Code Template for ACM-ICPC

wodesuck
SYSU_ArthasMenethil
Sun Yat-sen University

September 17, 2014

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 f)
{
	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 != f) {
			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;
                        grp[v] = cnt;
                } while (v != u);
                ++cnt;
        }
}
```

1.4 Lowest Common Ancestor(Tarjan)

```
void dfs(int u)
{
        anc[u] = u; v[u] = 1;
        for (edge *i = e[u]; i; i = i->next) {
                if (!v[i->t]) {
                        dfs(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)

```
int esz, psz, s, t;
int h[MAXV], vh[MAXV + 1];

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;
}

/*
Initialize:
        psz = 0; memset(e, 0, sizeof(e));
```

```
Usage:
        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)

```
int esz, psz, s, t;
int cost, dist, d[MAXV];
bool v[MAXV];

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

bool modlabel()
{
        deque<int> q;
        memset(v, 0, sizeof(v));
        memset(d, 0x3f, sizeof(d));
        q.push_back(s); d[s] = 0; v[s] = true;
        while (!q.empty()) {
                int u = q.front(); q.pop_front(); v[u] = false;
                for (edge *i = e[u]; i; i = i->next) {
                        if (i->u && d[u] + i->c < d[i->t]) {
                                d[i->t] = d[u] + i->c;
                                if (!v[i->t]) {
                                        if (q.empty() || d[i->t] < d[q.front()])
                                            {
                                                q.push_front(i->t);
                                        } else {
                                                q.push_back(i->t);
                                        }
                                        v[i->t] = true;
                                }
                        }
                }
        }
        if (d[t] == inf) return false;
        for (int i = 0; i < esz; ++i) {
                for (edge *j = e[i]; j; j = j->next) {
                        j->c += d[i] - d[j->t];
                }
        }
        dist += d[t];
        return true;
}

/*
```

```
Initialize:
	psz = 0; memset(e, 0, sizeof(e));
	cost = dist = 0;
Usage:
	while (modlabel()) {
		do memset(v, 0, sizeof(v));
		while (aug(s, INT_MAX));
	}
*/
```

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

```
int n; // n vars
struct edge {
```

```
        int t;
        edge *next;
}epool[CONDITIONS * 2], *e[MAXN * 2], *e2[MAXN * 2];
// sat variable Bi and !Bi are encoded as i<<1 and i<<1^1
int psz;
int stamp, dfn[MAXN * 2], low[MAXN * 2];
int top, sta[MAXN * 2];
bool ins[MAXN * 2];
int cnt, grp[MAXN * 2];
int deg[MAXN * 2], con[MAXN * 2], mrk[MAXN * 2];
int head, tail, que[MAXN * 2];
// Bi is true if mrk[grp[i << 1]] == 1

bool chk()
{
        cnt = 0; stamp = 0; top = 0;
        memset(dfn, 0, sizeof(dfn));
        memset(low, 0, sizeof(low));
        memset(ins, 0, sizeof(ins));
        memset(grp, 0, sizeof(grp));
        for (int i = 0; i < (n<<1); ++i) if (!dfn[i]) tarjan(i);
        for (int i = 0; i < n; ++i) {
                int u = grp[i<<1], v = grp[(i<<1)^1];
                if (u == v) return false;
                con[u] = v; con[v] = u;
        }
        return true;
}

void group()
{
        memset(e2, 0, sizeof(e2));
        memset(deg, 0, sizeof(deg));
        for (int i = 0; i < (n<<1); ++i) {
                for (edge *j = e[i]; j; j = j->next) {
                        int x = grp[i], y = grp[j->t];
                        if (x == y) continue;
                        add_edge2(y, x);
                        ++deg[x];
                }
        }
}

void tsort()
{
        memset(mrk, 0, sizeof(mrk));
        for (int i = 0; i < cnt; ++i) if (!deg[i]) que[tail++] = i;
        while (head < tail) {
                int u = que[head++];
                if (!mrk[u]) mrk[u] = 1, mrk[con[u]] = -1;
                for (edge *i = e2[u]; i; i = i->next) {
                        if (!--deg[i->t]) que[tail++] = i->t;
                }
        }
}

/*
Initialize:
        psz = 0; memset(e, 0, sizeof(e));
        add_edge(i<<1, j<<1^1); for Bi -> !Bj
Usage:
```

```
        if (chk()) {
                group();
                tsort();
        } else {
                // No solution
        }
*/
```

1.8 Divide and Conquer for Tree

poj 1741 Tree

