} -2x^2 & -2xy & -2xz & 0 \\ -2yx & -2y^2 & -2yz & 0 \\ -2zx & -2zy & -2z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\overrightarrow{(x,y,z)}$ should be a unit vector

Rotate, $\overline{O(x,y,z)}$ as axis, rotate α radian:

$$\begin{bmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ yx(1-c)+zs & y^2(1-c)+c & yz(1-c)-xs & 0 \\ zx(1-c)-ys & zy(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$s = sin(\alpha), c = cos(\alpha)$$

$\overrightarrow{(x,y,z)}$ should be a unit vector

Usage:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = M \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

4.15 Convex Hull in 3D

```
struct face {
	int v[3];
	face(int a, int b, int c) { v[0] = a; v[1] = b; v[2] = c; }
	int operator[](int i) const { return v[i % 3]; }
};

bool visible(point3 p[], face f, int i)
{ return dcmp(vol6(p[f[0]], p[f[1]], p[f[2]], p[i])) > 0; }

vector<face> ch3d(point3 p[], int n)
{
	static bool v[MAXN][MAXN];
	int i, j, k;
	for (i = 2; i < n && !dcmp(area2(p[0], p[1], p[i])); ++i) {}
	swap(p[2], p[i]);
	for (i = 3; i < n && !dcmp(vol6(p[0], p[1], p[2], p[i])); ++i) {}
	swap(p[3], p[i]);
	vector<face> cur;
	cur.push_back(face(0, 1, 2));
	cur.push_back(face(2, 1, 0));
	for (i = 3; i < n; ++i) {
		vector<face> next;
		for (j = 0; j < cur.size(); ++j) {
			face f = cur[j];
			bool vis = visible(p, f, i);
			if (!vis) next.push_back(f);
			for (int k = 0; k < 3; ++k) v[f[k]][f[k + 1]] = vis;
		}
		for (j = 0; j < cur.size(); ++j) {
			for (k = 0; k < 3; ++k) {
				int a = cur[j][k], b = cur[j][k + 1];
				if (v[a][b] && !v[b][a]) {
					next.push_back(face(a, b, i));
				}
			}
		}
		cur.swap(next);
	}
	return cur;
}
```

4.16 Half-space Intersection

```
struct plane {
	point3 p; vec3 n; // represent the half-space that n direct to
	plane() {}
	plane(point3 p, point3 n): p(p), n(n) {}
};
```

```

point3 inter(line3 ln, plane f)
{
        double d1 = dot(ln.s - f.p, f.n);
        double d2 = dot(ln.t - f.p, f.n);
        return (ln.t * d1 - ln.s * d2) / (d1 - d2);
}

struct face {
        point3 p[3];
        face(point3 a, point3 b, point3 c) { p[0] = a, p[1] = b, p[2] = c; }
        point3 &operator[](int i) { return p[i]; }
        vec3 normal() { return cross(p[1] - p[0], p[2] - p[0]); }
        void adjust(vec3 n) { if (dot(n, normal()) < 0) swap(p[0], p[1]); }
};

vector<face> cut(vector<face> h, plane f)
{
        vector<face> ans;
        point3 p0; int m = 0;
        for (int i = 0; i < h.size(); ++i) {
                vector<point3> c0, c1, c2;
                for (int j = 0; j < 3; ++j) {
                        int d = dcmp(dot(h[i][j] - f.p, f.n));
                        if (d == 0) c0.push_back(h[i][j]);
                        else if (d > 0) c1.push_back(h[i][j]);
                        else c2.push_back(h[i][j]);
                }
                if (c0.size() == 3) {
                        if (dot(f.n, h[i].normal()) < 0) h.clear();
                        return h;
                }
                if (c0.size() == 1) {
                        if (c1.size() > c2.size()) c1.push_back(c0[0]);
                        else c2.push_back(c0[0]);
                }
                if (c0.size() == 2) c2.push_back(c0[0]), c2.push_back(c0[1]);
                if (c1.size() == 3) ans.push_back(h[i]);
                if (c1.size() == 3 || c2.size() == 3) continue;
                point3 p1, p2; vec3 n = h[i].normal();
                if (c1.size() == 1) {
                        p1 = inter(line3(c1[0], c2[0]), f);
                        p2 = inter(line3(c1[0], c2[1]), f);
                        ans.push_back(face(c1[0], p1, p2));
                        ans.back().adjust(n);
                } else {
                        p1 = inter(line3(c1[0], c2[0]), f);
                        p2 = inter(line3(c1[1], c2[0]), f);
                        ans.push_back(face(c1[0], p1, p2));
                        ans.back().adjust(n);
                        ans.push_back(face(c1[0], c1[1], p2));
                        ans.back().adjust(n);
                }
                if (m++) {
                        ans.push_back(face(p0, p1, p2));
                        ans.back().adjust(f.n);
                } else p0 = p1;
        }
        return ans;
}
```