```
#include <stdio.h>
#include <string.h>
#include <algorithm>

using namespace std;

int n, k;
const int MAXN = 10001;
struct edge {
        int t, w;
        edge *next;
}epool[MAXN * 2], *e[MAXN];
int psz;
int rec[MAXN], tot;
int size[MAXN], msize[MAXN], dist[MAXN];
bool del[MAXN];
int w[MAXN], c[MAXN];
int ans;

void getsize(int p, int fa)
{
        rec[tot++] = p; size[p] = 1; msize[p] = 0;
        for (edge *i = e[p]; i; i = i->next) {
                if (i->t == fa || del[i->t]) continue;
                getsize(i->t, p);
                size[p] += size[i->t];
                msize[p] = max(msize[p], size[i->t]);
        }
}

int centroid(int p)
{
        tot = 0;
        getsize(p, 0);
        int k = rec[0];
        for (int i = 0; i < tot; ++i) {
                int t = rec[i];
                msize[t] = max(msize[t], tot - size[t]);
                if (msize[t] < msize[k]) k = t;
        }
        return k;
}

void getdist(int p, int fa)
{
        for (edge *i = e[p]; i; i = i->next) {
                if (i->t == fa || del[i->t]) continue;
                w[tot++] = dist[i->t] = dist[p] + i->w;
                getdist(i->t, p);
```

```
        }
}

void add(int p, int x)
{
        for (; p <= tot; p += p & -p) c[p] += x;
}

int sum(int p)
{
        int s = 0;
        for (; p; p -= p & -p) s += c[p];
        return s;
}

void dfs1(int p, int fa)
{
        ans += sum(upper_bound(w, w + tot, k - dist[p]) - w);
        for (edge *i = e[p]; i; i = i->next) {
                if (i->t != fa && !del[i->t]) dfs1(i->t, p);
        }
}

void dfs2(int p, int fa)
{
        add(upper_bound(w, w + tot, dist[p]) - w, 1);
        for (edge *i = e[p]; i; i = i->next) {
                if (i->t != fa && !del[i->t]) dfs2(i->t, p);
        }
}

void solve(int p)
{
        p = centroid(p);
        del[p] = true;

        dist[p] = 0; tot = 0;
        getdist(p, 0);
        sort(w, w + tot);
        tot = unique(w, w + tot) - w;
        fill(c, c + tot + 1, 0);

        for (edge *i = e[p]; i; i = i->next) {
                if (del[i->t]) continue;
                dfs1(i->t, p);
                dfs2(i->t, p);
        }
        ans += sum(upper_bound(w, w + tot, k) - w);

        for (edge *i = e[p]; i; i = i->next) {
                if (!del[i->t]) solve(i->t);
        }
}

void add_edge(int u, int v, int w)
{
        edge *tmp = epool + psz++;
        tmp->t = v; tmp->w = w; tmp->next = e[u]; e[u] = tmp;
}

void init()
```

```
{
        psz = 0; ans = 0;
        memset(e, 0, sizeof(e));
        memset(del, 0, sizeof(del));
        for (int i = 1; i < n; ++i) {
                int u, v, w;
                scanf("%d%d%d", &u, &v, &w);
                add_edge(u, v, w);
                add_edge(v, u, w);
        }
}

int main()
{
        while (scanf("%d%d", &n, &k) != EOF && (n || k)) {
                init();
                solve(1);
                printf("%d\n", ans);
        }
}
```

1.9 Heavy-Light Decomposition

```
void dfs1(int p)
{
        size[p] = 1; h[p] = -1;
        for (edge *i = e[p]; i; i = i->next) {
                if (i->t == f[p]) continue;
                f[i->t] = p;
                d[i->t] = d[p] + 1;
                dfs1(i->t);
                size[p] += size[i->t];
                if (h[p] < 0 || size[i->t] > size[h[p]]) h[p] = i->t;
        }
}

void dfs2(int p, int anc)
{
        top[p] = anc;
        if (h[p] >= 0) dfs2(h[p], anc);
        for (edge *i = e[p]; i; i = i->next) {
                if (i->t != f[p] && i->t != h[p]) dfs2(i->t, i->t);
        }
}

int lca(int u, int v)
{
        while (top[u] != top[v]) {
                if (d[top[u]] < d[top[v]]) swap(u, v);
                u = f[top[u]];
        }
        if (d[u] > d[v]) swap(u, v);
        return u;
}

/*
Data:
        f[]    -- father
        d[]    -- depth
        size[] -- substree size
```

```
        h[]     -- heavy child
        top[]   -- head node of chain
Initalize:
        f[1] = -1;
        d[1] = 0;
*/
```

Chapter 2

Data Structures

2.1 Segment Tree

2.1.1 zkw Segment Tree

```
// M should be 2^k, and should satisfy M >= n + 2,
// where the range of query indexes is [1, n]
int T[M<<1];

void change(int p, int x)
{
        v[p += M] = x;
        while (p>>=1) T[p] = max(T[p<<1], T[p<<1^1]);
}

int query(int s, int t)
{
        int l = INT_MIN, r = INT_MIN;
        for (s += m - 1, t += m + 1; s^t^1; s>>=1, t>>=1) {
                if (~s&1) l = max(w, T[s^1]);
                if ( t&1) r = max(w, T[t^1]);
        }
        return max(l, r);
}
```

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);
}

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);
}
```

2.2.2 Splay

```
void zig(int t)
{
        int p = parent[t], g = parent[p];
        if (right[t]) parent[right[t]] = p;
        left[p] = right[t]; right[t] = p;
        parent[p] = t; parent[t] = g;
        update(p); update(t);
        if (g) p == left[g] ? left[g] = t : right[g] = t;
}

void zag(int t)
{
        int p = parent[t], g = parent[p];
```

```
        if (left[t]) parent[left[t]] = p;
        right[p] = left[t]; left[t] = p;
        parent[p] = t; parent[t] = g;
        update(p); update(t);
        if (g) p == left[g] ? left[g] = t : right[g] = t;
}

void splay(int t, int header = 0) // header = parent[root]
{
        int p = parent[t], g = parent[p];
        for (; p != header; p = parent[t], g = parent[p]) {
                if (g == header) t == left[p] ? zig(t) : zag(t);
                else if (p == left[g]) t == left[p] ? zig(p) : zag(t), zig(t);
                else t == right[p] ? zag(p) : zig(t), zag(t);
        }
}
```

2.2.3 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));
}
```

2.2.4 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

```
int left[MAXN], right[MAXN], parent[MAXN], size[MAXN];
bool rev[MAXN];

void update(int t)
{
        size[t] = size[left[t]] + size[right[t]] + 1;
}

void revsub(int t)
{
        swap(left[t], right[t]);
        rev[t] ^= 1;
}

void sink(int t)
{
        if (rev[t]) {
                if (left[t]) revsub(left[t]);
                if (right[t]) revsub(right[t]);
                rev[t] = false;
        }
}

void sinkdown(int t)
{
        static int path[MAXN];
        int n = 0;
        for (;;) {
                path[n++] = t;
                int p = parent[t];
                if (t != left[p] && t != right[p]) break;
                t = p;
        }
        while (n) sink(path[--n]);
}

void zig(int t)
{
        int p = parent[t], g = parent[p];
        if (right[t]) parent[right[t]] = p;
        left[p] = right[t]; right[t] = p;
        parent[p] = t; parent[t] = g;
```

```
        update(p); update(t);
        if (p == left[g]) left[g] = t;
        else if (p == right[g]) right[g] = t;
}

void zag(int t)
{
        int p = parent[t], g = parent[p];
        if (left[t]) parent[left[t]] = p;
        right[p] = left[t]; left[t] = p;
        parent[p] = t; parent[t] = g;
        update(p); update(t);
        if (p == left[g]) left[g] = t;
        else if (p == right[g]) right[g] = t;
}

void splay(int t)
{
        sinkdown(t);
        for (;;) {
                int p = parent[t], g = parent[p];
                if (t == left[p]) {
                        if (p == left[g]) zig(p), zig(t);
                        else if (p == right[g]) zig(t), zag(t);
                        else zig(t);
                } else if (t == right[p]) {
                        if (p == left[g]) zag(t), zig(t);
                        else if (p == right[g]) zag(p), zag(t);
                        else zag(t);
                } else break;
        }
}

int expose(int t)
{
        int u = 0;
        for (; t; u = t, t = parent[t]) {
                splay(t);
                right[t] = u;
                update(t);
        }
        return u;
}

void link(int t, int p) // link subtree t to p
{
        parent[expose(t)] = p;
        expose(t);
}

void cut(int t)
{
        expose(t);
        splay(t);
        parent[left[t]] = 0;
        left[t] = 0;
        update(t);
}

void setroot(int t)
{
```

```
        revsub(expose(t));
}

void query(int u, int v) // query path u -> v
{
        expose(u);
        int t = expose(v); // t == lca(u, v)
        // analysis node t
        // analysis right[t]
        if (u != t) {
                splay(u);
                // analysis u
        }
}

void change(int u, int v) // path change
{
        expose(u);
        int t = expose(v); // t == lca(u, v)
        // change node t
        if (right[t]) // change right[t]
        if (u != t) {
                splay(u);
                // change u
        }
}
```

2.4.2 Euler Tour Tree