Chapter 5

Number Theory

5.1 Fast Fourier Transform

```
typedef complex<double> cp;
void fft(cp a[], int n, int inv)
{
        for(int i = 0, j = 0; i < n; i++) {
                if(j > i) swap(a[i], a[j]);
                int k = n;
                while (j & (k >>= 1)) j &= ~k;
                j |= k;
        }
        for(int k = 1; k < n; k <<= 1) {
                double arg = inv * pi / k;
                for(int i = 0; i < k; ++i) {
                        cp w = exp(cp(0, arg * i));
                        for(int j = i; j < n; j += k << 1) {
                                cp t = w * a[j + k];
                                a[j + k] = a[j] - t;
                                a[j] += t;
                        }
                }
        }
        if(inv == -1) for(int i = 0; i < n; i++) a[i] /= n;
}

/*
Usage:
fft(a, n,  1); -- dft
fft(a, n, -1); -- idft
n should be 2^k
*/
```

5.2 Primality Test(Miller-Rabin)

```
bool Witness(ll n,ll a)
{
     ll m=(n-1),j=0;
     while(!(m&1)) m>>=1,j++;
     ll ans=Make_Power(a,m,n);
     while(j--)
     {
         ll tmp=Make_Multi(ans,ans,n);
         if(tmp==1 && ans!=1 && ans!=n-1) return 1;
         ans=tmp;
     }
```

```
    if(ans!=1) return 1;
    return 0;
}
bool Miller_Rabin(ll n)
{
    if(n<2) return 0; if(n==2) return 1; if(!(n&1)) return 0;
    for(int i=0; i<max_test; i++)
    {
        ll a=rand()%(n-2)+2;
        if(Witness(n,a)) return 0;
    }
    return 1;
}
```

5.3 Integer Factorization(Pollard's ρ Algorithm)

```
ll Pollard_Rho(ll n,ll c)
{
    ll i=1,k=2,x=rand()%(n-1)+1,y=x,d;
    while(1)
    {
        i++;
        x=( Make_Multi(x,x,n)+c)%n;
        d=Gcd(n,y-x);
        if(d>1&&d<n) return d;
        if(y==x) return n;
        if(i==k) k<<=1,y=x;
    }
}
```

5.4 Extended Euclid's Algorithm

```
int exgcd(int a, int b, int &x, int &y)
{
        if (b == 0) {
                x = 1; y = 0;
                return a;
        } else {
                int g = exgcd(b, a % b, y, x);
                y -= (a / b) * x;
                return g;
        }
}
```

5.5 Euler's φ Function

```
void phi_table()
{
        for (int i = 2; i * i < MAX; ++i) {
                if (!phi[i]) {
                        for (int k = (MAX - 1) / i, j = i * k;
                             k >= i; --k, j -=i) {
                                if (!phi[k]) phi[j] = i;
                                // i is a prime factor of j
                        }
                }
        }
        phi[1] = 1;
        for (int i = 2; i < MAX; ++i) {
```

```
			if (!phi[i]) {
				phi[i] = i - 1;
			} else {
				int j = i / phi[i];
				if (j % phi[i] == 0) phi[i] = phi[j] * phi[i];
				else phi[i] = phi[j] * (phi[i] - 1);
			}
		}
}

// n = p1^a1 * p2^a2 * ...
// phi[n] = n / p1 * (p1 - 1) / p2 * (p2 - 1) ...
```