```
int n;
int left[MAXN * 2], right[MAXN * 2], parent[MAXN * 2], size[MAXN * 2];
// each node split into 2 nodes. i --> (i<<1) && (i<<1)^1

void update(int t) { size[t] = size[left[t]] + size[right[t]] + 1; }

int root(int t) { while (parent[t]) t = parent[t]; return t; }
int minnode(int t) { while (left[t]) t = left[t]; return t; }
int maxnode(int t) { while (right[t]) t = right[t]; return t; }

int prev(int t)
{
        if (left[t]) return maxnode(left[t]);
        int p = parent[t];
        while (p && t == left[p]) t = p, p = parent[t];
        return p;
}

int succ(int t)
{
        if (right[t]) return minnode(right[t]);
        int p = parent[t];
        while (p && t == right[p]) t = p, p = parent[t];
        return p;
}

void cut(int t)
{
        int x = prev(t<<1), y = succ(t<<1^1);
        splay(x); splay(y, x);
        parent[left[y]] = 0; left[y] = 0;
        update(y); update(x);
```

```
}

void link(int t, int p) // link subtree t to p
{
        p <<= 1; t = root(t<<1);
        splay(p); splay(minnode(right[p]), p);
        parent[t] = right[p]; left[right[p]] = t;
        update(right[p]); update(p);
}
```

2.5 KD Tree

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

double sqr(double n) { return n * n; }

double dis(kd a, 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, 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 f[MAXN][LOGN];

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

int rmq(int l, int r)
{
	int k = 0;
	while ((1<<(k+1)) <= r - l + 1) ++k;
	return min(d[l][k], d[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->f->ch[i];
	}
	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; i = i->last) {
                //      sum += i->val, i->val = 0;
                // }                # method 2
        }
        return sum;
}

/*
Initialize:
        psz = 1; memset(dict = pool, 0, sizeof(trie));
Method 1: counting appearance times
Method 2: counting appear patterns
** pattern that appear more than once counted 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 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(Gramham)

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

int gramham(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 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[])
{
	int head = 0, tail = 0, m = 0;
	for (int i = 0; i < n; ++i) q[i].a = arg(dir(q[i]));
	sort(ln, ln + 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);
}
```

4.5 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)
*/
```

4.6 Farthest Pair(Rotating Caliper)

```
double maxdis(point *p, int n)
{
        gramham(p, n, h, m);
        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;
}
```

4.7 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 = n + n; 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;
                }
        }
}
```

4.8 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);
}
```

4.9 Union Area of Circles

```
bool incir(circle a, circle b)
{ return dcmp(abs(a.c - b.c) + a.r - b.r) <= 0; }

void unionarea(circle c[], int n, double tot[])
{
        static pair<double, int> a[MAXN * 2];
        memset(tot, 0, sizeof(tot));
        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 (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
*/
```

4.10 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[])
{
	memset(tot, 0, sizeof(tot));
	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
*/
```

4.11 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);
}
```

4.12 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);
}
```

4.12.1 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;
```

```
}
```

4.12.2 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;
}
```

4.13 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;
}
```

Chapter 5

Number Theory

5.1 Fast Fourier Transform

```
typedef complex<double> cp;
void fft(cp *a, int n, int f)
{
        static cp b[MAXN];
        double arg = pi;
        for (int k = n >> 1; k; k >>= 1, arg *= 0.5) {
                cp wm = polar(1.0, f * arg), w(1, 0);
                for (int i = 0; i < n; i += k, w *= wm) {
                        int p = i << 1;
                        if (p >= n) p -= n;
                        for (int j = 0; j < k; ++j) {
                                b[i + j] = a[p + j] + w * a[p + k + j];
                        }
                }
                for (int i = 0; i < n; ++i) a[i] = b[i];
        }
}

/*
Usage:
fft(a, n, 1); -- dft
fft(a, n, -1); for(i) a[i]/=n; -- 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 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;
}
```

6.4 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;
	}
}
```

6.5 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);
}
```

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
se go= bs=2 "␣for␣gvim
syn␣on
```