Chapter 6

Others

6.1 Exact Cover(DLX)

```
int N, S[COL + 1], L[NODE], R[NODE], U[NODE], D[NODE], row[NODE], C[NODE];

void dlxinit(int c) // c Cumns, numbered from 1
{
        for (int i = 0; i <= c; ++i) {
                U[i] = D[i] = i;
                L[i] = i - 1; R[i] = i + 1;
                S[i] = 0;
        }
        L[0] = c; R[c] = 0; N = c + 1;
}

void addrow(const vector<int> &c)
{
        int h = N;
        for (int i = 0; i < c.size(); ++i) {
                U[N] = U[c[i]]; D[N] = c[i];
                D[U[N]] = U[D[N]] = N;
                L[N] = N - 1; R[N] = N + 1;
                ++S[C[N++] = c[i]];
        }
        L[h] = N - 1; R[N - 1] = h;
}

void remove(int c)
{
        L[R[c]] = L[c];
        R[L[c]] = R[c];
        for (int i = D[c]; i != c; i = D[i]) {
                for (int j = R[i]; j != i; j = R[j]) {
                        U[D[j]] = U[j];
                        D[U[j]] = D[j];
                        --S[C[j]];
                }
        }
}

void resume(int c)
{
        for (int i = U[c]; i != c; i = U[i]) {
                for (int j = L[i]; j != i; j = L[j]) {
                        U[D[j]] = j;
                        D[U[j]] = j;
```

```
                                ++S[C[j]];
                        }
                }
                L[R[c]] = c;
                R[L[c]] = c;
}

bool dance(int d)
{
                if (R[0] == 0) return true;
                int c = R[0];
                for (int i = R[0]; i; i = R[i]) {
                        if (S[i] < S[c]) c = i;
                }
                remove(c);
                for (int i = D[c]; i != c; i = D[i]) {
                        // select row[i]
                        for (int j = R[i]; j != i; j = R[j]) remove(C[j]);
                        if (dance(d + 1)) return true;
                        for (int j = L[i]; j != i; j = L[j]) resume(C[j]);
                }
                resume(c);
                return false;
}
```

6.2 Fuzzy Cover(DLX)

```
void remove(int i)
{
                for (int j = D[i]; j != i; j = D[j]) {
                        R[L[j]] = R[j];
                        L[R[j]] = L[j];
                }
}

void resume(int i)
{
                for (int j = U[i]; j != i; j = U[j]) {
                        R[L[j]] = j;
                        L[R[j]] = j;
                }
}

int h()
{
                static int v[COL + 1], m;
                int s = 0; ++m;
                for (int i = R[0]; i; i = R[i]) {
                        if (v[i] == m) continue;
                        ++s; v[i] = m;
                        for (int j = D[i]; j != i; j = D[j]) {
                                for (int k = R[j]; k != j; k = R[k]) {
                                        v[C[k]] = m;
                                }
                        }
                }
                return s;
}

bool dance(int d)
```

```
{
        if (!R[0]) return true;
        if (d + h() > limit) return false;
        int c = R[0];
        for (int i = R[c]; i; i = R[i]) {
                if (S[i] < S[c]) c = i;
        }
        for (int i = D[c]; i != c; i = D[i]) {
                remove(i);
                for (int j = R[i]; j != i; j = R[j]) remove(j);
                if (dance(d + 1)) return true;
                for (int j = L[i]; j != i; j = L[j]) resume(j);
                resume(i);
        }
        return false;
}
```

6.3 3D Partial Order(Divide and Conquer)

```
struct triple {
        int x, y, z;
        bool operator<(const triple &b) const
        {
                return x != b.x ? x < b.x : y > b.y;
                // return x != b.x ? x < b.x : y < b.y;
        }
}v[MAXN];
int f[MAXN];

/*
   solve LIS problem,
   the longest chain that (x[i] < x[i+1] && y[i] < y[i+1] && z[i] < z[i+1])

   for problem with (x[i] <= x[i+1] && y[i] <= y[i+1] && z[i] <= z[i+1]),
   use the commented code instead
*/
void solve(int l, int r)
{
        if (r - l == 1) f[l] = max(f[l], 1);
        if (r - l <= 1) return;
        static int p[MAXN];
        int mid = (l + r) / 2;
        solve(l, mid);
        for (int i = l; i < r; ++i) p[i] = i;
        sort(p + l, p + r, [](int i, int j) {
                return v[i].y != v[j].y ? v[i].y < v[j].y : i > j;
                // return v[i].y != v[j].y ? v[i].y < v[j].y : i < j;
        });
        for (int i = l; i < r; ++i) {
                if (p[i] < mid) bit.add(v[p[i]].z, f[p[i]]); // maintain maximum
                else f[p[i]] = max(f[p[i]], bit.query(v[p[i]].z - 1) + 1);
                // f[p[i]] = max(f[p[i]], bit.query(v[p[i]].z) + 1);
        }
        for (int i = l; i < mid; ++i) bit.clear(v[i].z);
        solve(mid, r);
}

void solve()
{
        sort(v, v + n);
```

```
        static int z[MAXN];
        for (int i = 0; i < n; ++i) z[i] = v[i].z;
        sort(z, z + n);
        int tot = unique(z, z + n) - z;
        for (int i = 0; i < n; ++i) {
                v[i].z = lower_bound(z, z + tot, v[i].z) - z + 1;
                f[i] = 0;
        }
        solve(0, n);
        return *max_element(f, f + n);
}
```

6.4 Power of Matrix

```
class matrix {
private:
        int row, col;
        vector<int> val;
public:
        matrix(int r, int c): row(r), col(c), val(r * c) {}
        matrix(int r, int c, int *v): row(r), col(c), val(v, v + r * c) {}
        int rows() const { return row; }
        int cols() const { return col; }
        int get(int r, int c) const { return val[r * col + c]; }
        void set(int r, int c, int v) { val[r * col + c] = v; }
};

matrix operator*(const matrix &lhs, const matrix &rhs)
{
        matrix ret(lhs.rows(), rhs.cols());
        for (int i = 0; i < lhs.rows(); ++i) {
                for (int j = 0; j < rhs.cols(); ++j) {
                        int s = 0;
                        for (int k = 0; k < lhs.cols(); ++k) {
                                s += lhs.get(i, k) * rhs.get(k, j);
                        }
                        ret.set(i, j, s);
                }
        }
        return ret;
}

matrix pow(const matrix &mat, int k)
{
        if (k == 1) return mat;
        matrix ret = pow(mat, k >> 1);
        return k & 1 ? ret * ret * mat : ret * ret;
}
```

6.5 Cantor Pairing Function

```
int cantor()
{
        // s = a[0] * (n - 1)! + a[1] * (n - 2)! + ... + a[n - 1]
        int s = 0;
        for (int i = 0; i < n; ++i) {
                int t = 0;
                for (int j = i + 1; j < n; ++j) if (a[j] < a[i]) ++t;
                s = (s + t) * (n - i - 1);
        }
```

```
        return s;
}

int uncantor(int s)
{
        memset(u, 0, sizeof(u));
        for (int i = 0; i < n; ++i) {
                int t = s / fac[n - i - 1];
                s -= t * fac[n - i - 1];
                int l = 0;
                for (int j = 0; l <= t; ++j) if (!u[j]) ++l;
                u[a[i] = --j] = true;
        }
}
```

6.6 Adaptive Simpson's Method

```
double simpson(double a, double b)
{
        return (b - a) / 6 * (f(a) + 4 * f((a + b) * 0.5) + f(b));
}

double rsimpson(double a, double b)
{
        double m = (a + b) * 0.5;
        double s = simpson(a, b);
        double s1 = simpson(a, m);
        double s2 = simpson(m, b);
        if (fabs(s1 + s2 - s) < ESP) return s;
        return rsimpson(a, m) + rsimpson(m, b);
}
```

6.7 Linear Programming(Simplex)

```
int n, m;
double a[MAXM][MAXN];
double x[MAXN];
int N[MAXN], B[MAXM];
const double eps = 1e-10, inf = 1e100;

// a[i][0]*x[0] + a[i][1]*x[1] + ... <= a[i][n]
// max(a[m][0]*x[0] + a[m][1]*x[1] + ... - a[m][n])
// x[i] >= 0

void pivot(int r, int c)
{
        swap(N[c], B[r]);
        a[r][c] = 1 / a[r][c];
        for (int i = 0; i <= n; ++i) if (i != c) a[r][i] *= a[r][c];
        for (int i = 0; i <= m; ++i) if (i != r) {
                for (int j = 0; j <= n; ++j) if (j != c) {
                        a[i][j] -= a[i][c] * a[r][j];
                }
                a[i][c] *= -a[r][c];
        }
}

bool feasible()
{
        for (;;) {
```

```
                int r, c;
                double p = inf;
                for (int i = 1; i < m; ++i) if (a[i][n] < p) p = a[r = i][n];
                if (p > -eps) return true;
                p = 0;
                for (int i = 1; i < n; ++i) if (a[r][i] < p) p = a[r][c = i];
                if (p > -eps) return false;
                p = a[r][n] / a[r][c];
                for (int i = r + 1; i < m; ++i) if (a[i][c] > eps) {
                        double v = a[i][n] / a[i][c];
                        if (v < p) r = i, p = v;
                }
                pivot(r, c);
        }
}

int simplex() // 0 - no solution, -1 - infinity, 1 - has a solution
{
        for (int i = 0; i < n; ++i) N[i] = i;
        for (int i = 0; i < m; ++i) B[i] = n + i;
        if (!feasible()) return 0;
        for (;;) {
                int r, c;
                double p = 0;
                for (int i = 0; i < n; ++i) if (a[m][i] > p) p = a[m][c = i];
                if (p < eps) break;
                p = inf;
                for (int i = 0; i < m; ++i) if (a[i][c] > eps) {
                        double v = a[i][n] / a[i][c];
                        if (v < p) r = i, p = v;
                }
                if (p == inf) return -1;
                pivot(r, c);
        }
        for (int i = 0; i < n; ++i) if (N[i] < n) x[N[i]] = 0;
        for (int i = 0; i < m; ++i) if (B[i] < n) x[B[i]] = a[i][n];
        ans = -a[m][n];
        return 1;
}
```

Appendix A

Snippets

```
const int INF = 0x3f3f3f3f; // ==  1061109567  < INT_MAX / 2
const int INF = 0x2a2a2a2a; // ==  707406378   < INT_MAX / 3
const int INF = 0xc2c2c2c2; // == -1027423550  > INT_MIN / 2
const int INF = 0xd5d5d5d5; // == -707406379   > INT_MIN / 3

// gcc stack enlarging
char *p = (char *)malloc(size) + size;
__asm("movl %0,%%esp\n" :: "r"(p));
// vc stack enlarging
#pragma comment(linker, "/STACK:16777216")
```

Appendix B

Java Example

```java
import java.util.Scanner;
import java.util.Arrays;
import java.math.BigInteger; // or BigDecimal

public class Main {
    public static void main(String[] args) {
        Scanner cin = new Scanner(System.in);
        int n = cin.nextInt();
        System.out.print(n);
        System.out.println(n);

        int[] arr = new int[5];
        int[] arrr = {1, 2, 3, 4, 5};
        int[][] f = new int[n][n];
        Arrays.sort(arrr);

        BigInteger a = cin.nextBigInteger();
        BigInteger b = BigInteger.valueOf(2);
        a = a.add(b);       a = a.subtract(b);   a = a.negate();
        a = a.multiply(b);  a = a.divide(b);     a = a.mod(b);
        a = a.shiftLeft(1); a = a.shiftRight(1);
        if (a.compareTo(b) < 0) System.out.println("a < b");
    }
}
```

Appendix C

Vim Configuration

```
se nocp nu cin ts=4 sw=4
syn on

se mp=g++\ -g\ -o\ %<\ %\ -Wall\ -std=c++11
map mk :make<cr>
map mr :!./%<<cr>
map mw :!./%< < %<.in<cr>
map mi :sp %<.in<cr>
```